Perspectives Technical Documentation

Table of Contents

1. Introduction to the docs . . . 2
1.1. Organization of this document . . . 2
1.2. Other resources . . . 3
2. Introduction to the PDR . . . 4
2.1. What we cover in this chapter . . . 4
2.2. About Perspectives . . . 4
2.3. Architecture: requirements . . . 5
2.3.1. Perspectives runtime . . . 5
2.3.2. Persistent store of information . . . 5
2.3.3. Authentication provider . . . 6
2.4. Connections between components . . . 6
2.4.1. UI – PR . . . 6
2.4.2. PR – Store . . . 7
2.4.3. PR – PR . . . 7
2.5. Actual architecture and underlying technologies . . . 8
2.6. Security Concerns . . . 9
PART I. HIGH LEVEL SYSTEM OVERVIEW . . . 11
3. Parser and Compiler . . . 12
3.1. Understanding a Perspective source text . . . 12
3.1.1. Types and scopes . . . 12
3.1.2. Perspectives . . . 12
3.1.3. Notifications and state . . . 14
3.1.4. More on states transitions; automatic effects . . . 15
3.1.5. Perspective and state . . . 18
3.1.6. About expressions and variables . . . 20
3.2. Parsing and compiling models . . . 27
3.2.1. Introduction . . . 27
3.2.2. Constructing a DomeinFile . . . 28
3.2.3. Transformation phases . . . 28
3.2.4. Interdependencies of models; packages . . . 30
3.2.5. Putting a model into operation for a PDR installation . . . 31
4. Functional Reactive Pattern . . . 34
4.1. The underlying network and queries . . . 34
4.2. Client requests . . . 34
4.2.1. Consequence of request order . . . 35
4.2.2. How to visit a context . . . 35
4.2.3. Deleting a context . . . 36
4.3. Queries . . . 36

4.3.1. Queries trace their steps . . . 36
4.4. Adding a role instance . . . 37
4.5. Deleting a role instance . . . 37
4.6. Changing the role binding . . . 38
4.7. Changing property values . . . 39
4.8. Adding a context . . . 39
4.9. The external function . . . 40
5. Assignment and state change . . . 41
5.1. Overview of state change mechanisms . . . 41
5.1.1. Introduction . . . 41
5.1.2. Five responsibilities . . . 41
5.1.3. Mechanisms . . . 42
5.2. Assignment . . . 44
5.2.1. Current context . . . 45
5.2.2. Assignment statements for roles . . . 45
5.2.3. Creating contexts . . . 49
5.2.4. Assignment statements for properties . . . 50
5.2.5. Why there is different syntax for properties and roles . . . 52
6. Synchronization . . . 53
6.1. Perspectives across context boundaries . . . 53
6.1.1. The importance of context . . . 53
6.1.2. Perspectives across borders . . . 53
6.1.3. Computed user roles . . . 54
6.1.4. Equivalence . . . 54
6.1.5. Assignment . . . 55
6.2. Query Inversion . . . 55
6.2.1. Inverting queries . . . 56
6.2.2. A mechanism for inverting a query . . . 56
6.2.3. Where we store inverted queries and how we use them . . . 57
6.2.4. Implementation complication: two types of trees . . . 59
6.2.5. Some cases . . . 59
6.2.6. What is inverted? . . . 63
PART II. THE LANGUAGE: DESIGN DECISIONS . . . 65
7. The Type System . . . 66
7.1. Semantics of the Perspectives Language . . . 66
7.1.1. Context . . . 66
7.1.2. Property . . . 66
7.1.3. Role . . . 66
7.1.4. Ordering of Role types as property sets . . . 69
7.1.5. Another ordering of Role types . . . 69
7.1.6. Creating products and sums of roles . . . 71

7.2. Abstract Data Type 73
7.2.1. Introduction 73
7.2.2. Representation 73
7.2.3. Ordering and type specificity 74
7.2.4. Property Sets 74
7.2.5. Other sets: views and aspects 75
7.2.6. Functions on ADT’s 75
7.3. Type reflection 78
7.3.1. Introduction 78
7.3.2. Responsibilities of authors 78
7.3.3. Responsibilities of the authors of screens 79
7.3.4. Reflection on model:System is guaranteed 79
7.3.5. Type reflection on locally created instances is guaranteed 79
7.3.6. Type reflection on instances made by peers is guaranteed 79
7.3.7. When the user fills a role 80
7.3.8. Type reflection on instances from public storage 80
8. Identifiers and Variables 82
8.1. Expanding prefixed names 82
8.1.1. Introduction 82
8.1.2. Declaring prefixes 82
8.1.3. Syntactic locations that support prefix expansion 82
8.1.4. Syntactic locations that don’t support prefix expansion 83
8.1.5. Implementation notices 83
8.2. Type and Resource Identifiers 83
8.2.1. Some definitions 84
8.2.2. On Uniform Resource Identifiers 84
8.2.3. Model identification 85
8.2.4. Model Manifest: a public context 89
8.2.5. Type identifiers 89
8.2.6. Instance identifiers 92
8.2.7. DomeinFile identifiers 93
8.2.8. Public instance identification 95
8.2.9. The default repository 99
8.2.10. Cross Origin Resource Sharing 100
8.3. Mapping Model Identifiers to Storage Locations 102
8.3.1. Pre-release information in semantic version numbers 103
8.3.2. Including pre-release information in the mapping 103
8.4. Current context and current role 104
8.4.1. What is a query applied to? 104
8.4.2. Re-basing the path 104
8.5. Indexed Names 105

8.5.1. Introduction 105
8.5.2. Modelling indexed names 106
8.5.3. Compiling a query with an indexed name 107
8.5.4. Starting to use a model 107
8.5.5. Looking up indexed names in runtime 109
8.5.6. How to write the right instances 109
8.6. Standard variables: technical design decisions 111
8.6.1. Lexical concepts in the state of the parser 111
8.6.2. Lexcial concepts in parse tree elements 112
8.6.3. Standard variables: the lexical concepts in QueryFunctionDescriptions 113
8.6.4. Actually computing the current context 116
8.6.5. Future extensions: currentobjects, currentsubjects 116
8.7. Unlinked Roles 117
8.7.1. Retrieve with a query instead of by identifier 117
8.7.2. Retrieving role instances by Aspect name 117
9. Models 119
9.1. Booting models 119
9.1.1. Models 119
9.1.2. Authoring a model: booting revisited 120
9.1.3. Booting the Perspectives Universe 121
9.2. Model Versions and Compatibility 122
9.2.1. Compatibility 123
9.2.2. Backward compatibility with respect to user data 123
9.2.3. Compatibility issues with respect to data received from peers 127
9.2.4. Solutions 128
9.3. Revision of Couchdb documents 131
9.3.1. Overview of the flow of entities and documents 131
9.3.2. Maintaining consistent revisions 132
10. Perspectives 134
10.1. The Operational Meaning of a Perspective 134
10.1.1. Perspective on an Enumerated Role 134
10.1.2. Compound binding: Binding Role Graph (BRG) 135
10.1.3. Perspective on a Calculated Role 135
10.1.4. Serialisation for Calculated Roles 136
10.2. Contextualizing Queries 136
10.2.1. Aspects 136
10.2.2. Solution 138
10.2.3. No consequences for serialization 139
10.3. Contextualization 139
10.3.1. Compile time or run time? 140
10.3.2. Contextualising a Perspective 140

10.3.3. State complicates matters . . . 142
10.3.4. Contextualisation of queries . . . 143
10.3.5. Contextualisation of actions . . . 144
10.3.6. Contextualization of automatic actions and notifications . . . 148
10.3.7. Contextualization of Properties . . . 148
10.3.8. Summary . . . 149
10.4. Universal Perspectives . . . 149
10.4.1. Public context . . . 150
10.4.2. URLS . . . 150
10.4.3. Identifying public contexts . . . 151
10.4.4. Multiple storage locations . . . 152
10.4.5. Authoring public contexts and roles . . . 155
10.4.6. Actions to create private contexts and roles from public ones . . . 155
10.5. Perspectives on role fillers . . . 156
10.5.1. Access to a property considered to be a query . . . 157
10.5.2. Binding . . . 158
10.5.3. Solving the general case . . . 158
10.5.4. A practical algorithm . . . 160
10.5.5. Relevant properties revisited . . . 162
11. Queries . . . 163
11.1. Binder versus Filler terminology . . . 163
11.1.1. Representation . . . 163
11.1.2. Functions: binding, binder <Role> . . . 163
11.1.3. Language: fills, filledBy . . . 163
11.1.4. Relation between the two; cardinality . . . 163
11.1.5. Inverted queries . . . 164
11.1.6. When we severe a connection . . . 164
11.1.7. Direction: in and out of the context . . . 164
11.1.8. When we remove a role . . . 165
11.2. State And Dependency Tracking . . . 165
11.2.1. Functions cached in global variables . . . 166
11.2.2. Dependency tracking administration . . . 166
11.3. Composing instance level and type level query functions . . . 167
11.3.1. Instance level functions . . . 168
11.3.2. Type level functions . . . 168
11.3.3. The problem, revisited . . . 169
11.3.4. End result: lifting from instance- to type level . . . 170
11.4. The Case for Database Query Roles . . . 171
11.4.1. Database Query Roles retrieve context type instances . . . 171
11.4.2. How Database Query Roles are different . . . 171
11.5. `selfOnly` implies `mineOnly` . . . 172

11.5.1. Mine only! 172
11.5.2. Other user roles 173
11.5.3. Exploring an improvement: "only for those playing a role" 173
PART III. IMPLEMENTATION DETAILS 175
12. PDR architecture 176
12.1. A Transport Layer for the PDR 176
12.1.1. The problem to be solved 176
12.1.2. Roadmap: distributed or not? 176
12.1.3. Current state 177
12.1.4. Alternatives 177
12.1.5. Proposed solution 179
12.2. STOMP 179
12.2.1. Technology chosen 180
12.2.2. Exchange type 180
12.2.3. Creating topic queues 180
12.2.4. Acknowledgements 181
12.2.5. Heartbeat 181
12.2.6. User accounts 182
12.3. Configuring Couchdb 182
12.3.1. Transport Layer Security 182
12.3.2. Configuring https for Couchdb on your computer 184
12.3.3. Apache proxy workaround for Couchb tls problems 185
12.4. Multiple databases and devices 186
12.4.1. A refresher: structural connections in Perspectives 186
12.4.2. Multiple databases 187
12.4.3. Linked and non-linked connections 187
12.4.4. Multiple Transaction Handling Points 188
12.4.5. Multiple User Interfaces 189
12.4.6. Multiple devices 190
12.4.7. Modelling freedom 191
12.5. Error Handling 191
12.5.1. Sources of errors 192
12.6. User and System Identifier 194
12.6.1. The origin of the system identifier 194
12.6.2. Persistence of system identifier, user name and password 194
12.6.3. System identifier as base name 195
12.6.4. Putting the systemIdentifier in PerspectivesState 195
13. A proxy for the PDR 197
13.1. Overview 197
13.2. Configuring the PDR proxy 197
13.3. SharedWorkerChannel and ChannelID 197

13.3.1. ChannelID . . . 198

13.3.2. Calling and receiving responses: Correlation identifiers . . . 198

13.3.3. Handling PDR responses . . . 198

13.3.4. Streams versus one-time-only responses . . . 198

13.4. SharedWorker . . . 199

13.4.1. Connection to the PDR: the InternalChannel . . . 200

13.4.2. A dance in four steps . . . 200

13.5. Tracing the entire process from client to PDR and back . . . 200

14. Modifying State . . . 202

14.1. State in Perspectives . . . 202

14.1.1. The use of states . . . 202

14.1.2. Role state versus context state . . . 203

14.1.3. How to use state in models . . . 204

14.1.4. Alternative modelling . . . 205

14.1.5. How to make it work . . . 205

14.2. State Change . . . 208

14.2.1. Model files . . . 209

14.2.2. Cache, database and synchronisation: mechanisms . . . 209

14.2.3. The dynamics of state change . . . 209

14.3. Execution Model . . . 210

14.3.1. Statements execute in order of appearance . . . 211

14.3.2. States associated with resource creation and deletion . . . 211

14.3.3. Staged evaluation of state changes . . . 212

14.3.4. Removing a role instance . . . 213

14.3.5. Removing a context instance . . . 214

14.3.6. Refining understanding of resource removal . . . 216

14.4. Creating and deleting contexts . . . 218

14.4.1. Local and remote . . . 218

14.4.2. Unbound contexts . . . 220

14.4.3. Removing and deleting contexts . . . 221

14.4.4. Synchronisation: delta representation . . . 222

15. Synchronization revisited . . . 226

15.1. Synchronisation . . . 226

15.1.1. What procedures to call? . . . 227

15.1.2. Types of Deltas . . . 228

15.1.3. Executing Deltas . . . 229

15.1.4. Declarative versus procedural synchronization . . . 229

15.1.5. Design considerations per DeltaFunction . . . 230

15.2. In depth treatment of synchronisation . . . 231

15.2.1. Different perspectives on non-user roles . . . 231

15.2.2. Perspectives on users: the *user graph* . . . 232

15.2.3. A synchronisation principle: *passing on* 233
15.2.4. Computing users to pass on to 234
15.2.5. Adding new peers 235
15.2.6. Implicit perspectives 241
15.2.7. Perspectives over model boundaries 243
15.2.8. An alternative way to *pass on* 244
15.2.9. Connecting users 246
15.3. Synchronizing subnetworks due to role filling 248
15.3.1. Binding: the way to build large structures 248
15.3.2. General approach 249
15.3.3. Finding nodes to be sent from query assumptions 250
15.3.4. Property set 251
15.4. Aspects require refinement of Inverted Queries 251
15.4.1. The problem in detail 251
15.4.2. Making sense of this situation 253
15.4.3. Solution 254
15.4.4. Comparison to Aspect Perspectives 262
15.4.5. Keys for aspects 263
15.5. Who authors? 264
15.5.1. Automatic actions 264
15.5.2. API calls 265
15.6. Query Inversion over Model Boundaries 265
15.6.1. Alternatives 266
15.6.2. Design 266
15.7. Optimizing Inverted Query Application 267
15.7.1. Problem statement 267
15.7.2. Solution 267
16. Extensions 269
16.1. External Function Interface 269
16.1.1. Outline of the design 269
16.1.2. Technical details 271
16.2. The Practical Guide to Writing an EFI module 272
16.2.1. Create a Core External Module 272
16.2.2. Declare function names in the Core External Module 273
16.2.3. The default argument 273
16.2.4. The other arguments 274
16.2.5. CallEffect 274
16.2.6. The type of the external functions 275
17. Various topics 276
17.1. The "mode" option on Fetch 276
17.2. Internationalization (I18n) and Localization (L10n) 276

17.2.1. What texts should be translatable? 276
17.2.2. Design decisions 277
17.2.3. Dynamic loading of language resources 277
17.2.4. MyContexts and perspectives-react 277
17.2.5. Namespaces for MyContexts and perspectives-react 278
17.2.6. To add a language 278
17.3. The range type PFile 278
17.3.1. A pattern to restrict MIME types 279
17.3.2. Creating a file in an assignment statement 279
17.3.3. Allowed MIME types 280
17.3.4. Allowed file names 281
17.3.5. Selecting files based on their name or MIME type 281
17.3.6. How files are stored 281
17.3.7. Handling files client side 282
17.3.8. The PerspectivesFile React Component 282
17.3.9. Storing files through the PerspectivesProxy 284
17.4. When the model ecosystem is broken 284
17.4.1. MyContexts lost, models and public instances available 284
17.4.2. MyContexts lost, just models are available 285
17.4.3. Everything is lost 285
17.4.4. Public databases have been lost but the MyContexts installation is intact 285
17.4.5. Precautions to be taken 285
17.5. Restoring from public resources 286
17.5.1. Referenced roles and their contexts 286
17.5.2. Backlinks 286
17.5.3. Limitations 286
17.5.4. Automatic actions 287
17.5.5. Triggering restoration 288
17.6. When resources cannot be stored or retrieved 288
17.6.1. Recognise the various situations 288
17.6.2. Options for handling these cases 289
PART IV. ESSENTIAL MODELS 290
18. CouchdbManagement 291
18.1. Overview 291
18.2. An aside: how does this work? 292
18.3. URL Rewriting Revisited 293
18.4. Naming instances 293
18.5. Configuring an Apache VirtualHost running Couchdb for a domain of perspectives models 293
18.5.1. Forwarding requests to Couchdb 294
18.5.2. Preflight requests 294

18.5.3. Protecting read databases 295
18.5.4. Certificates 296
18.5.5. Putting it all together 296
18.6. Configuring an Apache VirtualHost WITHOUT Couchdb for a domain of perspectives models 297

This is the documentation of the design of the Perspectives Language and the implementation of the Perspectives Distributed Runtime (PDR). Alternatively, open the PDF version instead.

Chapter 1. Introduction to the docs

This documentation grew out of some 50 design documents written over the course of the years from 2016 to 2022. They were first integrated into this single document in oktober 2022, made possible by a grant from the NGI ZERO Search program through the NLnet organization.

From that date on, the individual design documents are no longer updated: just this (online) documentation is.

> **NOTE** This documentation is a work in progress! If you think a topic is missing, please create an issue here.

1.1. Organization of this document

This document is organized into three parts.

PART I gives a high level overview of the Perspectives Distributed Runtime. It is organised around a functional decomposition of the Perspectives Distributed Runtime into four parts:

- The parser / compiler
- The Functional Reactive Pattern (FRP)
- State change through assignment
- Synchronization

PART II is devoted to the *design* of the Perspectives Language. It ranges from high level conceptual overviews - such as the *type system* - to implementation details. The topics covered are:

- The type system
- Identifiers and variables
- Perspectives
- Queries
- Aspects
- Model source texts

PART III goes into the nitty-gritty of the implementation. Expect the most technical issues here. We cover:

- The architecture of the PDR and MyContexts
- The mechanisms of state change
- The workings of state synchronization between peers
- The Foreign Function Interface (FFI)

PART IV discusses a number of models that are essential to the functioning of MyContexts, such as:

- model://perspectives.domains#System
- model://perspectives.domains#CouchdbManagement

1.2. Other resources

There are quite a few other resources on the Perspectives project.

- The starting page for all Perspectives documentation.
- The MyContexts User Guide, which is more of a reference.
- A small tutorial to get started.
- Reference of the Perspectives Language

NOTE If you find something wrong with this documentation, please create an issue at the Issues section on Github. Use the label `documentation`.

Chapter 2. Introduction to the PDR

2.1. What we cover in this chapter

The purpose served by this text is to provide a clear view of

- the functional components that make up the Perspectives software stack;
- the connections between them
- the required and / or desired non-functional characteristics of these connections.

The suitability of concrete technologies for the Perspectives Stack can be judged by comparing them with these requirements. This text is informal (and imprecise) in the sense that the protocols and message formats between components is not given in detail.

2.2. About Perspectives

Let’s introduce Perspectives in terms of some questions and their answers.

What is Perspectives for? We have created a modelling language from which working applications can be generated, that with some fine tuning of screens can have real practical usage. The language is well suited for a range of programs that support human co-operation; it is not meant to be a general programming language. Nor is it a high-throughput computing platform: the human pace of information processing / generation is the norm.

What is it like, conceptually? Think Access List Control (ACL). IT professionals are familiar with ACL as a way to describe and enforce limited access to computing resources for a variety of end users. ACL systems are applied to existing systems, as an add-on. With Perspectives we turn that on its head: from the specification of whom is entitled to what (viz. information and basic CRUD actions) we generate a working system. Thus, privacy is not an afterthought; it is the design itself.

What is the main technical challenge? Usually, ACL systematics are implemented as a restriction of an otherwise very permissive system (anything goes for any user). And this latter system almost invariably involves some form of shared persistent memory: central databases, a central server or a central bus providing access to modifiable stores. Not so in Perspectives. People form a network by their interactions; as users of Perspectives, each of them runs an installation of MyContexts (the Perspectives Distributed Runtime (PDR) plus a GUI, plus a private database, usually local). Considered as nodes in a network, a crucial observation is that all nodes are functionally equal (no clients, no servers, no hubs, etc). The main technical challenge for creating the PDR then is to figure out, for each possible modification of the allowed data structures that is initiated on some PDR, what other PDR’s should be informed. And, as a corollary, it is to check each received modification against the model to protect the user against unauthorised modifications.

How will it be used? We hope that MyContexts, the framework program, will attract a community of users and of makers, where the latter create models and screens that contribute functionality. Models + screens ('apps') will be distributed through repositories (webstore-like servers). Models will be downloaded once and stored in the (local) private database, too.

2.3. Architecture: requirements

The software stack should be built from four high level components:

1. a Perspectives Runtime (PR);
2. a persistent store of information (Store);
3. a user interface (UI);
4. an authentication provider (AP).

2.3.1. Perspectives runtime

Please note that in this paragraph we discuss Perspectives runtime in terms of *requirements*, rather than the actual Perspectives Distributed Runtime that is the main topic of this technical documentation. See Actual architecture and underlying technologies for a first introduction of the actual system.

A runtime serves a number of functions:

1. It provides answers to requests sent in by (a single instance of) the User Interface. There is a limited number of requests that is accepted, answered in terms of role instances, context instances and property values.
2. It handles commands sent in by the UI. Again, the number of commands is very limited and allows the end user to add or remove role instances to contexts, to create and remove context instances, and to modify, add or remove property values. It also allows the user to bind some role to another.
3. When we say the PR handles commands we mean that it ensures that subsequent requests reflect the changes ordained before. In practice, this means it stores contexts, roles and property values (or rather causes them to be stored, see Persistent store of information).
4. For each elementary change effected by a command, it constructs one or more Deltas. A Delta is like a remote procedure call. It contains enough information for another PR to reproduce the change. Deltas are collected in Transactions. Transactions are sent to other PR installations.
5. It handles incoming Transactions. After checking whether the sender is authorised to effect the change, each Delta is executed. This allows PR’s to synchronise their stored contexts, roles and property values.

All these are governed by one or more models that authorise user roles to perform some actions (and that includes requesting contexts, roles and property values through the ui). Models are written in the Perspectives Language.

A PR is not expected to be active at all times. Assuming so would simplify some aspects of the architecture.

2.3.2. Persistent store of information

The Store can be thought of as a list of all elementary changes sent in by a single end user, including endorsed changes sent to it by other end users. In practice, it will contain an accurate accumulation of all those changes. It does not allow ‘time travel’, that is, one cannot ask it to revert to a previous state.

It is important to realise that each Store will be updated by exactly one PR, just as a PR will be used by exactly one end user.

We consider the structure of the persistent form of contexts and roles to be internal, even though they are exchanged between PR and Store. This is because the Purescript data definitions are leading. The serialised forms derive from those data structures. The type of serialisation deployed is an implementation choice. Consequently, we only stipulate that these persistent forms are JSON.

Concretely, the store should

1. Provide the ability to create named compartments to store items in;
2. Be able to store and produce on request JSON documents in such a compartment;
3. Be able to provide a list of the documents in a compartment

There is a number of self-evident desired non-functionals, such as reliability, lots of capacity, etc.

Performance is important in the sense that the PR will very frequently request documents and update them. To some extent this requirement is relaxed because the PR caches each document it has requested. This cache is transparent with respect to updates. User Interface

There are hardly requirements for the user interface other than in the most general of terms: it should enable a single end user to

1. Enter requests and inspect the answers sent by the PR;
2. Enter commands;
3. Identify him- or herself to the PR.

A UI will often be a graphical UI presenting screens and forms. As of version v0.4.0 of Perspectives, a JavaScript proxy library is available through which a process can interface with the PR. Also, built on top of that, a ReactJS library of data containers is available that provides higher level abstractions to deal with the PR.

2.3.3. Authentication provider

The end user authenticates him- or herself through the UI to the PR. Currently, we have taken a very simple approach to authentication. The authentication provider is not a separate component. The required functionality is provided by the PR. User credentials are kept in the Store.

2.4. Connections between components

2.4.1. UI – PR

A single end user interacts with a single instance of the PR through a UI. The connection between them should be confidential. Transport of information between them should be fast enough that it does not stand in the way of a smooth user experience (this includes all aspects of transport, such as setting up a connection, applying measures to ensure confidentiality, etc).

The information items passing through the connection are usually quite small in terms of bytes when compared to current network bandwidth. The Deltas consist of alphanumeric information. Each Delta is the result of an end user action. There are no actions that lead to massive numbers of Deltas. Files can be handled as claim data: that is, the PDR is concerned with identities, not the actual items themselves.

The connection should also be reliable: whenever the end user fires up his UI, it should be able to connect to its PR.

Because of the nature of the UI (to enable an end user to access a PR) we assume that both components are active at the same time.

The connection should not only allow the UI to approach the PR; the PR should be able to initiate a contact, too. We need this to alert the end user to changes initiated by other end users.

2.4.2. PR – Store

A single PR interacts with a single Store (conceptually). We have not yet worked out an architecture where an end user deploys multiple devices. The simplest of architectures would be one where the Stores attached (through a PR) to UI's on multiple devices, synchronise between them. This, however, will not provide a limited user experience when the user simultaneously uses multiple devices.

So for the time being we assume the unique association between a PR and a Store. The connection between them should be

- Confidential
- Reliable
- Fast enough to handle the traffic resulting from several humans interacting through their UI (in the order of the number of relations a single person has).

Again, like with the connection between UI and PR, the required bandwidth is quite limited. Because of the nature of the Store (to persist information shed by the PR) we expect it to be active at the same time as the Store.

2.4.3. PR – PR

PR's send Transactions to each other. However, as we do not require that PR is always available as an active process, the connection between them should handle this. Consequently, this connection should have the characteristics of a mailbox. We require the following non-functionals:

1. The connection should be reliable;
2. The connection should be confidential
3. The connection should be restricted to two PRs.
4. The connection should have a push-character: that is, the receiver should be notified after the sender has sent a Transaction.

5. The connection should be reasonably fast, ideally fast enough to allow for a chat-like experience (i.e. time delay introduced by the channel should be low enough to provide a good user experience). This, however, is not a hard requirement.
6. The connection should be able to handle the fact that end users will interact through mobile devices and do move around.

2.5. Actual architecture and underlying technologies

Now let's concentrate on the concrete architecture and the languages / systems involved in building the PDR.

The core of the PDR is written in the Purescript language and compiled to Javascript. Purescript is a strict (non-lazy) variant of Haskell. The user interface consists of HTML screens, (currently, but not necessarily) written in terms of ReactJS (which is more Javascript, HTML and CSS).

The architecture consists of three components:

- the client (an HTML viewer)
- the PDR
- a private database (either Couchdb or the browser's IndexedDB)

See the diagram below.

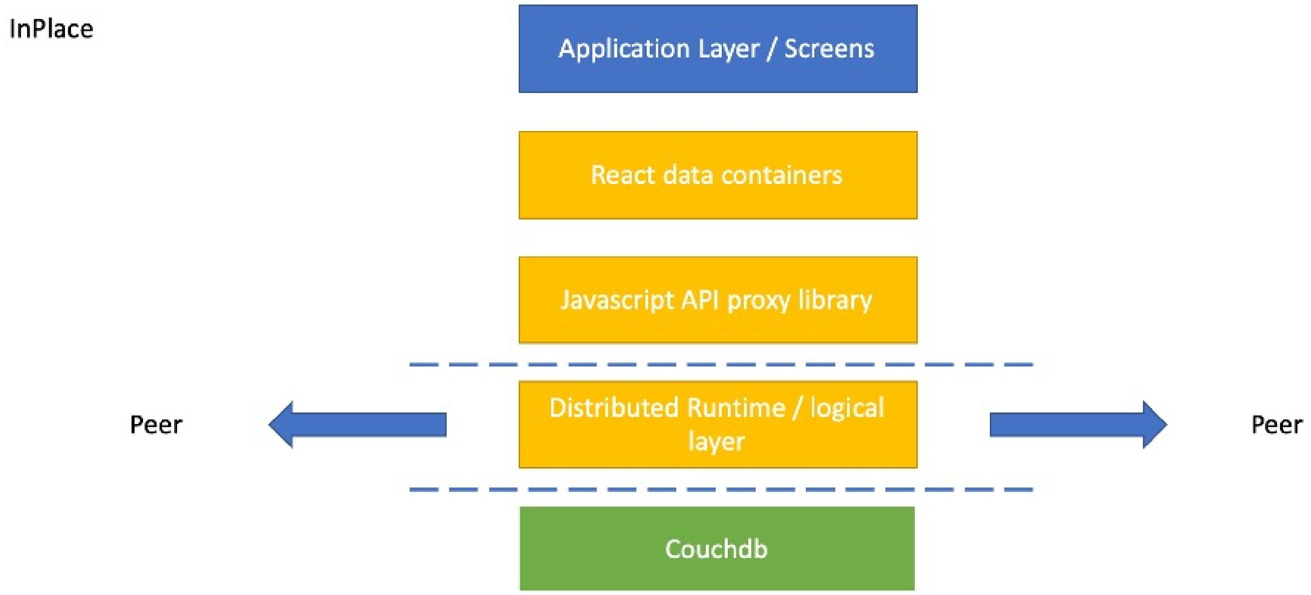

InPlace
Application Layer / Screens
React data containers
Javascript API proxy library
Peer
Distributed Runtime / logical layer
Peer
Couchdb

Figure 1. Application layer, React data containers and Javascript API proxy library form the client. The PDR runs in a SharedWorker. The client connects to the Distributed Runtime via window.postMessage. The PDR connects to Couchdb via HTTPS. Yellow components are 'propriety' (but open source). The Application Layer consists of models in PL and React-based screens. They are 'apps' that are 'hosted' in MyContexts, the end user framework program (the name of the entire ensemble).

All run on the end users' node (laptop, desktop; in the future, tablet & mobile phone too).

The PDR connects to the client through the window.postMessage method in the browser.

Both client and PDR access Couchdb over HTTPS; the latter for data and models, the former for screens (associated with a model as attachments).

> Notice that, if the private database is IndexedDB, the PDR does not connect through HTTPS to it but through API calls. This holds for screens and models, too.

PDR and client both run in the browser: the PDR in a SharedWorker, the client in a window or tab in a window .

PDRs exchange information and do so exclusively through the Advanced Message Queuing Protocol (AMQP). When two PDRs are connected, they have each other’s message queue address, to which both can post Transactions (a package of Deltas on the primary data). To prevent misunderstanding: only Deltas are communicated, never the primary data itself. Neither is the actual local data storage used by the PDR ever modified by any other process than the PDR itself.

To connect, two installations need each other’s message queue address, requiring IP address, port and a user account on the AMQP server. This server is considered to be outside the Perspectives system (e.g. RabbitMQ).

Finally. A node supports more than a single user. Each user has an account that is (currently) password protected. This account maps one to one to a Couchdb admin account for the relevant databases (a data database, a models database and a post database), or to a set of tables in IndexedDB.

The installing user must have a database admin account with the Couchdb installation. Currently, the local account administration used by the login screen is kept in files in an IndexedDB database ('localusers'). It is accessible to processes in the https://mycontexts.com domain.

2.6. Security Concerns

From the above, some obvious security concerns follow. We’ve identified the following:

1. The local user account secrets are accessible to anyone with (enough) access to the device and with enough knowledge to access the IndexedDB database 'localusers' for the https://mycontexts.com domain. In other words, a users' data is as secure as his own device. Once an attacker can log in to MyContexts, he can impersonate the user, and can steal his data.
2. An agent might try to inject Transactions into a message queue, targeting a specific user, impersonating another. This requires the message queue address of that user and the secret key of the impersonated user, to sign the Transaction. Encryption is used for authentication purposes.

   Note that even if an agent would successfully impersonate someone’s peer, they still can only change a persons' data in accordance with the modelled authorisations of that peer (an attacker cannot assume more authorized privileges, since the receiving peer himself compares the privileges he has stored with those claimed by the attacker).
3. An agent with access to the device running the private Couchdb installation might manipulate its data. This would require the users' credentials for Couchdb or it would require him to set up

and admin account on the Couchdb (which is as hard as Couchdb makes it, after the first Admin account has been established).

4. The window.postMessage method might be vulnerable in the sense that processes running in other domains can intercept or even manipulate messages going between the screens and the PDR.

PART I. HIGH LEVEL SYSTEM OVERVIEW

This part of the documentation is organised around a functional decomposition of the Perspectives Distributed Runtime.

Chapter 3. Parser and Compiler

The PDR *executes* models. A model is written in the Perspectives Language and stored as a file with the extension `.arc`. We call those file *source texts*. A source text cannot be interpreted directly by the PDR: we first have to convert it to a machine-readable form (a `.json` file in a particular format). In the next chapter we explain how to interpret a source text; that is, how we as humans can attribute meaning to a source text. In the chapter that follows, we explain how the PDR parses a source text and compiles a machine readable version out of it.

3.1. Understanding a Perspective source text

3.1.1. Types and scopes

With this text we explain how to understand a Perspective model source text. We do not give grammar definitions here, but focus on the meaning of the various expressions instead.

First of all, indentation matters. A model text consists of *blocks* of lines with equal distance from the left margin. Such blocks can be nested arbitrarily deep. We will call such a block a *lexical scope*, or just *scope*.

Second, a model is a number of *type definitions* (at least one: the *domain*). A type definition consists of a *type declaration line*, followed by an indented block that gives the type's details (we might call this block the *body* of the type definition text). A type declaration can be easily recognised because it must be preceded by a keyword:

- a *context* keyword: domain, case, party, activity;
- a *role* keyword: user, thing, context, external;
- or one of the keywords property, view or state.

Any point in a model text (a *lexical* position) will, of necessity, be enclosed in some scope (except for the domain type declaration line). As scopes can be nested, any given point might be inside an arbitrary number of scopes, each enclosing the other until we arrive at the domain's body. Some, but not all, of these scopes will be type definition bodies. Now we will say that all these definitions *are visible* from that lexical position. The first visible context definition that we encounter on 'moving upward', is the *current context*.

3.1.2. Perspectives

A perspective is not a type. A perspective governs what a particular end user, in some user role in a context, can perceive of a role in that context and how he can affect that role. We call the role that a perspective is about, the *object role* while the user role that the perspective is for is the *subject role*. A perspective may be built of three type of parts:

- the *role verbs* that the subject role may apply to the object role, such as Create and Delete;
- the *property verbs* that the subject role may apply to given properties of the object role, such as DeleteProperty and SetPropertyValue;

- the *actions* that the subject role may execute. An action is a sequence of *assignment statements* executed in order, bundled under a name.

> Assignment statements are not explained here. See the text Assignment. However, an assignment statement either changes a context by creating or deleting a role instance, or changes a role by filling it with a role (or remove that filler) or by changing the values of its properties.

Here is an example:

```
domain Parties
  case Party
    thing Wishes
    user Guest
      perspective on Wishes
        all roleverbs
```

User Guest can apply all role verbs to the Wishes role, in the Party context. In order to prepare ourselves for more involved texts whose meaning may not be immediately clear, lets dissect what we have.

We need two concepts: the *current subject* and the *current object*. We've already met the current context, and these two new concepts are just like it. Let's pick the line all roleverbs as our lexical position. We must be able to establish for what user role this line is meant (as part of its perspective). Moving upward through the blocks, we encounter the type declaration line user Guest. Here is a rule: each user role type definition determines the *current subject* for its body. So we now know all roleverbs is meant for Guest.

How about the current object? Well, a perspective on determines the current object for its body. So, putting it all together, for our lexical position all roleverbs we have

- Party as the current context
- Guest as the current subject
- Wishes as the current object.

Now it happens to be that any non-user role definition sets the current object. And we also have perspective of, setting the current subject. So the following model says exactly the same as our previous example:

```
domain Parties
  case Party
    thing Wishes
      perspective of Guest
        all roleverbs
    user Guest
```

Recapitulating: the current object is in scope in:

- the block following a perspective on;
- the body of a non-user role definition.

The current subject is in scope in:

- the block following a perspective of <user>;
- the body of a role definition that is a user role; idem
- the block of assignments following do for <user>, defining an action (we'll see examples below).
- the expression following notify <user>.

In each of these cases, exactly one user role type is specified. Hence the current subject is always a single named type (but notice this may be a calculated role type that defines the disjunction of two types).

3.1.3. Notifications and state

Often, we want to make sure that the users of an application take note of some changes. In our party example, it is important for the person throwing the party to note that his invited guests accept (or reject) the invitation. To make sure that some changes do not go unnoticed, we introduce the mechanism of notification.

A *notification* is something that draws the end users' attention. Usually, a notification also consists of some message (like a line of text). Crucially, a notification *should happen under specific circumstances*. In Perspectives, we model this with *state transitions*.

A context can be in a specific *state*, and so can a role. This state must be defined in terms of its roles (for a context), while the state of a role is defined in terms of its filling role and/or property values. In Perspectives we can write an *expression* with some truth (Boolean) value to define a state.

> We do not explain expressions in this text in detail. In short, an expression can be some comparison whose left and right terms trace paths through the web of contexts, roles and properties.

Here is an example:

```
domain Parties
  case Party
    user Guest
      property FirstName (String)
      property Accept (Boolean)
      state Accepted = Accept
        on entry
          notify Organizer
            “{FirstName} has accepted.”
    user Organizer
```

The state declaration line shows that role state Accepted is simply determined by the value of the Boolean property Accept. As soon as the end user playing the Guest role sets that property to true (e.g. by ticking a box in the invitation screen), the Guest role *enters* the state Accepted, or *transitions* to that state.

But what was the previous state of Guest? By default, each context or role is in its resting, or *Root,* state. So by ticking the box, role Guest transitions from Root state to Accepted state.

Let’s do some lexical analysis. notify Organizer is our lexical position of interest. What is the state transition it applies to? Following the nested scopes upward, we encounter an on entry line. This specifies a transition type (there is only one alternative: on exit). But entry of what state? Continuing our exploration of nested scopes upward, we encounter the state type declaration line for Accepted. Here’s another rule: in the body of a state type definition, that state is the *current state*. This completes our quest: notify Organizer applies to entering state Accepted.

A little more on the nature of notifications. In the example above, we might be tempted to notify the Organizer with the text “I will come” rather than “X has accepted.”. After all, it is the Guest who accepts and this could be his personal message to the Organizer.

But a notification is not a message from one user to another. It is more like an act of observation by the notified user. The difference is subtle: users should be observant and note what happens to contexts they play a role in. The notification mechanism merely aids them in actually noting that some changes have occurred. It is not a *conversation* mechanism; we have other means for that.

> This is not yet so at the time of writing, but will be added in a future release.

3.1.4. More on states transitions; automatic effects

Life is full of repetitious tasks. Automation can take care of them and Perspectives is no exception. But when we make something happen automatically using Perspectives, it will always be *delegated by some user role*. Here we mean by ‘something happens’ that the information recorded in terms of contexts, roles and their properties, changes; in other words, that the state (of some context or role) changes.

So to make this very clear: state changes are always traceable to some (exactly one!) end user.

Let’s consider a birthday party. On entering the Root state of the Party (which happens as we create it), we’ll create a role PartyPig (to be filled later by a person). This is the model:

```
domain Parties
  case MyParties
    user Organizer
      perspective on Parties
        only CreateAndFill
    context Parties filledBy BirthDayParty
      on entry
        do for Organizer
          createRole PartyPig in binding >> context
  case BirthDayParty
    user PartyPig
```

The first thing to note is that all birthday parties (all instances of BirthDayParty) are embedded in the context MyParties. The context role Parties holds them. The user role Organiser can create a new one and fill it automatically with an empty embedded context (an instance of BirthDayParty) as well, due to the role verb CreateAndFill.

> This is the *only* way we can create contexts. Thus, each context is always embedded through a context role in some other context. The role verb CreateAndFill governs this.

But this context is created empty and we always want to have an instance of the PartyPig role in it. This is where the on entry comes in. We then automatically create a role PartyPig, an automated task delegated by the Organizer user role.

You will notice the clause in binding >> context. It traces a path from the new Parties role instance (the current object) to the new embedded BirthDayParty context that it is filled with. This is where we create the new instance of the role PartyPig.

> Notice that the current object as we've defined it above is actually available as the value of the variable origin in expressions.

We might have written this model like this:

```
domain Parties
  case MyParties
    user Organizer
      perspective on Parties
        only CreateAndFill
        on entry of object state
          do
            createRole PartyPig in binding >> context
    context Parties filledBy BirthDayParty
  case BirthDayParty
    user PartyPig
```

There is no difference in meaning; just in the way we express it.

By now you will have inferred that the line do for Organizer sets the current subject to Organiser. But why do we write on entry of object state and not just on entry, like we did in the first formulation?

This has to do with the notion of current state. We've seen that in the body of a state definition, that state is the current state. But what is the state outside of such scopes?

By construction, in the scope of a context definition, the current state is the Root state of that context. Similarly, in the scope of a role definition, the current state is the Root state of that role. Finally, we also have in state expressions that, unsurprisingly, set the current state.

So in our second formulation of the model, the current state in the lexical position at the start of the line on entry of object state is the Root state of the Organizer role. But we obviously do not want the automatic effect to take place on creating an instance of the Organizer role – it should happen when we create an instance of the Parties role! This is what on entry of object state does for us: it sets the current state that the on entry applies to, to the Root state of the current object. And this happens to be Parties (it is the first enclosing scope that sets the current object).

Looking back to the first formulation, we can understand why on entry works. The current state at this lexical position is given by the declaration context Parties – and thus is the Root state of Parties.

There are a lot of ways to set the current state. Summing up:

1. A state definition sets the current state for its body.

> In all rules we list below, 'setting the state' holds for the lexical scope following the declaration or clause lines.

1. A context definition sets the current state to the Root state of that context.
2. A role definition sets the current state to the Root state of that role.
3. The in state X clause sets the current state to its substate X. When of and a state type (object, subject, context) is specified, the current state is set to the substate X of the current object, current subject or current context respectively.
4. The on entry and on exit clauses do by themselves not change the current state, but specify a state transition for the current state.
5. The on entry and on exit clauses can be augmented with three parts:
    a. of object state, optionally extended with a state name. It sets the current state to the Root state (or the named state) of the current object;
    b. of subject state, idem, for the current subject;
    c. of context state, idem, for the current context.

These definitions and clauses give us full control of specifying the conditions under which something may happen automatically, in various ways.

> In case you wonder why perspective on does not set the current state, see the paragraph Why *perspective sets no state*.

Some examples:

```
  on entry
  on entry of object state
  on entry of object state Published
  in state Published
  in object state Published
  in context state Published
```

3.1.5. Perspective and state

A user role might have different perspectives in various states. Let’s revisit our first example:

```
domain Parties
  case Party
    thing Wishes
    user Guest
      perspective on Wishes
        all roleverbs
```

What is the current state in the lexical position perspective on Wishes? Its narrowest enclosing state giving scope is the body of the user Guest definition, so it is the Root state of Guest. The implication is that this perspective on Wishes is always valid.

Why always? Would Guest not lose the perspective on the very first state transition? No, because whatever state Guest would transition to, it *must* be a substate of its Root state. This means that Guest then would be in both the substate and the Root state. In other words, perspectives for the Root state are always valid.

In contrast, in this model:

```
domain Parties
  case Party
    thing Wishes
    user Guest
      property Accept (Boolean)
      state Accepted = Accept
        perspective on Wishes
          all roleverbs
```

Guest would only acquire a perspective on Wishes in state Accepted. That is, the state Accepted of the role Guest.

We might call this *subject state*: the perspective depends on the state of the subject. It is also possible to define a perspective dependent on object state:

```
domain Parties
  case Party
    thing Wishes
      property Finished (Boolean)
      state Published = Finished
    user Guest
      in object state Published
        perspective on Wishes
          all roleverbs
```

Now Guest can only see the Wishes when they are published. The perspective no longer depends on the state of Guest.

As of yet, we cannot make a perspective dependent on both object and subject state.

Obviously, we can also define a perspective to be valid in some context state. That means, in this case, that we can actually make the perspective depend on both object and subject state:

```
domain Parties
  case Party
    thing Wishes
      property Finished (Boolean)
      state Published = Finished
    user Guest (Relational)
      property Accept (Boolean)
      state Accepted = Accept
        state WishesPublished = context >> Wishes >> Finished
          perspective on Wishes
            all roleverbs
```

Subject role state Accepted now has a substate called WishesPublished. Its definition depends on the same property Finished of role Wishes as the Published state of Wishes itself (but we need a path via the context to reach it). So, whenever Wishes transitions to Published, a Guest user role instance in state Accepted will transition to its substate WishesPublished and thus be in both states at the same time. So we succeed in mimicking the effect of making the perspective depend on both object and subject state.

This works, however, only because Wishes is a *functional* role (roles are by default functional: only by adding the qualifier Relational (in parentheses) we can make it have more than one instance). Obviously, Guest is not a functional role and this means we cannot mirror this solution by reaching out from the role Wishes:

```
domain Parties
  case Party
    thing Wishes (Functional)
      property Finished (Boolean)
      state Published = Finished
        state GuestAccepted = context >> Guest >> Accept
          perspective for Guest
            all roleverbs
    user Guest (Relational)
      property Accept (Boolean)
      state Accepted = Accept
```

Look at the declaration of GuestAccepted: exactly what Guest are we talking about? The expression context >> Guest >> Accept will return as many Boolean values as there are Guests. As a matter of fact, the Perspectives compiler will reject this state definition because the expression is not functional (can result in more than one value).

Summing up: only as long as at least one of subject and object are functional, can we mimic the effect of making a perspective depend on both object and subject state.

3.1.6. About expressions and variables

Expressions can occur in six different positions in a Perspectives source text (illustrated in the next paragraph). An expression is like a function, applied to either a role instance or a context instance. Until now we've glossed over the question: to what instance is an expression applied, in execution time? The next paragraph is devoted to answering that question, but first we turn our attention to a number of *standard variables.* These variables take on a single value in runtime.

For convenience, we can use in any expression the standard variable origin. Its runtime value is always the context- or role instance that the expression is applied to. The name 'origin' reflects the path-like character of an expression: you can trace it from context to role and vice versa, through the network of types of your model. Runtime, these are paths laid out through the network of connected instances. Below we will show how you can determine the type of origin from the model source text. It will turn out to be either the *current context*, *current subject*, or *current object*.

Furthermore, we can always include the standard variable currentcontext in an expression. It's type is the *current context* as we've used the concept above, in the lexical analysis of source texts.

In an action and an automatic action delegated by some user role, we can use the standard variable currentactor. It's type is, unsurprisingly, the *current subject* of lexical analysis.

Finally, in a notify construct we can use the standard variable notifieduser that, again, has the *current subject* type.

3.1.6.1. What expressions are applied to

The definition of a calculated role

Given this model:

```
case C
  role R = <expression>
```

<expression> is applied to the *current context*, which is an instance of C. The same holds for the other types of context (domain, party, activity). The value of origin is that same instance, and so is the value of currentcontext. Invariant: currentcontext == origin.

The definition of a calculated property

Given this model:

```
case C
  thing R
    property P = <expression>
```

<expression> is applied to the *current object*, which is an instance of R. The same holds for the other role types (context, external and user). The value of origin is that same instance, and the value of currentcontext is an instance of C. Invariant: currentcontext == origin >> context. NB: a (calculated) property is always embedded directly in the body of an enumerated role.

The condition of a context state

Given this model:

```
case C
  state S = <expression>
```

<expression> is applied to the *current context*, which is an instance of C. The same holds for the other types of context (domain, party, activity). The value of origin is that same instance, and so is the value of currentcontext. Invariant: currentcontext == origin.

The condition of a role state

Given this model:

```
case C
  thing R
    state S = <expression>
```

<expression> is applied to the *current object*, which is an instance of R. The same holds for the other role types (context, external and user). The value of origin is that same instance, and the value of currentcontext is an instance of C. Invariant: currentcontext == origin >> context.

The object of a perspective

Given this model:

```
case C
  user U
    perspective on <expression>
```

<expression> is applied to the *current context*, which is an instance of C. The value of origin is that same instance, and so is the value of currentcontext. Invariant: currentcontext == origin.

Expressions in do

Given this model:

```
  case C
    thing R
      property SomeProperty
    user U
      perspective on R
        on entry of object state
          do
            SomeProperty = <expression>
```

<expression> is applied to an instance of R. This requires some explanation. Why not to an instance of U?

Let's start with the question: what is the current state in the line that holds <expression>? Moving in from the outside: the body of case C has *context state*. But in the body of user U we have *subject state*. The body of on entry of object state changes that to the state of the current object. So what is the current object, at this position? It is determined by perspective on R, hence our <expression> is in the (root) state of role R: it is the *current state* for that expression. As a consequence, <expression> is applied to the thing that can be in that state, hence it is applied to an instance of R.

This turns out to be the way to find out, from analysis of the source text, the type of thing that an expression in the body of a do is applied to. Find the current state: the expression is applied to instances of the type that can be in that state.

Because we have context state, subject state and object state, expressions in the body of do can be applied to contexts, user roles and other roles.

origin consequently can be a context instance or a role instance.

There is another standard variable available for use in expressions in a do: currentactor. It is an instance of the current user for that expression, which, in our example, is U.

Finally, we have currentcontext: it is an instance of C.

Consider this variation on the example model:

```
case C
  thing R
    property SomeProperty
  user U
    perspective on R
      on entry
        do
          SomeProperty = <expression>
```

(we've omitted the of object state after on entry). Consequently, the current state for <expression> is *user state* and so <expression> will be applied to an instance of U. As U has no property SomeProperty, the system will complain about this and not accept your model.

Expressions in do for a remote (calculated) perspective

Given this model:

```
case C1
  thing R1
    property SomeProperty
  context C2S filledBy C2
case C2
  user U
    perspective on extern >> binder C2S >> context >> R1
      on entry of object state
        do
          SomeProperty = <expression>
```

<expression> is applied to instances of C1$R1 (meaning: R1 in C1), so origin is an instance of R1. currentcontext is an instance of C2, currentactor is an instance of U, just as we've seen before. Now take a look at origin >> context. Does it equal currentcontext? No!

This is new. In many of the examples above, we had the invariant currentcontext == origin >> context (and in the other cases, currentcontext == origin). But here, it is not so. This is exactly the meaning of a 'remote perspective': the user has a perspective on a role *outside* the current context. For <expression>, this remote role is the origin.

Having both origin and currentcontext, we can access both contexts in our expression if we need to: the current context from our lexical analysis (the context 'as we see it' surrounding the expression), and the context of the resource that the expression is applied to (the origin).

Expressions in do for a remote (calculated) user

Consider this model:

```
case C1
  user U1
  context C2S filledBy C2
case C2
  thing R
    property SomeProperty
  user U2 = extern >> binder C2S >> context >> U1
    perspective on R
      on entry of object state
        do
          SomeProperty = <expression>
```

Instead of having a remote object role, we now have a remote user role. This is reflected in the types and values of the standard variables: currentactor is an instance of C1$U1. currentactor >> context is not equal to currentcontext; the former is an instance of C1, the latter an instance of C2. origin is an instance of R.

Expressions in action

Given this model:

```
case C
  thing R
    property SomeProperty
  user U
    perspective on R
      action
        SomeProperty = <expression>
```

Is <expression> applied to an instance of U, or an instance of R? As with do, we have to ask ourselves: what is the *current state* for <expression>? Our state rules say that current state in the body of user is *subject state* (perspective on changes the *current object*, but does not change the current state). Hence, <expression> is applied to an instance of U and the system complains that U does not have property SomeProperty.

Now examine a variation on this model:

```
case C
  thing R
    property SomeProperty
  user U
    perspective on R
      in object state
      action
        SomeProperty = <expression>
```

The system accepts this, because we've changed the current state to object state and so

<expression> is applied to the current object, which is an instance of R.

In short: the rules for expressions in do apply to action as well, including the treatment of currentactor.

Expressions in notify

Given this model:

```
  case C
    thing R
      property SomeProperty
      property Name
      state S = exists Name and exists SomeProperty
    user U
      property Nickname
      perspective on R
        on entry of object state S
          notify
            □Hello {notifieduser >> Nickname}. instance {Name} of R now has value
{SomeProperty}
```

The expressions in the notification are applied to an instance of R, so origin holds that instance. notifieduser is an instance of the type of the *current subject*, which is U since we did not specify otherwise (given another user role U1, we might have written notify U1). currentcontext will be an instance of C.

In short: the rules for expressions in do apply to notify as well, but instead of currentactor we have notifieduser.

3.1.6.2. Summing up: what expressions are applied to

The table below summarises what expressions are applied to (and that resource is always available inside the expression as the origin variable). The standard variable currentcontext is also always available in each expression.

Expressions in	Applied to	standard variables other than origin and currentcontext
The definition of a calculated role	Current context	
The definition of a calculated property	Current object	
The condition of a context state	Current context	
The condition of a role state	Current object	
The object of a perspective	Current context	
Expressions in do and action	The resource in current state	currentactor

Expressions in	Applied to	standard variables other than origin and currentcontext
Expressions in notify	The resource in current state	notifieduser

3.1.6.3. Delegate to a functional user role only!

We’ve seen that resources that expressions in action and in do are applied to, are determined by the same rules. Yet there is a difference between automated actions and actions that must be executed by hand. Automated actions are *delegated* by a user role. Suppose that this role could have more than one instance (as specified with the keyword Relational). We then would have a situation in which multiple PDR installations would execute the same action and claim the authorship of the changes to ‘their’ user!

Instead, we stipulate that the user role on whose behalf an action is executed automatically, must be *functional*. This obviously also holds for calculated user roles, but that may be less easy to see. Luckily, the system checks this for us and flags down the model if the calculation could result in more than one user instance.

3.1.6.4. Why perspective sets no state

You may have wondered why perspective on sets the current object, but does not change the current state. Consider this model:

```
domain SomeModel
  case C
      thing R
      property SomeProperty (Boolean)
      user U
        perspective on R
          action MyAction
            SomeProperty = true
```

Its intended meaning seems clear: user U has an action called MyAction in her perspective on R. However, the system will raise an error to the effect that U does not have property SomeProperty. Let’s analyse why: the current state at the lexical position of MyAction is set by the user U declaration, so MyAction is *subject state*. Hence, the action holds for that state and will be applied to the current subject, being an instance of U. We can easily fix that:

```
domain ActionExample
  case C
      thing R
      property SomeProperty (Boolean)
      user U
        perspective on R
          in object state
            action MyAction
              SomeProperty = true
```

Now MyAction is in object state and will be applied to an instance of R. All is well.

This seems a perfect argument to have perspective on change the current state as well. But then we would run into problems with this model:

```
domain Parties
  case Party
    thing Wishes
    user Guest
      perspective on Wishes
        all roleverbs
```

The current state for the line all roleverbs would become object state, meaning that the perspective holds for the root state of Wishes rather than the root state of Guest. That is not very intuitive. As models will usually be a lot more about specifying perspectives than actions, we have decided to construct the language in favour of the former. Hence we must use in object state for an action in the ActionExample model.

3.2. Parsing and compiling models

3.2.1. Introduction

A model is in effect a collection of types. A model also introduces a namespace. Contexts and roles introduce namespaces, too:

- a context gives its name to the roles and actions defined in it;
- a role gives its name to the properties and views defined in it.

A context can also contain other contexts (notice we’re talking types here!) and so provides a namespace to them. This might lead us to think that a model is just another context. And it is. However, a model is a context *without a context*. In other words, it is the root of a namespace hierarchy. We call such a context a *Domain*. So a Domain is a context that contains all other types in the model, recursively.

However, a DomeinFile – derived from a Model’s source text – is not just a context. To start with, it is represented differently. In Perspectives, it plays the role of a *package*. All types (contexts, roles, properties, views, actions) are pulled up to the surface and are available in maps as members of a record. The PDR uses this representation to quickly look up any type.

Furthermore, Instances can be packaged with a DomeinFile, indeed, an instance of model:System$Model *must* be packaged with it. The external role of this context instance has properties that give its readable name and description, and the url where it resides (a repository on the open internet).

Last, but not least, a DomeinFile/Package lists the *other* DomeinFiles/Packages it’s definitions depend on, in the sense that they refer to types defined in them.

To sum up:

- a *Model* is a collection of types;
- a *Domain* is the root context of a Model, having no context itself;
- a *DomeinFile* is a *Package*, the physical transport format of the represented types constituting the model.

3.2.2. Constructing a DomeinFile

Perspectives models are written in the Perspectives Language

> Previously called ARC language, from Action Role Context. This name is frequently used in the Perspectives Source code files.

A model file is parsed and transformed into a DomeinFile in a process consisting of three phases. A DomeinFile is a data structure that lends itself to easy translation into JSON. It contains the type definitions as defined in the model. It also contains an unparsed CRL file holding the instances that belong to the model, such as the model's description.

In this text we give a high level overview of

- the various transformation phases;
- the interdependencies of models;
- how to put a model in operation for a PDR installation;
- the treatment of the model instances, emphasising indexed names.

3.2.3. Transformation phases

3.2.3.1. Parsing

The raw source text of a model is parsed with the functions in the module Perspectives.Parsing.Arc. The output is an Abstract Syntax Tree in terms of the data structures defined in the module Perspectives.Parsing.Arc.AST.

Expression parsing

Query expressions form a substantial subset of the Perspectives Language. They have their own parsers, in Perspectives.Parsing.Arc.Expression. These functions create an AST in terms of the data structures defined in Perspectives.Parsing.Arc.Expression.AST.

Assignments (as in the right hand side of bot rules) are treated by the expression parser, too.

3.2.3.2. Phase Two

The next step – simply called 'Phase Two' – is performed by functions in module Perspectives.Parsing.Arc.PhaseTwo. It is only now that a DomeinFile is created. During this phase, the identifiers of the definitions that have been declared, are qualified with namespaces. Contexts and roles are both namespace-giving (embedded definitions are scoped with the name of their embedder). However, *references* to definitions are left untreated, because a reference may be made in the text before that what it refers to, is fully qualified (the so-called *forward reference problem*). In this phase we also collect the models that the model under treatment depends on (its ‘referredModels’).

During this phase, we also expand all *prefixed names*. Prefixed names are shorthand for qualified names, making life easier for modellers. Each prefix must be defined for some (context) scope. Phase Two expands all these names (A full treatment of this subject is given in the text: Expanding prefixed names). Prefixes as such are lost beyond Phase Two (but the namespaces they refer to are saved in referredModels).

3.2.3.3. Phase Three

It is only in the third phase that references are qualified. These are:

- the object and indirect object of actions
- the binding of role definitions
- references to properties (in views)
- references to views (in actions)
- the type of value that is returned from a computed role.

Furthermore, Phase Three is responsible for several other processes:

- expression compilation
- inverted queries for properties and roles
- qualification of binding to a Calculated role
- rule compilation

Expression compilation

Expressions in the source text have, up till Phase Three, been stored as abstract syntax trees. Now that all identifiers are fully qualified, we can look up, for any identifier used in an expression, what its type is. Only now can we compile an expression into a *description of a function that can be executed runtime*.

Let’s step back. A query expression is used to define a so-called *Calculated Role* or *Calculated Property*. When an end user requests the value of a Calculated Role through the Api, a function is executed on a context instance and a set of (Enumerated) role instances is returned. It is these functions whose descriptions are compiled in Phase Three.

A QueryFunctionDescription is a data type (defined in Perspectives.Query.QueryTypes) that gives details of a function’s domain and range and how it should be executed. This can either be a primitive (implemented as a Purescript function) or a composition of QueryFunctionDescriptions. As queries consist for a large part of traversing from contexts to roles and vice versa, we need to be able to look up the type of each role. Hence this compilation can only be done in Phase Three.

The module responsible for this compilation is Perspectives.Query.DescriptionCompiler.

From the description of a query function, a Purescript function is computed with the functions in

Perspectives.Query.Compiler. This, however, is *just in time compilation*: each function is compiled just before it is applied for the first time during a PDR session (the compiled function is cached, so it is compiled only once during each session). In other words, actual query function compilation is a run-time process and does not happen in compilation time (the subject of this text).

Inverted queries for properties and roles

A big responsibility of the PDR is to make information available to those users that are modelled to have a perspective on it. Concretely, if user A adds a role instance to a context instance (as an example), all peers in that context with a perspective on that role should be informed. To this end we calculate, in Phase Three, *inverted queries* that compute in run time, from a role instance, the peers that should be informed. Whenever a role instance is added (run time), we run the appropriate queries and then add a Delta to a Transaction to be shipped to those peers. In order to prevent misunderstanding:

- inverted queries are *constructed* during Phase Three (compilation time)
- inverted queries are *executed* during run time.

Qualification of binding to a Calculated role

A role can be defined with a restriction on the type of roles that can be bound to it (that can fill it). This restriction is often in terms of another Enumerated Role, but may be in terms of a Calculated Role. But the type of a Calculated Role is nothing but a composition of types of Enumerated Roles (it is an *Abstract Data Type* consisting of Sum and Products). So, really, we need to record the type of the Calculated role as the restriction on the binding of such an Enumerated role.

But the type of a Calculated Role is the *range of its compiled expression*. Hence, only after the expressions of a model are compiled, can we really qualify the binding of an Enumerated Role restricted with a Calculated Role.

This step finds all such roles and sets the restriction on their bindings.

Rule compilation

Rules consist of a condition and assignments (where these assignments may be in the body of a let-expression). In this step we compile the condition (just an expression) and the assignments. As with expressions, an AST representing an assignment is turned into a QueryFunctionDescription.

> Strictly speaking we should separate expressions (that just have a value) from assignments (that change state). However, their treatment is sufficiently similar to justify lumping them together in the same modules.

3.2.4. Interdependencies of models; packages

We’ve mentioned in the introduction that a package (DomeinFile) lists the models that the model it represents depends on. How do we deal with these dependencies?

3.2.4.1. Consequences of interdependencies for modellers

A modeller is a PDR user who creates models. He uses functions of the PDR that ‘ordinary’ end users do not: to parse and compile a model source text to a DomeinFile. With special powers come special responsibilities. In this case that means that a modeller must take care that he has installed, for his PDR, for local use *all models that his model depends on*. During compilation these models are referred. Indeed, when we have IDE support, these models will be consulted to provide type information on referred types.

3.2.4.2. Consequences of interdependencies for end users

An end user cannot be expected to inspect a model’s source text, find the other models that it depends on, and download them from a repository *before* downloading the original model. We have to take care of these dependencies for them. So, on putting a model into operation for some PDR installation, all dependencies must be downloaded automatically first. We will see a little more detail of this in the next chapter.

3.2.5. Putting a model into operation for a PDR installation

To be able to use a model, a user should add the package that contains the models’ types to his PDR installation. That is, he should download the package file from some repository and store it in his local database of packages. The function that takes care of this is addModelToLocalStore from the module Perspectives.Extern.Couchdb. We highlight several facets of this function.

> This module makes functions available in the namespace model:Couchdb, as external core functions that can be executed by callExternal and callEffect.

3.2.5.1. Add model instances

As stated before, a package contains (the text of) a CRL file. This file holds a number of instances, usually of the types defined in the package. It **must** contain an instance of sys:Model, describing the model itself. The embedded CRL file is parsed and the instances are added to the users’ local database.

3.2.5.2. Customize indexed names

A model may both declare and use *indexed names* (see the text *Indexed Names* for more detail). Indexed names, such as sys:Me, must be replaced by personalised identifiers before they can be used. We must distinguish the occurrence of an indexed name in the model source text from that in a model instance.

In the type definitions, indexed names *only* occur in queries (calculated roles and properties, conditions of actions, right hand sides of binding in a let expression, right hand side of an assignment expression). Replacement is done when the query is executed – run time.

However, an indexed name in instances that come with a model, must be replaced before that instance is stored in the users’ local database. As an example, think of a role instance that is filled with sys:Me. Before that instance is stored, its binding should be replaced by whatever identifier is used to identify the user who ‘owns’ the PDR installation.

Moreover, the replacement should be remembered in run time. As the query interpreter executes a step that is, for example, sys:Me, it should look it up and find the unique local replacement for it. How do we set up the lookup table?

This is achieved by the requirement that the *modeller* should bind an instance of each indexed type in the model description (remember that the model description is an instance of sys:Model, a mandatory instance packaged with a model). Here is an example, take from model:System itself:

```
sys:Model usr:PerspectivesSystemModel
  extern $Name = "Perspectives System"
  extern $Description = "The base model of Perspectives."
  extern $Url = "http://127.0.0.1:5984/repository/model%3ASystem"
  $IndexedContext => sys:MySystem
    $Name = "MySystem”
  $IndexedRole -> sys:Me
    $Name = "Me"
```

Notice how sys:MySystem (an instance of the indexed context type sys:PerspectivesSystem) and sys:Me (an instance of the indexed role type sys:PerspectivesSystem$User) are bound in usr:PerspectivesSystemModel.

Run time, the PDR retrieves all instances of sys:Model$IndexedContext and sys:Model$IndexedRole from the local database and constructs a table from the indexed names to the customised identifiers (remember that the indexed names themselves have been replaced by unique identifiers *before* the instances were added to the users’ local database!).

3.2.5.3. Add new indexed names

Indexed contexts and roles introduced by the model will be added to the table of all indexed names that the PDR has compiled on startup.

3.2.5.4. Set `me` and `isMe`

The PDR keeps track of a number of computed properties for the sake of efficiency. One of these is the me member of a context instance representation; related is the isMe Boolean property of a role instance representation. The latter is true if the role represents the current (owning) user; the former is the role in the context instance that has an isMe value of true. (for more details, see the text *Overview of state change mechanisms*).

After parsing the CRL text with instances, the PDR computes the value of me and isMe for each of them.

3.2.5.5. Retrieve dependencies

A model may list other models whose types are used in its definitions. These *dependencies* should be loaded before the models’ instances are processed. Currently, the PDR supposes that model dependencies are available from the same repository as the model itself. We may need to change that in the future. The most likely way to do so seems to be to store the repository location with the model name as a dependency (the repository name must be known to the PDR of the modeller, as it must have a local copy of each of the dependencies).

Chapter 4. Functional Reactive Pattern

The perspectives Distributed Runtime implements the *functional reactive pattern*. To be more specific, it keeps requests issued by clients up to date with changes of the underlying structure of context- and role instances. This text explains how this works.

4.1. The underlying network and queries

We can think of context- and role instances, and sets of property values, as nodes in a graph. Property value set nodes (short: property nodes) are terminal nodes. The others are connected by role binding and by the relation between a context and its roles. In Figure 1 below, we've drawn contexts as rectangles and nodes as circles. Nodes are drawn within the contexts they belong to, or just outside if their context is of no concern. The external node of a context is drawn outside its rectangle and is connected to it with a line.

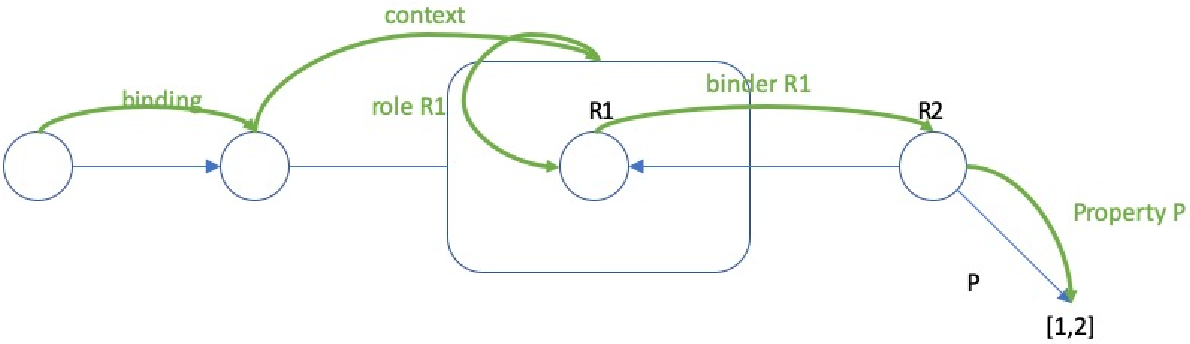


Figure 1. The thin blue lines are part of the structure; the green lines represent query steps.

Figure 1 also shows the jumps a query makes through the graph. In fact, it shows all possible types of jumps available:

- From a role to its binding (binding) and back (binder R1);
- From a role to its context (context) and back (role R1);
- From a role to a set of property values (property P).

NOTE In this document, we use the terms `binder` and `binding`. We found these terms are less clear than the terms `filled` and `filler`. See Binder versus Filler terminology for an explanation.

4.2. Client requests

A client sends requests to the PDR. The core keeps the client up to date with changes to the underlying network by a mechanism of callbacks. In short: the client sends a callback along with its request and the PDR sends (new) results by calling it with those results.

In essence, the API of the PDR allows the client to request any of the five types of query step. Each step must be complete with arguments: for example, the client should say which role instance it wants to see the binding of; similarly, it should additionally say what type of role binders it is interested in (i.e. the type of the role instances that bind a given instance).

It follows that the client always requests nodes that are direct neighbours of another node. This means that the order in which a client requests nodes, depends on the underlying structure, as the client has no means of knowing the identity of a node before it has received that node from the PDR.

> With the exception of *Named Nodes*. These are nodes that have a universal readable name besides their GUID-based identifiers. A modeler may use a Node Name (mostly context instance names) in a query; a screen programmer can start a data container with such a node.

4.2.1. Consequence of request order

Consider the situation in which a client (typically, a React screen) has requested a context, some roles in it and some of their properties. Now that context is deleted, let's say by another user. What requests does the PDR recalculate?

Let's take the React screen as an example. It is built, tree-like, from nested data containers that hold, as terminal leaves, simple display components. A role data container will be nested inside a context data container. Now if the context container disappears (is *unmounted)*, the client is no longer interested in updates for requests that originate from the data containers inside it. This is because it will scrap the entire context data container with all that is inside. As a matter of fact, it is even an error to try to update the data in unmounted components.

How does the PDR prevent updates to updated components? How does it ensure that other requests that are affected by this change, are updated?

4.2.2. How to visit a context

There are but three ways to enter a context, or, in other words, queries can alight in just three ways on the nodes that make up a context in the graph from outside that context.

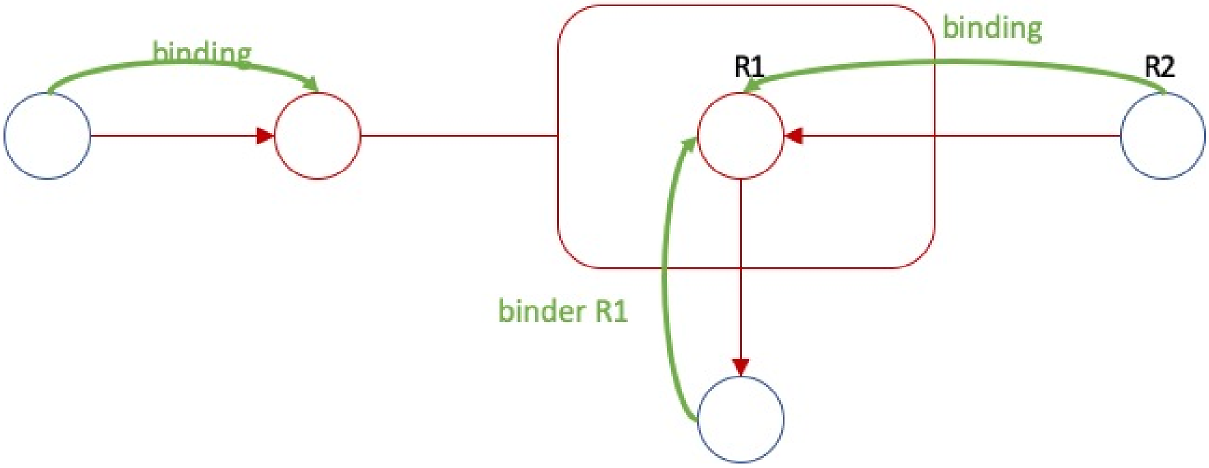


Figure 2. All ways to visit a context. Red parts will disappear when the context is removed.

As the figure shows, first contact with a context is always through a role instance; and this is either through the binding query step or the binder <RoleType> querystep.

4.2.3. Deleting a context

Figure 2 illustrates clearly what client requests should be recomputed if a context disappears. These are requests (green lines) from roles that will remain (in blue) to roles that will disappear (in red).

All other requests involving parts of the context that will disappear, must be ignored. Without exception they are requests that originate in parts that will have gone after the context has been removed; moreover, they will be context, role <RoleType> or property <PropertyType> requests.

By updating the former types of requests and ignoring the latter types, the PDR addresses both questions. All requests that need be updated are, in fact, updated; and no requests are updated that the client is no longer interested in.

4.3. Queries

A screen programmer can only make a client proceed step by step through the network, but a modeler may create a query that performs any number of steps with a single request. Such queries are stored as Calculated Roles or Calculated Properties. Incidentally, these calculated roles can be requested by a client, just like other roles and properties.

If we remove a context, how do we detect (in order to recompute) queries that have passed through that context? In general, how do we detect the queries that should be recomputed after any of the atomic changes that can be made to the network?

4.3.1. Queries trace their steps

The query execution engine records each step of a query. It saves the identification of the client request with the (function) type of each step, permitting the PDR to re-compute the request if the node visited by one of these steps is affected by a change to the underlying network.

Let’s look at the details of saving a query step. A step consists of the identification of a node, and the name of the step. This works out differently for the various steps.

Step type	Node identification	Step name
binding	role identifier	“model:System$Role$Binding”
binder	role identifier	Role Type
context	role identifier	“Model:System$Role$Context”
role	context identifier	Role Type
property	role identifier	Property Type

> Actually, we do not record the context step. See Deleting a role instance for an explanation.

The binding and context steps use a generic step name; for the other steps, we use role or property type names.

Now let’s look at what happens when changes are made to the network.

4.4. Adding a role instance

What queries can proceed once we add a role instance to a context? Figure 3 clearly illustrates there is only one kind of step that is enabled by adding a role instance: role R1. So if query that would have passed through instances of R1, should be recomputed. Because we just add a role (and no binding!) there are no other queries to consider.

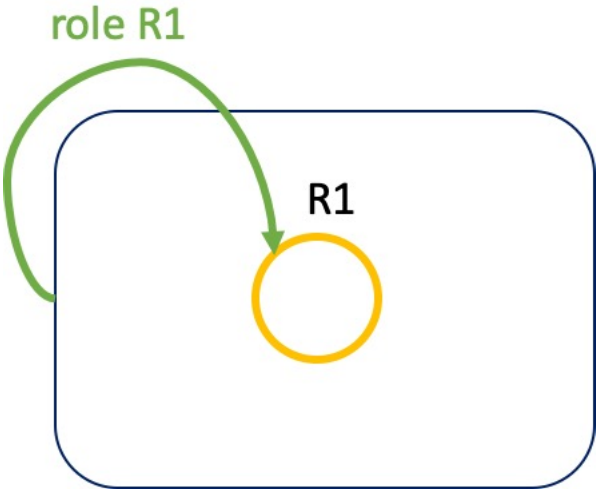


Figure 3. The yellow circle represents a new role instance added to the context instance.

4.5. Deleting a role instance

Much like when we remove a context instance, we have to consider the case of removing a role instance. How does the PDR ensure that requests that are affected by this change, are updated?

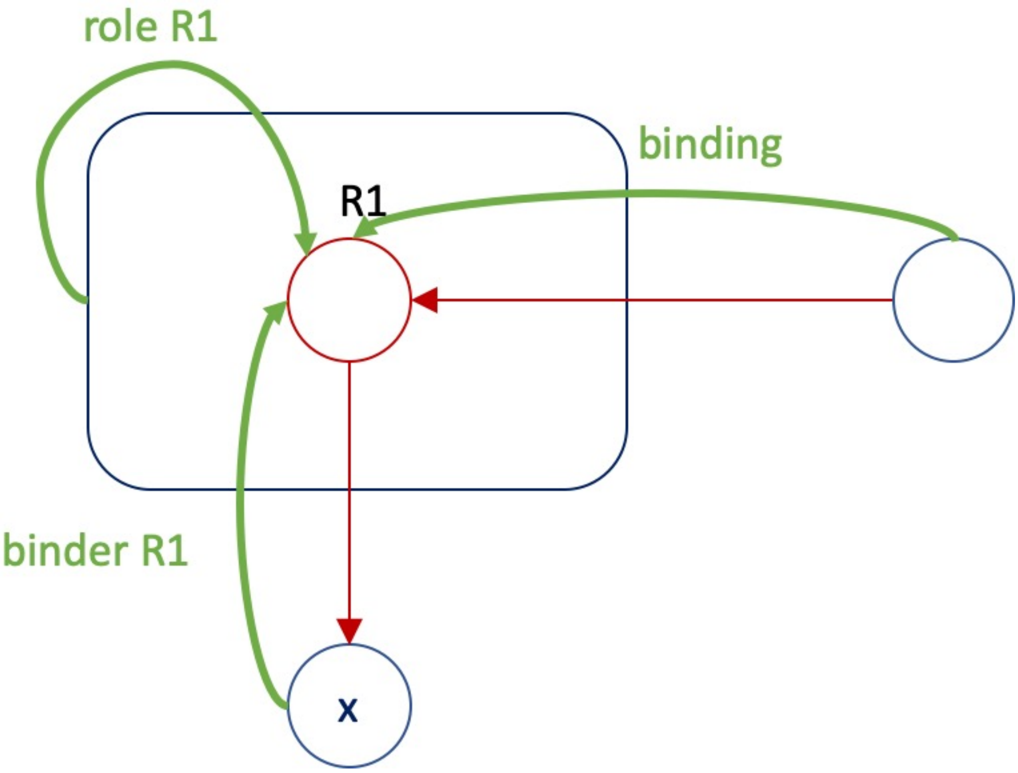


Figure 4. All ways to visit a role. Red parts will disappear when the role instance is removed.

From Figure 4 we learn that there are three ways for a query to alight on a role instance node. When that node is removed, all queries with such steps (the green lines) should be recomputed.

Shouldn’t we recompute queries that have one of the inverses of these steps? Just think how such a query would arrive at R1 in the first place: it can only be by one of steps illustrated by green lines. Hence, if we find all queries with the green steps, we’ve got them all.

4.6. Changing the role binding

What if we just remove the binding of role instance R1? A glance at Figure 5 tells us that all queries with the step binder x <R1> should be recomputed, and the queries that contain the step binding R1. This is because we’ve severed a connection between nodes and those are the two kinds of step that traverse this type of link.

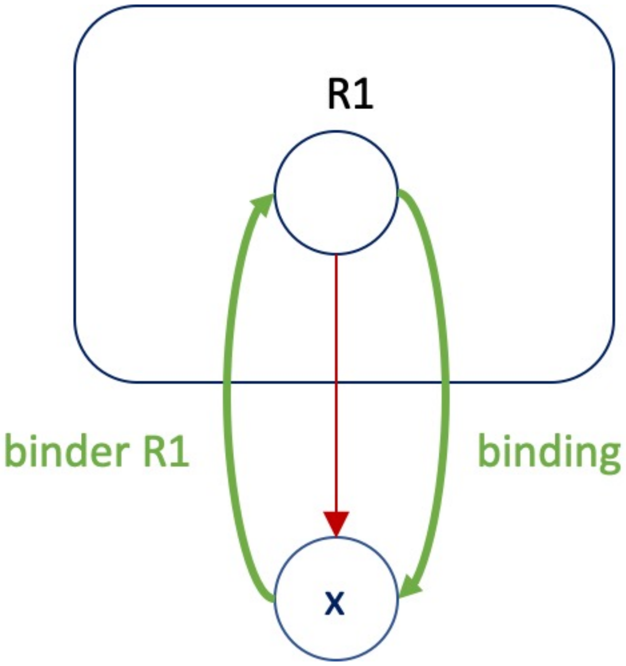


Figure 5. Remove the binding of node R1.

If we add a link, we do the same. Now one could wonder: can there be queries that have recorded this step, as they could not traverse the link before? Yes, there could be, because we record such steps, whether they succeed or not. So a query that tries to move from x to an instance of R1 by attempting the step binder x R1, records that step – even if it fails. Similarly, a step from R1 by following its binding records the step binding R1, even when there is no binding at all.

4.7. Changing property values

Any change to the values of a property P should lead to recomputing all queries with step property P.

4.8. Adding a context

We will consider the case of adding an empty context. To add a context with a role is like first adding an empty context, then add a role.

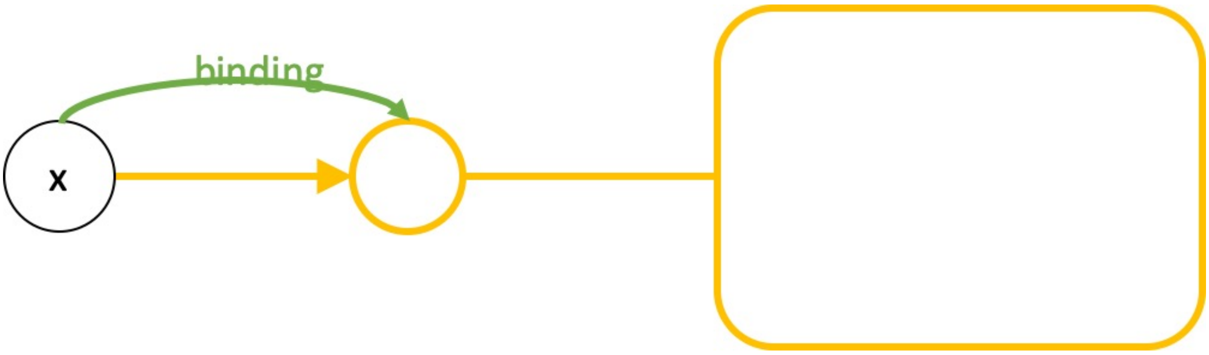


Figure 6. Adding an empty context.

Actually, adding a context is a simple case. It is very much like adding a binding because the only thing we can do – indeed, should do – with a new, detached empty context is to bind some role to its external role. Because we bind to a new role (the external role is created with the context) and because the external role can have no binding itself, we just have to consider queries with the step type binding <ExternalRoleType>.

4.9. The external function

We’ve left the external function out of our considerations. This function moves query evaluation from a context to its external role. Surely, this is a step very similar to the role <RoleType> step? But a moment’s reflection tells us that there is no mutation of the underlying network that would add or remove an external role. Such roles come with contexts and leave with them, too. So: yes, we could trace the step, but it would never figure in computing which queries to recompute.

This is especially relevant for contexts that are not bound. Such contexts are allowed, as long as there is a Calculated Role that retrieves them directly from the database. When we remove such a context (by deleting it ‘from’ the Calculated Role), we make no effort to trace queries that depend on its external role (there can be none that come from outside of the context).

Chapter 5. Assignment and state change

5.1. Overview of state change mechanisms

5.1.1. Introduction

Perspectives has straightforward declarative semantics, explained elsewhere (See Semantics of the Perspectives Language). Program use proceeds by changing that declarative state. In other words: the set of facts kept by Perspective users changes. In this text we give a high-level overview of the techniques involved in dealing with that state change, in order to preserve the declarative semantics in each new state. Where appropriate, we give pointers to other texts that have been written on the subject.

Where do changes come from? Ultimately, they come from the users of Perspectives. Keep in mind that there is a *Universe* of facts collectively tended to by many users. Each user 'sees' facts within his (or her) own horizon. Horizons overlap but are never equal.

When thinking about the origin of changes that arrive in the PDR, we must distinguish between those made by the owning user and those made by his peers. This is because it (the PDR) must send the owning users' changes to his peers – but just absorb and integrate send changes by those peers.

Finally, we must remember users can have bots that act on their behalf.

So when we think about the implementation of the PDR, we have to reckon with three sources of changes:

1. The API that enables a client program to translate the owning users' actions into updates;
2. Actions that are carried out automatically on state transitions, for a particular user;
3. Incoming transactions with changes made by peers of the owning user.

The first two translate into the application of a limited set of *update operations*. The third translates into incoming *Deltas*, where each Delta describes an atomic change.

Where do Deltas come from? They are created by the same update operations.

5.1.2. Five responsibilities

To preserve the declarative semantics, the PDR must recompute requests by client programs sent in through the API. Clients can request neighbouring nodes in the network of context- and role instances, as detailed in *Implementing the Functional Reactive Pattern*. A client should be informed of a change to those parts of the network it has requested (clients are said to have *subscribed* to parts of the network). Now it is important to realise that some of the relations between nodes may be *calculated*. We call a role calculated if it is retrieved by a *query expression* (see *Perspectives Across Context Boundaries*). So it may happen that part of the network traversed by the query engine to compute a role may have changed, even if the nodes that ended up in the query result have not. In such cases, the query must be re-run in order to be able to inform the client about the new declarative state: the *selection* of end nodes is likely to have changed. This is the first responsibility of the PDR.

Secondly, a change might imply that a bot rule must be triggered. Rule conditions are Boolean queries. This is similar to the situation with the client requests. However, rules do not make an explicit request. Models may contain many rules; in theory we need to evaluate them all after each change. The challenge is to create an efficient mechanism that prevents this, yet fires all rules whose conditions have changed.

Third, the PDR must find out where to ship the Deltas that arise from the update functions. Whom should we send a particular Delta? A first approximation is: to anyone who has a perspective on the changed node, as described by the Delta. But, again, as with the client request subscriptions and the rule triggering, queries complicate the matter. A change may happen in some context that user A has no role in, but that is passed through by a query for a calculated role that A has a perspective on.

Fourth, obviously, the PDR should hold on to changes. They should persist from session to session. Persistence relies on Couchdb and on a cache in memory.

Fifth and finally, the PDR keeps a record for each context, telling it what role instance represents the owning user. Many computations depend on that information, so it is prudent to keep a permanent record. This, however, is merely a matter of efficiency; we could do without. The bookkeeping is simple: for a context we record which role instance represents the owning user (dubbed me), and on that role instance we have a Boolean property isMe.

To sum up and give compact names to these five responsibilities, we have:

- request updates
- rule triggering
- synchronisation
- persistence
- current user computation.

5.1.3. Mechanisms

We use various mechanisms to shoulder the five responsibilities.

5.1.3.1. Request updates: dependency tracking

A query is in essence a series of steps through the network of context- an role instances (See Functional Reactive Pattern. and [State and Dependency Tracking] for more detail). A step is one of the five fundamental moves through the underlying network:

- from role instance to its context;
- from a context to the instances of a particular role type;
- from a role to its binding
- from a role to roles that bind it
- from a role to values for a particular property.

Each step is carried out by a simple function. It records its own application. A client request through the API also leads to application of one of these functions. In other words, we can equate each client request, whether it is just a request for the binding of a node or the value of a complex calculated role, with a query. The recorded steps are associated with a particular client request.

> To prevent misunderstandings: the client can also order changes to the network through the API; however, in this context we will not call them ‘requests’.

A Delta (We use Delta and ‘change’ as synonyms in this text) also corresponds to a step. So when we have a Delta, we can look up what client requests are affected by that Delta and recompute them.

5.1.3.2. Rule triggering: inverting queries

Even though a rule condition is just a query (with a Boolean result), we cannot reuse the dependency tracking mechanism. This would require us to evaluate all rule conditions at least once. The computational costs may be considerable, especially when we realise that most context instances are not loaded into memory, for any given session. In order to evaluate each rule, we should, indeed, load everything in memory and that is something we have taken great pains to avoid in the Perspectives implementation.

So another mechanism is necessary and we have found it in running the rule conditions in reverse. This is explained in detail in the text *Query Inversion*. Briefly, it consists of inverting conditions so they run from each node that would be visited, to the context that holds the rule. These inverted queries are stored with role- and property types. When a Delta comes into effect, we look up the inverted queries for the resource in question and run them to find *affected contexts*. Then we run the rules for the owning user in those context instances.

> Inverted queries are stored with the user type(s) that they are relevant for. In the case of a rule, this means the user role that the bot is for. Finding the owning user is just a matter of lookup, anyhow.

This mechanism will run rules even for contexts that do not reside in memory as the Delta takes effect.

5.1.3.3. Synchronisation: inverting queries, too

Inverted queries give us a solution for computing users that should receive a Delta, too. To implement it we do not just invert rule conditions, but, indeed, each query defined in the model. Remember that they define either (Calculated) roles or (Calculated) properties.

> Or the condition of a state, or the value expression in a let body or its bindings.

Requesting a Calculated role is trying to ‘move’ from a context to the instances of that role. Now, because the role is calculated, retrieving the instances will in general involve a number of moves, possibly outside the context of origin. When we invert those steps, where do we end up? In the context of origin, of course.

We invert the calculation for a CalculatedRole in design time (on processing a model). Doing so, we store the user types that have that role in their perspective, with the inversions. So when we later (in run time) run the inverted query and end up with some context instances, we can immediately look up the instances of those user types. They should receive the Delta.

5.1.3.4. Persistence

For persistence we have a number of functions that cache in memory and store in Couchdb. This task is straightforward.

5.1.3.5. Current user computation

The User role of model:System represents the owning (current) user. Consequently, any role instances filled by this role represent the current user, too. This definition can be construed recursively.

> Notice that these are indexed concepts! All users are the owning user with respect to some PDR installation. While doing its work, that user is the ‘current’ user for that PDR.

For now, we hold that a context instance can have only one role instance that is its current user. In other words, a user should take only one role in any context. This may change in the future, as we extend the language.

5.2. Assignment

A user role can have a perspective on roles in- or outside his own context. This brings with it the possibility to change those roles, i.e. to add or remove instances and to add or remove values to or from their properties.

A user agent exclusively changes state through (graphical) user interfaces. Automatic actions on his behalf when state transitions occur, have to be programmed however and to do so we need a vocabulary. This text contains the design for that part of the Perspectives language.

> Also note that the API of the Perspectives Distributed Runtime exposes these functions.

Statements that change the state are called *assignments*. There is a small vocabulary for assignments on roles, including a number of special *operators*: move, remove, delete, bind, bind_, unbind and unbind_ and the keyword combinations create role and create context. There is a smaller set for assignment on property values: =, =+, =-. We also use delete on property values.

The default case is assigning to roles and properties in the context that contain where the state transitions are modelled. We design our language to be intuitive for that default case, extending it for the situation where state of other contexts is changed (see the design text *Perspectives across context boundaries*).

5.2.1. Current context

An automatic action can happen only in a state transition. States are defined with conditions, which are just Boolean queries. The assignments that make up the automatic effect are executed when the condition becomes true. Conceptually, the system observes each context- and role instance. As soon as the state of one of these instances changes in such a way that the condition of one of its defined states evaluates to true, the automatic effect is carried out.

It is therefore important to realise, when reading below about the language for effects, that the effect is computed *relative to some context instance*. All queries that are part of effect statements are executed with that context instance as their starting point. Below, we call it the *current context instance*. The current context can actually be used inside an expression. Reflect on the nature of expressions: they trace a path through the network of contexts and roles. Using the keyword currentcontext, the modeller can re-base a subexpression to the context instance that the rule is executed in.

At the same time, the rule is defined in a perspective. The perspective has an object: a role in the context. When the automatic effect is defined on a role state transition, that role is the object. When it is defined on a context state transition, all instances of the role in that context are the object. These instances are bound to the query variable object. It is available in the state condition and in the statements of the effect. It is useful, for example in the situation where one wants to bind that object to a Role instance.

5.2.2. Assignment statements for roles

Below, we'll find that assignment operators on roles take one or more arguments, that specify what they operate on. For example, in the case of the remove operator its first argument says what we want to have removed automatically. These arguments can be arbitrary expressions – as long as they select roles of the same type as the Object of the Perspective. The system, after all, must operate within the limits set for the user that it works on behalf of!

Implicitly, we give the user that the system carries out an automatic effect for, the role- and property verbs required to do so.

5.2.2.1. Remove

Let's consider the situation where we have some instances of a role that we want to remove from their context. We have selected them with a query, <roleExpression>:

```
remove <roleExpression>
```

Remember, <roleExpression> is a query applied to the current context instance. For example, this could be such an expression:

```
filter SomeRoleType with Completed = true
```

This query will select some instances of SomeRoleType in the current context instance. If we prepend the keyword remove to that expression, all those role instances will be removed from the current context instance.

This works on any context, because role instances are bound to a context. We need not say *from which* context we want to remove them. So in order to remove role instances from an embedded context, we'd write for example:

```
remove filter AContextRole >> binding >> context >> AnotherRole with Completed = true
```

We select a context role, move to the context that is bound in it, move to one of its roles and filter it like before. As before we end up with a number of role instances: these will be removed *from the embedded context*.

What if we want to remove a role that has been added to a Database Query Role? In order to enable the system to decide whether removing the (external) role instance is allowed, it needs to know the Calculated Role type that it should be removed from. So we get:

```
remove <roleExpression> from RoleType
```

5.2.2.2. CreateRole

Create a new role instance in this way:

```
createRole RoleType
```

Here RoleType is not a query, but the identifier of the role we want to create. We can use an unqualified name but it will have to resolve in the type of the current context. The new role instance is attached automatically to that context. In that sense, createRole is an assignment statement, too.

We can also create a role instance in another context:

```
createRole RoleType in <contextExpression>
```

<contextExpression> selects one or more context instances. RoleType now has to resolve in the type of those instances.

5.2.2.3. Move

Usually, when we add a role instance, we create it at the same moment. However, it is possible to move role instances from one context instance to another. Then we use the move operator:

```
move <roleExpression>
```

This removes the selected instances from their origin context and adds them *to the current context*.

> Note that if <roleExpression> selects instances in the current context, this statement does not change state!

To move to another context, we use an extra clause: move <roleExpression> to <contextExpression>

Again, in order to make this useful, <roleExpression> should select from another context than that identified with <contextExpression>. Notice that <contextExpression> can only select a single context.

5.2.2.4. Delete

Sometimes we just want to remove all instances of a role. Then we use delete:

```
delete role <roleType> [from <contextExpression>]
```

Select the instances to be removed. Optionally, to select from another context, add the from <contextExpression> clause. Be careful with deleting the instances of a Database Query Role: all instances that are not bound to some other role will be removed from the users Bubble!

To remove all values of a property, use a slightly different syntax:

```
delete property PropertyType [from <roleExpression>]
```

Notice that we do not provide a query to select instances. We just want to remove all values of the property. By default, we remove them from the *current object set* (see *Object of the Perspective* below). To remove from another role, add a clause:

```
delete property PropertyType from <roleExpression>
```

Obviously, PropertyType has to resolve in the type of roles selected by <roleExpression>.

5.2.2.5. Bind

To fill a role with another role, use bind (when A fills B, we say that B *is bound to* A. A is the *binding*; B is the *binder*). Bind like this:

```
bind <binding> to RoleType

createRole RoleType filledBy <binding>
```

Here, <binding> selects instances of a role (the bindings) whose type must be equal to, or more specialized then, the possible bindings of RoleType. *A new instance of* RoleType *will be constructed automatically* (the binder) and attached to the current context.

To bind in another context, add a clause:

```
bind <binding> to RoleType in <contextExpression>
```

Again, RoleType should resolve in the type of the instances selected by <contextExpression>.

If RoleType happens to be functional, <binding> must evaluate to a single role instance as well. Notice that <contextExpression> may evaluate to multiple contexts; we just bind in all those contexts.

5.2.2.6. Bind_

The variant bind_ can be used to bind an instance of a role in a previously existing instance of RoleType:

```
bind_ <binding> to <binder>
```

The first expression (<binding>) selects a role instance that is going to be bound to the role instance selected by the second expression (<binder>). Notice the singular: this operation only works on singletons.

> If we allowed more bindings and binders, it would be unclear what should be bound to what.

Obviously, the binder must be legally able to attach to the binding. That is, the possible bindings of the binder must be equal to or more general than the type of the binding.

To bind in another context, just select binders in another context.

5.2.2.7. Unbind

The inverse of bind is unbind. Notice that unbind does not remove anything. Both the binder and the binding remain attached to their contexts.

```
unbind <roleExpression>
```

Here, <roleExpression> selects role instances as before. But do we consider them as binders, or bindings? Both are possible. By convention, we choose them to be bindings (fillers) and thus we release them from the roles that bind them (filled by them).

Notice the plural. A role can be bound many times, in many different other roles. By just using an unqualified unbind, we break all bonds that this instance has. Usually, we want to be more selective and this we achieve with another clause:

```
unbind <binding> from RoleType
```

Now, we just release the instance from a particular type of binder. Still, this is across all instances of the context with that type of binder. We can't be more selective with unbind, but we can with unbind_.

On removing the last binder of an external role, the context it belongs to may be removed, too! This process can cascade recursively to nested contexts.

5.2.2.8. Unbind_

Remember that bind_ allowed us to select roles that become a binding and a binder respectively. Similarly, with unbind_ we select a role that *is* a binder and a role that *is* a binding and break them apart:

```
unbind_ <binding> from <binder>
```

As with bind_, this only works with singletons.

There is another use case for unbind_. It is possible to bind a role instance to *more than one* other role instance, of different types. This enables us to create role instances as combinations of property packages, as it were: think of a role at the pharmacy that you'll fill both as patient and as bank account holder. Unbind_ allows us to pick those multiple bindings apart. We can just remove, say, the bank account role from the pharmacy client role.

As with unbind, on removing the last binder of an external role, the context it belongs to will be removed, too! This process will cascade recursively to nested contexts.

5.2.2.9. 'missing' statement types: add and set

One might expect an operator add, to add role instances to a context. However, just where would these instances come from? We don't need add for creating instances, because createRole 'adds' the created instance to the context anyway. The only possible source for the right kind of instances would be from another context than the current. However, for this we have the move operator. Notice also that a role instance can only be attached to *one* context instance. So to move, we have to detach and re-attach somewhere else, preferably in a single transaction. This is precisely what move accomplishes. By omitting the add operator, we protect the modeller from mistakes without compromising what he can express.

A set operator would replace the current instances of a role with a new set. There are use cases for this operator, but these can always be programmed by a combination of delete and move or delete and createRole.

5.2.3. Creating contexts

One of the design goals for Perspectives is that all context- and role instances must be *reachable*. This can be attained by direct indexing (e.g. a role is directly linked to its context), by deploying an indexed name, or by a database query that retrieves instances of a particular type. Such a query has to be the expression by which we define a Calculated Role. To differentiate such database-query-based Calculated Roles from those that are defined by a path query, we call them Database Query Roles.

> An indexed name has a different extension (reference value) for each end user, e.g. My System.
>
> They must be external roles, possibly filtered. Database Query Roles must be context roles.

The assignment statements for roles preserve this quality. In order to do the same for freshly constructed contexts, we have to bind them directly to a context role in another context, or we must ensure that they are available through a Calculated Role somewhere that performs a database query.

Notice there is no operator to remove a context. Contexts are deleted if they are no longer bound, or, in the case of a context that was never bound but added directly to some Database Query Role, as soon as they are removed from such a role (a role based on the same type).

5.2.3.1. Create context

With create context, we create a context of the given type and bind it to a new instance of the given Enumerated Role type in the current context:

```
create context ContextType bound to RoleType
```

In order to bind it in another context, we add a clause:

```
create context ContextType bound to RoleType in <contextExpression>
```

It goes without saying that actually the external role of the fresh context is bound to the new role instance.

If RoleType is a Calculated Role that qualifies as a Database Query Role, no role instance is created to bind the new context. However, we require ContextType to be equal to or a specialisation of the result type of the Database Query Role.

5.2.3.2. Create context_

Like with bind_, we may be in the situation that we already have a role instance that acts as binder. For that case, create context_ creates a context instance of the given type and binds it to the role instances selected with <roleExpression>.

```
create context_ ContextType bound to <roleExpression>
```

5.2.4. Assignment statements for properties

5.2.4.1. The object of the perspective

For role assignment, we discussed the importance of the current context. For property assignment, a similar importance is attached to the *object of the perspective.* Remember that automatic actions are run for a user role having a perspective. A perspective has an object. The object is selected as a is query applied to the current context (it follows that there may not be an object and that there may be multiple objects).

Again, when the automatic effect is executed, there is a *current object set* (possibly empty). A the assignments are executed on each element of that object set in turn, binding an instance to the query variable object. It is available in the condition of the role state and in the effect making up the automatic action.

When we change the values of a property, we really change a role's properties. If not stated otherwise, we change the properties of the current object set.

5.2.4.2. Operators

For assigning values to properties, we use a number of infix operators: =, =+, =-. We also re-use the delete operator we've seen for roles, but with an extra keyword property. However, property values are not moved or created, neither bound nor unbound.

5.2.4.3. =, =+, =-

The syntax for these three operators is the same. For example:

```
PropertyType =+ 10
```

would add the value 10 to the existing set of values for PropertyType for each element in the current object set.

In order to change the property values of another role, we provide an extra clause:

```
PropertyType =+ 10 for <roleExpression>
```

Here, <roleExpression> is a query executed on the current context. Of course it can select roles outside the current context, too.

An expression can be used on the right of the operator:

```
PropertyType =+ SomeRole >> AnotherProperty
```

The meaning of this expression is: add the value(s) of AnotherProperty, for each of the role instances of SomeRole, to those of PropertyType (of the same instance). The query expression is evaluated relative to the current context.

5.2.4.4. Delete

This is how to delete all values for a property on the instances in the current object set:

```
delete property PropertyType
```

And here is to delete the values on another role instance:

```
delete property PropertyType from <roleExpression>
```

5.2.5. Why there is different syntax for properties and roles

Superficially, assignment does not look that different for roles and properties. So why not adhere to the same syntax for both? There are three reasons:

1. There are assignment operations that work on roles but not on properties: bind, bind_, unbind, unbind_ and move.
2. There are assignment operations that work on properties but not on roles: = (set) and =+ (add).
3. The remove operator is quite different for roles than for properties.

To remove a role, we have sufficient information with:

```
remove <roleExpression>
```

The expression identifies the role instances that we want to remove. They are represented internally by identifiable data structures that we can find and destroy. Moreover, we can look up any references to them, so we can clean those up, too.

In contrast, to remove a property value, we not only have to find the role instance that bears the values to be removed, but we also need the name of that property (and, of course, the values to be removed). So we must write down both a <roleExpression> **and** the name of the property (an EnumeratedPropertyType):

```
EnumeratedPropertyType =- <valueExpression> from <roleExpression>
```

(we can omit the <roleExpression> from our expression if we want to operate on the current object set, but to effectually remove the values we do need role instances!).

These differences are great enough to justify different syntax for assignment to roles and assignment to properties.

Chapter 6. Synchronization

6.1. Perspectives across context boundaries

6.1.1. The importance of context

Context introduces privacy in the sense of providing a comprehensive view of participants. As a first approximation, we can provide this with the requirement that a user role can just have perspectives on other roles in his context. However, this requirement is too strict. Context roles and external roles of contexts broaden the horizon of user roles in a limited way, but it is not enough.

By incorporating calculated roles in the language, we provide access to a wider environment around the users' context. All contexts that can be reached through some role path can be consulted. This raises the issue of privacy, however. At the very least, a user participating in some context should be informed about who else can consult (parts of) that context.

6.1.2. Perspectives across borders

Let's examine Party, as modelled below. Guest has a perspective on a calculated role:

```
case: Party
  user: Guest
    perspective on: Giver
  user: Giver = WishInParty >> binding >> context >> Giver
  context: WishInParty filledBy: Wish
  case: Wish
    user: Giver
```

We could write this more compact by losing the calculated Giver role in Party:

```
case: Party
  user: Guest
    perspective on: WishInParty >> binding >> context >> Giver
  context: WishInParty filledBy: Wish
  case: Wish
    user: Giver
```

Notice how we moved the calculation of Giver to the perspective declaration. In a diagram, we just draw a line across context boundaries. There is, however, a condition on such lines: there must be a valid path from the source context (Party) to the destination role (Wish). In our example the path exists because we have bound Wish in the context role WishInParty.

Such a line would be a command to the system to compute the path. We assume the system would notify the modeller if no path exists. In future versions of the system we could allow the textual modeller to command the system to find a path, too, for example with this notation:

```
case: Party
  user: Guest
    perspective on: ... Giver
  context: WishInParty filledBy: Wish
  case: Wish
    user: Giver
```

6.1.3. Computed user roles

There is another way to interpret this example. We might say that there is, in the case type Wish, a *calculated user role*. Let’s call it GuestInWish:

```
case: Party
  user: Guest
  context: WishInParty filledBy: Wish
  case: Wish
    user: Giver filledBy: Guest
    user: GuestInWish = External >> binder WishInParty >> context >> Guest
      perspective on: Giver
```

The computation of GuestInWish proceeds in four steps: from Wish we move to its external role, then to its binding role instance of type WishInParty, then to its context and finally to the instances of the Guest role.

Notice that we’ve given this calculated user role a perspective on Giver.

Transparency is restored, with this model. At a glance we see who can consult the roles of Wish. At a glance, because we only have to examine the definition of Wish itself. We don’t have to scan other context definitions for calculations that provide access to Wish’s roles.

6.1.4. Equivalence

The models given above are completely equivalent. There is no difference, in effect, between a perspective on a calculated role, or a perspective on an enumerated role by a calculated user role. The system should be able to transform one into the other.

A difference might arise when we interpret the perspectives to screens. On first sight, one might think that in the original model, the Guest would consult Wish in the screen that is created for him for Party. In contrast, with the second model, he would be able to open a screen for Wish itself. Whether such decisions on screens should be decided by a choosing one of the two otherwise equivalent models, is an open question. Modelling process logic might dictate a course of action that runs contrary to user experience design.

6.1.4.1. Transparency restored

One way or another, the equivalence between both ways of modelling allows us to create a system that can, in principle, show a participant in a context who else can consult that context, however it is modelled.

We can imagine a query function that computes all user roles in a context – including the calculated users (This function is necessary for synchronization, too).

6.1.5. Assignment

Up till now we've written in terms of *consulting* perspectives on roles outside the context. Do the conclusions extend to *changing* those roles (and their property values)? We think so.

A primary example is the common case where a user role *becomes* another role. In the Party example, Guest can fill Giver. That is, he can bind the role instance that represents himself in Party, to a new instance of Giver in Wish. Guest becomes Giver.

There is – in this case – a restriction on this perspective, however, and that is that a Guest can only bind *himself* in Giver. Otherwise, we would have the situation where any Guest can make another Guest give a present! But, again, this is particular to the *become* Verb.

Becoming is almost the reason for perspectives on calculated roles (or for calculated users): without them, a model like Party-Wish would become very clumsy, as Guest would need a role in Wish before being able to fill the Wish role. An infinite regression threatens here, only to be broken by *another* user role. Becoming is an elegant solution for this problem.

We can extend these powers of change across context borders to other perspectives. For example in the bot rule below:

```
External >> binder Role1 >> context >> Role2 >> Prop1 = true
```

Here, the bot reaches outside its context to set a property on a role on its enclosing context. The left hand side of the assignment is a path; the property value on the end of that path is set to true.

In general, any perspective with Change powers on a calculated role allows the user having that perspective to modify that calculated role. These are great powers, indeed! With them come responsibilities for the modeller.

6.2. Query Inversion

In the text Perspectives across context boundaries we've described how computed roles and properties can extend across the border of a context. Such *queries* reach out of context and bring roles and values in the perspective of a user agent. As a user agent can have actions executed automatically on a state change, we need a mechanism to act on such state changes that would cause new query results. Automatic actions depend on state conditions. In short: a state change outside a context may trigger an action defined for a local user role.

The question is: how do we implement this mechanism? The situation is further complicated because at any moment, just a fraction of the contexts in which automatic actions may be triggered are in the computer memory. By far the greatest number will reside only on disk. In the context of this text we'll call this the *sleeping context problem*

6.2.1. Inverting queries

Imagine a query as a piping system, fanning out from some context with a user for whom actions are carried ot automatically. Call it an *automatic user context*. Role instances and property values ‘flow’ from *source contexts* through the pipes to the automatic user context and the combined and filtered result ends up as a Boolean value. It is as if the automatic user context ‘pulls’ items towards it.

> In general, queries are computed roles or computed properties. The condition of a state is a computed property with a Boolean value.]

Now switch your point of view and ‘look through’ the same pipes from the other end. Looking from some source context, we see automatic user contexts at the other end. In other words, invert the queries. As contexts and roles form a graph constructed from bindings that fans out and fans in, the inverse queries again travel over a subnetworks that are trees. But this time these queries will pull in automatic user contexts.

This solves the sleeping context problem. A change to a context or role or property can only be realised when that context is in memory. If the Perspectives distributed runtime (PDR) receives a delta (the unit of description of change) from some other PDR, it will first retrieve the affected context or role into the computer memory. Now, if it is a source with respect to the condition (query) of a state of some other context, it will have inverted queries. The PDR runs these queries and thereby retrieves all automatic user contexts that depend on it, fetching them from disk if necessary. In the process, it fetches all roles and contexts on the path between them. It then runs the rules in those contexts. The conditions of those rules will pull in further role instances and values. The changed or new item will be one of them (or it will be missing if it was removed), possibly leading to a different Boolean result from before the change.

6.2.2. A mechanism for inverting a query

Queries are stored in a model file in the form of a *description* (See module `Perspectives.Query.QueryTypes`). Such a description holds the domain and range of the query function, and a description of the actual computation. It turns out that we can turn this description inside out, as it were, ending up with a description of an inverted query.

Consider the straightforward case of a query that is just a composition of simple steps – a path. The inverse query is just the inversion of each individual step, run in inverse order (starting with the last step first). But can we invert each simple step? Yes we can (Except for roles computed by *external* functions that have no inversion). Remember that with a query we traverse the network that consists of contexts and roles, connected through role binding. The simple steps are:

- Move from a role to its context;
- Move from a context to some role;
- Move from a role to its binding;
- Move from a role to its binder.

Each role has a single context and a single binding (this is not so in the generalised version of Perspectives where a role can have multiple bindings). However, contexts can have many roles and roles can be bound by many other roles. So the inverse function of role-instance-to-context-instance must be a function that is informed with the *type* of the role instance. The same holds for the inverse function of role-instance-to-binding.

But this information is available in the query function description, so we can draw up a description of the inversion of each simple step.

A word on cardinality. If the original query condition moves from a context to the instances of a particular role, the path ‘fans out’ over multiple instances. However, the inverse path will come from a single instance. In contrast, if the original query moves from a role instance to a context, the inverse query will fan out. This is not a problem, because queries have sequences of values as result. So queries are particular functions, with multiple results.

Besides simple composition of steps, we have (very few) functions that combine simple paths through the context-role network, filter being the most prominent example. In a filter operation, the results of one path are filtered by the outcome of the result of another path.

6.2.3. Where we store inverted queries and how we use them

6.2.3.1. Where we store

Queries are inverted in design time, as we process a model. As we follow an inverted query back to its origin through the web of types, *we store the remaining query with the node we’ve arrived at*, along the way.

We can pass through a role instance node in a number of ways:

- From its context, using the role <type> step;
- To its context, using the context step;
- From its binding, using the binder <type> step;
- From a role that binds it, using the binding step.

We only store inverted queries with Role types. We let the type of the first step of the inverted query determine the member of the type where we store it: so if the first step is binding, we’ll store the inverted query in onRoleDelta_binding.

The step from a context to a role must be the odd man out (we don’t store inverted queries in context types). Here, we store the inverted query with the Role type *that the first step takes us to* (the role <type> step takes us from a context instance to a role instance: we’ll store the inverted query in the onContextDelta_role of the type of that role instance).

The table below gives the overview for all four steps.

step type of inverted query	query stored in	Query stored in node
binder R	onRoleDelta_binder	of departure
binding	onRoleDelta_binding	of departure

step type of inverted query	query stored in	Query stored in node
context	onContextDelta_context	of departure
role R	onContextDelta_role	of arrival

Figure 1 illustrates all four cases.

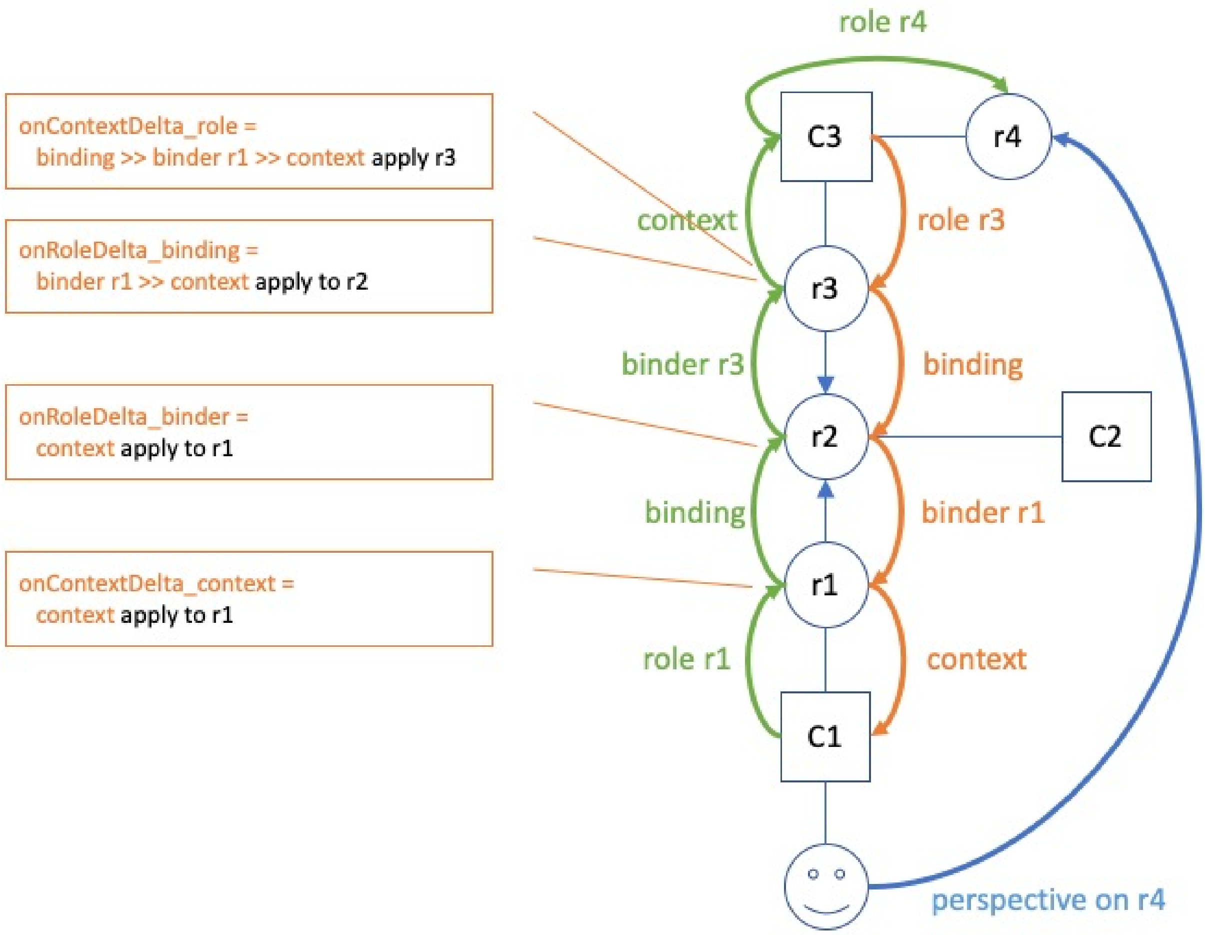


Figure 1. Inverted queries in relation to various nodes in the graph. Green lines and text represent the original query; red lines and text represent the inverted query. The user has a perspective on role (type) r4. Blue lines connect a binder to its binding. The boxes show inverted queries as stored in various members of role types.

6.2.3.2. What we store and what we apply it to

Consider the example of the inverted query stored in on_contextDelta_role of r3 in Figure 1. The query step that we would apply to c3 would have been: role r3. So we would expect on_contextDelta_role to hold the full query

Role r3 >> binding >> binder r1 >> context

That will take us from c3 to c1, as intended. Yet, as the diagram shows, we skip the first query step (storing just binding >> binder r1 >> context) and apply it to r3 (instead of c3). Why?

We will apply the inverted query when we handle a ContextDelta. Let’s assume the delta represents a new instance of r3. Now the *whole point* of applying the inverse query is to find contexts and roles that are now available to the user having the perspective, but were not so before. In other words: a new path has been formed and we want to travel that to its root. Obviously, the new connection must be part of the path we travel. But then we should start at the new instance of r3! Otherwise, on starting with c3, we *would also travel down all paths that begin with siblings of the new instance*.

Hence we shorten the query and start at the new role instance.

A similar consideration holds for the inverted query stored in onRoleDelta_binder stored in r2. Instead of applying the full inverted query to an instance of r2, we apply the shorter version to the new binder of type r1. This is because there can be many binders of r2!

This shortening-and-skipping does not hold for queries in onRoleDelta_binding and onContextDelta_context, because these steps are functional: there is always just one binding and just one context. No confusion can arise. So you see it is the cardinality of the step that determines how we handle it.

6.2.4. Implementation complication: two types of trees

Module Perspectives.Query.Inversion holds the code that actually inverts a query function description. This code deals with a complication. In this document, we've imagined query execution to trace a path through the graph of instances of contexts and roles, fanning out from a single automatic user context to many source contexts. Why the branching? Because of two reasons:

1. a context may have many instances of a role type;
2. a role may be bound by many other roles.

In other words, the path traced by executing a query stands out as a tree selected from the underlying graph of context- and role instances.

However, the way back from a source context (a path endpoint) is always a straight path without branches to the automatic user context (the path starting point).

Confusingly, the *description of a query itself can have a tree-shape*. This is a tree selected from the graph of *types* of contexts and roles. Why the branching? Because we have several operators on two arguments, for example:

1. filter
2. join

(Composition is an operator on two arguments, too, but we use it to construct a single path through the graph of types).

Being a tree, its inverse is, again, a collection of paths. This time, however, these are paths through the underlying graph of types of contexts and roles.

6.2.5. Some cases

6.2.5.1. Variables

letE and letA expressions introduce variables. Furthermore, in calculated properties the variable object is automatically bound to the current object set and in calculated roles we have the variable currentcontext. How should we treat an expression using, for example, this object variable? Consider:

perspective on: SomeRole

on entry

bind object >> binding to AnotherRole

If we invert the sub-expression between bind and to, we should get

binder SomeRole >> context

in order to arrive at the context of this rule from the role (whatever it is) that is being bound by it. Explanation:

1. the binding step inverts to binder SomeRole. SomeRole, because that is the type of the object of the perspective (it is the type of the step object).
2. the object step itself inverts to context, because *underlying the object variable* is the expression SomeRole, evaluated in the current context. That is how we arrive at the value of object (the inverse of SomeRole is context).

This gives us a recipe for the general case in which a variable is bound to an arbitrary expression. Substitute the inverted expression that defines the variable into the syntactical location occupied by the variable.

So while we invert queries, we add bindings to the compile time environment. Because the same variable name can be re-used arbitrarily often, we push a compile time frame before each block.

> In the perspectives language, we can use LetE and LetA. This translates to a QueryFunctionDescription with function name WithFrame. The query inversion code pushes a frame as it encounters this instruction. The variable bindings that follow, lead to additional bindings in this frame. Finally the expression (or statements) in the body of the LetE or LetA are inverted in this environment.

Can we look up the variables, in compile time?

In compile time, we store with the name of a variable a description of a function that will compute its value (an instance of QueryFunctionDescription): a *compile time variable binding*. A variable has a limited *visibility*; we will call the area of Perspectives Language code where we can refer to the variable, its *scope*. There are two scopes we have to consider:

- the condition of a state. It is the scope of the object variable.
- the letE or letA expression. Each binding (from left to right or top to bottom) introduces a new

scope: for the rest of the expression (i.e. the rest of the bindings and the body).

Scopes may be nested. We keep, in the state of the compiler, a stack of Environments to reflect that recursive structure. An Environment is a collection of compile time variable bindings. We introduce, in our Purescript code, a new Environment with the function withFrame. The argument to withFrame is a computation with state in which we save variables and their (compile time) binding.

This makes it as if we can read the Purescript code as a lexical Perspectives Language scope: the computation (Purescript) corresponds to a particular scope (PL).

It so happens that we invert all queries that can hold variables exactly in the withFrame computations that hold their definition, meaning we have all variables in scope: we can actually look them up and find their QueryFunctionDescription.

6.2.5.2. Treatment of properties

Consider a somewhat degenerated Calculated property:

property P1 = P2

We should invert this expression, for two reasons:

- if P2 changes, every user with a perspective on P1 should be informed (synchronisation);
- if P2 changes, P1 changes and it might be (part of) the condition of a rule somewhere.

So how do we go about it? The update function that actually changes the value of property P2 on a role, obviously has access to that role. We do not need to trace a path back from the property value to the role; property values are represented on role instances. In other words, to move from a Value to a Role is a no-op. On inverting queries, we represent this operation explicitly, because it carries type information:

Value2Role Propertytype

But an inverted query should yield contexts, not roles. Hence, for the update function to find the context in which a property has changed from the role on which it is represented, the no-op is insufficient. It needs to be followed by the context step. So, on inverting a calculated property, we postfix the context step on the inversion of the expression.

6.2.5.3. Functions that operate on values

Consider:

thing: SomeRole

property Sum = Prop1 + context >> AnotherRole >> Prop2

Can we invert that? We've seen above how we invert an expression that consists of just a single Property, so that deals with the first operand. If we invert the second operand, we get:

Value2Role Prop2 >> context >> SomeRole

Why SomeRole? Because the property is defined on it. Visualise the original query path, as it moves from SomeRole to its context, then to AnotherRole and then to Prop2. Moving back, we start with the no-op Value2Role (‘arriving’ at AnotherRole), then we move to the context, *and then we have to move back to* SomeRole.

But we’re not done yet, because we need a context as the result. In fact, we’re in exactly the same position as with the simple property P1 defined in the previous paragraph. So the easy solution is to postfix the inversion with a context step:

Value2Role Prop2 >> context >> SomeRole >> context

It is glaringly obvious we could, alternatively, have removed the last step of the original inversion, too:

Value2Role Prop2 >> context

This is an implementation detail.

So we now have two inverted queries for our two operands:

Value2Role Prop1 >> context

Value2Role Prop2 >> context

The first will be used when Prop1 changes value; the second when Prop2 changes value. Both will return contexts of the same type.

And we’re done with that. The (+) function does not change anything: it does not ‘move’ over the underlying graph of context and role instances. The end result of the application of the function invertFunctionDescription (module Perspectives.Query.Inversion) is an instance of Paths, the representation of a series of query paths (see the previous chapter for an elaboration).

6.2.5.4. Join queries

We can join the result of two (role) queries:

property: Channel = (binder Initiator union binder ConnectedPartner) >> context >> extern >> ChannelDatabaseName

The sub-expression (binder Initiator union binder ConnectedPartner) has a Sum type.

We invert queries of this type by treating them as two separate queries:

binder Initiator >> context >> extern >> ChannelDatabaseName

binder ConnectedPartner >> context >> extern >> ChannelDatabaseName

Both can be simply inverted.

6.2.5.5. Filter queries

A filter combines a source and a criterium:

context: UnloadedModel = filter ModelsInUse with not available (binding >> context)

Again, we invert these queries by considering them to be two separate parts:

ModelsInUse

ModelsInUser >> not available (binding >> context)

6.2.5.6. Functions with arguments

A function like available takes an expression as argument. On inverting, we just ignore the function. So we treat

ModelsInUser >> not available (binding >> context)

just like

ModelsInUser >> binding >> context

(both not and available are functions with a single argument). Functions with more than one argument just lead to multiple queries, as with the join and filter operators.

6.2.6. What is inverted?

Query inversion is all about *expressions* in the Perspectives language. Queries occur in a limited number of lexical situations (that is, in an Perspectives Language source text):

- in the definition of a *calculated role*;
- in the definition of a *calculated property*;
- in the *condition of a state*;
- in the position of operands of *assignment operators*;
- as *excecutable expressions in notification texts*.

The PDR inverts all of these queries, **except for the operands of assignment operators** and for **excecutable expressions in notification texts**. This shifts the burden of ensuring that a peer has sufficient information to perform a calculation, to the modeller. Consider the following situation:

```
case Test1
  user Him
    perspective on Source
      props (B) verbs (Consult)
  thing Source
    property A (String)
    property B (String)
    state A_has_a_value = exists A
      on entry
        notify Him
          "Role Source now has value {A} for property A"
```

Looking at this example, it is pretty clear that user Him will never be notified of the value of A, even though he may see the role instance of Source with its B value (if some other user creates it). This is because of the design decision mentioned above: that we do not invert expressions in notification texts. Hence, resources mentioned in such texts are not synchronized *because of* these texts. The fact that the modeller has made clear his intention to notify Him of the value of A does not mean the PDR actually *sends* such values to Him.

Less conspicuous is the following example:

```
case Test2
  thing Source
    property A (String)
    property B (String)
  thing Access = Source
  thing Destination
    property C (String)
  user Her
    perspective on Access
      props (A) verbs (Consult)
      action Doit
        C = B for context >> Destination
```

First, notice what *does work*: because of Her’s perspective on calculated role Access, another user’s PDR will send an instance of Source to Her. And if that other user provides a value to property A of that instance, this value will be sent to Her as well.

But the modeller may find to his surprise that after user Her has executed the action Doit, Destination’s property C will not have a value. Why not? Because Her has no perspective on property B of Source. Even though B occurs in an expression in the action (the singleton expression "B", as operand of the property assignment operation "="), this is no cause for the PDR to synchronize values of B to user Her. It is necessary to give Her an explicit perspective on property B. Only then will her PDR receive the B value for role instance Source.

Why this design decision? This is for reasons of complexity and efficiency. Were we to synchronise every resource mentioned in an expression, we would find that the number of inverted queries increases a lot. The same property may occur over and over again and each occurrence would lead to a (duplicated) inverted query.

We might, of course, try to prevent this duplication but that requires new complexity. Let’s introduce the term 'implicit perspective' for a role or property occurring in an operand or notification text. If we were to invert such expression, for each role or property, the PDR would have to find out whether a user has an explicit or implicit perspective on it; if not, it would have to make an implicit perspective and invert it. While not impossible, this requires a quite complex refactoring of the current code.

PART II. THE LANGUAGE: DESIGN DECISIONS

In this part we discuss all facets of the language, from types to identifiers, models, calculated roles and properties, perspectives and aspects.

Chapter 7. The Type System

7.1. Semantics of the Perspectives Language

7.1.1. Context

A Context is a product type. A Context is the product of a number of roles, at the very least just a single one. In our programs we attribute special meaning to that single role: it *represents* the Context. In this account of semantics of PL it does not figure, however, otherwise than that there must be at least a single role for a Context. There is no *empty context* in PL.

7.1.2. Property

A Property is an atomic type, as far as the semantics of PL concerns. There are but three things of interest to a Property (we call them *attributes*) and those are

1. Whether it is *mandatory;*
2. Whether it is *functional;*
3. What is its *range type*.

A Property is a single dimension to compare Roles on. Range types are for example Integer, Boolean, String and Date. However, an implementation may add other Range types.

Of theoretical interest is a Range type without values. We use it to define the ValueLess Property.

7.1.3. Role

Roles are by far the most interesting type in PL. A Role is one of three things:

1. A *Simple Role*, which is a Product of Properties;
2. A *Product Role*, which is a Product of Roles;
3. A *Sum Role*, which is a Sum of Roles.

Because we can transform a Product of Sums or a Sum of Products to a normal form that has just Simple Roles, we can simplify the above definitions by stating that a Product Role is a Product of Simple Roles (and a similar thing holds for Sum Roles). So, without loss of generality, we assume some normal form and just talk about Sums and Products of Simple Roles.

7.1.3.1. Role instances

Above we have defined Role types in terms of properties. Alternatively, we can define a Role type as a set of *Role Instances*. We say that the type is *inhabited* by its instances. An instance is a collection of values for Property Types.

A Role Type constrains the instances that can inhabit it. It requires that an inhabitant has a value for each of the Property Types that define the Role Type. Each such value must be one of the values of the Range type of the Property.

> Here, it is of importance to distinguish the requirement that a role instance has *at least* the required properties, or that it has *exactly* the required properties. The latter we can interpret as a form of the *closed world assumption* (CWA): the descriptions are definite and something that is not described, is not the case.

7.1.3.2. Properties of a Product Role

A (normalised) Product Role is just a product of Simple Roles. A Simple Role is a product of properties. Hence, the properties of a Product Role is just the Product of the properties of all Roles in the product.

An instance of a Product Role of A and B is an instance of both A and B and carries all properties of A and all properties of B. Consider an instance ‘Joop’ of the product of Person and of PhysicalObject. As a Person, Joop has a familyname (Ringelberg). As a PhysicalObject, Joop has a dimension length (1.93 meter).

7.1.3.3. Properties of a Sum Role

A (normalised) Sum Role is just a sum of Simple Roles. A Simple Role is a product of properties. An instance of a Sum Role of A and B is either an A, or a B. What properties can we be sure of? Those that are in the intersection of the properties of A and B.

Consider a limited definition of a Vehicle as the Sum of either a Car or a Bicycle. This sum will have a property NumberOfWheels, but it will not have a property CapacityOfFuelTank – even though occasionally an instance of this sum will have both properties.

7.1.3.4. The Empty Role

We can imagine a Role with no properties: it is called the Empty Role. A Product of a Role with the Empty Role is just that Role itself; a Sum of a Role and the Empty Role is itself the Empty Role.

Explanation of EmptyRole x Role = Role. When we form a Product of Roles, we take the *union* of their properties. However, the EmptyRole does not have properties, so contributes nothing. Hence the result is just the original Role.

Explanation of EmptyRole + Role = EmptyRole. A Sum of Roles A and B is the role with the *intersection* of their properties. But the Empty Role has no properties, hence its intersection with any other Role is just an empty set – in other words the Empty Role itself.

Does the Empty Role have instances? A Role type constrains its instances by requiring that they have a value for each Property in the type. But the Empty Role has no Property Types, so is not much of a constraint. Hence, all role instances are an instance of the Empty Role.

> An alternative interpretation, based on the CWA, holds that the role instance *can have no properties as they are not described*. Hence, we come to the opposite conclusion: just a role without any properties at all can be an instance of the Empty role. But what is a role without properties?

7.1.3.5. The Universal Role

Now turn to a Role with all properties: let’s call it the Universal Role. A product of the Universal Role and another Role is just the Universal Role; the Sum of the Universal Role and another Role is just that other Role.

Explanation of UniversalRole x Role = **UniversalRole.** When we form a Product of Roles, we take the *union* of their properties. However, the UniversalRole already has all properties, so Role contributes nothing. Hence the result is just the UniversalRole again.

Explanation of UniversalRole + **Role** = **Role**. A Sum of Roles A and B is the role with the *intersection* of their properties. But the Universal Role has all properties, hence its intersection with any other Role is just that other Role.

Universal Role instances

The Universal Role is funny when it comes to instances. To start with, an instance of the Universal Role is always an instance of any other Role (it always qualifies because it has values for all properties!).

But does the Universal Role have instances at all? Yes, infinitely many, because there are infinitely many properties and each instance of the Universal Role has a value for each of them. It follows that each such instance must be infinitely large.

Nevertheless, each ordinary Role has instances that the Universal Role does not have (because the Universal Role is the pickiest of all Role types). So, while being infinite, its instance set still is less than that of any other Role (for a discussion, see *Instances of a Product*, below).

Importantly, the infinite character of Universal Role instances severely limits its usefulness, however, because in our programs we only handle finite instances. In other words, in practice the Universal Role cannot be inhabited. So if we require that a Role be filled with instances of the Universal Role type, we say in that – again, in practice, in our programs - it will never be filled.

Empty Property construction

We have postulated a Property with a Range without values: the ValueLess property. If we concede that a Role instance with this Property cannot exist (because an instance must have a value for each Property), it follows that the Universal Role is an uninhabited type. After all, the Universal Role has all Properties, including the ValueLess Property.

So if we assume the ValueLess Property, the Universal Role has no instances, even in theory.

7.1.4. Ordering of Role types as property sets

Considering a Role to be a set of properties means we can compare Roles and partially order them. For example, the Sum A+B has just the properties that are in the intersection of A's properties and B's properties. Hence the properties of A+B are a subset of both A's properties and B's properties. We say that A+B is *less specific* than either A or B.

Definition: Role X is *less specific* than Role Y if X's properties are a subset of Y's properties (less specific = isSubset over property-set);

It follows that AxB is *more specific* than either A or B. It has more properties than either A or B.

7.1.4.1. Empty Role is least specific

What about the Empty Role? It's properties are the empty set. The empty set is a subset of every other set, hence **the Empty Role is less specific than any other Role**.

7.1.4.2. Universal Role is most specific

The Universal Role has all properties. It is a superset of all other sets, hence **the Universal Role is more specific than any other Role.**

7.1.4.3. Instances and ordering

A Sum Role like A+B has more instances than either A or B. After all, each instance of A is an instance of A+B and the same holds for instances of B. So while A+B is *less* specific than either A or B, it has *more* instances. This is understandable if we consider a Role type as a constraint on instances. So we can give the definition of less specific in terms of instances too:

Definition: Role X is *less specific* than Role Y if X's instances are a superset of Y's instances (less specific = isSuperset over instance-set).

Instances of a Product

The instances of a Product may confuse you. The Product AxB actually has *less* instances than either A or B. But nevertheless it is created by multiplying A and B together! In our minds eye the multiplication table is larger than the row of column labels or the column of row labels.

But consider: each instance of AxB is an instance of A (it has the required properties). But an instance *with just the properties required by A* is not an instance of AxB – though it obviously is an instance of A. So A has instances that AxB has, but not vice versa – AxB is a subset of A.

7.1.5. Another ordering of Role types

There is, however, another base for comparing roles and that is to consider instances as 'tagged individuals'. An instance is an instance of a role R by declamation, as it were; not because it has a particular structure (in terms of properties).

Let's briefly diverse into a discussion of Aspects to underline this (See the chapter Aspects below for a brief explanation). Consider a rather universal role Driver, that can be used in many contexts, e.g. as the driver of an Ambulance or the driver of a Taxi. Driver-in-an-Ambulance has a different meaning than Driver-in-a-Taxi. A model may, for example, contain a query to count the number of hospitals that the Driver-in-an-Ambulance has visited in a period of time. Clearly, if the same person drives both Taxi's and Ambulances, we want to separate out both 'contextualized' roles when performing that query.

7.1.5.1. Product types

There is no `natural` ordering of simple role types. But there is a way to consider compound roles to be ordered. Take, for example, the Product of two roles A and B and compare it to A itself. Surely, A is less specific than AxB. Any instance of AxB is, by construction, also an instance of A, but not the other way round.

From this we derive the notion that, for Product types, when we consider the products as the sets of their terms, subset means: less specific.

7.1.5.2. Sum types

What about a Sum of A and B? How does that compare to A? Well, an instance of A is always an instance of A | B, but not the other way round (after all an instance of B is an instance of A+B as well – but clearly not an instance of A!). So, A+B is less specific than A. Paraphrasing it may be easier to understand: A is more specific than A+B ("I will talk not to just any Parent, only to the Mother!").

So, for Sum types, considering them as sets of terms, subset means: more specific.

7.1.5.3. Arbitrary types

Taking this as the base of our alternative definition of ordering, how can we generalize it to any role defined in terms of Sums and Products?

Any construction in terms of Sums and Products can be converted into Disjunctive Normal Form (DNF). A formula is in DNF iff it is a disjunction of products, where the term of each product is atomic.

Let's work out how we can compare two types written in DNF. We'll investigate: SUB is more specific than SUPER.

For Sum types, more specific equals subset. SUB should be a subset of SUPER. The definition of subset is: each element of SUB should occur in SUPER.

What if the terms (elements) are Product types? Instead of using simple equality, we must now ask of each element in SUB if it is a superset of an element in SUPER. Remember that the elements are Products and that for them more specific means: superset.

In short, our entire algorithm is: is *each element* in SUB the superset of *some element* in SUPER?

7.1.5.4. Aspects

In the previous paragraph we used reduction to Disjunctive Normal Form (DNF) to prove the soundness of the function equalsOrGeneralisesADT. This function does not take Aspects into account, because we treat terms in Products and Sums as atomic. However, this is insufficient to capture the semantics of Perspectives types, as the following example shows. Let T have aspect A, then clearly A `equalsOrGeneralises` T. However, our function will return false! This is because union’ will be {A}, while intersection’ will be {T} and {A} is not a subset of {T}. Luckily, we can easily improve the function to cover aspects, too.

When we give a type T an aspect A, we actually state that any instance of T is an instance of A as well. We can think of a type-T-with-aspect-A as the Product TxA. Imagine that we expand all terms to the full set of their aspects (i.e. the type itself and its aspects). Then we would have:

- union’ = {A}
- intersection’ = \{A, T}

and now the first is a subset of the latter, hence clearly A `equalsOrGeneralises` T returns true! In other words, we can capture the semantics of Perspectives types in the function equalsOrGeneralises by expanding all types to their transitive closure over the aspect relation.

7.1.6. Creating products and sums of roles

So how do products and sums of roles arise? PL allows for several ways to construct them.

7.1.6.1. Filling

The primary way to construct a Product Role is by specifying a Role with a filler:

```
user Person (..) filledBy PhysicalObject
```

Here we create the product of Person and PhysicalObject. All properties of PhysicalObject ‘can be reached’ through Person. We can specify a View with all those properties.

7.1.6.2. FilledBy as a constraint

The type we specify as the possible filler of a Role constrains what instances we may actually bind to that Role. Only inhabitants of the filler type may be bound to an instance of the Role.

When we say that the filler of a Role is the Empty Role, we say in effect that anything goes. There is no constraint. So, by default, each role has the Empty Role as its filler type.

In contrast, the Universal Role as the filler type of a Role is a tall order. We’ve seen that in the practice of our finite programs there can be no instances of the Universal Role. So in effect, by requiring the Universal Role as filler type of a Role, we say that that Role can have no filler.

7.1.6.3. Multiple fillers

This is how we create a Sum of Roles:

```
thing Vehicle (..) filledBy Bicycle, Car
```

Vehicle can be filled by either a Bicycle, or by a Car. Notice that we can add Properties to the definition of Vehicle itself, too. Then we would have a product of Vehicle and the sum of Bicycle and Car. All instances of Vehicle would have Vehicle's properties.

When Vehicle has no properties of its own, it is in effect the EmptyRole. A product of R and the EmptyRole is R itself, so in that case Vehicle is just the sum of Bicycle and Car.

7.1.6.4. Multiple fillers in a product

We also have the option to specify multiple fillers in a product:

```
user ClientOfPharmacy (..) filledBy BankAccountHolder + Patient
```

> Attention: we use the + symbol here to denote a product in the sense that we've defined it above. This is because +, interpreted as 'and', reads more natural to a lay person than *.

Here, an instance of the role ClientOfPharmacy **must** be filled by both an instance of BankAccountHolder and an instance of Patient. This is because the pharmacy needs access to my medical records and it also needs access to my bank account number (assuming it will automatically collect the costs incurred).

Why do we need the BankAccountHolder + Patient syntax? Can we not instead accumulate properties by filling BankAccountHolder with Patient? We could, and we could do it the other way round, too. And that is the catch: by choosing either way, we *tie both roles together forever*. Obviously, I don't need my medical records in all situations where I need my bank account and the same holds vice versa. There is no 'natural model': it just depends on the situation.

Even worse, we can never be sure what combinations of properties might be required in future models. The best way we can prepare for that future is to lump everything together in the 'role at the bottom', whatever we choose that to be. It is an *include everything even the kitchen sink* approach to modelling we obviously want to avoid.

This is why we need the opportunity to bring in packets of properties as desired, using the product syntax given above.

Another way to think of this is that a product created by filling is 'on the supply side', whereas a product created by + is 'on the demand side'. We create that product when we need it, instead of creating it because we might need it in the future.

7.1.6.5. Aspects

By giving a Role an Aspect, we bring in the properties of that Aspect. So, on the type level, Aspects create a product of roles, too. For example:

```
thing HomeAddress (..)
  aspect: Location
```

creates the product of HomeAddress and Location. Instances of a HomeAddress will have properties of both, for example TypeOfHome for HomeAddress and X and Y from Location.

The following lines create the same product type:

```
thing HomeAddress (..) filledBy Location
```

However, on the instance level things are different. With the latter model, we would have to bind an instance of Location to an instance of HomeAddress. With the former model, there would be no such instance. Instead, the HomeAddress instance would have the Location X and Y properties, too.

Aspects of Contexts work in the same way. A Context Aspect introduces Role types into a Context. However, there is no alternative modelling to consider, as we cannot ‘fill’ Contexts.

7.2. Abstract Data Type

7.2.1. Introduction

There are three use cases that underline the need for *abstract data types* to reason about roles, contexts and properties. Without explaining the concept, we’ll give these three cases to build some intuition.

The binding of a Role can be a set of alternatives. We call such a set a Sum of types. Alternatively, we can stipulate that a Role must be bound to multiple other instances *at the same time* and that we call a Product of types.

> We’re talking *type level* here, so a Role is a type, its binding is a requirement on actual bound instances.

The query language supports concatenating two queries. This means we can create a set of role instances of two or more types. The type of such a set is a Sum type, too. In a sense we create a new type of role in this way, just like an Enumerated Role with multiple alternative bindings (but in a query we construct it on the fly and we do not add own properties, or aspect roles).

An EnumeratedRole is described by giving it aspects and a binding. If we reason about roles and their relations, it is convenient to lump these three facets together in a single Product type. The role itself (as a set of Property types), its aspects and its binding form a Product.

In the text Semantics Of The Perspectives Language we have explained these issues in more detail. In this text we describe the representation of such compound types and the functions we have for manipulating them.

7.2.2. Representation

We represent compound types with an Abstract Data Type, or ADT for short:

```
data ADT a =

ST a |

SUM (Array (ADT a)) |

PROD (Array (ADT a)) |

EMPTY |

UNIVERSAL
```

ST stands for Single Type or Simple Type. When we describe roles, the parameter a is bound to a RoleType.

> Not just EnumeratedRoles, e may want to bind to a CalculatedRole. Consider the case where you want to restrict a binding to grown up participants (bind to a filtered role). Or to parents (Sum of Father and Mother).

Empty and Universal are edge cases of, respectively, a role without properties and a role with all properties. The former we use to specify that no restrictions hold, on binding a role; the latter says that nothing can bind to a role. In the syntax we use the keyword NoBinding.

> Because all properties include the property without values and no role instance can have a value for that.

7.2.3. Ordering and type specificity

The compiler checks the validity of bind assignment statements. Such a statement, in the right hand side of a rule, is the intention of a user to have his bot make an assignment if the conditions are right. But is this binding allowed? Obviously, when the type of the required binding equals that of the binding, the answer is yes. But we should allow the binding, too, when the type of the required binding is *less specific* than that of the binding.

In Semantics Of The Perspectives Language we have defined specificity as the relation between the property sets of Roles: Role X is *less specific* than Role Y if X’s properties are a subset of Y’s properties (less specific = isSubset over property-set)

7.2.4. Property Sets

Specificity is defined in terms of the Property sets of Roles. This means that we must have a way to compute the Property set of an ADT. Here is an algorithm:

1. For an ST ENR EnumeratedRoleType just take the role’s own properties.
2. For an ST CR CalculatedRoleType, retrieve the calculation of the role and compute the property set of (the ADT in) its range.

3. For a Sum type, form the intersection of the properties of the terms. If one of the terms is Empty, the intersection is Empty. Ignore Universal.
4. For a Product type, take the union. If one of the terms is Universal, the union is Universal. Ignore Empty.

Because we have to handle Empty and Universal, we cannot represent a Property set merely by an Array or a List. Hence we define the following type:

```
Data PropertySet = Universal | Empty | PSet (Array PropertyType)
```

7.2.5. Other sets: views and aspects

What are the Aspects of an ADT? This is a relevant question for a modeller. Each EnumeratedRole can have Aspects, much like it can have properties. Hence, we can re-use the algorithm we used for computing a PropertySet for computing the set of Aspects of an ADT.

For Views, the situation is slightly different. We handle ST and Product just like with properties, but the Views of a Sum is not just the intersection of the Views of its terms. Rather, we have to compute the PropertySet of the Sum and then use it to filter each View in the *union* of the view set per term. This is because we have only use for a View if the role it is applied to, has all the properties it wants.

7.2.6. Functions on ADT’s

The Description Compiler translates the Abstract Syntax Tree that results from parsing a Query into a description of a function that, in its turn, can be translated into a function that actually runs the Query. While doing so, the Description Compiler checks whether the result of applying a query step is suitable as input for the next step. That is, it compares the range of a step with the required domain of the next step.

It therefore must be able to determine the range of a querystep, given the domain that is supplied to it, and the type of the function that is applied.

To make this less abstract, consider the query step type *binding*. Given a particular Role, what is the binding of that role? This is easy; we just look it up in the Role’s definition. But what if we then again take the binding of the result? Then we have to compute the binding of an ADT, because binding types are represented as ADT’s. In this chapter we explain how the five relevant functions work out for the various ADT type constructors.

The functions are: from context to role and vice versa; from role to binding and vice versa; and from role to property.

7.2.6.1. RoleX: From Context to Role

The simple case is ST ContextType. To take a particular role of such an ADT simply yields that Role type: ST RoleType.

If the modeller wants to take the role from Empty we have to throw an error. After all, Empty here means a Context without roles.

By similar reasoning, taking a role from Universal succeeds and yields the same result as with ST ContextType. However, notice that this would only work if the Role type has been specified completely by the modeller.

> But in reality this case will never arise. To see why, we must ask ourselves under what circumstances the Description Compiler would have to handle a domain of Universal. Universal is only introduced by the modeler as a requirement on role binding. He thereby specifies that that role can never be bound. So Universal is the range of the function binding, when applied to such a role. But, by definition, if the modeler does so, we signal an error. So no function will ever return Universal; and so the Description Compiler will never meet Universal as domain.

A role from a Product succeeds if the role is in the union of the roles of the terms of the Product. But what if there are multiple roles in that union that match a local role name? Then we can return a Product of those roles.

A role from a Sum of Contexts only succeeds if the role is in the intersection of the roles of the terms. Notice that this is only meaningful for local role names: each Context in the Sum must have a Role with that local name. Again, if multiple roles match a local role name, we return a Product of those roles.

7.2.6.2. Context: From Role to Context

The simple case (ST RoleType) is obvious.

For Empty, we return Empty. This is because we know nothing about the Empty role. To make this concrete: suppose no binding requirements have been set for a Role. We are allowed to take its binding, but from then on we are ‘off the chart’. Any role might be bound to such a Role, so any context can be the result of taking the context of its binding. And the only thing we know about every context type is that it is a subtype of Empty.

For Universal it can only work for fully qualified role types (but we have seen that the case will never arise).

What is the context of a Product of Roles? It is the Product of each Role’s context and that can be considered to be a ‘super-context’, holding all roles of the terms.

What is the context of a Sum of Roles? Again, it is just the Sum of Contexts.

7.2.6.3. Binding: From a Role to its binding

We find the binding of ST ENR EnumeratedRoleType simply by looking it up in the definition of the EnumeratedRole. For a CalculatedRole, we look up its calculation and take the binding of the ADT that is in its range.

> This range must be of the form: RDOM (ADT RoleType).

binding applied to Universal is simple, as the modeller uses Universal to stipulate that no binding is allowed. Hence, the modeller should not take the binding of Universal, so we throw an error.

Similarly, binding applied to Empty just results in Empty. Anything goes.

For Sums and Products, we construct the Sum of the bindings and the Product of the bindings, respectively.

A word on normalisation. There is no need to create a normal form of an ADT (e.g. Conjunctive Normal Form), though we could. However, if, on creating a Sum or Product, we detect Empty or Universal among its terms, it is advisable to normalise the result. Thereby we avoid unnecessary work later.

> A Product with Universal is just Universal; we can leave out Universal from Sums. We can leave out Empty from Products, and a Sum holding Empty is just Empty.

7.2.6.4. BinderX: from a role to its binders

The function binder takes an argument that identifies a role type. The Description Compiler checks if this is a legal move by looking up the required binding in the definition of that EnumeratedRole or CalculatedRole. The domain of the binder function (the assumed binding, an ADT RoleType) must be equal to or more specific than the required binding.

It is important to understand that we never try to compute the binder of an arbitrary ADT. That has no meaning.

Notice, however, that the ADT that results from the function binder is always of the form ST ENR EnumeratedRoletype. This is because only enumerated roles can bind other roles.

7.2.6.5. PropertyX: from a role to a property value

We can think of Properties in abstract terms, too. But a Property has no sub-parts, as do roles. Instead, a Property is characterised by its Range. However, we have also found that we cannot ignore the Property name itself (Not explained in this tekst). It is the carrier of semantics we cannot afford to lose. Hence we have the Description Compiler handle abstract descriptions of Properties in terms of

- Its name, and
- Its range.

We can have abstract data types constructed from these pairs.

The Description Compiler constructs descriptions of functions on Property values. For example, it may construct a function that adds the values of two properties, or counts the number of values. It guards the compatibility of these functions with the ADT’s that describe the properties.

So how do we construct an ADT if we take the property value of an arbitrary ADT RoleType?

The simple case is ST ENR EnumeratedRoleType. We find the ADT of the results of the property-taking function by looking up the range in the definition of EnumeratedRoleType.

For an ST CR CalculatedRoleType we take the range of the calculation and work from there (i.e. compute the ADT of that ADT RoleType).

Taking a property of Empty should result in an error. The empty role has no properties.

Taking a property of Universal should be allowed, as the Universal role has all properties. On the other hand, this case will never occur. The modeller uses Universal to stipulate that no binding can exist for a particular Role. Taking the binding of such a Role results in an error. Hence we can never arrive in the situation that Universal is the domain of a function.

Taking the property of a Sum of roles succeeds only if the property is in the PropertySet of that Sum. If multiple Role types have names that match with a local property name, the result will be a Product of those properties.

Similarly, taking the property of a Product of roles succeeds only if the property is in the PropertySet of that Product. Again, if multiple Role types in that set match the local name of a property, we combine those properties in a Product.

7.3. Type reflection

7.3.1. Introduction

Each instance has a type. That goes for contexts, roles, and also property values. In this text I hold it to be an invariant that the core can always reflect on the type of its instances, because the model describing the type is available in the local installation.

Terminology: a type falls into a namespace; a namespace is described in exactly one model. A type is known locally if it is described in a model present in the local installation.

A model can be added by the end user to his local installation. He does so to be able to use new perspectives. But a model can also be implicitly added as a result of another action. Such a model is added:

- because it is an import of a model imported explicitly;
- because the user performs an action through the user interface that constructs a context or role instance from a description contained in the code of that user interface, where the type is not locally known;
- because a peer constructs an instance that the end user (the recipient) has a perspective on while the type of that instance is not locally known;
- because the end user is retrieving an instance from a public store whose type is not locally known;
- because the end user uses the core to parse and compile a model text.

7.3.2. Responsibilities of authors

Currently we require the author of a model to add the dependencies to his installation by hand.

7.3.3. Responsibilities of the authors of screens

We consider the API **not** to be a system boundary. In other words, the author of code that addresses the API must behave responsibly in the sense that his calls

- only use instance identifiers from instances with a locally known type.
- only use type identifiers from locally known types.

> With one exception: the type identifier used in a RolDescription or ContextDescription is checked and, when not locally known, the corresponding type is loaded.

This may be easier than it looks. In the first place: instances are only created in the core ('behind' the API) and for those we guarantee that their types are known locally. Thus, because programs that use the API can only work with instances coming from the core, their type must be known locally. The only exception are indexed names. The author must spell them right; however, errors with this become apparent very quickly during development.

In the second place: almost all type identifiers that API users can use in their calls, also come to him through the API. These are either types of instances (which are known locally for each construction), or types that are retrieved directly from a model (and are therefore known locally, too).

Only some type identifiers, that are hard-coded into the client code, are the responsibility of the client's author. An example is the Role component. The value of the 'role' prop must be traceable to a locally known type. If the author makes a mistake, this will become apparent very quickly when trying out the screens.

The React library (the primary means to build client programs) behaves responsibly.

7.3.4. Reflection on model:System is guaranteed

The setupUser function ensures that model:System is present in the local installation.

7.3.5. Type reflection on locally created instances is guaranteed

We want to be able to guarantee type reflection on existing instances. Instances have two sources: the 'own' user, and peers.

The user can only create instances through the API. If a ContextDescription or RolDescription is included in the API call, we check whether the given types are present locally. This is because these can be passed as data (value of React props) to the react components CreateContext, CreateContext_, CreateContextInExistingRol, RoleInstance and RoleInstances.

Because of this check, we know that type reflection is possible on all locally created instances.

7.3.6. Type reflection on instances made by peers is guaranteed

Instances created by peers come in in the form of deltas in transactions. Deltas come in in a particular order:

- first the external role of a context is created;
- then the context itself (and we add the external role to it automatically without there being a delta for it);
- then the role instances of the context;
- finally, the roles are placed in the contexts.

If the type of the context instance is locally available, all of its role types will be locally available, too. Conversely, if a role type is locally available, so will be its context type.

The same applies to a role instance: if the role type is locally available, all of its property types will be locally available as well.

On receiving a UniverseRoleDelta, the Perspectives.Sync.HandleTransaction module checks whether the type to be created is present locally. This also applies to the external role of a context. Hence, after receiving the delta that describes the external role, we can be sure its context type is locally available, so we do not need to check types for a UniverseContextDelta. And if we have the role type, we also have the property types. In short, type reflection on instances created by peers is guaranteed.

7.3.7. When the user fills a role

At first glance, it seems that if the core fills a role, the filler type should be locally available, too. After all, the role instance has been retrieved from the local storage and we are guaranteed to be able to reflect on its type. That type description includes the required type of the role filler, so it would seem the type of the filling instance must be locally known.

But it is possible to fill a role with an instance that is a *specialization* of the type required in the model. That specialization may have a different namespace and the model in question may not yet be stored locally. For this reason, the setBinding function checks whether the filler type is locally available. If not, it adds that model to the local installation.

Basically, when constructing a bond, the invariant is enforced.

7.3.8. Type reflection on instances from public storage

A core adding a role instance to a public store will certainly be able to reflect on the type of that instance. But a core that retrieves that instance cannot be sure of it. So this is a very different situation than retrieving from local storage, because there the core that saves and retrieves is one and the same.

Therefore, we should always check that we have the model that describes the type of a role instance that we retrieve from public storage.

We (almost) always retrieve a context instance from the reference of a role instance. The type of a role instance's context is described in the same model as that role instance's type itself, so we never need to check that we have the model needed for reflection on context instances. Never, except in the case of Uniquely Identifying Readable Names (UIRN): these are URLs in the cw subdomain without a query part, such as https://cw.nlnet.nl. Such an identifier can, but does not have to, identify a context instance whose type is not known locally.

A circumstance under which it is not known locally is, for example, if NLnet has defined its own model (model://NLnet.nl for example) and has made an instance of one type available under the identifier https://cw.nlnet.nl. A user who enters that name for the first time in his browser will not have model://NLnet.nl.

Chapter 8. Identifiers and Variables

8.1. Expanding prefixed names

8.1.1. Introduction

The syntax of the Perspectives Language describes a construct prefixedName (see *Syntax of the Perspectives Language*). An example of a prefixed name, taken from model:System, is sys:PerspectivesSystem. The prefix sys: expands into model:System$, which gives us model:System$PerspectivesSystem. However, not all positions that allow identifiers support this expansion. This text gives the details.

8.1.2. Declaring prefixes

Prefixes are local to the model in which they are used. Even stronger, they are local to the *context* in which they are declared (where domain is a context, too). The modeller declares a prefix with a line like this:

use: sys for model:System

(notice we omit the ‘:’ and ‘$’ for clarity).

Prefix declarations shadow each other; if we were to declare a prefix sys local to, for example, sys:TrustedCluster, binding it to model:SomeOtherNamespace, this means that an identifier prefixed with sys nested inside sys:TrustedCluster will expand to model:SomeOtherNamespace.

8.1.3. Syntactic locations that support prefix expansion

Some syntactic locations where an identifier is expected, support prefix expansion. These are the locations listed below (enclosed in < and > in the examples). In these situations we must use a **segmented name**, such as PerspectivesSystem$User (but notice that a single segment such as User is ok).

1. aspect <aspectName>.
2. indexed <indexName>.
3. callExternal <functionName>.
4. callEffect <effectName> returns <resultType>.
5. filledBy: <roleName>.
6. view: (<propertyName>).
7. all references to views:
    a. perspective on: SomeRole (<AView>). Here, <AView> is the default view.
    b. Change with <AnotherView>. <AnotherView> is a verb-specific view.
    c. indirectObject SomRole (<IndirectObjectView>). The view specific to the indirect object of an action.

8. All occurrences of role- or property names in query expressions.
9. All occurrences of *indexed names* in query expressions.

8.1.4. Syntactic locations that don’t support prefix expansion

We also have syntactic locations where we must use a **local name** (i.e. a capitalised name: the production segment in the identifier grammar):

1. All declared names:
   a. all context kinds (domain, case, party, activity, state), e.g. case <caseName>
   b. all role kinds (thing, context, user, external), e.g. role <roleName>
   c. properties: property <propertyName>
   d. views: view <viewName>.
2. perspective on: <object>
3. indirectObject: <object>
4. bot: for <user>

All these local names are automatically scoped to their enclosing namespace. This means that, for example for a role, the name of the context is tacked on front.

8.1.5. Implementation notices

A model source text is parsed using the functions in the module Perspectives.Parsing.Arc. Query expressions within the text are parsed using the functions in the module Perspectives.Parsing.Arc.Expression. Neither of these modules expand prefixes.

Instead, this is done exclusively in the modules

- Perspectives.Parsing.Arc.PhaseTwo;
- Perspectives.Query.ExpandPrefix,

for, respectively, identifiers in the main text and in query expressions.

A model source text is compiled to a DomeinFile of type data. The last stage of this compilation is performed by the module Perspectives.Parsing.Arc.PhaseThree. We can safely assume that the data structures traversed by functions in this module contain just expanded identifiers; prefixed names do no longer occur in them.

8.2. Type and Resource Identifiers

Dealing with a Perspectives model and instances involves many things that must be *identified*, for example:

- Models
- Types (contexts, roles)

- Instances (contexts, roles)
- Storage locations (stores)

The Perspectives Distributed Runtime must be able, too, to *find* all physical resources that represent these models, types, instances and stores. In this text I explain how we build identifiers, when identifiers are locations and how we find resources whose identification is not a location.

Furthermore, we have to deal with the fact that things change. Models evolve, meaning that type definitions may change and sometimes even their names might change. The physical location of a store may change, too. This of course is complicated because the Perspectives Universe is full of things that *refer* to each other – by identifier. Change introduces the concept of *versioning*.

Finally, there is the dimension of *context of usage* in which an identifier must, well, identify. For some identifiers we can firmly establish that they will never leave a particular context of use: this holds for the identifier of a type in a model, for example. Others, however, may be used worldwide and should be globally unique.

The text does not begin with a full, final definition of the shape of identifiers but works towards them gradually, to build understanding of the parts and their function.

8.2.1. Some definitions

A **model** is a collection of types.

A **namespace** is an identifier, used to qualify types in a collection. Hence a model identifier is a namespace, but so is a context type identifier, for its roles.

A **model text** is a readable text file that defines a model. The PDR can transform it into a domeinfile.

A **domeinfile** is a resource (a json file) that holds the types in a model in a form that is used without modification by the PDR to reflect on types.

An **instance** of a context or role is represented by a resource (json file).

A **store** holds a collection of resources. Perspectives distinguishes domeinfile stores from instance stores.

A **resource** is a file.

8.2.2. On Uniform Resource Identifiers

We make use of the definitions given in rfc3986, *Uniform Resource Identifier (URI): Generic Syntax*. Briefly: a URI consists of a *scheme name*, separated by a colon from an identifier that is constructed according to the scheme. The http and https schemes are well-known examples. The URIs defined by these schemes may be classified as *locators* and hence can be called a Uniform Resource Locator (URL).

In the text below, we define three custom schemes for use in Perspectives.

8.2.3. Model identification

As stated above, a model is a collection of types. These types are organised hierarchically. Context (types) contain context types and many others, role (types) contain property types, etc. The hierarchy begins with a context type that has exactly the same identifier as the model itself.

Models can be authored by anyone, all over the world. Models can also be *used* by anyone, in any combination. This requires model identifiers to be globally unique. We do not assume there will be a dedicated global register of models, hence an author cannot check whether the model name he intends to use is unique by comparing it to existing models. Instead, we need to provide a method for constructing a globally unique identifier.

Model names need to be communicable, too, for example because authors may want to advertise their models. This precludes Globally Unique Identifiers (GUIDs) as defined in rfc4122, *A Universally Unique IDentifier (UUID) URN Namespace*. These identifiers are anything but human-readable, are very hard to remember and so do not lend them for human communication.

Instead, we construct a custom URI scheme with the name *model*. rfc3986 states: *Many URI schemes include a hierarchical element for a naming authority so that governance of the name space defined by the remainder of the URI is delegated to that authority.* For Perspectives, we will rely on the domain name system (DNS) as the authority that governs name spaces for the internet, to ensure that model names are unique.

What identifiers can we construct under this scheme? Here is an example of a full URI:

```
model://perspectives.domains#System
```

model is the scheme name, separated with a colon (as prescribed) from the identifier constructed under the scheme. The authority has to be delimited up front by two forward slashes and at the end by another forward slash, a question mark or a hash sign.

> We forego the possibility to end the authority with a forward slash or a question mark: we just use the hash sign. This decision is heavily influence by the fact that forward slashes cannot be part of Couchdb file names.

In the model scheme, the authority itself has no user- or port information. Following the authority is the local name for the model. This name must start with an upper case character and must be unique in the domain identified by the authority to make the URI unique.

Referring back to rfc3986 we define this syntax for the identifiers under the model scheme:

```
Identifier = ▯//▯ reg-name ▯/▯ segment-nz-nc
```

A reg-name (registered name) intended for lookup in the DNS uses the syntax defined in Section 3.5 of RFC1034 and Section 2.1 of RFC1123. A segment-nz-nc is a non-zero-length segment without any colon ":". We add the constraint that it must start on an upper case character.

Our model scheme has the advantage that it can be easily mapped onto a URL. As a matter of fact, the identifiers constructed under the model scheme are a subset of those that can be constructed under the https scheme (after substituting *model* by *https*). This means we can locate a unique resource by means of a model identifier (more on this below).

8.2.3.1. Mapping URIs to URLs

We use a simple mapping from URI to the URL where the DomeinFile (the compiled model text) can be found. The authority (a reg-name) is used thrice:

- Once as the authority of a URL in the https scheme;
- Once as part of a single step in the path of that URL. That single step is prefixed with “models_” and all dots are replaced by underscores. It identifies the Couchdb database that holds the DomeinFile;
- Once as part of the actual file name that the DomeinFile is stored under (see DomeinFile Identifiers.)

For example:

`model://perspectives.domains#System`

maps to:

`https://perspectives.domains/models_perspectives_domains/perspectives_domains-System.json`

Another example:

`model://social.perspectives.domains#System`

maps to:

`https://perspectives.domains/models_social_perspectives_domains/social_perspectives_domains-System.json`

In this way, we can have multiple hierarchical levels in our model namespaces, while always mapping them to a location in a simple domain (By which we mean: a top level domain, such as “com” and one subdomain (such as “google”)).

8.2.3.2. Model stores

The Perspectives Distributed Runtime uses Pouchdb, that relies on the conventions of Couchdb, to access resources. For our purposes, this means that a webserver must map URLs of the form `https://perspectives.domains/models_social_perspectives_domains/social_perspectives_domains-System.json` to a database in a Couchdb installation (for example, a local installation). It is up to the webserver to provide that mapping (But there are three restrictions. See Mapping Model Identifiers to Storage Locations). However, we suggest a simple scheme that just uses the first step of the path as the database name. In this case, the json resource might be retrieved from Couchdb using this string: `models_social_perspectives_domains/perspectives_domains-System.json`. That is, we want the resource perspectives_domains-System.json in the database `models_social`.

> According to its documentation, Couchdb allows forward slashes in its database names. In the practice of version 3.1.0 this runs into problems. Hence we replace slashes by underscores.

8.2.3.3. Storage service providers

Suppose Perspect BV io, owner of the domain `perspectives.domains` would provide storage services to third parties, how would it handle them? It could offer an author like me to use a subspace of their namespace, e.g. `joopringelberg.perspectives.domains`. I could then create a model with this name: `model://joopringelberg.perspectives.domains#JoopsModel`. This would map to the following url: `https://perspectives.domains/models_joopringelberg_perspect_perspectives_domains/joopringelberg_perspect_perspectives_domains-JoopsModel.json`. Their server would consequently look for the resource `joopringelberg_perspect_perspectives_domains-JoopsModel.json` in the database `models_joopringelberg_perspectives_domains`.

While perfectly usable, I'd have to rename my model if I wanted to move it to a different provider, because I'd have tied my namespace to theirs (it is a subspace of theirs). That would be very impractical.

It so happens that I own the domain name `joopringelberg.nl`. Suppose I created a model named `model://joopringelberg.nl/JoopsModel` (notice the .nl part!), the PDR would map it to:

`https://joopringelberg.nl/models_joopringelberg_nl/joopringelberg_nl-JoopsModel.json`

My server does not host a Couchdb. However, I could redirect (See Cross Origin Resource Sharing: the domain joopringelberg.nl must be indicated by the server to be allowed for CORS) that to:

`https://perspectives.domains/models_joopringelberg_nl/joopringelberg_nl-JoopsModel.json`

The perspectives.domains server would then request the resource joopringelberg_nl-JoopsModel.json from the database `models_joopringelberg_nl`. All is well!

Can I have subspaces in my namespace? Yes:

`model://professional.joopringelberg.nl/JoopsModel`

maps to

`https://joopringelberg.nl/models_professional_joopringelberg_nl/professional_joopringelberg_nl-JoopsModel.json`

and is forwarded to:

`https://perspectives.domains/models_professional_joopringelberg_nl/professional_joopringelberg_nl-JoopsModel.json`

and leads the server to request the resource professional_joopringelberg_nl-JoopsModel.json from the database models_professional_joopringelberg_nl/joopringelberg/nl. Again, all is well.

The takeaway is that I could identify my models like this: `model://joopringelberg.nl/JoopsModel`. Identifiers like this would remain valid as long as I own the `joopringelberg.nl` domain, while I could switch storage providers at will.

8.2.3.4. Model versions

We want to introduce model versioning in Perspectives using semantic versioning. Version numbers defined according to this scheme are: MAJOR.MINOR.PATCH, where each of the three parts are non-negative integers, and MUST NOT contain leading zeroes..

Version numbers will be appended to model identifiers in such a way that

- They are accepted as part of the segment-nz-nc;
- They are accepted as part of couchdb document names.

The semantic version number as such (consisting of numbers and “.”) can be part of both. The “@” character can be, too, so we extend our definition of the model scheme to the following production:

```
Identifier = □//□ reg-name □/□ segment-nz-nc □@□ version core

version core = numeric_identifier "." numeric_identifier "." <numeric identifier>
```

(see semantic versioning for the production of <numeric identifier>).

The versioned version of the System model:

`model://perspectives.domains#System`

would be, for example:

`model://perspectives.domains#System@1.0.0`

and be mapped to the url

`https://perspectives.domains/models_perspectives_domains/perspectives_domains-System@1.0.0`

while the server would map this to the document `perspectives_domains-System@1.0.0` in the database models_perspectives_domains.

8.2.3.5. Pre-release versions

An author needs to maintain her model and this involves creating versions that are not accessible to the public. Yet, with the mapping from model identifier to storage location, we seem to have precluded this practice.

In order to restore it, we extend the semantic version with *pre-release information*. In short, we require model storage providers to use that information in the mapping of URLs to databases. See Mapping Model Identifiers to Storage Locations for details.

As a result, the identifier under the model scheme becomes:

```
Identifier = □//□ reg-name □/□ segment-nz-nc □@□ version core [ - <pre-release>]
```

An *optional* pre-release string may be appended to the model name, separated from it by a hyphen.

NOTE pre-release model version mapping to separate databases has not yet been implemented.

8.2.4. Model Manifest: a public context

A domeinfile is a resource that holds the machine readable version of a model. However, end users will want to inform themselves about a model before taking it into use. For this we introduce the convention of a *model manifest*. The manifest describing a model is itself a resource, a context instance to be precise. Its type is defined in `model://perspectives.domains#System`. The instance contains descriptive text, an expanded name, etcetera.

A model manifest should be accessible to everyone, or at least to everyone subscribing to a model repository (subscription may require a fee). A model manifest qualifies as a *public context*: it's type defines a Visitor role (see the text Universal Perspectives).

Crucially, a public context is the same for everybody (except for the model author): each participant has the same (consulting) perspective. This means that end users can share a single resource representation.

By convention, we will have a model manifest instance in a location that can be derived from the model URI. We have seen before that

`model://perspectives.domains#System`

maps to:

`https://perspectives.domains/models_perspectives_domains/perspectives_domains-System.json`

But we can also map it to:

`https://perspectives.domains/cw_perspectives_domains/perspectives_domains-System.json`

and at this location the model manifest instance is found.

8.2.4.1. Public context stores

A server that manages a models database for the domain X should therefore also manage a cw database for X, to store public instances in.

For example:

`https://perspectives.domains/cw_perspectives_domains/perspectives_domains-System.json`

looks for `perspectives_domains-System.json` (the ModelManifest for `model://perspectives.domains#System`) in the database `cw_perspectives_domains`.

Obviously, like we saw above, it may forward these URLs to another domain, if that is convenient.

NOTE the document needs updating below this point.

8.2.5. Type identifiers

Top level model types, like contexts and roles, have names that are scoped to model namespaces. This means that their name is prefixed with a namespace identifier. For example, the type PerspectivesSystem is identified by:

```
model://perspectives.domains#System$PerspectivesSystem
```

8.2.5.1. Type versioning

Peers send deltas to a Distributed Runtime, so the installation may update its instances. A delta contains type information. We have to accommodate the situation where a peer might have another version of the model containing the type, than the receiver. Therefore we need to version types, too:

```
model://perspectives.domains#System$PerspectivesSystem@1.0.0
```

Consider a domeinfile, representing a model at version 1.0.0. Now the author modifies the context PerspectivesSystem, but nothing else. This means that just the identifier of PerspectivesSystem changes: all other identifiers will retain their previous version.

```
model://perspectives.domains#System$PerspectivesSystem@1.1.0
```

Obviously, all references to PerspectivesSystem in the model will be updated, too (but this does not cause those referring types to have version 1.1.0, too).

This allows for quick checks when a delta comes in to create an instance. All deltas from a peer using model version 1.1.0 will be allowed, only a delta to create an instance of PerspectivesSystem will be reason for further analysis.

> Newer type versions may be downward compatible. For example, a context with an extra role is shape-compatible with instances without that role.

8.2.5.2. Type renaming

A frequent kind of change is when the author chooses a *new local name* for a type. For example, PerspectivesSystem might be renamed to PSystem. This is not a structural change and has no consequences whatever, in runtime. Obviously, it does have consequences in model time:

- Existing references to the name in the model text must be updated;
- Existing references to the name *in other model texts* must be updated, too.

However, type renaming does not cause an increase of the semantic version of a model. If there are other reasons to increase the version, renamed types retain their original version.

Nevertheless, instances refer to types by identifier. How can we make that work? How can a type identifier change, while existing instances do not change their reference?

This is because a reference to a type name is not by its visible name (the local name entered by the author, prefixed by namespace), but by a generated local name (prefixed by namespace). The domeinfile contains a table that maps the two to each other.

When a model is first parsed and saved, all local names are replaced by an integer. Integers start with 0 (for the root type, i.e. the namespace itself, the model identifier) and then increasing by 1 for every next type that the parser encounters. For example:

```
model://perspectives.domains#System$PerspectivesSystem@1.0.0
```

is referred to in the domeinfile and in instances with:

`model://perspectives.domains#System$0@1.0.0`

If the author modifies PerspectivesSystem to PSystem, he should provide an instruction to the parser:

```
context PSystem [renamed from PerspectivesSystem]
```

After a successful parse and save, he may (but need not) remove the instruction. The parser looks up the old name in its table and replaces it with the new name.

8.2.5.3. Handling backwards-incompatible changes in instances

Let's say that an author changes the type of a property from Boolean to Integer. Role instances that have a value for that property are no longer described by the new type. A property change like that needs to be followed by a change of the shape of the value in the instances.

We may construct a scheme of automatic repairs to be carried out on data on the occasion of such model changes. Lacking that, some changes can be carried out automatically to ensure proper functioning, but possibly to the cost of semantics. For example, every type can be mapped to a String. Booleans may be mapped to Numbers according to some scheme (e.g. 0 for false, 1 for true), etc.

It turns out that very few model changes do actually cause a problem with the shape of the instances (see [Model versions and compatibility]).

8.2.5.4. Imports

A model text imports dependencies in this way:

```
use sys for model://cw.perspectives.domains#System@1.1.0
```

Type names imported from another namespace will be replaced by using the name table of the corresponding domeinfile.

If the author updates the version of an import, the parser MAY compare the name table of the previously used version with the that of the new version, if the author provides an instruction:

```
use sys for model:cw.perspectives.domains#System#1.15.0 [up from 1.11.0]
```

Imported names must be fully qualified (either written in full, or with a prefix). Hence the parser can scan the model text for names that are replaced in the import (using the prefix, if applicable) and replace them automatically in the text.

The next parse is then guaranteed to be able to replace the each imported identifier by its number.

The model text may refer to types that have been dropped in the new version of the import. The parser MAY report these to the author.

8.2.6. Instance identifiers

NOTE text is up to date from this point on.

Instance identifiers identify context- and role instances. When two PDR installations exchange Transactions, we identify instances in them in terms of GUIDs (with one exception that will be explained below). However, within an installation, we use an extended format to store and look up resources. A *resource* is the technical term that encompasses both instances and *DomeinFiles*: the json documents that hold type definitions. This is because, not on the logical level of the language, but on the technical level of the implementation, all json documents that we want to persist are treated as members of the classes `Persistent` and `Cachable`.

We introduce the notion of a 'scheme' to identify resources locally. Schemes tell us where to find and store resources. End users have some influence on where resources are stored through the *local* and *remote* schemes, the former specifying a local database name and the latter a URL where the end user has facilities to store documents. However, DomeinFiles, with the *model* scheme, are stored in a local database and *public* schemed resources are stored in a location determined by their creator. Finally, *default* schemed resources are stored in a local database. See module Perspectives.ResourceIdentifiers for more details.

However, we allow the end user to associate each type in a model with a scheme. This is accomplished through a special model that provides the end user with a suitable user interface. Using this he can specify a mapping from types to storage schemes in as much detail as he prefers. Any type name functions as a namespace for syntactically embedded types. The system will use the most specific mapping for an instance. For example,

```
model://perspectives.domains/System$PerspectivesSystem$User
```

is more specific than

```
model://perspectives.domains/System$PerspectivesSystem
```

If a mapping has been specified for the latter but not for the former, instances of the former will be stored according to the mapping of the latter.

8.2.6.1. Default scheme

Identifiers following this scheme point to resources stored in the PDR’s default resource store. Currently, this is implemented as a store accessible through Pouchdb, named <userID>_entities.

8.2.6.2. Local scheme

The local scheme allows us to store resources in *another* database than the default resource store, albeit locally (i.e. through Pouchdb with a non-url database name).

8.2.6.3. Remote scheme

The remote scheme allows us to fetch and store resources through some REST interface. The URL represents the endpoint where we fetch and store resources. The resources themselves are stored using the <guid> part as key.

8.2.6.4. Public scheme

The public scheme is just like the remote scheme. However, is is used exclusively to publish resources in a public Umwelt, or in other words, according to a public-facing perspective. It is possible to fill a role with an identifier in the public scheme. Unlike the other schemes, the public scheme is preserved in deltas (the other identifiers are stripped to their guid parts, as the receivers themselves decide on where to store them). Note that the storage location of resources in the public: scheme is under control of the modeller. She can include the location verbatim in the model, or she can specify an expression that allows the end users of the model to set it. But this is different from the control that the end user has over storage in the local and remote schemes (the latter can be set per user, while the former is set once for all users).

8.2.6.5. Model scheme

The model scheme form is designed to be equal to the model URI. It was introduced to be able to handle DomeinFiles through the classes Persistent and Cacheable.

8.2.7. DomeinFile identifiers

A model is identified by a URI:

```
model URI = 'model:' '//' Authority '#' LocalModelName

Authority = dot-separated string parts

LocalModelName = a string compatible with Couchdb filename rules
```

We derive a name from that model URI to store a DomeinFile locally on a users PDR:

```
DomeinFileName = Authority_ '-' LocalModelName.json

Authority_ = Underscore-separated string parts
```

This is a name that is compatible with Couchdb file name rules. Why don’t we just use the LocalModelName? Well, that may identify the model uniquely within the repository for a given namespace, but not so locally, where the end user collects models from all kinds of domains. There may very well be a model called 'Automotive' in two separate domains:

- model://bigcars.com#Automotive
- model://hobbies.org#Automotive

Were we to store both under the name Automotive, this would cause no problem in the repository databases `models_bigcars_com` and `models_hobbies_org` on their respective servers, but it would cause a name conflict for an end user who wants to use both models because their filenames would both be `Automotive.json`.

Why do we replace "#" with "-"? Both characters are allowed in URNs and thus in URLs and even in Couchdb file names, but in URLs the hash sign has a special interpretation: it separates the resource (file) name from an anchor. Hence, when we ask for a resource like `https://perspectives.domains/models_perspectives_domains/perspectives_domains#System.json`, Couchdb is requested to provide `https://perspectives.domains/models_perspectives_domains/perspectives_domains` which does not exist.

We treat DomeinFiles as instances of classes Persistent and Cachable. The Persistent class requires its identifiers to be in the form specified as ResourceIdentifiers in module Perspectives.ResourceIdentifiers. These are strings that can be parsed into a DecomposedResourceIdentifier instance.

We achieve this for DomeinFiles by having a scheme 'model:' (along with loc:, def:, remote: and public:). This means that

- we can identify a DomeinFile with its full model URI w.r.t. Persistent and friends (in other words: the _id member contains the model URI);
- the actual files are stored with a name in the form specified above as DomeinFileName (but we never handle that name as such outside class Persistent and the module Perspectives.ResourceIdentifiers).

Couchdb particulars: we serialise the DomeinFile using generic encoding. This results in a JSON structure with a member "contents" and a member "tag". However, Couchdb expects the members "_id" and "_rev", too. Just before storing the file, we add both members to the serialization. Both members are also part of the "contents". There is the following relation between both:

- the inner "_rev" is always one behind the outer "_rev" (this is because Couchdb sets it on updating: Couchdb is in the lead. On retrieving the resource from Couchdb, we quickly set the inner "_rev" to the outers value);
- the inner "_id" is the model URI, the outer "_id" is the DomeinFileName as defined above.

The model: scheme is implemented in such a way that all models are stored in a local database with a name that is composed from the system identifier and the suffix "_models".

Class Cacheable will just store the DomeinFile under the full model URI.

To prevent misunderstandings: **we should never reach out to a model in a repository using classes Persistent and Cacheable**.

8.2.7.1. Looking up a type

Consequently, when we have to look up, say, a context type, we split its identifier at the '$' character. The left part is a model URI. This we can send to getPerspectEntiteit directly and we will receive a DomeinFile either from cache or from the database.

8.2.7.2. Saving a DomeinFile

Likewise, when we add a new model to our local set of copies, we can use saveEntiteit with the model URI to store it.

8.2.8. Public instance identification

By publishing a resource, we make available to the general public a specific perspective on that resource. Concretely, we associate the resource with an identifier in the RemoteScheme (see Scheme Based Design of Resource Identifiers), which allows any PDR to retrieve it from the internet.

NOTE We deliberately introduce an error here and that is that public resources should have an identifier in the RemoteScheme. It should not; it should be in the PublicScheme. This sections builds up to that conclusion.

While we publish both context- and role instances, an end user will start by visiting a public context. In this context, he is given a Visitor role (not necessarily by that name) that is defined as a calculated role. The result of the calculation should be sys:Me (the indexed instance of sys:PerspectivesSystem$User).

The modeller publishes a context using the following syntactical construct:

```
case CouchdbServer
  external
    -- The location of the CouchdbServer_.
    property Url (String)
    property Name (String)

  publish at extern >> Url for Visitor
    perspective on extern
      props (Url, Name) verbs (Consult)
```

The keywords `publish at` should be followed by an expression that yields a Url in runtime, as evaluated with respect to the current context.

NOTE The result of this expression **must** end on a forward slash! The compiler cannot check this; only a runtime error will reveal that the forward slash is missing.

On processing the ARC file, the PDR creates two roles with the given perspective:

- a calculated Visitor role: user Visitor = sys:Me
- an Enumerated role VisitorProxy, with a Calculated Property Url = extern >> Url

Actually, for `publish at <loc> for <Name>`, it creates both <Name> and <Name>Proxy.

Moreover, a context type with a `publish at` clause is marked as having a public facing perspective. Whenever some user creates an instance of such a context type, his PDR creates an instance of the proxy role in it automatically. The instance has no filler, but it functions as a 'ghost' user that the PDR will create deltas for.

Either the initiator communicates that instance to other users, or each peer re-creating that context instance also re-creates the VisitorProxy instance.

This does NOT mean that the user who created an instance of CouchdbServer, has specified that instances of this type are associated with the RemoteScheme, at the server located by Url. He may have stored his instance locally, in the default store, or any other place of his choosing. But it DOES mean that a *version of the CouchdbServer, as defined by the published perspective, is available at that location*.

It also means that any other resources that a Visitor has access to according to this perspective, are stored there. Or at least, versions that the Visitor is allowed to see.

It's almost as if there was a real user filling an instantiated VisitorProxy role in that context, whom the creator of the CouchdbServer instance synchronised with. As if the creator's PDR sent a Transaction with all deltas required to construct the resource versions, according to VisitorProxy's perspective. And as if VisitorProxy had ordained that resources of this type should be identified with the RemoteScheme and thus be saved to that particular location.

Actually, that is pretty much how it is implemented. But instead of there being another PDR receiving the Transactions for Visitor, we have the creator's PDR do it for him, as it were impersonating the VisitorProxy. Transactions for VisitorProxy are caught just prior to sending them and then interpreted locally, making sure that the resources are saved at the remote location.

8.2.8.1. Interpreting Transactions for VisitorProxy

MonadPerspectivesTransaction is defined as ReaderT (AVar Transaction) MonadPerspectives. Transaction has a member called typeSchemeMap and this contains associations between a resource type and a scheme for identifying its instances with. Almost always we have that member carry the local user's associations, but on interpreting a Transaction for the VisitorProxy role, we replace it with a scheme that maps any resource to RemoteScheme with a url value as calculated in VisitorProxy$Url.

8.2.8.2. Switching to the Visitor perspective

A user with access to a context that has a public facing perspective, may want to switch to the Visitor role so he can see things from that perspective. We actually have the user interface to do so. We can automatically detect that a context has such a perspective (remember we have marked the context type as such). This allows us to add GUI functionality to switch to the public perspective at once.

Note that all peers in roles that have another perspective, will actually store the context and its roles privately (wherever they have ordained).

Will these two versions not conflict? No, because resources are looked up **and cached** using their full (i.e. schemed) identifier. So a user might have the same context in his cache twice: once in a public version, once in his (the user's) private version. When looking at it from a public perspective, he will see different things (likely less than in his private role). But that is as it is meant.

8.2.8.3. Making known where a public context can be found

So far, we have only shown how those who have access anyway (in a non-visitor role) to a context with a public perspective can switch to the visitor role. How do we publish the existence (and location) of such a context? First, it should be really easy to obtain the url of such a context through the GUI. This allows us to use it as an ordinary link in an ordinary web page. Notice, however, that the plain url will not cause MyContexts to become active and capture it; to do so, we should add the url as an url argument to https://mycontexts.com.

How do we include a link to a public context on a page displaying a user’s perspective on another context? If the origin page is public, it is easy: the link identifier should automatically be in the RemoteScheme format. Note: we speak of 'link' but really it is a nested context role whose filler is a public context.

However if the origin page is private, by default the filler of the nested context role will be in a format that leads to the user’s local copy of it. This is a problem for enclosed contexts that should have lists of public contexts. This is rather common: think of the Repositories in MySystem.

We can create a role instance with a public filler in an (automatic) action because we can say:

```
bind publicrole <expression> to RoleX
```

where <expression> can yield an identifier in the remote scheme. We can create a perspective with a property that can be filled with the plain url of a public context and then we can have an action create a role with the value of that property as the identifier of the filler. So peers can create links to public contexts in enclosed contexts.

It can be easier than that. The author of an enclosed page might drop a role taken from a public context onto it, thereby creating a new role whose filler has the remote scheme.

8.2.8.4. No backlinks from a public role!

In order to be able to traverse a role filler relationship quickly, we not only keep a link from a role to its filler, but also to all the roles it fills. We call these *backlinks*. However, if a *public role is used as filler*, we register no backlinks on it. In most cases, the resource in the public location cannot be changed by the Visitor of that location. This would prohibit creating a private role filled with the public role.

8.2.8.5. When a Visitor adds a role to a public context

Let’s start out by noting that a Visitor should not have a perspective that allows her to modify the *public facing* perspective on a context. However, should there be an unlinked role in the public context, she might be able to add an instance to that. The scope of that instance would be restricted to her own umwelt. But it might be shared, too, with for example the contexts' Author. He would then keep the instance in his own umwelt (the scope of the instance would include both this single Visitor and the Author). Now think of the deltas the Visitor sends to the Author. It should include a ContextDelta. What shape does the identifier of the context take? Is it the full remote address, or just the guid? Since the Author keeps the instance in his own Umwelt, he gets to decide where to store it - so the delta should just have the guid. As a consequence, this situation is no exception to the general rule that says that deltas should be in terms of guids without their storage schemes.

8.2.8.6. Dropping a private role onto a public page

A Visitor might drop a role from a closed context onto the public context and then create an (unlinked) role instance that is filled by it. For example, he might sign up by dropping his Me role in the Volunteer role drop zone. This will create a Volunteer instance filled by a private role; but notice that this Volunteer instance is not linked to from the public context. There may be (indeed, must be for this example to be useful) another role that has a perspective on Volunteer, e.g. Organizer, but then the new Volunteers PDR will send a delta to him or her describing the new Volunteer role and its filler. The invariant is that there cannot be a role instance in a public facing perspective that is filled by a role from an enclosed context.

8.2.8.7. The case for the 'public' scheme

Above, I wrote that the author of an enclosed page might drop a role taken from a public context onto a private context, thereby creating a new role whose filler is public. If we adhere to the rule that a delta should be expressed entirely in terms of guids without their storage schemes, we run into a problem. The receiver of the delta may or may not have that filler role in his umwelt - and likely he has not, as the entire purpose of this excercise was to share a *public* resource.

The author’s PDR should create a binding delta whose filler slot contains an identifier in the remote scheme. But how does it distinguish this situation from that of a role filled by a private role stored in the remote scheme?

It cannot. Remember that, even though the filler role is available in a published perspective, it is not declared public itself in the model. The role **is** not public; it has **also been published**. So type reflection will not decide the issue.

Neither can it decide on grounds of the storage location. The Author may have that storage location in use to store his private copies of resources, or he may not. If he does not, he clearly intends to share a reference to a public resource. But if he has, we cannot know!

There seems but one solution and that is to encode in the filler identifier itself that the author intends to refer to the version of its referred role that is in the Visitor (public) Umwelt. We could do this with a new scheme: 'public:'.

It would be quite convenient if the role that was dropped was already identified in the public: scheme. But that seems to imply that **all** identifiers of resources in a public Umwelt should be in the public: scheme. And this is how we implement it. So, any resource that is to be available publicly, has an identifier in the public: scheme. All resources stored in a publicly accesible store on the internet have public: identifiers. Note, however, that for example the author of such a resource has stored it locally, for example with the default: scheme.

The public: scheme is just like the remote: scheme, but for the scheme name itself. There is only one situation in which public: identifiers are created, and that is when a Transaction for a Proxy role is executed.

The rule for deltas is then that all default:, local: and remote: schemed identifiers should be reduced to their guids, whereas the public: identifiers are not.

8.2.8.8. Overlapping public perspectives

In model://perspectives.domains#CouchdbManagement, both a Repository and a ModelManifest have a public role. In fact, the ModelManifest contextrole property `LocalModelName` falls in both public perspectives. Either perspective publishes to a different location. Where does the contextrole of ModelManifest end up?

Let's step back here and reflect for a moment on the meaning of public roles. A public role is one that can be taken on by any member of the public (this is not entirely true, as the model's author may require particular role instances to fill the public role - but the general thrust of the argument remains). As a consequence, the resources in the public perspective are available to the public - regardless of the exact url where they are published.

One approach to the problem posed above, therefore, is just to publish the contextrole of ModelManifest and its property `LocalModelName` to *both locations*. Consider for a moment however, what would happen if a visitor of Repository opens a ModelManifest instance. Doing so he assumes the ModelManifest$Visitor role, which gives him access to, a.o., the Versions role. But as Repository$Visitor does not have a perspective on that role, it is not represented at the particular url where Repository$Visitor has published. No Versions will turn up! In contrast, were the user to visit ModelManifest directly, he would find instances of Versions. Clearly this is not what we want.

For that reason we have adopted another strategy. We define, for each context, an optional public url. It will be available if there is a public user role in that context. Now, for each role instance, if its context does have a public url, we will publish it there. If it has no public url of itself (and we are indeed publishing) we use the public url of the user we are actually publishing for.

Let's consider some examples.

In case of the role ModelManifest$External (with property LocalModelName), we find that ModelManifest does have a public url - it is the url where its Visitor role publishes to. So, even if we're in the middle of publishing *for* Repository$Visitor, we'll publish ModelManifest$External to the url of ModelManifest$Visitor rather than to Repository$Visitor's url.

In case of, say, the System$User role, things are different. Both Repository$Visitor and ModelManifest$Visitor have a perspective on User - but System doesn't have a public role of its own. So, an instance of User ends up at both public locations.

8.2.9. The default repository

An MyContexts installation cannot function without the system model. Moreover, every installation needs access to certain basic models (such as model://perspectives.domains#BrokerServices). These models and their manifests are stored in databases on the server https://perspectives.domains. This model database is described by an instance of Repository that is identified by (and located in) https://perspectives.domains/cw_perspectives_domains/BaseRepository.json. Fetching the system model is hardwired into the PDR.

Part of the installation routine is to create an instance of `sys:PerspectivesSystem`. This instance is created complete with an instance of the role `sys:PerspectivesSystem$Repositories`, that is filled with this public repository.

> This role should be computed by fetching the instances of Repositories from the (local) database; but then this should be an Aspect role that can be reused in model:CouchdbManagement.

NOTE Document should be updated below this point.

8.2.10. Cross Origin Resource Sharing

The 'same origin policy' implies that a script is allowed to request resources just from the same domain it itself is served (see: https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS). The PDR is served from https://mycontexts.com. This would imply that the PDR could only request models (and other resources) from that same domain. It would preclude the repositories at arbitrary locations as described in this document.

However, a server may be configured such that it sends CORS headers. Couchdb supports such configuration. Part of that configuration is to declare a set of *origin domains*. In our case, that would be https://mycontexts.com.

Every hosting party that supplies a Couchdb server for repositories, should therefore configure CORS in the same way. As a consequence, PDR sources, served from https://mycontexts.com, are allowed to see resources served from such servers.

8.2.10.1. HTTPS and certificates

All domains should be approached using the https scheme. This holds for domains that redirect, too. So, in our example, the server that redirects from joopringelberg.nl should have a certificate for that domain.

8.2.10.2. Redirection: problematic

In paragraph Storage service providers we suggest that if a domain is hosted by party A, while the repository where models in that domain are stored is hosted by party B on another domain, party A *redirects* requests to party B's domain. However, CORS does not always allow this (see: https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS/Errors/CORSExternalRedirectNotAllowed).

This holds especially for so called 'pre-flight requests' (made with the OPTION verb). From MDN:

Not all browsers currently support following redirects after a preflighted request. (…) The CORS protocol originally required that behavior but was subsequently changed to no longer require it. However, not all browsers have implemented the change, and thus still exhibit the originally required behavior (https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS#simple_requests).

In contrast, redirect is always allowed on *simple requests.* The PDR requests models in a way that seems to satisfy the criteria for such simple requests, excepting that the content-type header is application/json (which is not allowed). Nevertheless, in Chrome (version 100.0.4896.88) no preflight request seems to be done.

The redirecting party should implement CORS for mycontexts.com, too.

We implement the PDR on the assumption that browsers allow redirection on our CORS requests.

An example redirection directive for Apache, for example to be used in the configuration file for VirtualHost joopringelberg.nl, where the actual hosting is done on the address perspectives.domains:

```
RedirectMatch permanent "^/models(.*)$" https://perspectives.domains/models$1
```

A similar effect (but without redirect HTTP status code) can be achieved by a reverse proxy:

```
ProxyPassMatch "^/models(.*)$" https://perspectives.domains/models$1
```

Observations

On the local version of perspectives.domains, we observe that

- the redirection fails because of the preflight problem with CORS (The preflight request cannot be observed in Chrome);
- the reverse proxy works, but only if the database is public (i.e. if no members or admins are defined.

Obviously, the PDR does not send credentials for the reverse proxy (mycontexts.com) with the request for the domain (perspectives.domains). So, while the reverse proxy works, no credentials are sent along with it.

While *retrieving* models without credentials might be ok, uploading models certainly needs credentials. This is a problem to be solved.

8.2.10.3. Rewriting

Instead of redirecting or proxying, we may *rewrite* an url. Continuing the example above for VirtualHost joopringelberg.nl:

```
  RewriteEngine On
  RewriteCond %{REQUEST_URI} ^/models(.*)$
  RewriteRule ^(/.*)$  https://perspectives.domains/$1  [R=204]
```

A url of the form https://joopringelberg.nl/models_joopringelberg_nl will now be rewritten to https://perspectives.domains/models_joopringelberg_nl, while other URLs in the same domain would be left untouched.

Now it turns out that Couchdb does not handle OPTIONS requests or at least not in the way we need it. Therefore, we handle all preflight requests here. We redirect an OPTIONS request to the default `index.html`, while stipulating that the response has the 204 (no content) code. We also add all the required headers and make sure that there are no doubles by first removing any existing values.

```
RewriteCond %{REQUEST_METHOD} ^(OPTIONS)$
RewriteCond %{REQUEST_URI} ^/models.*$
RewriteRule ^/.*$  index.html  [R=204]

Header unset Access-Control-Allow-Credentials
Header unset Access-Control-Allow-Methods
Header unset Access-Control-Allow-Origin
Header unset Access-Control-Allow-Headers

Header always set Access-Control-Allow-Credentials "true"
Header always set Access-Control-Allow-Methods "GET, PUT, POST, DELETE, OPTIONS"
Header always set Access-Control-Allow-Origin https://mycontexts.com
Header always set Access-Control-Allow-Headers "content-type"
```

This code rewrites an url with the OPTIONS verb, when its path starts with "models". Notice that we allow the origin https://mycontexts.com.

We now should adapt our rewriting for URLs whose path start with "models_", stipulating that it should not run for the OPTIONS verb:

```
RewriteEngine On
RewriteCond %{REQUEST_METHOD} !^(OPTIONS)$
RewriteCond %{REQUEST_URI} ^/models(.*)$
RewriteRule ^(/.*)$  https://perspectives.domains/$1  [R=204]
```

Again, this only works for databases that can be accessed without credentials. That will be enough for reading DomeinFiles (models) from public repositories.

8.3. Mapping Model Identifiers to Storage Locations

The storage service provider is free to choose its own scheme for mapping URLs in a models subdomain to a Couchdb database, as long as it obeys two restrictions:

- URLs with a path that starts with "models" should be mapped to a store that stores just models;
- URLS with a path that starts with "cw" should be mapped to a store that stores just instances.

An author needs to maintain her model and this involves creating versions that are not accessible to the public. Yet, with the mapping from model identifier to storage location, we seem to have precluded this practice.

We restore this possibility with an extra restriction on the mapping of model identifiers to storage locations, to be carried out by the providers of a storage service.

8.3.1. Pre-release information in semantic version numbers

Semantic versioning provides for pre-release information:

> A pre-release version MAY be denoted by appending a hyphen and a series of dot separated identifiers immediately following the patch version. Identifiers MUST comprise only ASCII alphanumerics and hyphens [0-9A-Za-z-]. Identifiers MUST NOT be empty. Numeric identifiers MUST NOT include leading zeroes. Pre-release versions have a lower precedence than the associated normal version. A pre-release version indicates that the version is unstable and might not satisfy the intended compatibility requirements as denoted by its associated normal version. Examples: 1.0.0-alpha, 1.0.0-alpha.1, 1.0.0-0.3.7, 1.0.0-x.7.z.92, 1.0.0-x-y-z.–.

8.3.2. Including pre-release information in the mapping

We map a model identifier to a URL using a simple scheme. The identifiers under the model scheme are:

```
Identifier = □//□ reg-name □/□ segment-nz-nc □@□ <version core> [ - <pre-release>]
```

We map them to URLs using the following rule:

```
model://\{host}/\{modelname}@\{semver}

=>

https://f(□models□, prerelease(\{semver})).\{host}/\{modelname}@\{semver}
```

where the function prerelease maps the semantic version string to its first alphanumeric string and f is a function from the string “models” and the result of prerelease.

What exactly the function f is, is to the discretion of the storage service provider. For example:

`https://models.perspect.it/System@1.1.0-alhpa.1`

may be mapped to:

`/alpha_models/System@1.1.0-alhpa.1`

or, just as good, to:

`/models_alpha/System@1.1.0-alhpa.1`

In effect, this gives the modeller control over numerous repositories to be used for pre-release versions.

In other words: from the pre-release string we take the first part (the part up to the first “.”). This we use to construct a database name: it is the database that holds the pre-release versions of models.

The storage provider may want to limit its clients in the number of databases they can create. It can require its clients to create these databases before using them (databases will not be created on-the-fly). As a consequence, the modeller must be conscientious in choosing and using <pre-release>.

Also, note that for these databases to be private, a modeller must use a pre-release name that is unique within the Couchdb.

8.4. Current context and current role

8.4.1. What is a query applied to?

The Perspectives Language consists partly of *expressions* used:

1. to create a CalculatedRole or CalculatedProperty;
2. as the condition of a bot rule (the left hand side of the rule);
3. to retrieve values for assignment expressions in the right hand side of the rule;
4. to retrieve values to be bound in a let-expression.

An expression traces a path through the network of roles and contexts, possibly ending in a property’s values. Obviously these paths must start somewhere:

- For a CalculatedRole, the path starts the context that we calculate the role for;
- For a CalculatedProperty, it starts with the role that we calculate the property for;
- For all expressions in a rule (condition and those that provide arguments to assignment operations) it starts with the context that the rule is executed in.

A note on let* expressions. Though they introduce a new scope for the modeller to bind names in to values, they are just expressions that continue the path.

8.4.2. Re-basing the path

Sometimes it is necessary to return, in the middle of a calculation, to the base of a calculation in order to retrieve some value from it. An example might be when we want to filter one role with a property value sitting on another. While in theory it should be possible to retrace the path back to the root, this is at least inconvenient.

For that reason we introduce the notion of *current context*. The current context is, for each expression, the base context with respect to which it is calculated (as given above). The modeller can start a subexpression with the keyword currentcontext.

Similarly, he can use the keyword currentrole in the definition of a CalculatedProperty.

8.5. Indexed Names

8.5.1. Introduction

Indexing is about instances. With an indexed role, we provide a *universal name for an instance*. However, that name resolves to a *different instance for each user*. Indexing is a mechanism restricted to functional roles (which of the instances of a non-functional role would the indexed name resolve to?

We create indexed roles and contexts with assignment statements. If a role is functional and declared to be indexed, it is automatically created as an indexed role instance.

We have two use cases for Indexed Names:

- modellers may use them in queries (e.g. to retrieve a subset of one's hobbies)
- end users may use them to navigate.

In both cases, we want the Indexed Name to be effectively replaced by a unique name, so the unique (indexed) resource can be retrieved. We will focus on the use in queries. We can extract three requirements:

1. We have to be able to recognize a name as 'indexed' on parsing a query;
2. We have to be able to look up the type of the resource identified by the indexed name, because, on compiling a query, we make a description of the composition of a series of functions;
3. We have to actually produce a function as the compilation result of the querystep, that, indeed, looks up the indexed name and comes up with the name of a unique resource.

8.5.1.1. Uniqueness

An indexed name is qualified with a model name. Within a model, the local part of such a name must be unique. Model names must be unique, too.

> In order to ensure this uniqueness, model names must be unique, too. This is realised by using the Domain Name System. See: Type and Resource Identifiers.

8.5.1.2. Private-ness

An indexed context or role is not a *private* context or role, where we mean with 'private' that only one user has a perspective on it

> Note that perspectives are on *roles*, rather than contexts. Here we use the term loosely, actually meaning a perspective on the *external role of the context*.

Actually, we have to be careful when speaking about perspectives in relation to *instances*. Imagine a context type with a single, functional user role. We give that role a perspective on the context. Now, any instance of this context type will be private – at least if we do not give a perspective on it to user roles in *other* contexts. It is private, precisely because the user role having the perspective is functional. Were it not, there could be *two* (or more) users in the same context and it would no longer be private. So we reframe our definition of ‘private’ to: only one *functional user role* has a perspective on it.

As a matter of fact, an indexed role or context should be indexed in the first place precisely because it is *not* private. Were it private, we could use a universal name for the instance. Each user would have his own instance, never sharing it with anyone else – but all instances would have the same name.

So here are the design rules:

- If we like to use a universal name for a role instance, as a query step, that should resolve to a different instance for each user, make the role type indexed;
- If the role is private (just a single, functional user role has a perspective on it) it need not be indexed.

In both cases, construct the instance in design time, with the model.

8.5.2. Modelling indexed names

To begin with, we extend the language with another keyword: indexed. We use it in the definition of Contexts and Roles:

```
case PerspectivesSystem
  indexed sys:MySystem
  ...
  user User (mandatory, functional)
    indexed sys:Me
```

The keyword is followed by the name we intend to be indexed. This name must be qualified with the namespace of the model that is being defined. So here we would have sys:Me, or, expanded, model:System$Me.

8.5.2.1. Representation

We extend the data type Context that we represent types of contexts with, with another member: indexedContext. It’s type will be Maybe ContextInstance. Here we store the indexed name itself, e.g. ContextInstance model:System$MySystem. Context types that are not indexed will have Nothing as value for this new member.

Similarly, we extend the data type EnumeratedRoleType with a member indexedRole. Note that Calculated Roles cannot be indexed.

8.5.2.2. Two useful functions

We construct (in module Perspectives.Instances.Indexed) two functions that will extract all indexed contexts and roles from a model file:

```
indexedContexts :: DomeinFile -> Object Context

indexedRoles :: DomeinFile -> Object EnumeratedRoleType
```

We use these functions on compiling models and queries. Implementation is straightforward: e.g. run through all EnumeratedRoleType definitions and discard the ones that have Nothing for their indexedRole member.

Notice how we provide two bits of information in these function results:

- The keys are the indexed names themselves, as they are stored in the members indexedContext and indexedRole;
- The values are the *types* of the indexed resources.

8.5.2.3. Usage in queries

An indexed name, let’s say sys:Me, may occur in a query. It can function as a complete query in itself. When used, it should be properly qualified, so either with a valid prefix defined in the model for the right namespace, or as an expanded name. Semantically, it should be thought of as a query producing a singleton result.

Because an indexed name in a query is always qualified, from the text we know exactly in which model it is defined.

8.5.3. Compiling a query with an indexed name

As we encounter qualified names in queries, we have to establish whether they represent a type, or an indexed individual. We cannot say from the name only. So we find the model the name is qualified with, apply the functions indexedContexts and indexedRoles defined above, and look the qualified name up in the tables produced by both (function compileSimpleStep, case ArcIdentifier, of Perspectives.Query.DescriptionCompiler).

If the name is, indeed, indexed, we produce a query step that describes a function that will, in runtime, look up the indexed name. Note we cannot do that in compile time! There must be, by definition, a different result for each user.

However, the description of this step must contain the type of its result. So we must know what type of role or context this indexed name represents. This is exactly what is returned by the two functions above.

8.5.4. Starting to use a model

When a user first starts with a model, its indexed contexts and roles must be created. The modeller must provide the adequate assignment statements that are executed automatically when the model context itself is created. Below is an example:

```
domain model://perspectives.domains/TestFields
  use sys for model:System
  use tf for model:TestFields

  state ReadyToInstall = exists sys:PerspectivesSystem$Installer
    on entry
      do for sys:PerspectivesSystem$Installer
        letA
          -- We must first create the context and then later bind it.
          -- If we try to create and bind it in a single statement,
          -- we find that the Installer can just create RootContexts
          -- as they are the allowed binding of IndexedContexts.
          -- As a consequence, no context is created.
          app <- create context TestFieldsApp
        in
          -- Being a RootContext, too, Installer can fill a new instance
          -- of IndexedContexts with it.
          bind app >> extern to IndexedContexts in sys:MySystem
          Name = "TestFields Management" for app >> extern

  aspect user sys:PerspectivesSystem$Installer

  -- The entry point (the `application`), available as tf:TheTestFields.
  case TestFieldsApp
    indexed tf:TheTestFields
    aspect sys:RootContext
```

The pattern followed here is that we add an aspect user to the model: `sys:PerspectivesSystem$Installer`. This user, defined in `model://perspectives.domains/System`, first creates an instance of the root context of the model (in this case: TestFieldsApp) and then fills a new instance of the role IndexedContexts in the system context of the user. It’s perspective allows it to do so:

```
domain System
  ...
  case PerspectivesSystem
    ...
    user Installer
      perspective on IndexedContexts
        only (CreateAndFill)
    ...
    context IndexedContexts (relational) filledBy sys:RootContext
```

Notice that IndexedContexts can only be filled by instances of sys:RootContext. For that reason, we give TestFieldsApp the aspect `sys:RootContext`: it allows Installer to fill an instance with the TestFieldsApp instance. Because TestFieldsApp is declared to be indexed, this instance is added to the administration of the users indexed contexts.

Notice, too, that, in effect, there are no restrictions on creating context type instances. Perspectives only apply to roles. But creating a context instance that is not connected to the web of roles and contexts is pretty useless, so in the end the perspective restrictions practically apply to contexts as well.

8.5.5. Looking up indexed names in runtime

As stated at the start of the text, we have two use cases for Indexed Names:

- modellers may use them in queries (e.g. to retrieve a subset of one’s hobbies)
- end users may use them to navigate.

Let’s again focus on the first use case. When the PDR starts, it must build a translation table. This table is used by the functions that our querysteps are compiled into. So how do we build that table?

Notice that there is nothing special about the representation of an Indexed context or Indexed role. They are not even singletons, even though there is just one instance for each user. For example, sys:Me represents an instance of sys:PerspectivesModel$User, but we have many of them in a PDR installation; one for each peer and the indexed one. But which is which?

WARNING This section is not up to date. The implemenation is currently in flux with regard to indexed names.

We solve that problem by requiring the modeller to list the indexed resources in the model description. We enable him to do so by introducing two roles in sys:Model:

```
context: IndexedContext filledBy: sys:NamedContext

thing: IndexedRole
```

Notice how we have put no restriction on the binding of IndexedRole. Similarly, sys:NamedContext is a very lightweight context type, merely requiring a name.

On starting the PDR, we run two queries on Couchdb that produce all instances of these two roles. This gives us all the information needed for our translation table:

- the actual resource identifier is the unique value for this user;
- from the resource we retrieve its type and from the type we retrieve the indexed name itself.

From this we build a table with indexed names as keys and unique identifiers as values.

8.5.6. How to write the right instances

The success of this system depends critically on writing the right instances in the CRL file that accompanies the model file.

We require two things:

1. For each context type that is declared indexed, we require a binding for the role sys:Model$IndexedContext in the model instance in the CRL file.
2. For each role type that is declared indexed, we require a binding for the role sys:Model$IndexedRole in the model instance in the CRL file.

Notice that the *type* of the model instance in the CRL file is a unique specialisation (through the use of aspect model:System$Model) for each model.

8.5.6.1. IndexedContext

We furthermore require the following for the binding of IndexedContext:

1. The name of the context instance we bind to, must be the indexed name! So, for example, in model:System we would have the name of this context instance to be sys:MySystem; for model:SimpleChat it would be chat:MyChats (the prefixed name is allowed).
2. The role must have a value for the property IndexedContext$Name. That name must be, again, the indexed name – however, **without the** model: **part**. So we would have SimpleChat$MyChats and System$MySystem respectively.

> The reason for this is rather arcane: we do textual search and replace on the string value of the crl file, replacing all occurrences of the indexed names. However, here we need a hook back from the indexed name to its replacement. By omitting the “model:” part, we prevent replacement of the property value.

8.5.6.2. IndexedRole

The same requirements hold for IndexedRole:

1. The name of the role instance we bind to, must be the indexed name! So, for example, in model:System we would have the name of this role instance to be sys:Me.
2. The role must have a value for the property IndexedRole$Name. That name must be, again, the indexed name – however, **without the** model: **part**. So we would have System$Merespectively.

8.5.6.3. Example

The relevant fragment of the CRL file for model:SimpleChat is this.

chat:Model usr:SimpleChatModel

…

sys:Model$IndexedContext ⇒

chat:ChatApp chat:MyChats

…

sys:Model$IndexedContext$Name = "SimpleChat$MyChats"

Notice

- The specialised type chat:Model;
- The indexed context name chat:MyChats;
- The value of the property IndexedContext$Name: it is the (expanded) indexed name without the “model:” part: SimpleChat$MyChats.

8.6. Standard variables: technical design decisions

The modeller can use a number of standard variables in his expressions. The semantics of these variables is given in the text Understanding a Perspective source text, but here is a summary:

- `origin` can be used in every expression. It is the role or context instance the expression is applied to.
- `currentcontext` can be used in every expression. It is the lexical current context of the expression.
- `currentactor` can be used in the scope of do and action. It is the lexical current subject.
- `notifieduser` can be used in the scope of notify. It is the lexical current subject.

In this text I explain how these variables are implemented.

8.6.1. Lexical concepts in the state of the parser

The parsers keep current context, current subject and current object in state. This means that any parser can retrieve a representation from state that represents these lexical concepts for the lexical position it starts to work on (if they are defined at all, of course, at that position).

8.6.1.1. Current context

The current context is represented in ArcParserState (module Perspectives.Parsing.Arc.IndentParser) as the member `currentcontext`. Every parser can extract the lexical current context from state. For our purposes we need to know that this current context is part of the representation of current subject, current object and current state. The representation is a fully qualified type.

8.6.1.2. Current subject

In the body of the following constructs, a current subject is defined:

- the body of a perspective of <user>;
- the body of a user role definition;
- the block of assignments following do for <user>, defining an action
- the expression following notify <user>.

In ArcParserState, the current subject is represented by a RoleIdentification:

```
data RoleIdentification =
ExplicitRole ContextType RoleType ArcPosition |
ImplicitRole ContextType Step
```

The ContextType part represents the current context (as stated in the previous paragraph). For ExplicitRole, this means that RoleType is defined in the context of the ContextType. In an ImplicitRole, the Step is the parse tree of an expression. It should be evaluated with respect to its ContextType: we interpret an expression in its current context.

It turns out that in all of the situations above, an ExplicitRole is constructed.

> In fact, as shown below, an ImplicitRole is only ever constructed on parsing a perspective on clause.

In all cases *except for* perspective of <user>, the RoleType part is a fully qualified name (we qualify this case in PhaseThree of the parser, when all role types of the model are available).

ArcParserState holds a member subject that represents the lexical current subject.

A note on calculated roles. The role type in ExplicitRole can be calculated (except when it is the external role) and will then be a CalculatedRoleType.

8.6.1.3. Current object

The current object is in scope in:

- the block following a perspective on;
- the body of a non-user role definition.

As with current subject, current object is represented in the parse tree by a RoleIdentification. A role definition results in an ExplicitRole with a fully qualified name, but a perspective on results in an ImplicitRole.

ArcParserState holds a member object that represents the lexical current object.

8.6.2. Lexcial concepts in parse tree elements

The lexical concepts relate to the parse tree in a straightforward way. We discuss the case of AutomaticEffectE, the lexical representation of do (module Perspectives.Parsing.Arc.AST).

8.6.2.1. current subject and current object

The newtype AutomaticEffectE has members subject and object. The former is a RoleIdentification; the latter a Maybe RoleIdentification (not every do has a current object in scope). The RoleIdentification that is subject is precisely the value of the member subject of ArcParserState as it was available to the parser handling the body of the do. In other words, it represents the current subject in the scope of the do.

Mutatis mutandis, the same holds for object.

8.6.2.2. Current context

The AutomaticEffectE is constructed with a member transition whose value is a StateTransitionE:

```
data StateTransitionE = Entry StateSpecification | Exit StateSpecification

A StateSpecification is:

data StateSpecification =

ContextState ContextType (Maybe SegmentedPath)

| SubjectState RoleIdentification (Maybe SegmentedPath)

| ObjectState RoleIdentification (Maybe SegmentedPath)
```

Let’s dissect the three data constructors. For a ContextState, the ContextType represents the current context from lexical analysis.

For a SubjectState and ObjectState, the RoleIdentification contains (as we’ve seen above) a part ContextType, that represents the current context.

How do we know? The parsers keep members onEntry and onExit in ArcParserState. These end up in AutomaticEffectE. They are built from `currentcontext`, subject and object in ArcParserState: meaning they are constructed from the representation of the corresponding lexical concepts for the body of the do.

8.6.2.3. Other lexical representations

We’ve discussed AutomaticEffectE above in detail. It happens that the following lexical representations exhibit the same pattern of a subject, object and transition, or, in some cases, a state member:

- RoleVerbE
- PropertyVerbE
- SelfOnly
- ActionE
- NotificationE
- AutomaticEffectE

The first three represent various qualifications of a perspective. The latter three have to do with state transition, assignments and notification.

8.6.3. Standard variables: the lexical concepts in QueryFunctionDescriptions

Expressions are parsed to Step and Step is compiled to a QueryFunctionDescription. Runtime, a

QueryFunctionDescription is compiled to an executable function. We've seen above how the lexical concepts current context, current subject and current object are represented in parse tree elements. How do these representations relate to standard variables?

8.6.3.1. The expression compiler

First, a quick recap on QueryFunctionDescription. These represent functions, with a domain and range and some information on functionality and being mandatory, along with a representation of the actual function to compute. The expression compiler (module Perspectives.Query.ExpressionCompiler) uses these representations to reason about the correctness of the function expressions entered by the modeller (it checks, among others, whether properties that are used do in fact exist on the roles they are supposed to be on, whether both sides of (in)equalities have the same type, etcetera).

8.6.3.2. Inserting standard variables into expressions

Standard variables are just like the variables that the modeller himself can introduce, in a letE or letA construct, but he need not bind them himself. Consequently, expressions will only have *references* to those variables but never their *bindings*. This would cause the expression compiler to throw an error. To prevent these errors, we *add bindings automatically for the standard variables to the parse tree of expressions*. This allows the expression compiler to reason about their use as usual.

These bindings are added to existing letE and letA constructs, or an expression or statement is wrapped in a letE to hold the new bindings. We don't actually always add all standard variables to expressions: the code analyses the expression to see what standard variables appear.

By modifying the expression itself, we can evaluate it in the standard way in any location in the code of the core. We guarantee that each expression is self-contained: the core code does not have to add standard variables on computing the value of an expression.

This is very convenient, as calculated roles and properties can be thought of as named functions that are, indeed, called by name from other queries. We can therefore freely compose such functions without the need to add bindings to the runtime environment.

8.6.3.3. Treatment of statements

Statements are treated differently. We do always add standard variables to statements (single statements or sequences of them, or a letA construct)(This happens in module Perspectives.Parsing.Arc.PhaseThree). This is because when we execute an automatic action or a notification, the core code itself needs the value of these variables. The `origin` is available anyway, because it is the argument that the compiled expression function is applied to. But in order to distribute delta's for the changes incurred by an automatic action, we need to know the current actor. And to compute the current actor, we need to have the current context – because the actor is computed relative to the current context. So, before actually executing statements, we have the value of all three standard variables available.

It would therefore be a waste of resources to recompute them with the statements while it introduces almost no overhead to add the already computed values to the runtime environment that holds values for our variables.

We handle this in runtime as follows, in the execution machine (modules Perspectives.RoleStateCompiler and Perspectives.ContextStateCompiler):

1. we let the execution machine push a runtime environment (the data construct that we store variable bindings in);
2. we then add the values of the standard variables to that environment.

But what about the bindings added to the statements? Well, in the very last step that creates an executable function (in module Perspectives.Query.UnsafeCompiler), we remove the bindings for the standard variables. The executable functions will never compute their values, nor add them to the runtime environment. A variable reference will just pick up the value added by the runtime execution machine.

> You may have noted that an extra environment is pushed by the compiled statement (we just remove the variable bindings). This is unnecessary, but has no effect: variable lookup scours the entire environment stack, so the empty environment is just passed (adding very little overhead).

8.6.3.4. Expressions in statements

We just add bindings to entire statement groups, not to each and every individual expression in them. This is because each expression is applied to the same `origin` and because there are no syntactical constructs within statements that change the current context, subject or object. Neither can statement groups be nested inside other statement groups (do, action and notify do not contain lexical constructs that introduce scopes). Hence a single set of bindings suffices.

8.6.3.5. Simple treatment of compile time only bindings

As the values of standard variables are never computed with the assignments that they occur in, we do not actually have to construct a full function to compute it. Just having a QueryFunctionDescription with the right domain, range, functionality and being mandatory information, will do: it is all the expression compiler needs to do its checks. For that reason we introduce a QueryFunction that can be considered a no-op. It is actually this QueryFunction that makes the unsafe compiler remove a binding.

Consider the computation of the current context from the `origin`, for a do. Now when we have context state, it is trivial: just the identity function. But when we have role state, we take the current context value from the RoleIdentification and put that into this parse tree construct:

```
Simple $ TypeTimeOnlyContext pos <currentcontext>
```

(where <currentcontext> is the name of the context type that is the current context) The expressioncompiler will turn it into:

```
SQD currentDomain (QF.TypeTimeOnlyContextF <currentcontext>) (CDOM (ST $
<currentcontext>)) True True
```

This can be used to reason with: whenever `currentcontext` occurs in the statement, the compile knows its runtime type.

8.6.3.6. Overview: standard variable computing in runtime

What variable should be computed when? With both context- and role situations, actions, automatic actions and notifications, we have a lot of cases. We do not always have to compute a value:

- `origin` never needs to be computed runtime;
- `currentcontext` must be computed for roles;
- `currentactor` (or `notifieduser`) must always be computed.

8.6.4. Actually computing the current context

The execution machine must compute the value of the current context. This is trivial in many cases, as expressions are quite often applied to contexts. However, in a do (or notify or action) things can get difficult if they hold for role state. Remember that the expressions in the body of such a do will be applied to a role instance.

Isn't the current context just the context of that role instance? That may be the case, but it is not so if the object- or subject state is for a *calculated role*. The current context should be the lexical context, i.e. the context that one can verify, in the source code, to contain the do expression somewhere in its body. But the `origin` that the expression function is applied to, may be a role in an entirely different context. That is the usefulness of calculated roles!

Let's consider the case of object state. The state specification will have a RoleIdentification that represents the `origin`. Consider the case of an ImplicitRole: the RoleIdentification contains a context type that represents the current context, and a Step that represents the expression that computes the role from that context.

To find the context from the role, we just have to reverse the expression. This we can do; we have machinery to invert QueryFunctionDescriptions (for triggering state changes and query updates, for example). So we compute a QueryFunctionDescription for the Step with respect to the current context type, and reverse it.

The execution machine uses that inverted function to compute the context from the role instance that changes state.

8.6.5. Future extensions: currentobjects, currentsubjects

It seems possible to add yet more standard variables. A do expression might be in scope of both a current subject and a current object, but the state transition it is contained in will be just one of them (or, of course, it will be on a context type transition).

So, for example, an expression in a do in subject state cannot refer to the current object. Yet, as we can compute the current context, we can compute the object from it.

However, if the object role is relational, there may be more than one instance available. This causes a curious phenomenon: when an expression occurs in the body of a do in object state, it can refer to *one* object as `origin`, and to *all* objects as currentobjects.

8.7. Unlinked Roles

Role- and context instances are represented in the PDR by mutually referring data structures. A reference consists of an identifier that we use to retrieve the data structure from the database.

For a non-functional role, a context holds an array of references: one for each of its instances. The number of instances may grow very large for some types of roles. This may make the memory cost of caching their context prohibitive.

8.7.1. Retrieve with a query instead of by identifier

We offer the modeller a ‘compiler instruction’ to use with a role definition: the keyword unlinked. It can be used as follows:

```
context Chats (unlinked) filledBy Chat
```

(taken from model:SimpleChat).

The role Chats will not figure in instances of its context, ChatApp. Instead, when query evaluation proceeds with the step Chats, as in:

```
user: Chatter (mandatory, functional) filledBy: sys:PerspectivesSystem$User
  perspective on Chats >> binding >> context >> Initiator
```

the PDR will perform a query on the database for all role instances of the type `model:SimpleChat$ChatApp$Chats`, whose context is the context the query tries to get the Chats from.

Semantically, there is no difference between linked roles (the default) and unlinked roles.

8.7.1.1. Reversing over an unlinked role

Interestingly, we outfit the role instances with a direct reference to their context – just like instances of linked roles. This means that when the query evaluator encounters a context step, it handles both cases in the same way.

8.7.1.2. Deleting unlinked role instances

On deleting an unlinked role instance, we remove it from the database, just like with an instance of a linked role. However, there is no need to remove its reference from its context.

8.7.2. Retrieving role instances by Aspect name

When a role has been declared to have an Aspect, we say that it has that Aspect as its type, too:

```
case CouchdbServer
  aspect acc:Body
  ...
  user Admin filledBy CouchdbManagementApp$Manager
    aspect acc:Body$Admin
```

An instance of `model:CouchdbManagement$CouchdbServer$Admin` is an instance of `model:BodiesWithAccounts$Body$Admin`, too.

We can use, in a query, the step `Body$Admin` to retrieve role instances from an instance of context type CouchdbServer. This is important because it allows us, for example, to create an automatic action in the Aspect that is carried out on an instance.

But how do we (technically) retrieve such an instance using the Aspect role name?

For ordinary (linked) roles, we keep, in the context instance, a table of *aliases*: this table maps each aspect type to the role that uses the aspect. So, in an instance of CouchdbServer we would be able to look up `model:BodiesWithAccounts$Body$Admin` and get `model:CouchdbManagement$CouchdbServer$Admin`: we then lookup the role instance in the context instance using the latter name.

8.7.2.1. Unlinked roles pose a challenge

However, as we do not record unlinked role instances in their context instance, we have no aliases, either. Moreover, we look up unlinked role instances in the database, using a double key consisting of

- the context instance identifier
- the role type identifier.

We send that information to a Couchdb View that constructs, for each document in the instances database, a similar pair. Only documents with matching pairs are returned (this might seem a computing intensive process, but Couchdb prepares and caches views in the form of a B-tree that allows for quick lookup and easy maintenance in the face of new documents).

Obviously, we cannot retrieve an unlinked role instance using an Aspect name.

To repair this situation, we have extended the view. Each document produces not only the pair given above, but also pairs for each type that the role instance has: we record all (aspect) types in an instance. This is an optimization: we could just as well use type reflection, but that would be a lot slower.

So, while this process produces too many key pairs (we get pairs for non-ground states too), we at least get a pair for each of an instances types and hence we can retrieve an unlinked role instance by one of its Aspect type names, too.

Chapter 9. Models

9.1. Booting models

A Perspectives Distributed Runtime installation – e.g. as included in MyContexts – needs Perspectives Models to operate. The basic requirement is `model://perspectives.domains/System`. This model is retrieved automatically from the website perspectives.domains while installing MyContexts. However, the end user will want to acquire more perspectives and thus needs to add models.

This text explains how this is achieved in the PDR.

9.1.1. Models

Conceptually, a model is a collection of types. Physically, it is a file (the *domeinfile*). Models reside physically in a repository on a server on the internet. A PDR installation (such as MyContexts) keeps a private collection of models, usually locally.

End users want to extend their perspectives and therefore need to be able to explore models published by authors. To this end, we have the context type Manifest. Instances of this type are public contexts.

End users also need an overview of locally available models. These are currently collected in the end user's instance of the sys:PerspectivesSystem context. This is an *indexed context*, meaning the user can access it by entering its name in the address bar of the browser.

> Its full name would be model://perspectives.domains/MySystem. However, MyContexts applies fuzzy matching to names, so MyModels stands a good chance to land you in the right place.

But the MySystem context can be accessed from the menubar of MyContexts, anyway. It is accessible through the Home icon.

An end user whose installation does not yet have this model, will acquire it automatically upon visiting the description. (S)he then can boot the model by executing an action on the screen.

9.1.1.1. Booting a model

Using `model://perspectives.domains/Models` as an example, we'll show the boot sequence that applies for any model. The purpose of booting is to instantiate *indexed contexts* and *indexed roles* of the model and register them with the system and to add the model description to the collection in MyModels.

1. The first step is to create an instance of the context type ModelsOverview. The system will create a unique identifier for it.
2. As this type is indexed, we then create a new instance of the role IndexedContexts in sys:MySystem (the indexed context of the system model). We fill it with the context instance we

created in the first step.

3. On this role we add a value to the property Name. This value is precisely the indexed name itself: `model://perspectives.domains/MyModels`. This construction enables the PDR to maintain a table of indexed names and their local (private) identifiers.
4. Finally, we add the model's description to the list of such descriptions in MyModels.

So, in the case of `model://perspectives.domains/Models`, we end up with an instance of MyModels that has just one entry: the description of its own model.

These steps are implemented *in* `model://perspectives.domains/Models` itself, in the form of an Action in the perspective of the Visitor role in the model description.

Acquiring `model://perspectives.domains/Models` is different from other models in one respect: the type of the model description is contained in this model itself. That is the reason that this model is added automatically to the local installation upon visiting its description. This is the *only* model for which this is the case. In all other models, the model is only loaded on booting.

9.1.1.2. Booting System

As stated above, model://perspectives.domains/System is a requirement for a PDR installation to operate. We cannot use a client such as MyContexts to acquire this model. Hence, on installing MyContexts, System is installed by the code that runs MyContexts. At that time, we do not have an instance of MyModels; indeed we do not even have `model://perspectives.domains/Models`! As a consequence, System is not listed in MyModels.

System differs from other models in another respect. It's indexed names (MySystem and Me) are hard-coded and they are not listed in MySystem.

9.1.2. Authoring a model: booting revisited

The author of a model must include the Action that allows the end user to boot it. This action is fairly standard, but its details need to be written again for each model. Here is the code for model:perspectives.domains/SimpleChat.

```
perspective of Visitor
  action Boot
    -- Create the indexed context:
    createContext cht:ChatApp bound to IndexedContext in sys:MySystem

    -- Add the indexed name:
    Name = "model://perspect.it/SimpleChat$MyChats"
      for sys:MySystem >> filter IndexedContext
        with binding >> context >> contextType == cht:ChatApp

    -- Add the model description to MyModels:
    bind https://cw.perspect.it/SimpleChat to LocalModels
      in mod:MyModels
```

Obviously, the value `model://perspect.it/SimpleChat$MyChats` should be customised in each model.

9.1.3. Booting the Perspectives Universe

A special case of booting has happened during development of Perspectives. We describe it here just in case it has to happen again.

We would like a fresh MyContexts installation to have a list of elementary models that the end user can add to her installation. In other words, we want the repository for the namespace `model://perspectives.domains` to be available. We could try to include the corresponding Repository description by adding this line to the boot sequence instructions:

```
bind publicrole
pub:https://perspectives.domains/cw_servers_and_repositories/#perspectives_domains$Ext
ernal to BaseRepository in sys:MySystem
```

However, to compile `model://perspectives.domains#System` with this line, the compiler needs to be able to look up the type `model://perspecives.domains/CouchdbManagement` locally - that is in an installation that already has this model. This introduces a catch-22 because this latter model requires the former - hence, we have introduced a circular dependency even though only one model (CouchdbManagement) refers to the other.

We've taken the following approach to avoid this problem:

- compile `model://perspectives.domains#System` with this line commented out;
- set up an installation with this version of System;
- compile `model://perspectives.domains#CouchdbManagement` with that installation;
- uncomment the line;
- recompile `model://perspectives.domains#System` and add it to the Repository

Now, each next installation will have a reference to the base repository (it can be found at https://perspectives.domains/cw_servers_and_repositories/perspectives_domains)

9.1.3.1. Compiling the System model

How do we compile a model *before* we have a working installation? The developer of the PDR can do so, using a testscript. Here is an example:

```
  test "upload model://perspectives.domains#System to repository from files" $ runP do
    -- Add a password for "https://localhost:6984/"
    modify \s@({couchdbCredentials}) -> s {couchdbCredentials = insert
"https://localhost:6984/" "geheim" couchdbCredentials}
    addAllExternalFunctions
    _ <- loadCompileAndCacheArcFile' "couchdb" modelDirectory
    _ <- loadCompileAndCacheArcFile' "serialise" modelDirectory
    _ <- loadCompileAndCacheArcFile' "sensor" modelDirectory
    _ <- loadCompileAndCacheArcFile' "utilities" modelDirectory
    errs <- loadCompileAndCacheArcFile' "perspectivesSysteem" modelDirectory
    dCache <- domeinCache
    if null errs
      then uploadToRepository (DomeinFileId "model://perspectives.domains#System")
      else liftAff $ assert ("There are instance- or model errors for
model://perspectives.domains#System: " <> show errs) false
```

Now notice the call to `uploadToRepository`. It infers the location of a database on the internet from the model identifier. It will be: `https://perspectives.domains/models_perspectives_domains/`. This database **must be available** for the script to run! That is, Couchdb must be installed on the server that hosts perspectives.domains and the database (and its read-version `models_perspectives_domains_read`) must be there. See ConfigConfiguring an Apache VirtualHost running Couchdb for a domain of perspectives modelsuring for details on how to set up Apache for this domain.

9.2. Model Versions and Compatibility

Perspectives parses an ARC text and transforms it into a DomeinFile (a JSON file format). On doing so, it **checks** the model. The purpose of this check is to prevent runtime errors and to ensure the intended meaning of the model.

As an example of the first category, consider a query that traces a path through the network of role and context instances. On each step, the query interpreter gathers resources from the database. If the modeller wrote a query with two consecutive context steps, the interpreter wrongly assumes it can read a property context from a context resource, which is a runtime error.

As an example of the second category, consider a query with a role-step in it for a role type that is not modelled. Runtime no error will rise, as the role instances getting function returns an empty sequence of instances. However, the program obviously cannot realise the intended meaning.

Of course model checking will never ensure intended meaning completely. In the end, that is the modellers' responsibility. More broadly, a model serves a purpose: to support a group of end users in their cooperation. It is the modellers craft to build a model that achieves that purpose.

In this text, I will reserve the word `consistent` to refer to a quality that a model is said to have, if and only if:

- it does not cause runtime errors
- it does not demonstrably violate the modellers' intended meaning.

This latter concept is, for now, operationally defined as the set of constraints that is checked.

This text needs to be extended with an analysis of changing perspectives. A sketch follows.

Newer model versions may have different perspectives, causing two problems:

- deltas sent by peers with an older version may be refused because in the receiving peers version the role that authored it, is not authorised to do so;
- changes are not communicated while peers expect them because they have a newer model where they have a perspective on the change, while the model version of the authoring peer doesn't.

9.2.1. Compatibility

A model may refer to another model. We call this second model a dependency of the first; it has a dependent. On checking a model with dependencies, we automatically establish the consistency of the model, extended with the types defined in those dependencies. If a model error is detected, the two models are said to be incompatible; otherwise we conclude them to be compatible.

9.2.1.1. Model versions

This notion of compatibility is of special significance in the light of model versions. It may happen that a newer version of a dependency is not compatible with a model. This is important information for the users of those models; they would be advised against updating the dependency, or, vice versa, would be advised to update the dependent. That is, if a compatible version is available! This is of concern to modellers; they should care about updates of dependencies of their model. To be more precise, they should check the compatibility of such updates with their models and preferably update their models to maintain consistency with the latest versions of those dependencies.

Referring back to the discussion above, they should also check if those latest versions do not compromise the intended meaning of the program in the sense of good support of the end user.

9.2.2. Backward compatibility with respect to user data

Up till now, we've only discussed compatibility of models. What about compatibility of a model and data? In Perspectives, data consists of context and role instances (collectively called `resources`). Obviously, resources that are constructed with respect to a particular model version cannot - indeed, should not! - cause problems. However, a new model version's types might not fit existing data.

Let's first examine what kind of problems can arise in this relation.

Resources turn out to be remarkably resilient with respect to types. First of all, notice that the **shape** of context and role instances does not depend on their type. This means that no problems can arise in encoding and decoding data when it is moved in and out of the database. In other words, the infamous problem of schema evolution does not arise in the context of Perspectives with respect to the database it makes use of.

9.2.2.1. Backward compatibility of role instances

How can a role type change?

A role type can

1. acquire or lose a property
2. have the type of a property change
3. have the allowed type of its binding change
4. acquire or lose a aspect
5. change into another kind (kinds are user, context, thing)
6. acquire or lose a view
7. change its cardinality (functional or not)
8. become indexed or no longer indexed
9. become linked or no longer linked

Some of these changes will cause the PDR to throw runtime errors in combination with existing representations of the instances of the type. We'll discuss each of them in turn.

Losing or acquiring a property

This causes no runtime trouble. A new property will, in time, lead to values be represented on role instances. Existing values represented on role instances will simply never be used.

Change the type of a property

This will cause problems for some types, depending on the change. As a String type is not interpreted, a change to that type is frictionless. However, change to Boolean, Number or Date may invalidate some existing string values. Errors would arise on interpreting a string representation as a typed value.

This can be detected automatically by comparing the new model version to the old model version.

We can prevent runtime errors automatically, too, by removing property values whose type has acquired an incompatible change.

Change the allowed type of a role type binding

If the new type is more general than the old type, no problems arise. Otherwise, some existing bindings may be invalid. This mostly boils down to 1, as properties are expected that are not represented. It may happen that the more specialised type shadows an existing property that has another type. This, however, will be detected by the model checker: it is a model time error.

Acquire or lose an aspect

This reduces to 1.

Change into another kind

This would be a huge semantic change. However, there seem to be no runtime errors that can arise because of such a change.

Acquire or lose a view

There can be no runtime errors as a consequence of such a change.

Change of cardinality

There can be no problem if a role no longer is functional. A role that becomes functional will give unforeseeable semantic effects, but there will be no runtime errors. There is no automatic way to fix this semantic problem. In the end, the user should select which of the role instances in a context should be selected as the one that may continue to be present.

Become indexed, or become not indexed

An indexed role is by definition functional. It means that we can refer in model texts to this role instance by a fixed name. Internally, this name is changed into a unique name for each end user. So, while references in the model text must change, there is no consequence for the runtime.

Become linked or unlinked

As a role becomes unlinked, the context instances that hold that role will no longer use the indices stored within them to retrieve the instances of the roles. This causes no particular problems.

However, the other way round is problematic. Without further action, each context instance would lose its role instances for the particular type. This problem can be fixed automatically.

Nevertheless, no runtime errors would occur.

Wrapping up: compatibility of role instances

We first notice that runtime errors only will occur for 2, property types that change their range type. Can such problems be diagnosed before models are put into use? Yes, but only by comparing a new model version with its predecessor. Such a check is really straightforward.

Second, we should notice that semantic errors may arise, too, in cases 7 and 9.

9.2.2.2. Backward compatibility of context instances

As with roles, let’s explore the ways in which a context type can change.

A context type can

1. change its kind
2. lose or acquire aspects
3. lose or acquire roles, where we must distinguish user roles from context roles and thing roles
4. lose or acquire nested context types
5. move to another context type

6. become indexed or become not indexed

Change the kind of a context

Context kinds come in flavours like Domain, Party, Case, Activity and State. There are semantic restrictions on embedding context kinds in other context kinds. However, these are not yet checked. Anyway, these are model issues only.

Lose or acquire aspects

This reduces to losing or acquiring roles.

Lose or acquire roles

The situation is very similar to roles losing or acquiring properties. Again, this causes no runtime problems, in general terms. Role instances that are represented on context instances that have a type that is no longer modelled for that context, just are ignored, runtime.

There is one exception to this rule and that concerns user roles. When a user role is removed, it may be that a particular context instance refers to an instance of that role that represents the end user in that context. This value may no longer be used (as it no longer has a type!). However, just ignoring it means that the end user no longer has access to the context. This obviously is a big semantic problem. But, without handling it properly, there will be runtime errors, too.

Runtime errors can be prevented by automatic action (remove the me member from affected contexts).

Lose or acquire nested context types.

This is not a runtime problem. The consequence is that type names change; this should be handled in model time and such changes should be transparent in runtime.

Move to another context type

This is the same issue as 4, but now viewed from the other side.

Become indexed or become not indexed

As with roles, this is not a runtime problem.

Wrapping up: compatibility of context instances

We see only one issue for run time and that is 3, when a context loses a user role type. However, this problem can be handled automatically.

9.2.2.3. Conclusion: backward compatibility of user data

Based on this analysis, we will say that data is **compatible** with an updated model version, if and only if it causes no runtime errors.

If the updated model version requires changes to the data that can be performed automatically, we say that the data is **fixable** with respect to the new model version.

Existing data (created with a particular model version) may turn out to be either compatible or fixable with respect to a new version of that model. Remember that this applies just to the issue of runtime errors, not to semantic issues!

9.2.3. Compatibility issues with respect to data received from peers

In the above, we’ve focussed on data created by the end users' PDR. What about data created by the PDR of peers, who may have another version for a particular model?

Let’s first consider the situation where the peer uses a *newer* model version. Three kinds of potential problem may occur:

- incoming Deltas may refer to types that are in the new model version, but not in the old model version;

This causes a runtime problem, because the authorisation process will reflect on that type. The problem arises during authorisation. We can catch these problems and choose to ignore the deltas that caused them.

- incoming Deltas may contain property values whose shape is incompatible with the type in the old model.

This does not cause immediate problems upon handling the delta, but will cause runtime errors later on.

We can add a check to the transaction handling and reject any Property deltas whose data have the wrong shape.

- a Context Delta arrives that puts a particular role instance in a context while the role type is unlinked in one of the models and linked in the other.

This turns out to cause no runtime problem. The PDR will either add or not add the instance to the context. The new instance will be found by query, or by following a link. Local logic is consistent!

Now let’s consider the reverse case: peer data refers to types from an *older* version of the model (than that the user himself has installed). All of the problems signalled above may arise. However, all we can say is that the peers' model is incompatible with that of the receiving user; we cannot say which is older.

9.2.3.1. Handling the problems

We can handle both problems, to the cost of ignoring the problematic deltas. This may cause further delta handling problems, but all can be handled to the extent that no runtime error arises.

Semantic problems persist, however. We would like to handle these situations based on knowledge about whose model is older:

- if the receiving users' model is the oldest, one course of action may be to stop handling deltas and ask the end user if he wants to upgrade his local model.
- otherwise, we might want to advise the sender to update his model. However, this would

require an entirely new form of communication between peers. As an alternative, we might simply wait. As soon as the user modifies a resource governed by this model himself, and the peer has a perspective on the modified resource, its PDR will find that it uses an outdated model.

9.2.3.2. Some special cases

Moving a role type to another context type

A modeller might decide that a particular user role is in a context but would be better of in the context that embeds contexts of the first type. As a consequence, the perspectives of that role would have to change - the model checker will signal any omissions.

Runtime errors can be prevented automatically for such a change, but semantic problems will arise. In this special case, however, a precise modification of the data would mend the semantic problems: move the role instances from one context to another, too. This can be done automatically but the modeller should decide if this action is desired.

Moving a property type to another role type

This case may be more frequent than the previous. It will probably be quite common for a modeller to move properties around in the role graph. As with roles, the required adaptation of the data could, in principle, be done automatically. Again, the modeller should decide whether that should be done or not.

Specialising a role’s binding

There is no automatic solution to this problem. However, we might find all role instances whose binding has become illegal and present the end user with a means to re-bind them.

Change type names

We can change either the namespace (by embedding the type in another context or role) or the local name. These changes are quite important for models, but should be invisible in the end user data.

9.2.4. Solutions

9.2.4.1. Recognizing versions

If we include version identifiers with type names, we can quickly establish what situation we have at hand, when a delta arrives and we cannot find a type name. On looking up a type name, we currently split it into a model name and a local type name. Including version numbers requires us to split the name into a third part, its version number, which is not a big deal.

We would immediately recognise a version mismatch. But we can do better. Compatibility over versions can be established per type, with each type recording the earliest (oldest) compatible version (this would need to become part of the model parsing and checking process). Upon recognising a model type mismatch, we can then establish whether the type in our model version is compatible with that of the incoming version (notice this may require us to retrieve a newer version).

Second, the problem of the wrong shape for an attribute value. We would immediately establish whether the problem exists, by checking if the incoming version of the property is compatible with our local version.

9.2.4.2. Unique model names

The issue of type name changes ties in with another, related potential problem and that concerns the uniqueness of model names. We need our model names to be both readable by modellers and to be unique - worldwide. That should either be solved by a global register (such as the DNS) or by GUIDs. Barring a global register, we must rely on GUIDs. As GUIDs are not readable, we must use a (local) name translation system that gives the modeller the illusion of uniqueness of readable names, while protecting them from name conflicts if they arise. We will equip a DomeinFile with both a readable and a unique name. The modeller can write his text in terms of the readable name: on parsing, the PDR substitutes the unique name. Now if a modeller wants to use two dependencies that have the same readable name, he must use a substitute in his text. The issue is: how to associate a substitute with a model? The association must be in the model text. The straightforward way would be to include the unique name in the text. For example:

```
use: sys for model:System
use: esys for model:System (<guid>) -- Electrical system modelling.
```

IDE support might assist in inserting the guid.

9.2.4.3. Handle type name changes

Modellers might change the name of a type from one version of the model to the next.

We check a dependent model, however, by parsing its source again, using the new DomeinFile as dependency. This will cause any reference to a renamed type to be thrown up as an error. An error that could be prevented, if only we knew how to substitute an old name for its new value!

So we need, as the result of parsing an adapted model text, a table that maps old names to new ones.

There appears to be no robust way of building such a table automatically. Instead, we offer the modeller a way to provide hints, so reference errors for dependent models can be avoided. A hint consists of the old name in square brackets immediately behind the new name. The parser builds those hints into a full table. Notice that this must handle changing namespaces, as well.

It should be noted that this is a solution for consecutive model versions only. Clearly, a name that changes in two successive versions, needs a different translation table for dependents that refer to different versions.

9.2.4.4. How to adapt a source file given a translation table

Given a table that maps, for a given dependency, the old names to new names, how can we adapt the source file to the new situation?

First, notice that references to types outside the model must be fully qualified. Such names come in two forms: expanded and prefixed.

Expanded names are easy to map to their new versions. Replacing them in the source file is just a matter of string replacement.

Prefixed names are more difficult. The main problem is that prefixes only hold for a given syntactical block. A modeller may use the same prefix as a substitute for **different** context names. If the source then contains the **same local name**, in different blocks, with the same prefix (but another mapping) we can no longer substitute globally. Luckily, we can detect that situation and warn the modeller that he must change his source before dependency name substitution can be carried out.

9.2.4.5. Unique names to protect instance data

Were we to substitute internal names for model type names that would persist through model versions, we could protect references in instances to types to those changes. Succinctly: instead of using the readable name as the type reference in instances, we would use the unchanging internal name.

This is very important. Without this facility, a model type name change must be followed by type reference changes in all instances of that type.

We can achieve this by including in the DomeinFile a translation table from readable names to internal names. Notice that this is a **different** table from the one that maps old names to new names. On parsing the source text again, we look up each readable name in this table and use its internal substitution in the data structures we're building.

By running each name through the old-new table, we can find the internal name for a new name.

9.2.4.6. A diff function for model source files

By comparing DomeinFiles, we can construct a structure of data that details the differences between a model's types. This structure could be encoded as JSON and we could generate a readable report from that, to be consulted through the MyContexts GUI.

We can put this diff structure to good use by providing hints for the modeller that can be shown in the IDE. Particularly, we might draw his attention to references to type names he failed to update after renaming the type (e.g. a property name in a view). We can do this because the DomeinFile refers using the internal name and we can look up the new readable name and compare it with the name in the text.

Another service is that we can detect a new name in a particular namespace, notice that another name has disappeared, and suggest that the modeller may have forgotten to include a renaming hint for the parser. E.g. when a role with local name MyRole may have changed to SomeRole, the modeller **should** have included the name change in the text (SomeRole [MyRole]). We can provide the following hint, in the form of a question: "Did you rename MyRole to SomeRole?" Obviously these hints are not infallible.

9.3. Revision of Couchdb documents

Perspectives stores its data in Couchdb: PerspectContext and PerspectRol are the primary representation. However, we also store DomeinFiles. Each of these entities are stored as JSON documents and Couchdb provides them with a Revision in the form of an extra parameter (usually _rev).

> We use *entity* as the group name for the data instances that represent Perspectives runtime information. We use *document* as the group name for the JSON-serialized version of an entity, in the context of Couchdb.

> Couchdb uses Revision. We distinghuish this from *version*. An entity may change over time and we might call these successive reincarnations *versions*.

When we want to overwrite a document, we have to provide its current revision string as a query parameter

> We can also provide it as the member _rev in the document itself. Even though we include that member in our definitions, we cannot use it in communicating with Couchdb. This is because we rely on Generic classes for serialising and this means an extra object layer around the resource itself. Hence, our modelled _rev member is buried and invisible for Couchdb.

For sake of efficiency, we keep the revision in entities themselves. Obviously we have to keep that value up to date. This turns out to be complicated, given the asynchronous interaction with Couchdb.

NOTE this text is **not** about *published versions* of a model. A model may go through many revisions before it is published as a new version.

9.3.1. Overview of the flow of entities and documents

Each type of entity can be cached and, its serialised version is stored as a document in Couchdb. In this sense, cache and Couchdb are both a *destination* and a *source*. However, we have more sources and destinations.

9.3.1.1. Sources

Besides cache and Couchdb, an entity can come from:

1. A creation statement, executed by or on behalf of the end user.
2. Parsing an Arc model (a DomeinFile is the serialisation of a model).
3. Starting to use a model (usually this creates some context- and role instances).

4. The query that retrieves the external role of model descriptions from a repository.
5. A UniverseContextDelta or a UniverseRoleDelta, originating from creation statements executed by *other* users.
6. A query on Couchdb (e.g. all instances of an Enumerated Role type).

Of these, only the last one gives us revisioned entities. Each other case has to be handled carefully.

9.3.1.2. Destinations

Contexts and Roles may be serialised as either Deltas, or JSON in a format that is suitable for exchange between PDR installations that are not connected (it is also used to create instances client side). None of these have revision information in them. Revisions will be re-established in the receiving PDR.

9.3.1.3. Why do PDR instances not share version information?

Many end users can share a context. So why don't their PDRs share revisions? The reason is that each user *may* have a different perspective on that context (or, for that matter, on roles, too). Obviously if the entities themselves are different, their serialisations must differ and hence we're talking about different documents altogether! Different documents have different revisions. This might be confusing, so let's summarize all kind of variants we've seen:

1. A *revision* applies to a Couchdb document (the serialised entity).
2. A *perspective* gives a user a unique variant of a context, depending on his role in it.
3. As a context (or role) evolves over time, we can say a user's perspective on it has different successive *versions* (and their serialisations have different revisions).

So, because the perspective on an entity has different successive versions, its serialisation will have different revisions. And because perspectives are unique, each user has a unique serialisation and hence revision for a context that all participants identify as the same!

9.3.2. Maintaining consistent revisions

First, we explain the revision handling between Couchdb and cache. Then we turn to the other sources of entities, explaining what will happen when we move an entity from such a source to cache (we never move an entity to Couchdb unless it is cached).

9.3.2.1. From cache to couch and back

We aim to keep the revision of an entity in cache equal to its serialisation in Couchdb. In moving entities in and out of Couchdb, we accomplish this by

- Storing the revision that comes from Couchdb in the entity in cache (all these entities must have a Revision instance; Revision is a class defined in the Couchdb package).
- Sending the revision in cache – if it exists! - as a query parameter to Couchdb.
- Putting the new revision string that comes from Couchdb after storing the serialisation, back in the entity in cache.

9.3.2.2. Creation

When an entity is created and stored in cache, it obviously has no revision. It has never been sent to Couchdb! This situation is easy and requires no special handling. The function saveEntiteit (or saveEntiteit_, a variant that takes both an id and an entity) handles this case by not sending a query parameter with revision information.

9.3.2.3. Parsing an Arc model

An Arc model is serialised as a DomeinFile. A DomeinFile has a Revision instance. A DomeinFile will be stored in Couchdb. Typically, parsing and storing an Arc model is an activity for the modelling user; end users just download DomeinFiles and start using them.

Obviously, a freshly parsed Arc model has no revision. However, the modeller may have parsed that file before and so there may be an equally identified DomeinFile in Couchdb with a revision!

Before moving such a parsed DomeinFile from cache to Couchdb, we have to update its revision by looking in Couchdb whether it exists and, if so, taking its revision. This is handled in the function Perspectives.DomeinCache.saveCachedDomeinFile.

9.3.2.4. Starting to use a model

A user may decide to start using a model; this involves downloading a DomeinFile from a repository. Later on, he can decide to no longer use that model. And then even later on, he may again start using the model. This entails that, if we again download the DomeinFile from the repository, we have to update its revision in cache before saving it (again) to Couchdb.

9.3.2.5. External model descriptions

In order to show the user the models that are available on a repository, we query it to send us a list of the external roles of the descriptions of these models. These role instances are cached. Now some of the available models may have been taken into use before and this means that the corresponding external role of its description is already in Couchb. If this is the case, we prefer the version that is in Couchdb over the one that comes from the repository. This is handled in the function getExternalRoles in package Perspectives.Extern.Couchdb.

NOTE This way of retrieving model descriptions will become obsolete soon.

9.3.2.6. Receiving a UniverseContextDelta or a UniverseRoleDelta

It probably may happen that we receive a UniverseContextDelta for a Context instance that is already in Couchdb. Such a delta is an *idempotent* operation, meaning it will not change Perspectives state and thus there will be no revision change.

Chapter 10. Perspectives

10.1. The Operational Meaning of a Perspective

We have four categories of use for a perspective:

1. We use perspectives when creating screens so the user can consult his data. This usage ranges from creating screens automatically, to informing components on the client side about what properties are available on a role, to the user.
   Part of this use is to keep the results of queries made by a client up to date with respect to changes to the underlying data.
2. We also use perspectives to enable the user to change data and prevent him from making unauthorised changes.
3. Perspectives guide us when a change to a context, role or property occurs (is made by the user). They tell us which peers should be informed of that change (a process called synchronisation). A variant on this theme is when a peer is added to a context and we need to send him all its details (a process called serialisation).
4. Finally, when we receive a change (as a delta packed in a transaction), we use perspectives to judge whether the author of that change was authorised to do so before we apply the delta to local data.

In analysing the usage of perspectives, it is useful to distinguish roles from properties. We will use the words role-perspective and property-perspective.

Perspectives have an object. These objects are, invariably, roles. It is useful to analyse perspectives on Enumerated roles apart from perspectives on Calculated roles.

In our analysis we'll be mainly concerned with usages 3 and 4 above: synchronisation and authorisation, respectively.

10.1.1. Perspective on an Enumerated Role

In Perspectives, we treat an instance of an Enumerated role and its binding as a unit with respect to properties. By that we mean that a perspective on a role provides access to all properties of the role **and** those of its binding (if not limited by a View).

So, for a user U having a perspective on an Enumerated Role ER:

- the role-perspective means:
  - an instance E of ER should be synchronised for U;
  - an instance B of type BR that is the binding of ER should also be synchronised for U;
  - U can legally Create, Delete and Fill an instance of ER (depending on the verbs in the perspective);
  - However, U cannot legally Fill, Create or Delete an instance of BR.
- the property-perspective (possibly limited by the view of the perspective) means:

  - all values for properties of E and B should be synchronised for U;
  - U can legally Create, Delete and Change values of properties of E and B.

Notice an asymmetry here. We consider **all** properties on a role to fall within a users' perspective on that role, be they on the role itself or on its filler; but that perspective does **not** allow us to change the filler role instance itself (neither create it, delete it, or fill it).

10.1.2. Compound binding: Binding Role Graph (BRG)

A complication arises because the binding of a role may be an expression constructed out of multiple Role types with the operators AND (for multiple bindings of a single instance) and OR (for alternative binding types of a single instance). The type of such an expression is represented as an ADT EnumeratedRole, using the constructors SUM (for expressions with OR) and PRODUCT (for expressions with AND).

So, in general, we say that the binding of an EnumeratedRole is specified as an ADT EnumeratedRole, where the simple case is ST <EnumeratedRole>.

This complication works out differently for role-perspectives than for property-perspectives.

The role-perspective applies to all role types in the ADT. So if the binding of a role is compound, we have to synchronise the relevant instances of **all** types in the binding ADT. However, as a user does not have the right to Fill, Create or Delete an instance of the binding of a role just by virtue of having a perspective on that role, we have to reject any (incoming) modification to instances of the types in the binding ADT (when justified solely by the perspective on the role that has that binding).

The property-perspective is different and depends on the constructors in the ADT.

- when role types are combined with SUM, the perspective applies just to the property types in the **intersection** of the property types of the members of the SUM;
- instead, on combining with PRODUCT, it applies to the **union** of those properties.

To further clarify the role-perspective: it applies to all types, **regardless** of the constructor (SUM or PRODUCT).

10.1.3. Perspective on a Calculated Role

A Calculated role is defined by a path expression. As with bindings, the basic building blocks of those expressions are Role types (Calculated and Enumerated alike). However, the operators are different: they represent traversals of the graph consisting of roles and contexts.

A simple path leads from a Context type to a Role type, often in another context. Either that role is Enumerated, or it is Calculated, leading (recursively, in the end) to an Enumerated Role itself (it is an error to define two Calculated roles in terms of each other). So we see that a Calculated role in one context really is an Enumerated role in another context.

> Note we cast the object of a Perspective on a role in the same context as its user role, as a (singleton) path as well for the sake of uniformity of representation and analysis.

This means that we can express the type of a Calculated Role in terms of Enumerated Roles, using the constructors of ADT. For a simple path-based Calculated Role, its type is ST <some enumerated role>.

However, we can construct expressions with operators that combine paths: union and intersection. The type of a Calculated role constructed with union or intersection is an ADT constructed with SUM.

We can apply the insight gained from analysing compound bindings to this case. We've seen that for role-perspectives, irrelevant of the constructor used (SUM or PRODUCT), the perspective applies to all role types occurring.

However, for property-perspectives, the perspective just applies to the intersection of the properties of the individual types.

Notice that we have no path operator that constructs expressions with an PRODUCT type.

10.1.4. Serialisation for Calculated Roles

In the analysis above we've focused on the endpoint(s) of a Calculated role expression, treating a Calculated role as a remote Enumerated role. This falls short when it comes to serialisation (in the third category of usage of perspectives). If we introduce a peer in a context where he has a perspective on a remote Enumerated role, we cannot suffice with just sending the context and its roles. We also have to provide the remote role, its context, **and all roles and contexts on the path to it**.

10.2. Contextualizing Queries

In this chapter we explore a theme that we will continue in the next chapter: contextualization. Contextualizing of queries is a simple case of the more general phenomenon, so we use it to build some intuition.

10.2.1. Aspects

Perspectives relies on a powerful abstraction mechanism called *aspects*. An aspect is a Context (we talk about *types* unless specified otherwise). that may be added to another Context, which we call a *specialiser* of the aspect. It thereby brings its roles into that specialiser. Furthermore, a role in the specialiser may be augmented by a role of the aspect. The role thus becomes a specialiser, too. The effect of Role specialisation is that

1. The aspect Role properties are added to the specialiser's role;
2. The binding of the specialiser *must be equal to or more specific than* that of the aspect role;
3. If the aspect Role is a User role, its perspective (a collection of Actions) is added to that of the

specialiser role.

10.2.1.1. Example

model:SimpleChat contains the definition of Chat, whose definition follows (partly) below:

```
case Chat
  aspect sys:Invitation
  user Partner (not mandatory, functional) filledBy sys:PerspectivesSystem$User
    aspect sys:Invitation$Invitee
```

Chat has aspect sys:Invitation:

```
case Invitation
  state Invite = exists Invitee
    on entry
      do for Invitee
        ...
  user Invitee ...
```

The role Partner in Chat is a specialisation of the role Invitee of Invitation. Notice that we run an automatic action on behalf of Invitee in Invitation, as soon as there is one.

10.2.1.2. Problem statement

When we add an instance of Partner to an instance of Chat, we expect the automatic action to execute.

> In the rest of the text we will write ‘when we add a Partner to a Chat’, leaving out all references to instances, meaning the same thing.

After all, Partner is just an Invitee in disguise. But prior to Perspectives v0.5.0 that did not happen. The state condition, being the query `exists Invitee`, applies the step Invitee to the instance of Chat, looking for a *literal* occurrence of “Invitee”(properly qualified, so really: `model:System$Invitation$Invitee`). And it will not find it: it has an instance of Partner instead. As a consequence, the rule will not fire.

We could, of course, rewrite the condition:

```
  state Invite = exists Partner
```

This would work. We would have *contextualised* the state condition in the type Chat. Obviously, we would have to add the state definition to Chat (the model checker would refuse it as part of Invitation, because Partner is not defined in model:System). So in effect we would have to overwrite this rule in the specialising context.

That is not what we want. We want a mechanism that contextualises automatically.

10.2.2. Solution

There are several ways to solve this problem. In principle, we would like to solve it entirely in compile time (let the modeller wait so the end user has better performance). This would involve automatically rewriting queries from aspects to fit their specialisers. It would also entail inverting those specialised queries. And these inverted specialised queries would, by definition, cross more model boundaries than the originals. It is not impossible, but cumbersome.

Instead, we have chosen to do a little more work in runtime. The key observation is that *we need aliases when looking up roles*. When looking up Invitee, we should know that Partner is an alias – and lookup with that key, too.

We can find aliases by reflecting on the model. After all, the definition of Partner references Invitee. How and when do we use this information?

NOTE The description below is but partial, because it omits a complication introduced by the fact that we sometimes want to add an extra context clause to looking up filled roles.

10.2.2.1. Double indexing: aliases in context types

In v0.4.0, a PerspectContext instance contains an Object (Array RoleInstance) where the keys are the (string values) of Role(types). Retrieving the instances of a Role requires just one lookup.

We will change that by

- Adding a new member to Context (the representation of context *types*): roleAliases. This will be an Object EnumeratedRoleType. The keys are, as before, Role(types); the values are Role types that are actually represented on context instances;
- Looking up a particular Role in a context instance in two steps: first we find, in the roleAliases of the context instance's type, the represented role type that we then use to look up the actual instances in roleInstances.

On transforming a model source file, whenever we find a role R that has another role as an Aspect, we add an entry to the aliases registration of R's context type.

10.2.2.2. Reverse lookup: the filled role step

There are two ways for a query to visit a role instance. The first, which we've discussed above, is when we move from a context instance to a role instance. The second is when we move from a role instance to a particular (set of) filled roles (filled by the the instance we depart from).

The filled role step requires a Role(type) because any Role may fill many others. Obviously, this step is affected by contextualisation, too. As an example, consider this small query:

```
User >> fills Invitee >> context
```

As Invitee is filled by model:PerspectivesSystem$User, we can navigate back from an instance of model:PerspectivesSystem to all Invitation(s) its user role fills an Invitee role in. But now consider a Chat with a Partner. Partner will be bound to User as well. However, in version v0.4.0, this query will not find any instance of Chat.

Again, we have to rely on aliases to make it work. But this time, as the direction is reversed, the alias table is constructed and located differently.

We add a new member to PerspectRol: roleAliases. This is an Object (Array RoleType). Its keys are the string values of Role(types). Its values are the Roles that specialise the key, including itself.

The representation of the filled roles does not change. But lookup does. In our example, we first look up Invitee in roleAliases *on the instance*. It finds Partner and then looks that up. Thus it finds the Partner instance that is filled by the user instance.

We build this structure gradually. If there is not yet an entry for a Role(type), we reflect on the model, find all aspects of the type and add the type under the entry for each aspect (adding an entry when necessary).

10.2.3. No consequences for serialization

The association between a Role(type) in the object aliases, and a particular index in roleInstances, depends on the historic order in which role instances were added to that particular context instance. These histories could be different for users sharing that context: one may have a perspective on a Role that another has not.

This learns us that we cannot communicate these indices in Deltas. But this need not worry us, because a Delta is like a *remote procedure call* rather than a data item. The receiver of a Delta executes the call and that will lead to appropriate and possibly unique association between role types an indices.

Similarly, a context Serialization is translated into calls to functions that reconstruct contexts and roles.

10.3. Contextualization

The functional languages Haskell and Purescript have a feature that is called *type classes*. They may be compared with *interfaces* as known in other languages. In Haskell/Purescript, a type class can have *instances*, where an instance is a type that must supply an implementation for the types (usually functions) that make up the type class. In some cases, the compiler is able to derive instances automatically.

Type classes allow us to decorate a type *multiple times* with ‘behaviour’ in terms of functions. It is in that respect superior to (singular) OO inheritance.

Perspectives allows contexts to have another context as an aspect, effectively giving the former to the latter as an extra, or super type. It also allows us to add the roles of other contexts, effectively allowing *context composition*. The same holds for roles: a role can have an aspect role, adding the aspect roles properties to it and ‘inheriting’ the aspect role’s restriction on its possible fillers.

Aspects are, in a way, to Perspectives what type classes are to Haskell and Purescript.

User roles have Perspectives. They, too, can be composed of Aspect (User) roles. What if the latter have Perspectives, too? In this text we explore the space of meanings we can give to such compositions. It turns out that Queries, Actions and Automatic Actions must be ‘contextualised’ when taken from an Aspect and incorporated into a decorated context.

Contextualising can be compared to supplying the implementation of a type class’s types in an instance. It turns out that the Perspectives compiler can derive all these ‘instances’ automatically – that is, it can carry out contextualisation automatically without requiring any action on the part of the modeller.

10.3.1. Compile time or run time?

Queries, actions and perspectives are described as particular data structures that are part of a (compiled) model. We may contextualise them in compile time to new versions that we can store in the model. Alternatively, it turns out, we can adapt the runtime engine to carry out the contextualisation of an abstract structure in a concrete situation. As this text will show, we pursue a mixed strategy on further expanding Perspectives, contextualising some structures in compile time, others in run time.

10.3.2. Contextualising a Perspective

We work with an example, taken from model:BodiesWithAccounts:

```
case Body

  user Test
    property UserName (String)

  user Admin filledBy sys:PerspectivesSystem$User
    aspect bwa:WithCredentials

    perspective on Accounts
      only (Create, Fill, CreateAndFill, Remove)
      props (UserName, Voornaam, Achternaam) verbs (SetPropertyValue, Consult)

  user Accounts (unlinked, relational) filledBy sys:PerspectivesSystem$User
```

The user roles in this model fragment are used as Aspects in another model, model:CouchdbManagement:

```
case CouchdbServer
  aspect acc:Body

  user Admin filledBy CouchdbManagementApp$Manager
    aspect acc:Body$Admin

    perspective on CouchdbServer$Accounts
      props (ToBeRemoved) verbs (Consult, SetPropertyValue)

  user Accounts (unlinked, relational) filledBy sys:PerspectivesSystem$User
    aspect acc:Body$Accounts
```

Notice the relations between the latter model and the former:

- `CouchdbServer` has Body as Aspect;
- `CouchdbServer$Admin` has Body$Admin as Aspect;
- `CouchdbServer$Accounts` has Body$Accounts as Aspect;
- Both Admin roles have a perspective on the Accounts role in their context.

10.3.2.1. What we want

Clearly, we want that `CouchdbServer$Admin` can apply Body$Admin's facets of the perspective on Body$Accounts to `CouchdbServer$Accounts`. Specifically, we expect `CouchdbServer$Admin` to be able to create an instance of `CouchdbServer$Accounts` in virtue of the perspective of its aspect Body$Accounts.

But we also expect that the verbs applicable to the property ToBeRemoved are available to `CouchdbServer$Accounts`.

In other words, we want to 'add' both perspectives in such a way that the resulting perspective:

- grants access to the union of the properties of both;
- allows, per property, the union of the propertyverbs allowed by each perspective individually;
- allows the union of the roleverbs allowed to each perspective individually.

To add two perspectives is conditional on the relation between their objects: we may only add an Aspect perspective to another perspective if the latter's object is a specialization of the former's object.

Compile time contextualisation

For roles that are added *as is* to a context, we'd have to contextualise

- their calculation (if any): this would be query contextualisation and the paragraphs below will show we will apply run time contextualisation to queries. Consequently, no specific action is required for the compile time contextualisation of the calculation of roles.
- Their perspective object. As these are queries as well, this falls in the same case as calculations.

No action required.

- Their Actions (actions with an object). As the relevant paragraph will show, Actions will be contextualised in run time.

In short: nothing needs to be done to contextualise a role that is added *as is* to a context.

For roles that are added as an aspect to a role, things are different. First of all, we have to distinguish two cases:

1. An aspect adds a perspective on an object that the specialised user role does not have a perspective on. In this case, the perspective is added *as is* to the specialised role (is simply copied).
2. The specialised user role already has a perspective on a local object role, that turns out to be a specialisation of the object of the aspect user role.

In the second case, we have to adapt the perspective of the specialised user role, but only with respect to its

- role verbs,
- property verbs.

Role and property verbs are additive over aspects: a verb applicable according to an aspect or the specialised role is applicable to the specialised role.

Design decision I:

we rely on compile time perspective contextualization w.r.t. verbs for composed roles; on run time contextualization of calculations, perspectives objects and actions.

Where perspectives are used, does contextualization matter?

Synchronization. The query that is the object of the perspective, is inverted in order to find user roles that should be informed when a change affects the outcome of that object query. Obviously, the query must be contextualized if it is taken from an aspect to a special context. However, because queries are contextualized run time, no further action is required for synchronization.

Authorization. A Delta is only accepted if the author of the change it describes, does have a perspective that authorizes him to do so. Contextualisation of aspect perspectives is important for this process, as the required authority may derive from an aspect. Compile time contextualization as described above will do the trick neatly.

Screens generation. Obviously, the perspectives contributed by aspect user roles to a user for whom we request a screen giving access to a context, cannot be missed. The mixed strategy contextualization of perspectives described above will take care of this.

10.3.3. State complicates matters

In the example above, perspectives were valid in the root state of their objects or subjects. When a perspective is only valid in some state, things get complicated very quickly when we want to combine perspectives.

The relation between two (macro) states can be thought of as a Venn-diagram of two circles representing the micro-states of both (macro) states. The diagram allows for three separate regions, where the perspectives may only be added in the union (the overlap of the two circles).

This would require we compute, in type time, the logical conjunction of both states' condition. This we will not do. It requires logical reasoning beyond the complexity we are willing to tackle.

Of course, we can just combine both conditions and see what happens in runtime. However, we then might create conditions that will never evaluate to true but consume resources nevertheless.

Also, picture once again the Venn-diagram. It may be that one state is entirely inside the other. We would have no way of knowing it and would still have a separate perspective for the contained state. Again, a waste of resources.

10.3.3.1. What we will allow

Instead of reasoning about state conditions, we will allow

1. a perspective to be conditional on an Aspect state (this is a modelling facility: one can refer to Aspect state in an in state clause);
2. a non-ground state Aspect perspective to be added to a ground state specialised role's perspective;
3. an ground state Aspect perspective to be added to a non-ground state specialised role's perspective;
4. two perspectives to be added when conditional on the same state.

With regard to the formerly discussed Venn-diagram, this reduces to these cases:

1. the states of the specialised role perspective and the Aspect perspective are equal;
2. the specialised role perspective is valid in the (a) ground state, while the Aspect perspective is valid in a named state. In this case, the intersection coincides with the Aspect state;
3. The intersection coincides with the specialised role state perspective.

10.3.3.2. Aspect states that may be used

We only allow an Aspect state to be used in contexts or roles that have the type that has that state, as aspect.

10.3.4. Contextualisation of queries

A query consists of a series of steps. Some of these steps must be contextualised when a query moves into a context as an aspect (e.g. in the form of a Calculated role, or as the condition of a state):

- the role step: moving from a context instance to a role instance of a particular type R;

- the filled role step: moving from a role instance to another instance of a particular type R, that is filled by it.

Referring back to our example above: assume a query defined for case Body, that moves to instances of role Accounts. Clearly, when we start with an instance of CouchdbServer (having Body as aspect), we will not find role instances of Body$Accounts on it; instead, we should contextualise the query step from Account to `CouchdbServer$Accounts`.

It turns out that runtime contextualization is rather easy for the role step (for a more detailed treatment we refer back to Contextualizing Queries). For a given context type, we can compile an *alias administration* that maps, for each Enumerated role type available in the context, its supertypes onto that Enumerated role. Then, when a role step has to be carried out on a context instance by the query evaluator, we have it

1. look up the type of the context instance;
2. fetch the alias administration from it;
3. look up the role step type to find the appropriate Enumerated role type
4. use that to look up instances on the context instance.

This causes some overhead during query evaluation, but saves a great deal of space with inverted queries. Remember that a query is *inverted* for reasons of synchronization and state transition and that the inversions are stored with each role and context type that is part of the query. When we contextualize a query in compile time, we have to invert the contextualized version, too.

If we rely on runtime contextualization, when a role instance is added to a context instance, we have to look up inverted queries on the type of the role instance *and on all its aspect types*. Climbing the aspect hierarchy takes a little time, but in the end the same number of queries has to be evaluated.

The **filled role step** can be contextualized in runtime using a variant of the alias administration. We then need to build this administration in the *filler role* instances. An example will make this clearer. Suppose we have a pattern with a Driver and a Vehicle role and we use it to specialise a context with a Pilot and a Plane, respectively. The first time we fill a Pilot role instance with some role instance R, we'd have to record on R that Pilot is an alias for Driver. Then, when the filled role step Driver is carried out on R, we look up Driver in the alias administration of R, find Pilot, then read from R what role instances are recorded as filled under the key Pilot.

Overall, we find that run time contextualisation of queries is conceptually simpler, imposes less storage overhead and introduces an acceptable runtime penalty.

Design decision II:

we rely on run time query contextualization.

10.3.5. Contextualisation of actions

Actions consist of statements. Some statement types mention a role- or context type to create:

```
create role RoleType [in <contextExpression>]

bind <binding> to RoleType [in <contextExpression>]

unbind <binding> [from RoleType]

create context ContextType bound to RoleType in <contextExpression>

create_ context ContextType bound to <roleExpression>
```

In these statements, RoleType refers to some role that should be defined for the current context, or the type of context that results from <contextExpression>.

We have two kinds of Action: those defined in lexical positions with a current object (where the action will be applied to that object), and without a current object (these actions will be applied to the current context). We call the former kind *perspective actions* and latter kind *context actions*.

Both kinds must be contextualised, when the user role with access to the actions is used as an aspect for a specialised user role, or is included *as is* in a context. The point is that the RoleType (and ContextType) may be specialised in the contextualising context, too, in which case this specialisation should be substituted for the original in the assignment statements.

Returning to the Driver-Pilot example: let’s assume for the sake of the explanation that the Driver can create the Vehicle role. Clearly, we want the Pilot to create a Plane, rather than a Vehicle. So we substitute Plane for Vehicle in the actions of the Pilot.

10.3.5.1. The consequences of compile time contextualisation

Skipping the details of *how* to contextualise an action (more about this later), we ask ourselves: where can we store the contextualised actions?

A contextualised *context action* cannot be tied to its user role’s representation, since we can incorporate an aspect user role *as is* in as many contexts as we like. Nevertheless, we’d have to contextualise the action for each such context. Obviously, a contextualised action is specific to the *combination* of a user role and a context. We can save Actions directly in the DomeinFile in a map with keys constructed from context- and user role types.

What about contextualised *perspective actions*? Currently (version v0.18.0) we store perspective actions in the perspectives. We could contextualise the perspective holding contextualised actions, but then store perspectives in the DomeinFile, again under a key constructed from context- and user role type. In other words, if we incorporate a(n aspect) user role into a context, we create a specialised version of the perspectives of that user role.

This requires a major change to the implementation, however. This is because the above implies we conceive of a perspective as a relation between *two role-in-a-context combinations*, rather than between two roles. On the perspective object side, we’ve already tackled that issue (because of query inversion). The abstract data types that we use to describe query domains and ranges are in terms of combinations of role and context. However, on the perspective subject side it means a complete overhaul of many modules.

Compile time contextualisation of actions requires a major refactoring.

10.3.5.2. How to contextualise an Action

Before deciding on compile time versus runtime contextualisation of actions, we explore how to contextualise individual statements.

Create role

The create operator in conjunction with the role keyword:

```
create role RoleType [in <contextExpression>]
```

mentions a RoleType that must be contextualised. Let us work with a different example to create some intuition: we have an Aspect context Meeting with a role Organizer and a role Participant. Now, we create a MedicalAppointment with:

- a role Physician and a role Patient, both having aspect Participant
- a role Assistant, having aspect Organizer.

Organizer can create Participant role instances and has an action to create one. What happens when Assistant executes that action? As there are two specialisations of Participant, two instances will be created: a Physician and a Patient.

Contextualisation of this action in **compile time** would result in *two* statements in the action:

```
create role Physician [in <contextExpression>]

create role Patient [in <contextExpression>]
```

However, we have a variant of this assignment operator that lets us save the result in a letA variable. This introduces a dilemma: should we create an ad hoc new variable? Or can we assume that the variable can hold multiple values, and will semantics of the rest of the action be conserved, under this change?

Contextualisation of this action in **run time** would require an adaptation of the code that is compiled from the CreateRole datastructure. We would have to look up the specialisations of Participant in MedicalAppointment, keep only those that the user executing the action has a sufficient perspective on and then iterate over those specialisations.

Both approaches need the same lookup (of local specialisations of the RoleType). While compile time contextualisation is not quite clear, run time contextualisation is conceptually simple.

Bind

The bind operator:

```
bind <binding> to RoleType [in <contextExpression>]
```

needs to be handled just like the create role operator. It, too, creates a role instance.

Unbind

The unbind operator:

```
unbind <filler> [from RoleType]
```

works by clearing those roles filled by filler that have type RoleType. However, no roles will be cleared that have RoleType as a supertype (an aspect of their type). In terms of our example: if the instruction is to unbind from Drivers, no Pilot roles will lose their filler. In compile time, we should replace RoleType with the specialised role type to contextualise the action. Again, if multiple role types are specialised, we'd have to duplicate the statement for each of them.

However, we can handle it in runtime, too, relying on the *alias administration* in the filler role instances described above for the filled role step. Each time we fill a Pilot role instance with some role instance R, we'd have to record on R that Pilot is an alias for Driver. Then, when unbind is carried out, we look up Driver in the alias administration of R, find Pilot, then read from R what role instances are recorded as filled under the key Pilot. Finally, we'd clear the filler from those instances.

Create context

The create operator can also be used in conjunction with the context keyword:

```
create context ContextType bound to RoleType in <contextExpression>
```

The RoleType must be contextualised just like in the create role situation. The ContextType must be replaced, too: by the type of context that may be bound to the substitution of RoleType.

To extend our Meeting and MedicalAppointment example, let's assume there is a Calendar context with a role Meetings that holds contexts of type Meeting. And let's assume that Calendar is added as an aspect to HospitalCalendar, with a role HospitalMeetings that has aspect Meetings. It, however, restricts its fillers to MedicalAppointment. Now, some user in HospitalCalendar with an aspect role that is allowed to create Meetings, should create an HospitalMeetings role instance rather than a Meetings instance. And it should fill it with a MedicalAppointment, instead of a Meeting. To accomplish this, the runtime first finds that HospitalMeetings is the local substitution for Meetings and then discovers MedicalAppointment as its filler restriction, being a specialisation of Meeting.

Create_ context

The create_ operator is a variant of the create operator, in that it omits the RoleType to create and instead retrieves an existing role instance that should be filled with a new context:

```
create_ context ContextType bound to <roleExpression>
```

Looking up the ContextType goes exactly in the same way as with create.

10.3.5.3. Wrapping up: contextualising Actions

Compile time contextualisation of actions requires a major refactoring. In contrast, run time contextualisation requires just part of the mechanisms that are required for compile time contextualisation (the lookup of type substitutions for statements with the operators create, create_, bind and unbind.

Design decision III:

we rely on run time action contextualization.

10.3.6. Contextualization of automatic actions and notifications

Automatic actions are predicated on state change. A modeler may refer to a state change defined in an imported model. Such actions, however, are stored with the state definition itself. This poses a problem: a modeler may not change an ‘upstream’ model (he need not have authoring rights of that model).

This is a problem similar to that of calculated perspective objects, where the object role type is defined in another model. We have devised a mechanism that consists of a store of inverted queries in a model, that are distributed over imported models *in the installation*. In other words, we extend upstream models – but only locally, each time as a user installs a model.

We will apply a similar mechanism to automatic actions that predicate on an aspect state (a state of an aspect context or role that are defined in an upstream model).

Automatic actions itself need no contextualization, as we’ve shown above.

The same reasoning applies to notifications, that predicate on state changes, too.

Design decision IV:

automatic actions and notifications are distributed over upstream models in runtime.

10.3.7. Contextualization of Properties

When we add a role type A to a role type R, all A’s properties become available on R. That is, we can add values for a property A$P to the representation of an instance of R. It is as if A$P was defined as one of R’s own properties.

In some circumstances, R might already have a property that can be seen as a local version of A$P. This is much like the situation where we have a role in a context that we want to add an aspect role to. So, in analogy, we would like to be able to specify that a property R$P should be considered as the specialization of A$P; that is, we want to add A$P as an ‘aspect property’ to A$R.

Notice that the defaults are different for roles than for properties. No aspect role is added implicitly to a specializing context, whereas we do want all aspect properties to be added implicitly to a specializing role. As a consequence, we need a different syntax to indicate what we want (explicitly adding all aspect properties would make our models very verbose).

Instead, we will write that *an aspect property is **replaced** by a local property*:

```
user Driver filledBy sys:PerspectivesSystem$User
  property License (String)
user Pilot
  aspect ta:Transport$Driver where
    License is replaced by Certification
  property Certification (String)
```

We expect the following effects from such a replacement:

1. That a specialized perspective does not hold the aspect property, but its specialization (e.g. a perspective on Driver with property License, when specialized to Pilot should show Certification).
2. That a calculation in the aspect that refers to License should, in effect, be computed as if License was replaced in the source text by Certification.
3. That an assignment in the aspect on replaced property should be carried out on the replacement (e.g. when an assignment line sets License for Driver, it should set Certification for Pilot).

Obviously, the expected behavior 2 follows from runtime query contextualization. Behavior 1, in contrast, should be carried out on perspectives that are added from the aspect user role to a specializing user role – in compile time. Behavior 3, finally, will be carried out in runtime much as we do for action contextualization of role assignments.

10.3.8. Summary

This text contains the following design decisions:

1. We rely on compile time perspective contextualization w.r.t. verbs for composed roles; on run time contextualization of calculations, perspectives objects and actions.
2. We rely on run time query contextualization.
3. We rely on run time action contextualization.
4. Automatic actions and notifications are distributed over upstream models in runtime.

10.4. Universal Perspectives

In this text we explore the notions of *universal perspective* and *public context*. These turn out to be closely related to the subject of *multiple storage locations* for Perspectives resources (context- and role instances).

10.4.1. Public context

With Perspectives we recognise that various participants in a context may have a different view of the same role. Consequently, each of them keeps his or her own version of it on his or her own computer. Because the distributed runtime doesn't keep a central store of data, but instead sends updates of contexts and roles to those whom it will concern, only participants in a context will have access to data concerning its roles. This maximizes correct interpretation of data: not just anybody can see it. Thus, Perspectives is built for the fine-grained structure of society consisting of personal interactions.

But society also knows public spaces. Political debate is carried out in public; the government works in public space; newspapers, television and radio make information available to the public, indeed, by 'publishing' it.

Perspectives supports public spaces, too. For this we introduce the concept of a **universal perspective**. This is the perspective of a role that can be played by anyone; no restrictions apply (other than that one should have a computer with Perspectives on it to one's disposal). This implies that this role should be *calculated*: the calculation should work for every member of the public. By convention, we'll call that role **Visitor**. If all participants share the same perspective on a role (or context), all of them can do with the same version. This allows us to store such entities in a *public store*. Someone wanting to access the information on such a role will have to fetch it from that public store.

We call a context with a Visitor a **public context**.

How do we find a public context? For this Perspectives relies on the Domain Name System (DNS) of the internet. Concisely, the DNS is a distributed system of internet servers that allows anyone to exchange a readable name for an internet address. Ipso facto, the readable name identifies a *location* and, indeed, its technical name is a Universal Resource Locator or URL.

Now most of us are familiar with URLs: using a web browser program with access to the internet, we can retrieve a document 'from' the URL. Actually, we retrieve it from some internet-connected server: both the server and the document are identified by (different parts of) the URL.

10.4.2. URLS

Two parts of the anatomy of a URL are of interest to our purposes: the **authority** and the **path**. For example, in: https://www.nlnet.nl/somedocument.html, "www.nlnet.nl" is the authority, while "somedocument.nl" is the (very simple) path. The authority is built from simple strings of characters separated by dots. Each part is called a domain. The rightmost part - nl - is the top domain, while the parts to its left are subdomains, successively deeper nested. So "nlnet.nl" is a subdomain of the top domain reserved for the Netherlands.

What about the "www" part? Well part of the nlnet.nl domain is called "www". So www is a subdomain of nlnet.nl This sits uneasily with our intuition of the world wide web as a kind of 'umbrella' over all websites. Actually, technically, all domains that take part in the world wide web, address *part of themselves* - one of their subdomains - with the name "www". So really, all those domains have a bit of the world wide web *that they happen to call by the same subdomain name*. The world wide web is a convention, rather than a technical unit.

The *path* is used to identify resources in a particular subdomain. In our example the path is just the identifier of a document. However, we may construct a path of segments separated by slashes, for example: /foundation/anotherdocument.html. We can interpret "foundation" as a named collection of documents, or, in other words, as a *folder* holding documents and other folders.

Having studied relevant parts of URLs we can now explore how to use them in Perspectives.

10.4.3. Identifying public contexts

First we notice that the world wide web usually is understood as a system for locating resources, like documents or programs. In Perspectives, the main concepts are on a different level of abstraction: in presenting it, we take care to avoid concepts like document, message and information. Context and role are not just different names for information objects, even if, when we discuss the operation of the distributed runtime, we recognise that instances of both are represented as documents.

In Perspectives, the end user usually visits a context by navigating to it from another context. If alternatives exist - for example, a list of chats - these will be presented with some description entered by the user himself. The *identification* of a document representing a context, however, consists of a GUID and is hidden from his view.

Navigating will have to start somewhere and for that the end user can choose from a small number of readable names that also *identify* a context. These names are defined in Perspectives models. Now, an observant reader might object that a conflict must exist between

- each end user having a unique version of a context, and
- models giving a universal context identification.

> Two users would both have a System context and they would both create a User role in it. But these roles **must** be different, hence the System contexts must be different, hence they must have different names. I will call the building *you* live in, *your* house.

This conflict is solved by a system of *indexed names*. A prime example of an indexed name is "Home". In the abstract, it means the same for everyone; but concretely, each of us points to a different dwelling (except, of course, those living together at the same address). So each end user enters the same indexed name, but, under water, these are exchanged for unique identifiers *by each installation of the distributed runtime*.

Back to public spaces. A public context, being the same for everyone, can be identified by the same name for everybody too. We don’t need to switch them for unique private names, like with indexed names. This is how we use the DNS in Perspectives. **Domain names identify public contexts**. We call such a context a **named public context**; we call the URL a Uniquely Identifying Readable Name (UIRN).

Suppose the NLnet organisation wants to provide a public context. What domain name should it choose? Here we would like to introduce a convention. Just like the "www" subdomain holds resources to be shared over the internet, we would like to propose to keep public contexts and roles in a subdomain, too. We'll call it the **context web** and abbreviate it with 'cw'. Hence, NLnet should call its public context `http(s)://cw.nlnet.nl`.

However, given that MyContexts is served from a secure domain, Cross Origin Resource Sharing (CORS) requires that we should use the https scheme rather than http. In other words, a server operator would be required to obtain security certificates for the "cw" subdomain (and in what follows we'll see that quite a few more are required). This is a practical (financial) obstacle. For that reason we'll introduce another convention. Instead of using a subdomain, we'll introduce a convention for a particular way to create *path names*. To continue the example, NLnet would store its public contexts at the location `https://nlnet.nl/cw_nlnet_nl/`. See [Model description: a public context] for an extended example.

10.4.3.1. How does it look?

So, given the MyContexts application, how does the end user navigate to a public context? Actually, she can just enter the UIRN in the browser address bar. Again, by convention, the web server monitoring the cw subdomain will *redirect* the browser to MyContexts, providing the UIRN as a parameter: https://mycontexts.com?https://nlnet.nl/cw_nlnet_nl/some-context.json (notice that the characters :, / and ? should be *encoded* in order not to confuse the internet servers). As a consequence, the MyContexts program will run in the tab and it will use the parameter to access the document some-context.json in the public storage .

This means that an ordinary webpage can hold a url to a public context. Clicking it will navigate into the context web!

10.4.3.2. Model descriptions: Manifest

A model is a collection of types. A *domeinfile* is a description of these types in a form that can be used without modification by the distributed runtime. We identify models by a URN in the model scheme (see Type and Resource Identifiers). Intuitively: in the model scheme we use the DNS to maintain unique model names. A model is *described* by a public context whose URL we derive from the model identifier.

For example: the System model, created by Perspect BV io, has the URN model://perspectives.domains/System. From it we derive the URL of its model description: https://perspectives.domains/cw_perspectives_domains/System.json.

So some public contexts are the description of a model; others are not.

10.4.4. Multiple storage locations

The concept of public contexts stored in a publicly accessible location - a **public store** - presents us with the notion of multiple locations where the distributed runtime can store contexts and roles. In versions of the system up to v0.11.0 all resources went into a single local store. As we introduce the notion of multiple stores, this local store will become the **default private store**. The current code base realises this through Pouchdb, where the default private store goes by the identifying name "{systemIdentifier}_entities" (where {systemIdentifier} is replaced by the unique system identifier for the installation).

This store may be based in the IndexedDB of the browser, or it may be a database of a Couchdb installation anywhere.

Multiple stores present us with the following questions:

1. How should the runtime decide where to store a given resource, or where to retrieve it from?
2. Where do we keep information on the various stores that are not the default private store?
    a. what is their name
    b. what is their location
    c. where do we store credentials, if necessary?
3. Should it be possible for the end user to transfer resources *after their creation* to a different store, i.e. from store to store?

10.4.4.1. Where a new resource should be stored

By default, a new resource is stored in the default private store. However, the end user has functionality to determine, for each context or role type, the store of its instances. This requires a Perspectives model and a user interface.

Hence, on creating a new resource, we should use type reflection to look up the store where it should go.

The author of a model may specify that instances of a type should be stored in a public location. But the author cannot specify a *specific location* - just that it be public. In fact, the end user is ultimately in control. He will have to specify a location for types that are deemed public - but as it is up to him to choose a location, he can choose a private storage location, too (in fact, we base a best practice on that possibility below).

To specify that instances of a type should be stored in public space, the type definition should include the keyword public.

10.4.4.2. Where a resource should be retrieved from

Currently, resources are identified by a string of the form "model:User${GUID}". The resources identified by this form **must** be stored in the default private store. Each store must have a symbolic unique name, associated with an address that can be interpreted by Pouchdb. We will interpret "model:User" as the symbolic name of the default private store. We will prepend this name to the GUID identifying a resource, separated with a $ sign from it. So, for example, if MyOtherPrivateStore is another store that I keep some of my contexts and roles in, a resource identified by the form `{GUID}|MyOtherPrivateStore` will be stored in MyOtherPrivateStore.

10.4.4.3. Saving resources

Consequently, the identifier of a resource holds the name of its storage. This is a symbolic name. The runtime has a lookup table, kept in a runtime state, to translate that symbolic name to a physical location. Because we rely on Pouchdb, any location that is not IndexedDB should be identified by a URL.

10.4.4.4. Authenticating

The runtime core assumes it has a session (is authenticated) with the store it is about to retrieve a resource from, or store into. If not, it authenticates. Currently (version v0.11.0) it uses credentials stored in Perspectives State. To accommodate multiple private stores, it will separate the store name from the resource identifier, look up the store's credentials (always stored in default private store!) and authenticate before retrying.

As a consequence, we put the credentials for the default private store in Perspectives State at the start of the session. For the browser db, no credentials are needed; for a Couchdb installation, the user must enter them on starting a session.

10.4.4.5. Moving resources

The end user might decide that, after having stored instances of a type in the default local store for some time, to move them to another location. As location is encoded in the identifier, this requires us to rename the resources. This may be an expensive operation in terms of processing power, memory and storage access operations:

- each resource must be moved to a different location under a different name;
- each occurrence of the resource identifier must be replaced by the new name. This applies to
  - context names of role instances;
  - role instance references in contexts;
  - role instance references in bindings;
  - role instance references in inverse bindings.

This requires us to retrieve all instances of a given type from a database.

10.4.4.6. Moving an entire store

Because of the dereference of the symbolic name of a storage to its address, we can move the contents of an entire storage location to another storage location *without changing resource identifiers*. Furthermore, the function from symbolic name to address need not be a bijection. Hence, we may create two or more symbolic names for the same storage address.

This gives rise to a good (best) practice. It may, for example, be a good idea to store all one's financial stuff in the same location. Start out with an alias for the default public store, e.g. FIN. When the time comes, move the entire store to some safe storage location at relatively low resource cost, by associating FIN with a new storage address.

In case of a store that multiple symbolic names map to, when we move one of them, say X, we have to pick *only* the instances of the types that were saved to X!

10.4.5. Authoring public contexts and roles

To author a context (instance) that will be available to the public, requires three things:

- a storage location on the internet that you have writing rights for and that interprets the HTTP verbs in the way Couchdb does;
- that you associate the type of the instance you want to create with this storage;
- the ability to enter the identifier of the context through the user interface.

That's all. Just create the context in whatever way the author of the model you're using, has made available. After creating it, each modification will be published automatically.

This requires some planning. As soon as you put up your contexts and roles, they are available. That may not be prudent; you may want some time for yourself to edit your contexts and roles until you're satisfied with them. This is where the best practice described above comes in handy. Associate your types with a symbolic name that represents a place to work, and have it point to your default private store. Then, when the time comes to publish, point it to its definite public location.

Notice that while this gives you some breathing space, it is a trick that cannot be repeated for more instances of the same type after publishing the first instances. Should you publish something like a periodical, you'll have to find another way to temporarily bar the public from work in progress. A good way is to prevent the public from navigating from accessible public contexts to the stuff you're authoring, until you're ready. In effect, they are then hidden from the public eye (As you are creating a new public context, by remembering its public name you yourself can revisit it before publication!).

10.4.5.1. Co-authoring

How many authors may a public context (or role) have? It's in the model authors hands, of course, depending on the perspectives (s)he creates. It may be prudent to limit the authors of a public context, preferably to one. Various authors can overwrite each other's changes and there is no arbiter, neither will authors be notified of changes made by other authors. In other words, it would be a very rough authoring experience.

10.4.5.2. State

Contexts and roles have state. A model may prescribe automatic actions and notifications on state change. State change occurs on creating a context or role, and when they are modified. All this means that state can only be useful for the *authors* of a public context. Assuming that the universal perspective does not allow for changes, those playing that role will never benefit from notifications or automatic actions on their behalf.

10.4.6. Actions to create private contexts and roles from public ones

A public context can be useful in the same way that a static web page can be useful. But Perspectives is not primarily a system to publish information; it is a system to co-operate. If the Visitor cannot change a public context, how can she ever enter in a co-operation starting from such a context?

It turns out that Perspectives already has a provision for this situation, in the form of the *unlinked role*. An unlinked role instance refers to its context, but the context has no pointer to the role instances of an unlinked role.

Hence, a Visitor can create, for example, an instance of an unlinked Context role. The context role, the external role of the context and the context itself are all stored in private store. By providing a role for the author of the public context - let's call him Admin - in that new context, the distributed runtime of the Visitor will send deltas to the Admin so he can construct the new roles and context, too. This way, Visitor and Admin participate in a private context that was created from the public context. The author of the public context can have a perspective on the context role; his runtime will retrieve the instances by database query.

This is a good general scheme for subscription, where a Visitor explores public contexts and decides to subscribe by creating a (private) contract context between herself and the Admin. Creating the private instances may be done by providing the Visitor with an Action that can be accessed through the GUI.

How does the Visitor (re)visit his Contract? To navigate to it from the public context would require a perspective for him on Contracts. But this would disclose other contracts, too, a privacy breach. Instead, the Contract is modelled as a context with an IndexedName. Consequently, he can visit it by entering that name (it may be given on the public context page; indeed, after it has been constructed, a link may appear). Remember that, under water, the runtime creates a unique identifier for an indexed name

> The Admin will receive the unique name; not the indexed name! Notice that the Visitor can only create a single contract, using this mechanism (there is just one indexed name).

An alternative to indexed names comes into being as soon as we make filters work on inverted queries. We can then give the Visitor a perspective on the contracts filtered with the criterium that he himself plays a role in it. The PDR of the Admin would apply this filter on the inverted query and this would cause a contract to be sent to just the relevant Visitor (the version that exists currently, v0.11.0, does not support inverted query filtering).

In a different approach, we would allow the Visitor create a new user role – let's call it an Account- with more perspectives. Usually, restrictions on Accounts apply. Because the Account will be an enumerated role, an end user visiting a context in which (s)he plays the Account role, will use the perspectives that come with that role (because enumerated user roles have precedence over calculated ones, when the client establishes the perspective for its end user). See the text *The Body-with-Account Pattern* for more details.

10.5. Perspectives on role fillers

The properties of a Role's binding are as accessible as if they were the Role's own properties. The modeller may add a View to an Action that includes any of these binding properties. And this applies to the binding of the binding, recursively.

In this text we address the technical question: how do we make sure that changes to such properties are distributed properly?

A strongly related question involves adding a binding to a role. The point is that behind that binding, an entire graph of other bound roles and their contexts that were previously hidden, may suddenly come in scope for some Users. How do we make sure that updates are sent to them that ensure they actually have access to that graph? For an explanation, see [Sync subnetworks as a consequence of a new binding].

10.5.1. Access to a property considered to be a query

We invert CalculatedRoles – being query definitions - so we can compute the users that should be informed about some ContextDelta (remember, ContextDeltas describe adding or removing Role instances to or from a Context instance). A CalculatedRole is like a teleport into some other context for a User in his origin context, allowing him to see and manipulate a role in that faraway context.

Now consider a User Role U with a perspective on a role R in his origin context C. ‘Having a perspective’ means that U has access to some property P on R. We can construct that as U being in the position to request (from the PDR) a query on (some instance of) C:

> I will write ‘User role U can see role R’, where U and R are types, in the understanding that it is *instances* of U that actually ‘see’ or ‘receive’ *instances* of R.

```
role R >> property P
```

We now see that we could apply the same mechanism sketched above for CalculatedRoles to this perspective of U on R. Invert the query and run it when P changes. Now, because we have two steps, we actually have two inverted queries:

```
ValueToRole P >> context
```

context

The first inverted query we store with the Property P in its onPropertyDelta member. When an update function like addProperty is executed, a RolePropertyDelta is stored in the currently open Transaction. When it comes to distributing that Transaction, we execute the InvertedQuery’s in the onPropertyDelta member of the Property mentioned in the Delta. This finds us contexts. An InvertedQuery lists relevant User Roles, too. We then ship the Delta to the instances of those User Roles in the contexts we’ve found.

In short, by storing the first inverted query in the onPropertyDelta member of P, we make sure that U is informed of any change to P (please note that the change to P might come from some other User in C, for example. We do not distribute changes that a user makes to himself!).

We store the second inverted query (just context) in the onContextDelta_context member of R. Now, when an instance of R is added to an instance of C, by similar reasoning as for properties, instances of U are informed of that change.

10.5.2. Binding

R might have a binding, let’s say of type B. Let’s suppose U has a View on R that includes some property of B, named P_b. Obviously, when P_b changes, U should be informed. How do we make this work?

First, consider, again, U’s perspective to imply an implicit query:

```
role R >> binding >> property P~b~
```

The inverter now yields three queries:

```
ValueToRole P >> binder R >> context

binder R >> context

context
```

This is what happens:

- The first query will be stored in onPropertyDelta of P_b. This will make sure that changes of that property are sent to U.
- The second query will be stored in onRoleDelta_binder of B. A RoleBindingDelta on B will lead to its execution, finds us instances of C and then the instances of U that will receive the Delta.
- Finally, as above, the last inverted query is stored with onContextDelta_context of R. ContextDeltas that change the instances of R will be distributed to U.

So we see that all possible changes along the path from C to the binding property P_b are communicated to U.

These examples set the stage for the generic solution of the problem: how to distribute changes somewhere on the binding role graph of R to users with a perspective on R?

10.5.3. Solving the general case

The binding of a Role is given as a ADT EnumeratedRoleType. This allows us to construct elaborate Role graphs as the binding of a Role. The children of a node may be conjunctive (meaning that an instance can have multiple bindings), or disjunctive (an instance may have one binding, albeit of different types). The simple case is a straight path (like the examples above). We want to prepare our types, in compile time, with inverted queries to make sure that Deltas reach all relevant Users.

The examples illustrate the approach for the case in which the binding is ST EnumeratedRoleType. We now consider the other cases.

10.5.3.1. UNIVERSAL

When we declare the binding of a Role to be UNIVERSAL, we prohibit instances from having a binding at all

> The syntax for this is: R filledBy: None. Intuitively one would expect None to mean EMPTY. However, a universal role must carry all possible properties, including the property without value. No instance can ever have a property that has no value – so it is not possible to bind to a role whose binding is required to be UNIVERSAL.

Any properties in a View on that Role must be the Role's own properties. We just have the inverted query context to consider. We'll store it, as above, with onContextDelta_context of R.

10.5.3.2. EMPTY

Instances of a Role with a binding of EMPTY can be bound to anything. We do not pose any restriction. This is the default case, if we have not specified a binding type in our model text.

However, there is no practical preparation we can do, compile time, to handle this situation. In theory we could construct a role graph that is a huge product of all known roles. This would make us add an inverted query to each other Role. For obvious reasons, we do not do that.

This means that our implementation is incomplete for the EMPTY case. That leads to a best practice for modellers: specify a binding for each role, possibly UNIVERSAL.

10.5.3.3. SUM

By giving multiple bindings for a Role, we implicitly construct a Sum type. The modeller is restricted to properties that can be found on all members of the Sum. Interestingly, the queries leading to those properties do not have to be equal. Along one path, one may reach a Role with the requested property in two steps, while another path may take three steps (to given an example). There is a generic recipe for all such queries:

```
role R >> binding* >> property P
```

The number of binding steps varies from 0 to some arbitrary number. However, the inverted queries show the differences clearly. As an example:

```
Value2Role P >> binder R >> context

Value2Role P >> binder X >> binder R >> context
```

When navigating backward, we have to choose the right binder and this clearly illustrates the node graph underlying the binding of R.

This also gives us our recipe to handle this case. We should follow each path through the role graph, construct a query on the way, and then invert that query and store it with all stations.

10.5.3.4. PRODUCT

Finally, the PRODUCT case. Somewhat surprisingly, the SUM recipe works for PRODUCT, too: work through all paths, and invert them.

> There is currently no syntax yet to construct PRODUCT binding types. However, by constructing a Calculated query with a join expression and specifying that as the binding, we can in effect make a Role have a PRODUCT binding.

10.5.4. A practical algorithm

While the above is a complete approach, it is more conceptual than practical. It would involve constructing a query for each and every property in a view. Their inversions would have a lot of overlap. Is there a faster algorithm to construct all inverted queries? It turns out there is.

10.5.4.1. Relevant properties

First, consider the properties relevant to a perspective. A perspective consists of a number of Actions and each Action has a View. The union of properties in those Views is relevant to the perspective and to our algorithm (but see the chapter Relevant properties revisited below).

10.5.4.2. Simple case: a role ladder

Next we will consider the case of ST EnumeratedRoleType bindings: they form a graph that is a *ladder*. We will descend that ladder, carrying an inverted query with us that grows on each step. We start with the context step. Remember that, in order to access a property on the ladder, the User requests a query whose first step is role R. We start with its inversion!

First step

So our first step lands us on the (the rung with) Role R with the inverted query context. We now ask ourselves two things:

1. Does R carry a property that is in the set that is relevant to the perspective?
2. Does the binding of R carry such a property?

If either is true, we store our inverted query in onContextDelta_context of R. Moreover, for any property P of R in the relevant set, we compose a query from Value2Role and context and store that in onPropertyDelta of P.

Having finished the work on this first step, we descend to the next level down. That is, we apply our function to the binding of R. Doing so, we extend the inverted query we carry by prepending a binder step to it:

binder R >> context

Next step

We now arrive on rung with Role B, the binding of R. Again, we ask ourselves two questions:

1. Does B carry a property that is in the set that is relevant to the perspective?
2. Does the binding of B carry such a property?

And we handle the answers like before. However, we would now store

```
binder R >> context
```

with onRoleDelta_binder of B, since we carry an extended inverted query. Similarly, the query we would store in onPropertyDelta of any Property of B is:

```
Value2Role >> binder R >> context
```

Now suppose we do, indeed, find a Property in the relevant set that is a property of B – or suppose that question 2 yields true. We then know the answer to question 2 of the first (previous) step on the ladder!

We make our function return true to signal that to ourselves in the waiting recursive call.

In other words, the answer to the second question is always provided by the recursive call to our function.

Final step

How does it end?

We have arrived at the bottom of the ladder when the binding of the role on the current rung is EMPTY or UNIVERSAL. We handle both in the same way. If we have no properties on the current role that are in the relevant set,

- We do nothing with our inverted query
- We make our function return false.

Otherwise, we store the inverted query (prepended with Value2Role) with the onPropertyDelta of the properties we've found. We then make our function return true.

10.5.4.3. Branching case: SUM or PRODUCT

Branching is simple. We step down once for each term in the SUM or PRODUCT. Notice that each time we carry a different inverted query (with a binder step for the term we select).

In the SUM case, all steps down must return the same Boolean value. That is because we must encounter the same properties along each path through the role graph.

In the PRODUCT case, if any step down returns true, our function must return true, too.

10.5.5. Relevant properties revisited

A modeller *may* specify a View on the Object of the perspective for each separate Action. But he is not required to specify a View. Omitting it, he signals that all properties are relevant.

Conceptually,

- The View for an action is unspecified,
- unless the modeller has given a View for the Object, or
- unless the modeller has given a View specifically for the Action.

The View given for a specific Action may either extend or limit the View given for the Object. However, if no view is given for the Object, the relevant properties for the perspective (as we've defined them above) include all properties (any more specific Views for selected Actions have no influence on the relevant properties for the perspective).

If all properties are relevant, we need not check, in our algorithm, whether, having arrived at a particular role in the graph, if any properties are in the set. By definition they all are in the set.

Rather than first traversing the role graph to collect the properties and then, superfluously, checking for each property whether it is in that set, we introduce a special case when no View has been set on the Object. We do that by introducing a data type:

data RelevantProperties = All | Properties (Array EnumeratedPropertyType)

10.5.5.1. Aspect properties

Notice that a View may refer to a Property that is defined on an Aspect of the Role it belongs to. Hence, to check whether a Property is in the Role namespace would miss those Aspect Properties. Instead, we have to collect all Properties defined on a Role and its Aspects.

Chapter 11. Queries

11.1. Binder versus Filler terminology

Unfortunately, in the implementation of the PDR we use a different set of terms for connecting roles than in the language definition.

11.1.1. Representation

In the implementation we have a member in the representation of a role instance that is named binding. This points to the role that fills it.

Conversely, from roles we keep tabs on what other roles are filled by it. These are the filledRoles (actually, the implementation uses a Dutch term: gevuldeRollen).

11.1.2. Functions: binding, binder <Role>

We have a function for traversing the link in both directions:

- binding traverses from filled to filler;
- binder <filled> traverses from filler to filled.

11.1.3. Language: fills, filledBy

A role type definition is given partly in terms of the keyword filledBy. This uses a different metaphor. The converse would be fills.

11.1.4. Relation between the two; cardinality

The translation is simple:

Implementation	Language	Cardinality
Binder filled	fills filled	Multiple
Binding filler	filledBy filler	Just one

Obviously, a role can be filled by just one other role, but it can fill many. The following figure depicts the relations.

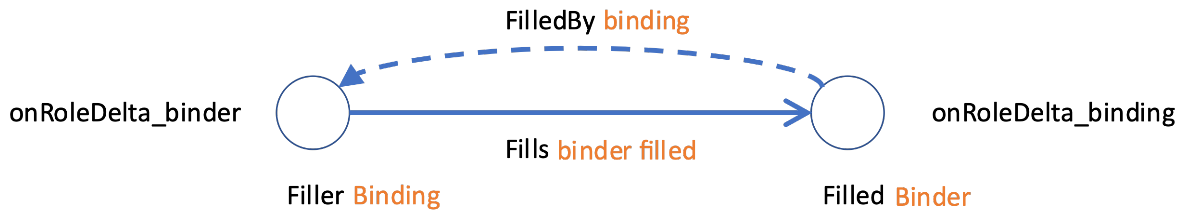


Figure 1. The two links that connect two roles.

11.1.5. Inverted queries

We store, with role types, inverted queries that lead back to the origin of perspectives (i.e. from the object of the perspective to its user role). We collect the inverted queries that start with a binder step in onRoleDelta_binder, and inverted queries that start with a binding step in onRoleDelta_binding.

The illustration shows how, for the same link between two roles, the inverted queries that traverse it are thus stored on opposite ends.

11.1.6. When we severe a connection

When the links between two roles are severed, we must trace inverted queries back to their origins to find (contexts with) users that should be informed. Two links will disappear (fills and filledBy), hence we must evaluate two sets of inverted queries:

- those in onRoleDelta_binder of the Filler;
- those in onRoleDelta_binding of the Filled.

Refer to Figure 1 for better understanding.

11.1.7. Direction: in and out of the context

A role R is attached to a context. R might fill roles in other contexts. We call a connection whose fills link departs from R *outgoing*.

Conversely, R might be filled by a role from another context. Therefore we call the connection whose fills link ends in R *incoming*.

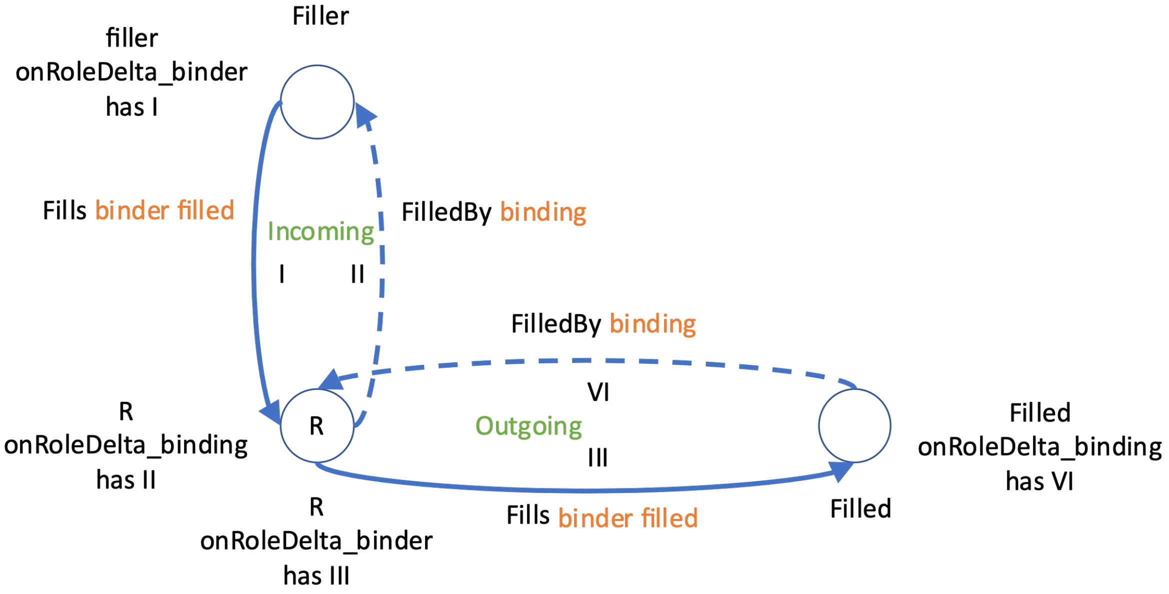


Figure 2 A single role in two relations: both as filled role and as filler role.

But, both inverted queries in:

- onRoleDelta_binding of R and

- onRoleDelta_binder of R

constitute links that move out of its context.

And, in terms of operations on R:

- filledBy moves out of its context;
- fills moves out of its context.

> In both cases: unless both roles come from the same context, of course.

11.1.8. When we remove a role

When we completely remove a role, we have more work to do than when we just remove a binding.

Removing a role gives us a direction, because it is natural to reason from the context that the role is removed from. Again, we must trace inverted queries back to their origins to find (contexts with) users that should be informed.

However, we should now trace *all links in and out of the context* through the role to be removed. Referring to Figure 2, they are:

- those in onRoleDelta_binder of R;
- those in onRoleDelta_binding of Filled (and there may be many such roles);
- those in onRoleDelta_binding of the R;
- those in onRoleDelta_binder of Filler (just the one).

11.2. State And Dependency Tracking

The runtime keeps state in an Avar in a Reader monad:

```
type MonadPerspectives = ReaderT (AVar PerspectivesState) Aff
```

Part of that state is for dependency tracking. Dependency tracking means that we recompute certain values when their input values, taken from representational state, change. This is the *functional reactive pattern*. With representational state we mean the information that is stored, in terms of contexts and roles, that represents part of the environment for end users.

We realise this by recording the relation between parts of that state, and functions that need to be recomputed.

It turns out, however, that we cannot store functions of type MonadPerspectives a in that state. Nevertheless we need to store such functions. They are kept in two caches that are stored in global, mutable variables.

The actual dependency administration can be kept in PerspectivesState.

11.2.1. Functions cached in global variables

We have three use cases:

1. A client of the PDR requests the value of a query. Whenever the underlying representation of contexts and roles changes, the PDR needs to update the query result and send it to the client (*functional reactive pattern*).
2. A bot has a rule with a condition that is true in certain states of a context instance. When the rule is triggered because of a state change, the right hand side of the rule must be executed. This right hand side consists of *assignments* and (context- and role-) *creation statements*. These change the representational state.
3. External functions. An external function can be used by the modeller by using the callExternal keyword. Such functions are defined in Purescript modules that are compiled with the PDR, but are not part of the Perspectives Language (and are, in that sense, ‘external’ to the language).

In the first case, the PDR constructs functions of the following type:

```
type ApiEffectRunner = Unit -> MP Unit
```

In the second case, the PDR constructs functions of this type:

```
type Updater s = s -> MonadPerspectivesTransaction Unit
```

Here, s is either a ContextInstance or a RoleInstance.

The third case stores structures of this form:

```
type ExternalFunction = \{func :: HiddenFunction, nArgs :: Int}
```

Here, a HiddenFunction is a type not penetrable to the Purescript Compiler.

Updaters are stored in a cache defined in the module Perspectives.Assignment.ActionCache. They are indexed by context instance identifier and action type (a so-called ActionInstance)

ApiEffectRunners are stored in a cache defined in the module Perspectives.DependencyTracking.Dependency. This cache is indexed by correlation identifiers that are communicated over the external API with clients of the PDR.

ExternalFunctions are stored in a cache defined in the module Perspectives.External.CoreFunctionsCache.

11.2.2. Dependency tracking administration

11.2.2.1. Queries

The administration for query-dependency tracking is defined in the module

Perspectives.CoreTypes:

```
type AssumptionRegister = F.Object (F.Object (Array CorrelationIdentifier))
```

The Foreign Objects are indexed by strings that represent the two elements of an Assumption. An Assumption is a combination of

- a resource (ContextInstance or RoleInstance)
- and a type (EnumeratedRoleType, CalculatedRoleType, EnumeratedPropertyType or CalculatedPropertyType), or the special identifier "model:System$Role$binding"

An assumption points to the value of a role in a context, a role binding, or of a property in a role (see [Implementing the Functional Reactive Pattern] for a better explanation of assumptions (called ‘steps’ in that text)). When changes to such values are made, the code reviews the assumption register to find correlation identifiers that identify functions that compute queries dependent on those values.

An instance of AssumptionRegister is stored in PerspectivesState under the key queryAssumptionRegister.

11.2.2.2. Object versus GLStrMap

The AssumptionRegister is created as a Foreign.Object, while the other two caches are created as a GLStrMap. The underlying representation is the same. However, the former is modified purely functional, while the latter is modified destructively. We believe that there is no functional difference between the two in the way it is used in Perspectives. The destructive operations are faster. The implementation difference reflects the current state of the program and is likely to change in the future.

11.3. Composing instance level and type level query functions

Consider the following two functions:

```
contextType :: ContextInstance ~~> ContextType

contextRole :: ContextType ~~~> RoleType
```

The former gives us the type of a context instance; the latter gives us all RoleTypes defined for a context type. A reasonable use case is to compose these two:

```
Z = contextType >=> contextRole
```

However, the Purescript compiler flags a type error. The first function is *defined on the instance level*, while the latter is *defined on the type level*.

We will explore what these two terms mean and how we can reconcile both types so we can safely compose these functions.

11.3.1. Instance level functions

Let’s examine the type of contextType. It expands according to the following types:

```
infixl 5 type TrackingObjectsGetter as ~~>

type TrackingObjectsGetter s o = s -> MonadPerspectivesQuery o

type MonadPerspectivesQuery = ArrayT (WriterT (Array Assumption) MonadPerspectives)
```

to:

```
contextType :: ContextInstance -> ArrayT (WriterT (Array Assumption)
MonadPerspectives) ContextType
```

So we see that this function computes a result in a monad stack that has MonadPerspectives at the bottom, WriterT above that and on top ArrayT. The WriterT monad transformer allows us to gather Assumptions. This is for query dependency tracking, explained elsewhere. As a spoiler: we do not track dependencies on the type level (because we do not allow type level queries through the API).

> Well, not really: MonadPerspectives itself is ReaderT (AVar PerspectivesState) Aff, but this need not concern us here as we will dig no deeper than MonadPerspectives.

Furthermore, ArrayT allows us to work with Arrays of values and still compose functions as if they yielded single values. It abstracts away *un-determinedness* with respect to functional result, as it would be put in FP terms.

Let’s turn to type level functions next

11.3.2. Type level functions

The type of contextRole expands as follows:

```
infixl 0 type TypeLevelGetter as ~~~>

type TypeLevelGetter s o = s -> ArrayT MonadPerspectives o
```

to:

```
contextRole :: ContextType -> ArrayT MonadPerspectives RoleType
```

This type is less involved. We merely get an undetermined (Array) result in MonadPerspectives. As said before, this type allows us to use Kleisli composition on such functions.

11.3.3. The problem, revisited

We can now state the problem clearly. We want to compose two functions with the following types:

```
contextType :: ContextInstance -> ArrayT (WriterT (Array Assumption)
MonadPerspectives) ContextType

contextRole :: ContextType -> ArrayT MonadPerspectives RoleType
```

The problem is that both stacks have ArrayT on top and MonadPerspectives as the bottom, but that the first stack has WriterT (Array Assumption) in between.

Notice that we apply the instance level function first and that it results in a type. This type is input for the type level computation. In other words, we move from instance- to type level. While the other way round is certainly possible, there seem to be far less use cases as we seldom ask for the instances of a type.

How do we make these types compatible?

11.3.3.1. runArrayT

We have function runArrayT to strip away the ArrayT layer:

```
runArrayT :: forall m a. ArrayT m a -> m (Array a)
```

Applying these to our functions, we get:

```
runArrayT <<< contextType :: ContextInstance -> (WriterT (Array Assumption)
MonadPerspectives) (Array ContextType)

runArrayT <<< contextRole :: ContextType -> MonadPerspectives (Array RoleType)
```

Now, remembering we want to move from the instance- to the type level, it seems appropriate to reframe our problem as: how can we lift `runArrayT <<< contextRole` to the type of runArrayT <<< contextType? So, we want to lift

MonadPerspectives (Array RoleType)

To:

```
WriterT (Array Assumption) MonadPerspectives) (Array RoleType)
```

That is actually not too difficult, we can just lift `runArrayT <<< contextRole`:

```
lift <<< runArrayT <<< contextRole :: ContextType -> (WriterT (Array Assumption)
MonadPerspectives) (Array RoleType)
```

We're almost done. We now want to abstract over the Array RoleType result, so we can deal implicitly with the un-determinedness of the result. That's what ArrayT is for:

```
ArrayT <<< lift <<< runArrayT <<< contextRole :: ContextType -> ArrayT (WriterT (Array
Assumption) MonadPerspectives) RoleType
```

Reducing this long expression using our type definitions, we get:

```
ArrayT <<< lift <<< runArrayT <<< contextRole :: ContextType -> MonadPerspectivesQuery
RoleType
```

Or

```
ArrayT <<< lift <<< runArrayT <<< contextRole :: ContextType ~~> RoleType
```

And we're done: we now have a function we can compose with contextType:

```
f :: ContextInstance ~~> RoleType

f = contextType >=> ArrayT <<< lift <<< runArrayT <<< contextRole
```

11.3.4. End result: lifting from instance- to type level

As this is a pattern of composition we will encounter quite a few times, it is worthwhile to introduce a special lifting function to abstract the pattern:

```
liftToInstanceLevel :: forall s o. (s ~~~> o) -> (s ~~> o)

liftToInstanceLevel f = ArrayT <<< lift <<< runArrayT <<< f
```

So we can rewrite our function f above as:

```
f :: ContextInstance ~~> RoleType

f = contextType >=> liftToInstanceLevel contextRole
```

11.4. The Case for Database Query Roles

One of the design goals for Perspectives is that all context- and role instances must be *reachable*. This can be attained by direct indexing (e.g. a role is directly linked to its context), alternatively by deploying an indexed name (an indexed name has a different extension (reference value) for each end user, e.g. My System), or finally by a database query that retrieves instances of a particular type.

Such a query has to be the expression by which we define a Calculated Role. To differentiate such database-query-based Calculated Roles from those that are defined by a path query, we call them **Database Query Roles** (DBQ Roles).

Database Query Roles must be context roles: their extension consists of external roles, either all instances of the type, or a filtered subset of them. The extension of a DBQ Role may contain otherwise **unlinked** roles.

Why should we have Database Query Roles?

In this text I describe use cases in the abstract and explore some of their characteristics.

11.4.1. Database Query Roles retrieve context type instances

This is the essence of a Database Query Role: a role defined as such in a context has as its extension all instances of a particular Context type (in fact, their external roles. In this text I will loosely equate a context with its external role, both on the instance and the type level) (possibly filtered). A use case might be:

- All natural persons known to a user through Perspectives (through a calculated query that retrieves User instances from their PerspectivesSystem instance);

In general, it seems that these role types might be used to avoid direct indexing if such indexing would be unpractical:

- Because there would be very many roles in one context, slowing down the system when it context must be loaded, even if just a single role would be accessed;
- Because it would be tiresome to retrieve all instances through a complex path query. This typically happens if the primary structure of contexts has many branches and the wish to have a single list of instances is only of secondary importance.

11.4.2. How Database Query Roles are different

To prevent misunderstanding: these roles need no special handling in terms of creating or deleting their instances. The normal assignment and create syntax applies. Furthermore, these roles can be part of a path query.

11.4.2.1. The operational semantics of deleting is different

We stipulate that an external role instance and its context are removed permanently from the end users bubble (the part of the Perspectives Data Universe that is accessible to him) whenever

1. It’s last binding is removed, or:
2. It is being removed (by user or bot) from a Database Query Role while it turns out to have no bindings.

The consequence of this is that when (1) occurs, no DBQ Role based on the same type will show the instance any more.

11.4.2.2. The extensional semantics of a DBQ Role

The extension of a DBQ Role is the set of role instances of a particular type (that comply with some optional filter criteria) that

1. either are bound in some role,
2. or have been added directly to some DBQ based on the same type.

11.4.2.3. Unexpected behaviour

As a consequence of its extensional semantics, these roles show some unusual behaviour. Consider the case of two different contexts (of the same type or of different types, but with a similarly defined Database Query Role). A sub context constructed in either context will be reachable through both. This is unusual behaviour: by default a sub context is only reachable through its containing context.

Still, there is no ‘privacy leak’. One might assume such a leak occurs in the following situation: a context A defines three user roles. Two of them have a perspective on a Chat role in the same context, the third role is excluded. The Chat role is defined as a Database Query Role. Now, in another type of context, B, a similarly defined Database Query Role exists and the end user that was excluded in A has a perspective on it in B. Will he be able to see, in B, the Chats that were hidden from him in A?

The answer is no. Even though this user will see (in B) all Chats that are represented in his system, *the other users never sent him their private Chats in A in the first place!* So privacy as modelled is preserved.

11.5. `selfOnly` implies `mineOnly`

The selfOnly modifier can be used to restrict a perspective of a user role on itself – a self-perspective – to just itself. Take, for example, the role of Account in context Body in model:BodiesWithAccounts: here the Account has a perspective on himself, but the selfOnly modifier restricts the available instances of Account to just one role – filled, ultimately, by himself.

This may be taken as a way to protect the privacy of Account holders; neither can see the others (note that to complete this pattern, the Account role needs to be unlinked, too. See the document *The Body With Account Pattern*).

11.5.1. Mine only!

Surprisingly, this has an interesting consequence. Suppose that, within the same context, the Account can make another role. Let’s call it a Contract, to stay within our example. Account must have a perspective on Contract to be able to create it (and therefore can see it as well).

It would, of course, be appropriate if an Account could see just his own Contract. Hence, we would like to have a modifier for Contract like ‘mineOnly’, to prevent the PDR of an Account from sending a freshly created Contract to all other Accounts.

However, pausing for a moment: what other Accounts? The very fact that the self-perspective of Account is selfOnly, means that the PDR of an Account has no information on any other Accounts. Hence, by implication, for an Account his perspective on Contract is ‘mineOnly’.

11.5.2. Other user roles

Suppose that the Admin of Body has a perspective on Contract, too. This would cause the PDR for any Account to send deltas for constructing a Contract to the Admin PDR. That is entirely in order.

But what happens if Admin creates a Contract? According to the model, Account has a perspective on Contract, hence Admin will send deltas – to the PDR of each Account it knows about. And Admin can see all Accounts.

This would constitute a privacy breach.

We see, then, that the ‘mineOnly’ effect implied by selfOnly is limited. Limited, in fact, to roles that just the user role with the selfOnly modifier has a perspective on.

11.5.3. Exploring an improvement: "only for those playing a role"

From the perspective of Admin, the restriction on Contract should be something like: only those Contracts that are relevant to an Account. But how can we make that computable? It turns out we cannot, without creating a special relationship between roles (we’d need to annotate, somehow, the Contract role with the User role that it ‘belongs to’).

But we can define ‘relevant’ for context roles. Suppose that Contract is a context role filled by a Contract context and that Account fills a user role in that context, let’s say Party (to the Contract). Then we could define context X is relevant to user role Y if Y plays a role in X.

We’ll name the keyword we’d like to use onlyWhenInvolved. It can be applied to qualify a perspective. In this case, we’d like to qualify Account’s perspective on Contract with onlyWhenInvolved.

Let’s have Admin create a Contract context (and bind it in Body in the Contract role). At this point, no user role is present in Contract, so the Admin PDR does not send any deltas. onlyWhenInvolved prevents the it from doing so. It first notices that Account has a perspective on Contract, but then finds no user role in Contract, so no Account is involved.

But as soon as Admin creates an instance of the Party user role in Contract *and fills it with a particular Account*, the PDR should send deltas to that Account. How is that done?

It starts with filling Party in Contract with Account. Party will have some perspective in Contract (otherwise it would not be a user role). For simplicities sake we can assume it has a self-perspective. The PDR of Admin will find that deltas for the Party role instance should be sent to the PDR of the

Account. It never sends a free-floating role, so the Contract context is sent, too, complete with its external role.

At this point, things will stop. The Contract *role* in Body is not sent to Account, unless we activate a special mechanism.

What can that be?

PART III. IMPLEMENTATION DETAILS

This is the most technical part of the documentation.

Chapter 12. PDR architecture

12.1. A Transport Layer for the PDR

The Perspectives Distributed Runtime (PDR) interprets models in the Perspectives Language for clients that display the current state of the Perspectives Universe to a particular end user (that part of the Universe that falls within his *horizon*). It also enables them to change that state. The Perspectives Universe is persisted as data saved on disk in an extremely distributed way – each user keeps his own stuff. Because users' horizons overlap, their data collections overlap, too. Because of the overlap, the PDR has to *synchronise* state with the PDR's of relevant peers. This synchronisation – considered on a technical level – necessitates the exchange of *deltas* to the state, in collections called *Transactions*. Usually, the PDR's of various end users operate on different computers, connected to each other on the internet. So an important architectural decision concerns the *transport protocol and mechanism* to use for exchanging Transactions.

In this text, we report on an exploration of the alternatives and the roadmap we follow to connect PDR instances.

Before we start, we want to point out that Perspectives can be considered to be distributed on the *application level* without necessarily having to rely on a distributed transport layer. Having written that, 'application level' is somewhat inappropriate for Perspectives, as we have no clear notion of application. Usually an application is considered to be a functional unit of deployment with a clearly defined collection of data. This does not apply in a clear way to Perspectives. Nevertheless we want to make that distinction because – spoiler alert! – we initially choose for a Transport Layer that relies on a client-server model. This does not, in any way, compromise the distributed character of the PDR on the application level.

12.1.1. The problem to be solved

PDR's, considered as nodes in a network, cannot be expected to be online all the time. They are end user devices! Moreover, the network itself cannot be expected to be 100% reliable – especially not so for mobile devices.

Nevertheless, a Transaction sent must arrive at its destination (to avoid misunderstanding: each Transaction is destined for exactly one other PDR). We cannot solve this problem on either the sender's node or the receiver's node. Even while the sender might notice that the Transaction it sent hasn't arrived, saving it to be sent for a later time is not a full solution. It may happen that sender and receiver are never online at the same time!

We would like to solve this problem in the transport layer itself.

12.1.2. Roadmap: distributed or not?

There is no doubt that we want, in the end, to build the PDR on a fully distributed transport layer. This would probably be a mechanism that connects each peer to each other peer on the level of TCP/IP. However, as it is, the current state of the internet makes this no sinecure. The problem can be stated simply as a lack of directly addressable nodes: most of our devices (laptops, desktops, tablets and sometimes mobiles too) root in some private network, having an address that cannot be reached through the open internet. More technical, these devices are behind routers that assign local addresses to them and prohibit or sincerely hinder direct addressing from the other side of that router.

To be able to proceed the PDR development without getting bogged down in this very technical problem, we will build the PDR on existing transport protocols and software instead. This does not preclude the other approach, leading to a full peer to peer stack as far down as we can push it; we merely postpone it to the future.

12.1.3. Current state

At the time of writing, the PDR relies on Couchdb for transport. This consists of series of databases that we let Couchdb synchronise. In short:

- Each pair of connected peers share a unique database, their *Channel*;
- Both users have a local copy of that database;
- There must be a server, accessible to both, that has a copy of that database, too.

Thus, a Transaction is written by A into A's copy of the Channel; replicated by Couchdb to the server copy; replicated by Couchdb to B's copy, where it is picked up by B's PDR. Actually, all local channel copies replicate to the *incoming post* database for the local user and the PDR listens to the stream of events on that aggregate

This is secure and robust and does the job. It is, however, not particularly fast. In the end, we abuse a database to function as a messaging mechanism. Of particular concern is the question of scalability. For n users, each with an average of m connections, n x m databases must be kept in synchronisation with remote copies. While Couchdb is built for database synchronisation, emulating a messaging system no doubt was not one of the use cases its designers had in mind.

12.1.4. Alternatives

Messaging systems – or message queue systems – are available in many kinds. Two spring out as likely candidates for our purposes: XMPP and AMQP. Projected onto our needs, both do the job. However, they differ substantially from each other. XMPP was designed for human to human text message exchange. AMQP was designed for program to program message exchange, clearly closer to our situation. In what follows, we go into some detail of various quite different facets that have played a role in our decision making.

12.1.4.1. Availability in the browser

XMPP is readily available in the browser in the form of various browser-Javascript libraries. This is not so for AMQP. However, there is another protocol, STOMP, for which browser-Javascript libraries are available and that is 'spoken' by important AMQP implementations like RabbitMQ. STOMP is a simplification when compared to AMQP but rich enough for our purposes.

So both systems can be used by a browser-based application.

12.1.4.2. Who controls the infrastructure?

Being motivated by the desire to prevent undue power that falls to server owners, we should be careful lest we lock our system in a monopolist on the level of the transport layer.

Both protocols are public. AMQP (https://www.amqp.org/) is ratified by the IEEE; XMPP (https://xmpp.org/) is maintained by the XMPP Standards Foundation (also known as the XSF). Both protocols have been implemented as Open Source software (for AMQP there are quite a few robust implementations; for XMPP the de facto standard is Jabberd (https://jabberd2.org/).

However, software is one thing, running the necessary infrastructure another and here XMPP and AMQP go different ways. XMPP is run mostly by volunteer organisations. There are quite a few and some of them have a good track record. Nevertheless, one cannot expect a guaranteed service level of these organisations. Moreover, Perspectives users would probably compare unfavourably with those who use the service for its intended purpose (chat): they would send messages at a far higher rate. Depending on these free services for the Perspectives Transport Layer might be unwise.

> For a curated list of awesome XMPP servers, libraries, software and resources, go to: https://github.com/bluszcz/awesome-xmpp. For a lost of public XMPP servers, see this (curated list.

We have found two commercial XMPP service providers:

1. https://tigase.net/xmpp-server
2. https://fluux.io

For AMQP the situation is different. AMQP is used intensively for professional, industrial applications and hence there is a mature industry of service providers. An example (and market leader) is https://www.cloudamqp.com/.

12.1.4.3. Who makes the choice for a service provider?

Is it us, the designers of Perspectives, who choose a Transport Layer service provider for all future Perspectives users? Or can individual users choose different providers?

We would like the situation to be like with email providers. Two end users, signing up with different providers, should be able to connect.

This is guaranteed with XMPP: the system was designed with this use case in mind. Not so for AMQP. However, we will use AMQP much like a postbox service. Knowing the IP address (and postbox, or, in the parlance, queue identification) should be enough to deliver Transactions. So in theory, a PDR could drop Transactions with several providers. In practice, however, it would have to authenticate with each of these providers. This may prove to be a problem. We judge it not to be insurmountable.

Why authenticate to drop a message?

Let’s start out by stating that a PDR examines each Transaction, checking that it has indeed been signed by the claimed sender (authentication). It will further scrutinize all deltas, checking that they were performed by a user with the required role (authorization).

The transport layer is not responsible for either authentication, nor authorization.

However, we can imagine a kind of Denial of Service Attack where a malicious agent drops overwhelming numbers of Transactions on a single PDR. This is what authentication at the service provider would discourage (as such attacks could be traced to a known user).

12.1.4.4. Transport Layer user administration

If we rely on XMPP, we expect Perspectives end users to sign up to some XMPP provider and enter their credentials in the PDR, so it can use the account to send transactions.

However, if we rely on AMQP, we must handle the signup process ourselves. As a matter of fact, we – Perspect IT - must act like a value-adding service provider ourselves. AMQP is not free and providers contract clients based on high volumes of transactions. A client of such a service provider must set up an ‘exchange’ and can then provision *its own customers* to use the service.

In terms of the Roadmap, we don’t have to start exploiting a service commercially straight away. A provider like CloudAMQP has a free plan that offers up to 100 queues, translating to 100 connected devices for Perspectives. Above that number, we’ll have to pay

> $19,- per month, to be exact, in 2020. This is not an insurmountable problem in the short run. However, it makes clear, too, that, in case of (exponential) success, we must quickly start charging customers!

12.1.4.5. Matureness of the technology

Both AMQP and XMPP are very mature technologies. For both excellent documentation exists. However, AMQP is used at far bigger scale with much higher message throughput that XMPP.

12.1.5. Proposed solution

I propose to build on AMQP

1. It was designed for a use case like ours (messaging between applications).
2. There is a mature service providing industry, offering managed services.
3. It does not technically lock us in with a specific service provider.
4. There is good software support in the browser environment.
5. Excellent documentation is available.

Admittedly, for XMPP points 3, 4 and 5 hold, too. It is points 1 and 2 that make the difference.

12.2. STOMP

This text elaborates the previous chapter A Transport Layer for the PDR. In it we describe in some detail the technical design decisions underlying the implementation of the message layer.

12.2.1. Technology chosen

The implementation is built on RabbitMQ (https://www.rabbitmq.com/documentation.html) and Stompjs (http://jmesnil.net/stomp-websocket/doc/). The latter library caters for Stomp version 1.0 and 1.1, not 1.2. The Stomp web plugin for RabbitMQ handles all versions.

12.2.2. Exchange type

While peers may use the Perspective User Identity (PUI) to send Transactions to, only the intended receiver must be able to subscribe to the relevant queue on the AMQP server. To achieve this end, we use a Topic Exchange where

- the routing keys are PUIs;
- the binding keys are PUIs, but only the PDR for a particular PUI knows the identification of the queue that binds that PUI.

Note that not even the server administrator has to know the queue that uses a particular PUI as binding key. The subscribing PDR can keep it to itself.

A Topic Exchange matches routing keys to binding keys, where the latter may include wildcards. We have the simplest possible situation, where both keys are equal. It is the binding rule that connects the key to a particular queue, that protects the receiver from others marauding his post box!

12.2.3. Creating topic queues

The web client using Stomp can create a queue with a particular binding key in a Topic Exchange; we don’t need the RabbitMQ administrator for that.

It turns out that when a new vhost is created using the management console of RabbitMQ, all types of Exchanges are created for it. Stomp sends a frame with a destination string that starts with “/topic” automatically to the amq.topic Exchange of that vhost.

A queue with a particular binding key and queue identification can be created from the client as follows:

```
const \{id, unsubscribe} = client.subscribe(

"/topic/" + topic,

function(message)\{…}, // handle the message

\{ durable: true

, "auto-delete": false

, id: "secred-id" // the secret queue identification.

});
```

Notice the fields in the object that is provided as last argument to subscribe. They specify that, apart from the queue identification, the queue is not to be deleted when no one subscribes to it and will be available after the server restarts, too.

This behaviour is governed partially by the semantics attributed to the *destination string* that, by default, Stomp assigns neither structure nor semantics to. For RabbitMQ this is described in https://www.rabbitmq.com/stomp.html.

12.2.4. Acknowledgements

We don’t want Transactions to get lost. To prevent the RabbitMQ server from deleting a Transaction before it has been handled by the receiver, we make the receiver send explicit acknowledgements.

By default the server removes a message after it has been delivered. To change that behaviour we give the object supplied as third argument to subscribe with another key:

```
ack: ▯client▯
```

Now the subscribing client has to acknowledge the message, using the function that is the value of the field ack on the message that is received.

12.2.5. Heartbeat

By default, the Stomp server sets up a heartbeat (RabbitMQ by default sends a beat every 10.000 milliseconds). However, as we have the client send explicit acknowledgement, it seems not necessary to have a heartbeat on the socket level. This is how to disable it:

```
const client = Stomp.client(url);

client.heartbeat = \{incoming: 0, outgoing: 0};
```

12.2.6. User accounts

The RabbitMQ manager must create user accounts; there is no self-registration.

12.3. Configuring Couchdb

Starting with version 8.0, the screens that provide a perspective on contexts are run as ordinary webpages in a browser. MyContexts is from now on a Web App.

> A Progressive Web App, to be precise. However, there is not much progressive to MyContexts: it either works or does not work in your browser, there is no notion of graceful degradation.

This has many advantages, one of them being able to have multiple windows open side by side. Each page communicates with the same Perspectives Distributed Runtime and this PDR executes in a *service worker*. Standards require PWA's to be served over the secure protocol https and as a consequence, the PDR can now only make https calls.

The PDR stores data in Couchdb, which operates as a local webserver. Consequently, starting with version 8.0, Couchdb must be accessible over the https protocol (otherwise the PDR cannot reach it). This requires some extra configuration.

This situation will change again with coming releases, as we move away from Couchdb to IndexedDB. From then on you will be able to run MyContexts without these extra configuration steps.

If you are familiar with concepts like https, encryption and certificates, you can skip the next section to the procedures that follow. However, if you are not, we encourage you to read on.

12.3.1. Transport Layer Security

A browser uses the hypertext transfer protocol to fetch pages and other resources from a webserver. Websites are run by people or organizations, but the web can be an unsafe place and it has happened that unsuspecting users were deceived into believing they communicate with a party while in reality they were dealing with imposters. To mitigate this risk, many websites now serve their resources over https: the http protocol with added 'transport layer security' (TLS). This serves two purposes:

1. To prove the identity of the parties involved;
2. To protect the information they send to each other from eavesdroppers between their computers.

In practice, a webserver proves its identity by sending a *certificate*, while the end user usually has to sign in with a username and password.

12.3.1.1. Protecting information

Let's focus first on the second purpose, protecting information. This is achieved using cryptography.

The particular form of cryptography used in TLS relies on two ‘keys’: a public one and a private one. The webserver sends the public one to the browser; it is part of the certificate.

Now the imagery of two keys is not very enlightening. It is better to consider the public key to be a kind of *box*, that can only be unlocked by the (private) key. So the server sends a box to the browser; when the user wants to send some sensitive information to the server, the browser puts it into the box, clicks it shut and sends it back. The server can open the box with its key. No one intercepting the box along the way can open it, as they do not have the key. This is the practice of encrypting information to communicate it privately.

But wait! How do we know that the box received by the browser is, indeed, the one sent by the server? If we reckon with interceptors, what if a crook should insert his *own box* in the response sent by the server!? The browser puts the sensitive data in it and gone is the users secret, as the crook intercepts the reply and opens the box with his own (private) key.

So the webserver should prove its identity, or, rather, prove that the public key (box) it ships with its responses is indeed owned by it (the first purpose of TLS). This is where the certificate comes in, and with it the notion of *signing*.

12.3.1.2. Proving identity: signing

The King’s signature on a banknote should convince people it is genuine. Only the King can create that signature and all can verify that, the King being a well-known public figure! Obviously, long gone are the days that all we needed was that signature, but the principle is clear. This is signing: only one party can put their sign on something and all others have the ability to verify it.

Now if this sounds familiar, you are right. It is just like protecting information, but the other way round:

- Everyone can put something in the box, but only one person can open it;
- One person can sign, but everyone can verify the signature.

The roles are reversed, or, rather, the keys are switched. On protecting information, the (private) key holder sends out the boxes (the public key). On signing, the (public) key holder puts something in a box, and everyone holding the (not so) private key can open it.

Because we can open the box with the King’s key, we know for sure that he is the one that put something in it (cause he keeps the unlocked boxes (=the public key) private). So signing requires distributing the (private) key; protecting requires distributing the box (public key).

12.3.1.3. Chains and authorities

To sum up where we stand: to protect information flowing from the browser to the server, we encrypt it using the public key sent by the server (the box). To make sure that public key is really owned by the server, it is signed and so becomes a certificate.

But, you may observe, haven’t we just pushed the problem a bit further away? For what is checking a certificate other than using a key (the private one, this time, issued by someone who certified the servers’ public key)? *So how do we know that key is genuine?*

In our minds eye we can see a chain of certifiers, each certified by someone higher up in the chain. As a matter of fact, this is exactly how it works. But the buck stops somewhere and that is with a certificate we call a *root*, issued by an *authority*.

Who decides who is an (certificate) authority and who not? This is largely the outcome of a historical process. There is now a smallish number of certificate authorities (CA's) that are regarded as trustworthy.

12.3.1.4. Root programs and you

You will have accessed countless websites over https and yet never spent a conscious thought on which CA you can trust. So when exactly did this happen? When did *you* decide to trust, say, the certificates signed by VerySign?

Well, you did not, otherwise than by using your browser and the list of CA's trusted by the *Root Program* active on your computer. Such lists are compiled by the suppliers of operating systems, such as Apple and Microsoft.

> There are a few other Root Programs. Most are by private companies but the Mozilla organization runs a well-known and public Root Program, too.

When you use a Mac, you can inspect that list using the KeyChain App (it keeps your passwords safe, but it also holds the list of CA's and certificates to trust).

To sum up: when you access a website protected with TLS, your browser receives its certificate, that holds a public key it will use to encrypt your information prior to sending it back. Before doing so it checks the certificate is signed by an authority that is in the list of authorities kept in the operating system of your computer.

12.3.2. Configuring https for Couchdb on your computer

The purpose of the procedures outlined below is twofold:

1. To make the Couchdb webserver send out a certificate with responses to requests coming in over https;
2. To make your browser trust that certificate (by adding it to the computers list of trusted certificates).

Only then will MyContexts version 8.0 and higher run properly on your computer.

12.3.2.1. Obtaining the certificate

You can create a self-signed certificate if you wish. The *common name* field in it should be https://localhost:6984. This is the secure endpoint of Couchdb on your computer.

Alternatively, you may download a certificate for this endpoint signed by Perspect IT from https://mycontexts.com/couchdb.pem. And https://mycontexts.com/privkey.pem. Notice that Perspect IT is not a certificate authority. Neither do you have to trust us; you will have to declare trust in the certificate yourself. You can safely do so, as it only certifies an endpoint on your own computer (Couchdb, that you have installed yourself).

12.3.2.2. Storing the certificate

Following the instructions for configuring https options in Couchdb (see below), put the certificate in the directory /etc/couchdb/cert. You probably may be able to store the files in another directory. If so, adapt the lines in the ssl section in your local.ini file accordingly (see below). Make sure the directory where you put the files is readable by the Couchdb process!

12.3.2.3. Configuring Couchdb

The official Couchdb documentation describes https options in https://docs.couchdb.org/en/latest/config/http.html#https-ssl-tls-options. From that section we've copied the relevant steps. Before carrying them out, stop Couchdb; after finishing, restart again.

```
Now, you need to edit CouchDB□s configuration, by editing your local.ini file. Here is
what you need to do.

Under the [ssl] section, enable HTTPS and set up the newly generated certificates:

{empty}[ssl]

enable = true

cert_file = /etc/couchdb/cert/couchdb.pem

key_file = /etc/couchdb/cert/privkey.pem
```

Where to find the Couchdb configuration files is described here: https://docs.couchdb.org/en/stable/config/intro.html#configuration-files.

12.3.2.4. Make the browser trust the certificate

If you use Firefox, open the Certificate Manager (Tools > Options > Advanced > Certificates: View Certificates).

If you run Chrome or Safari on Mac OSX, open KeyChain Access. Select the Certificates Category. Drag and drop the couchdb.pem file on the right. Double click it and change the settings to “Always trust”.

We have no instructions for Microsoft Windows.

12.3.3. Apache proxy workaround for Couchb tls problems

It turns out that Couchdb does not add the qualification “Secure” to cookies sent over https. The Chrome browser requires that qualification (together with SameSite=None) before it accepts cookies from third party domains. As a workaround, we skip the TLS in Couchdb (it cannot effectively connect to most browsers, anyway!), handle it in Apache and rewrite the cookies to add Secure to it as well. Below is a working configuration:

```
<VirtualHost *:6984>

ServerName localhost

ProxyPass / http://127.0.0.1:5984/

ProxyPassReverse / http://127.0.0.1:5984/

SSLEngine on

SSLCertificateFile "/private/etc/apache2/ssl/127.0.0.1+1.pem"

SSLCertificateKeyFile "/private/etc/apache2/ssl/127.0.0.1+1-key.pem"

Header edit Set-Cookie (.*) "$1; Secure"

</VirtualHost>
```

12.4. Multiple databases and devices

We can analyse the MyContexts application in terms of three parts:

- an interface (graphical, as it is, but not necessarily so),
- a database
- and a component that sits between the two.

This latter component

- receives instructions given by the end user through the interface on how to modify, delete or extend contexts and roles;
- saves new versions of those contexts and roles to the database
- compiles and sends transactions of deltas for each peer that should be informed about those changes. A *delta* is an atomic change to a context or role; however, some deltas must be carried out together in order for the data to remain consistent, hence the notion of *Transaction*.

In this text we focus on the question: can an end user be meaningfully supported by a constellation of more than one UI, database and middle component? This will concern mostly the transaction handling aspect of that component, so here we call it the *Transaction Handling Point* or THP.

> The Perspectives Distributed Runtime is a THP; but a THP need not be a PDR, as we will see later.

12.4.1. A refresher: structural connections in Perspectives

Before we delve into these subjects, let’s quickly recap some aspects of the representation of Perspectives data. To start with, there are only two entities: contexts and roles. They are connected in just two ways:

1. Contexts have roles;
2. Roles fill other roles.

However, because we want to traverse these connections efficiently in both directions, we represent each by two links:

1. In a context, we point to the roles, and
2. a role contains a pointer to its context.
3. In a role, we point to the role that fills it, and
4. a role also contains pointers to roles that are filled by it.

And that’s it! In this text we will speak of the *roles of a context*, the *context of a role*, the *filler of a role* and the *filled roles of a role.*

12.4.2. Multiple databases

To prevent misconceptions, please remember that each user keeps his own stuff. That is, a database of entities *just contains the entities as seen by a single user!* Their peers have a database, too, and they will contain their versions of the entities accessible to both.

InPlace v0.7.0 stores all data in a single Couchdb instance on the local computer of the end user. What might be reasons for changing that? Here are three use cases:

- MyContexts-in-the-cloud: a user decides to store his data with a service at some endpoint on the internet, for reasons of availability, safety, whatever.
- A home server, with more capacity than, say, a mobile phone.
- The wish to store some entities, e.g. relating to a particular domain such as insurances, with a third party while keeping other data local.

This requires a change in the nature of the links themselves. Currently, they merely *identify* entities; but we need them to *locate* entities. It is the change from an URI to an URL.

To make it practical, more is required. A user needs safe and private access to endpoints that keep his data and that will require authentication and possibly encryption. This is all usual stuff; however, the THPs need to be able to authenticate and de- and encrypt on their users’ behalf.

Furthermore, a remote database may be unavailable for some time. The THP should be able to handle this. This has repercussions for the current implementation because it is built on the assumption that an entity identifier can always immediately be exchanged for the representation of the identified entity and that would no longer be the case.

12.4.3. Linked and non-linked connections

As stated above in *A refresher: structural connections in Perspectives*, each connection is represented by two links so we can traverse connections in both directions. We left unsaid that there is an asymmetry in numbers: a single context can have many role (instances), but each role instance has just a single context. Similarly, a single role is filled by just one other role, but each role can fill an unlimited other number of roles. So, the cardinality of each connection is 1 to n.

There may be use cases where directly indexing (linking) all role instances in a context, or directly indexing all filled roles, becomes impractical. Consider, for example, a witnessing service where a single witness is involved in millions of financial transactions. Clearly, we should not require the PDR of this witness to load as a single entity these millions of references! Similarly, for the other kind of connection, think of a model repository where many thousands of users want to have a role in a particular model context, as end users that want to be notified of updates.

Instead, we want to be able to represent some links with a *database query*. To find all roles filled by role X is conceptually straightforward and, indeed, it is the type of query that every database system is optimised for. We will call such links *Database Query Links* (DQL).

12.4.3.1. Accessing DQLs

Traversing the connection between roles in the direction of the *filled roles* would occur, in Perspectives, in the form of a Calculated role that has the keywords *filled by* in its definition (a path query). As the typical use case for a DQL involves high volumes, it is desirable to limit the number of responses that come from the database. This is usually done by deploying two different techniques:

- paging, i.e. retrieving a set of consecutive results on user demand;
- filtering, i.e. just returning results that conform to some criterium

or a combination of both.

We can envision paging keywords as an extension of the Perspectives Query Language. Role properties would parameterize the paging query.

Similarly, we can conceptualise a different compilation of the existing filter expression, where the criterium would translate to a query carried out on the server. This is a well-defined, albeit possibly complex technical extension of PQL.

12.4.4. Multiple Transaction Handling Points

Two peers synchronise the representation of the state of entities located within the Perspectives Universe that falls in their respective horizons. Their Transaction Handling Points construct and process deltas and ship them in transactions.

> But again, note that synchronization does not imply they send copies of these resources to each other. They send deltas that deal with each other’s perspectives.

A THP is also the place where bots execute actions on behalf of their users. Now reconsider the use case of witnessing. The actual actions that should be carried out by the witness lend themselves perfectly to bot handling. As a witnessing service grows, the bot should scale with it. Above a certain volume it will not be practical for the THP of the witness (the human) to handle all those transactions: the bot will compete for resources with the actions the human user wants to carry out himself.

So here arises the wish to have some types of transactions to be handled by another processing unit: a THP dedicated to a few types of transactions. In other words: we want the witnessing bot to run on some server or in the cloud and not on the laptop of the witness himself!

The PDR (as our only current THP implementation) looks up, for each delta, to what THP it should be sent. It finds that information with the User of the System and it finds this system user by following *filled by* links, starting with the role in the context where the delta applies. In our example this would be a financial transaction, where the witness role would be filled by a ‘telescope’ of roles that bottoms out in the system user of the employee designated responsible for the service by his employer.

To paint the picture in a little more detail, let’s assume that we have a company context (WeWitness), with Employee roles, with a WitnessEmployee role in the actual WitnessingDepartment, with WitnessContract contexts with a customer role and a Witness role. We then would have the following telescope:

- Witness in FinancialTransaction
- Witness in WitnessContract
- WitnessEmployee role in the WitnessingDepartment
- Employee role in WeWitness
- User role in System.

Conceptually it is simple to add THP properties to the WitnessEmployee role that will *shadow* the THP properties in the User role. So when the PDR of a client looks up the THP properties starting from the FinancialTransaction, it will find the location of the THP that is run on WeWitness’s high volume servers. The transaction of deltas will be sent to that server.

12.4.4.1. A THP that is not a PDR

The service that executes the bot actions on behalf of the WitnessEmployee needs not be implemented as a full Perspectives Distributed Runtime. We know, by design, exactly what the form of the transactions is that it receives and that it must send. The actual processing is extremely simple. So we can write such a service in any language, as long as it accepts and produces the correct transactions for this use case.

Such simplifications will bring great scaling benefits.

12.4.4.2. A use case for Database Query Link

As the number of financial transactions grows, it will soon be impractical to represent the filled roles link in the WitnessEmployee role with a list of all those filled Witness in FinancialTransaction roles. So here we have a perfect use case for a DQL.

12.4.5. Multiple User Interfaces

Actually, the PDR v0.7.0 is fully prepared for multiple GUI’s. In the near future we will have a version that runs the PDR as a Web Worker in the browser. Pages on multiple tabs or windows will then be as many clients for that single THP.

12.4.6. Multiple devices

Useful as multiple UI's can be, in order to be able to use MyContexts seamlessly on both laptops and mobiles, for example, another setup is required. One such setup would be to run a THP as a service and just run the clients locally. While this is perfectly defendable, scaling the service might become a challenge as a generic THP is quite resource-hungry.

Another setup would be to run UI and THP locally but store data in a place that can be accessed from multiple devices. Again, a commercial service on the internet might be a solution; but so might be a home server.

12.4.6.1. Database subscription service required

However, this introduces another functional requirement for the database (and some extra handling by the THP). This is because MyContexts operates according to a *what you see is what you have* principle. To be more precise: the contexts and roles visible on screen are guaranteed to be a faithful representation of the state of that part of the Perspectives Universe.

Now consider the situation where an end user enters, for example, an utterance in a chat, using his mobile phone. Were his laptop to show the same chat, his words would not appear automatically there, too. This is because the update of the GUI is driven by transactions. Let's delve into this a little deeper.

When chatting on his mobile, the end user clearly wants to receive his answers there – not on his laptop. We have seen above that we may specify different THP's for various user roles, but in the end a transaction is sent to just one such THP. So the mobile should receive the chat-related transactions while the end user is on it – and that leaves the laptop in the dark.

We can remedy that situation by requiring the database to offer a subscription service that provides the subscribing THP with a stream of modified, new and deleted entities. On receiving such a changed entity, a THP would compare the arriving entity with its existing version, work out the difference in terms of new deltas and then check its dependency administration to see if one of its GUI clients has subscribed to a query for which the delta is an assumption.

So, to return to our example setup, the laptop would subscribe to the database and be informed of new utterances. As its GUI client has an active query regarding these utterances, the PDR would re-run the query as soon as the database sends a new one.

12.4.6.2. What is the active THP?

Notice that a new notion has crept into our discussion: that of a THP that can be the intended receiver of transactions or not, *based on the device the end user is handling*. This is a subtle notion and its implementation is not trivial. For what is 'handling'? When two devices are simultaneously active, how should peers know where to send a transaction?

There are no easy answers here. For the time being, we will satisfy ourself by a simple priority list, combined with a notion of being available or not. Each peer will come to send transactions to the same THP, based on this approach.

Notice that while the priority list is stable and can suitable represented in Perspectives itself in terms of contexts and roles, this is not the case for availability. Preferably the channel we use for sending transactions would handle this issue(see [State and Notification] for a notion of *availability* that could be useful for this issue).

12.4.6.3. Analysis: why each transaction is handled only once

Could the same transaction be handled by multiple THP's? Executing a same delta a second time is an idempotent operation, so at first sight it seems harmless to have two THP's handle the same transaction (but notice that storing the same value twice in a database need not be idempotent!). But changes can trigger bot rules and thereby change the Perspectives Universe again. Care is taken, in the design and implementation of the Perspectives language, to provide a clear and deterministic semantics to these operations. However, having multiple processors carry out the same process, interleaving the results in unpredictable order (due to varying transport times) probably really complicates these issues. And each THP will compile the same transactions and send them on to the relevant peers, where the process might trigger yet more consequences, etc.

For that reason, we will require that just a single THP handles a particular transaction.

12.4.7. Modelling freedom

The exploration above gives us considerable leeway for modelling, because we can see several techniques on the horizon that will make models practical that otherwise might seem wieldy and inefficient. The model repository is an example. The most straightforward way to describe the exchange and exploitation of models would give the modeller and the end user a role in some context that represents a particular model. That context would obviously also include a role for the model itself, in its various versions. We would further include some constructs to record end user payments for the model, etc.

But this would quickly become a burden for the author of a popular model. His THP would become buried under thousands of end user in the role of client. Update information would have to spread to them, invoices, etc., in volumes that could grow into a problem for the modellers laptop.

We now know that we can 'offload' such transaction handling to dedicated servers. This is exactly the business use case for repositories as a kind of app store, their raison d' etre.

In other words, by implementing the above features, we further separate the conceptual modelling of co-operation from issues of scaling and deployment.

12.5. Error Handling

Purescript offers the possibility to **catch** runtime errors. Having caught an error, one must handle it. The aim of handling is to protect the end user from situations that he cannot handle himself. Ideally, this means that the end user can continue to use the program as if nothing had happened. This may for example be the case when the source of the error was unavailability of the network connection. The end user may than be advised to restore the connection and/or try later. In general we try to inform the user that the action he initiated could not be completed. We also try to inform him on how to behave to remedy the situation that has arisen.

In the future we may add functionality to send these errors to developers, if the end user chooses so.

We have a second objective when handling errors and that is to provide good debugging information. This means not too low in the function call stack (it is of little use to know that the error occurred on accessing the database), neither too high ('something went wrong when you tried to open this screen' is not very useful, either).

12.5.1. Sources of errors

An obvious source of errors is the programmer. The purescript compiler catches most of these errors before the program can run, thus preventing them from becoming runtime errors. Nevertheless, some programming mistakes may persist and cause runtime errors. By testing we try to eliminate them all. We do not try explicitly to handle such errors.

The program relies on secondary services for its normal operation, such as a database and an internet connection. These are a major source of runtime errors: the services may be unavailable, or may malfunction. In the case of the database, some artefacts may be missing, for example. We try to catch and handle all these potential problems.

12.5.1.1. Fetching type instances from the database

In particular, the PDR is written on the assumption that a resource identifier (such as e.g. a role or context) can always be exchanged for the resource itself at the database. This assumption may be violated by malfunctioning of the database or because someone has removed a resource by other means than the PDR. Consequently, we have protected each and every access to the database by error catching- and handling code.

12.5.1.2. Fetching models from the database

A special case of resources taken from the database are the documents that represent type information. A model file (written in ARC) is translated into a DomainFile (a JSON document). These resources are taken from the database just like context- and role instances. We do protect functions that access these resources by error handling code.

What if we cannot exchange the identifier of a model for a DomeinFile, in the database? When this happens on handling incoming deltas, or JSON structures that represent resources coming in as dropped files or as fragments sent in through the GUI, we try to retrieve the model from a repository.

All other cases constitute a serious error condition. It is an error of a type that we can only protect the PDR against on a high level: that of processes that rely on model reflection but can be skipped entirely without compromising the state of the PDR.

Probably this will go all the way up to the end user, or his bot. We may not be able to answer a request by the end user, or to perform a side effect ordered by him. Practically, this means catching the error in the API itself. Similarly, we may be unable to compile a bot rule. Obviously we should inform the end user about such a situation.

Providing good debugging information is difficult for these cases. The moment we signal the error is not the moment it was caused. Neither does it matter much to know why we were looking up a type definition; it is the fact that the model is missing, that matters.

Fetching a model when handling an incoming transaction

It turns out we only need to check for a missing model when a UniverseRoleDelta arrives. A UniverseContextDelta is always preceded by a delta for its external role and the type of the external role and the type of its context are invariably in the same model. A property delta refers to a role that must be present, hence a role delta always precedes a property delta that would require a new model. Even if the property was defined in an aspect, we would have imported the model that defines that aspect as a dependency of the model that defines the role.

12.5.1.3. Looking up type definitions in a model file

Should we check, in the source code, if we can find a type name in its model? Or do we assumethat type names can always be found?

To answer this question, we must first establish the source of the type names that are run through these lookup functions. There are four such sources:

1. They may be hard coded, i.e. be part of the program source code.
2. They may occur in model source texts.
3. They may occur in DomeinFiles.
4. They may be a type reference in a context- or role instance.

Hard coded type names are the sole responsibility of the programmer. We should not include error checking in code to catch such programming mistakes!

Handling unknown type names in model source texts is the job of the parsing system.

Type names should only be included in DomeinFiles if they are defined in the model source files. Again, this is taken care of during parsing a source file and translating it into a DomeinFile.

That leaves us with type references in instance data. We have addressed this problem in the text model versions and compatibility. To summarise the discussion there:

- by using persistent internal names in DomeinFiles and as type references in instance data, we protect ourselves against errors that would otherwise arise when type names change in successive model versions. In other words, even if the readable name of a type has changed in a model, we can still look up the type definition in that new model version by using the internal name of the type.
- by including version identifiers in type names, we can recognise incompatibility of incoming data (deltas) with the version of the model that is being used locally. Only type references that can be looked up in the locally available DomeinFiles will pass the screening at the gate performed when handling deltas.

So we can finally answer the question: Should we check, in the source code, if we can find a type name in its model? with a resounding: NO!

12.5.1.4. Authentication errors

The PDR relies on two services that require authentication: the (local) database and the message broker.

Couchdb has a notion of session with a time out period. We want the PDR to authenticate automatically if necessary, after the initial login by the end user (once the end user has authenticated himself with the PDR). This requires we catch messages returned by the database server to the effect that the session has expired.

The same holds for RabbitMQ, the message broker. However, the library we use to connect to RabbitMQ handles this by itself.

12.6. User and System Identifier

sys:MySystem and sys:Me are replaced by unique ‘private’ names, like all other indexed names, but we handle them a little different. This is because we have to generate a unique name for a particular PDR installation *before* the functions are executed that replace all other indexed names. This text explains why and how. It also explains the relation between these two, the user database names and some facilities for testing.

WARNING This text is not entirely up to date. The area is currently in development, which is why we have not yet updated this section, even though it no longer reflects the implementation.

12.6.1. The origin of the system identifier

When MyContexts is first fired up on a computer, Couchdb is supposed to be in PartyMode. The user, having no account, enters a username and a password of her choice. The PDR then calls the function setupCouchdbForFirstUser. This function will, eventually, generate a guid that will be the base of the system identifier. However, for now, during development, for easy testing, we just use the user identifier that the user just typed in. In this text we call it the systemIdentifier.

We also have a function setupCouchdbForAnotherUser. Like for the first user, it will in the end generate a guid but now just uses the user name provided to one of its parameters.

12.6.2. Persistence of system identifier, user name and password

We put the user name and password and the system identifier into a data structure CouchdbUser:

```
newtype CouchdbUser = CouchdbUser UserInfo

type UserInfo =
\{ userName :: UserName
, couchdbPassword :: String
, couchdbHost :: String
, couchdbPort :: Int
, systemIdentifier :: String
, _rev :: Maybe String
}
```

This structure is serialised and stored as a file in Couchdb in the database localusers. On logging in, the PDR fetches the document with the name entered by the user (it uses the special account authenticator to do so. The password for this account is kept in the source code. This is not particularly safe, but remember the database resides on the user's own machine) and checks the password. If all works out, the PDR starts up with the above data structure as part of PerspectivesState.

12.6.3. System identifier as base name

From the systemIdentifier we construct replacements for both sys:MySystem and sys:Me:

Purescript function	Indexed name	Private name
getMySystem	sys:MySystem	model:User$<systemIdentifier>
getUserIdentifier	sys:Me	model:User$<systemIdentifier>$User

These values are constructed by the two functions given in the first column. They take the value of systemIdentifier out of PerspectivesState.

User database names are derived from systemIdentifier, too:

- <systemIdentifier>_instances
- <systemIdentifier>_models;
- <systemIdentifier>_post.

These databases are constructed by the functions setupCouchdbForFirstUser and setupCouchdbForAnotherUser.

12.6.4. Putting the systemIdentifier in PerspectivesState

When the PDR fires up, it constructs a PerspectivesState that holds, among others, the systemIdentifier (as we've shown in *Persistence of system identifier, user name and password*). It then calls runPerspectivesWithState on that state and some value in MonadPerspectives to compute.

However, there is another function, runPerspectives, that takes, among others an argument bound to its parameter systemId, that constructs an instance of PerspectivesState on the fly and then computes a value in MonadPerspectives. So while computing that value, for systemIdentifier we have that argument. This means that during that computation

- Models and instances and transactions are taken from and written into a specific set of user databases, based on that argument value;
- sys:Me and sys:MySystem are replaced by values based on that argument value.

This is very useful for testing. We can run one computation for one user, then another for another user, all in the same test code!

Chapter 13. A proxy for the PDR

13.1. Overview

The Perspectives Distributed Runtime is accessed from an HTML page that runs in a browser. For most browsers (excepting Safari) we run the PDR in a SharedWorker process and its client HTML pages in their own (page) processes. They interact by exchanging messages through the Channel Messaging API.

This level of communication is programmed in the package Perspectives Proxy. The package provides several Javascript classes, some functions and a number of promises. The most important client-side facing are:

- the `PerspectiveProxy` class, that holds an extensive API of functions to query the PDR and change state;
- the function `configurePDRproxy`
- the promise `PDRproxy` that exposes a PerspectivesProxy instance
- the promise `SharedWorkerChannelPromise` that exposes the SharedWorkerChannel.

13.2. Configuring the PDR proxy

Each page that wants to connect to the PDR must create an instance of the PerspectiveProxy class. The creator of this class has a 'channel' parameter that must be bound to an instance of either `SharedWorkerChannel` or `InternalChannel` - the latter only for Safari.

The page does so through calling `configurePDRproxy` with a string argument that signals what kind of Channel should be created.

NOTE Safari is the exception. The rest of this text is oriented to other browsers. The Safari case will be treated in its own paragraph at the end of this chapter.

`configurePDRproxy` creates both a Channel instance and a PerspectivesProxy instance (and provides the former as argument to the latter's constructor). It also fulfills two promises: `PDRproxy` and `SharedWorkerChannelPromise`.

13.3. SharedWorkerChannel and ChannelID

The Channel classes handle the actual communication through the Channel Messaging API. The Channel instance stores the MessagePort instance that can be used to send messages to other processes (in our case, from the client page to the PDR). In the case of the SharedWorkerChannel, it receives that port by creating a SharedWorker with the script `perspectives-sharedworker.js`. Notice that only de FIRST page actually creates the SharedWorker (as a separate process) but that the others merely connect to it! Nevertheless, each page has its own instance of SharedWorkerChannel and its own MessagePort. Once the SharedWorkerChannel has been created, we resolve the SharedWorkerChannelPromise with it (using the sharedWorkerChannelResolver).

Notice that the SharedWorkerChannel has the MessagePort that is the *client side endpoint* of the connection between PDR and client page. So the SharedWorkerChannel uses its port to connect to the PDR that runs in a SharedWorker:

- its `send` function calls its postMessage function, relaying calls to the PDR;
- it puts an event handler on the onmessage property of the MessagePort it has received, listening for responses from the SharedWorker/PDR.

13.3.1. ChannelID

Each time a new Perspectives page creates its SharedWorkerChannel, it receives a ChannelID from the SharedWorker. The SharedWorkerChannel instance stores this and sends it along with each request to the PDR or SharedWorker. The SharedWorker uses this ID to send responses from the PDR to the right client page. The PDR itself has no notion of different clients; it is the SharedWorker that keeps them apart. This is entirely in line with the way the PDR works. It handles client calls that modificate Perspectives State in exactly the same way that it handles similar modifications from peers through the Transactions they send. It is quite democratically in that respect!

13.3.2. Calling and receiving responses: Correlation identifiers

It is important to realise that the SharedWorker communicates *out of process* with the PDR. So it performs a kind of *remote procedure call* through the Messaging API. All functions in the PerspectivesProxy API expect a response, but because of the out of process character we must treat them as *asynchronous* functions (they either take a callback or return a promise and this translates to a callback under the hood). The SharedWorker class takes care of all callback functions, storing them internally with a key called a `Correlation Identifier`. The Correlation Identifier is sent along to the SharedWorker/PDR and when it returns a result, the Correlation Identifier will be part of it. The SharedWorker class then selects the associated callback function and applies it to the result.

All this is transparent from the perspective of the PerspectivesProxy class. It just delegates calls to its Channel, sending the client-supplied callbacks along and never thinks about then again.

13.3.3. Handling PDR responses

The PDR exposes two APIs:

- one defined in the module `Perspectives.Api`. These are functions to query Perspectives State or to modify it.
- one loosely defined in the `Main` module of the PDR. These are functions to start the PDR, to register a new account, etc. All these 'housekeeping' functions return promises to the client. They are one-time-only functions.

13.3.4. Streams versus one-time-only responses

A modification to Perspectives State should be made just once and will return a success or failure message (its format depending on the nature of the call). In contrast, a query returns a result that depends very much on the current state of affairs. Whenever that changes, the client wants to be informed. Hence, we can regard the result of a query call through the proxy as a *stream* of responses at successive moments in time, as measured by relevant state change.

This translates, in the Channel classes, to callbacks that are registered to be used just once or to be called again and again. The Channel `send` function has a parameter `fireAndForget` that decides this. The value to this parameter is supplied by the various functions in the PerspectivesProxy API.

There may be circumstances that the client is not interested in any updates on a query result. It may call the query function with a value FIREANDFORGET to have the Channel classes treat it like a one-time-only call.

Those API functions that are one-time-only by their nature return a promise and do not take a callback. All others take a callback.

NOTE In the future we may replace the callback interface with a Stream-like abstraction.

13.4. SharedWorker

The SharedWorker faces towards the PDR. It comes into being when a client (let's say a page) calls the 'new' constructor on 'SharedWorker'. The first page actually causes a new process, whereas the next page merely connects to the now existing SharedWorker. Hence 'shared'!

It is important to understand that when this new process is created, the PDR does not start running immediately. The client initiates the PDR by calling the runPDR function of the SharedWorkerChannel.

When a page actually creates a SharedWorker from its script, an event is raised at the SharedWorkerGlobalScope in the SharedWorker: the 'connect event'. The same happens when the next page connects to the now existing SharedWorker. The script 'perspectives-sharedworker.js' handles this event by creating a new ChannelID and sending it through the port that represents its end of the Channel between PDR and this client. The SharedWorkerChannel instance (facing towards the client) stores it internally.

The SharedWorker faces a problem similar to the SharedWorkerChannel: it communicates directly (through function calling) with an in-process program and at the same time communicates through the Channel Messaging API with an out-of-process program.

The PDR is itself a highly asynchronously operating program. The SharedWorker handles this asynchronous character by providing callback functions to the Perspectives.API. It constructs these functions itself and what they do is this:

- it selects the right MessagePort to send a PDR reply to, based on the ChannelID;
- it returns the PDR result through that port, identifiying it with its Correlation Identifier.

When received by the SharedWorker channel instance, the Correlation Identifier is used to retrieve the *original callback function provided by the client*. It finally is applied to the result created by the PDR.

13.4.1. Connection to the PDR: the InternalChannel

The PDR operates on a stream of requests that flow in from the various clients. All requests are coming in on the same stream. This stream is created when the PDR calls the function `createRequestEmitterImpl`, provided by the package perspectives-proxy.js. The PDR does this when starting up.

Now this function creates an instance of the class InternalChannel. It provides its constructor with three functions supplied by the PDR. The first two are stream data constructor functions. The third is an `emit` function that should be called with a stream data element. Whenever the InternalChannel calls `emit`, the stream in the PDR emits another element to be processed by the module Perspectives.API.

13.4.2. A dance in four steps

Summing up, this is how the entire process is booted on the SharedWorker/PDR side:

- the client starts the SharedWorker. Through the SharedWorkers onConnect event, it establishes connection through the Messaging Channel API with the client. The SharedWorker, however has not yet started the PDR and thus cannot relay requests from clients yet;
- the client then instructs the SharedWorker to start the PDR;
- the PDR calls `createRequestEmitterImpl` as part of its startup routine, which constructs the InternalChannel instance and fulfills the InternalChannelPromise.
- only then is the InternalChannel ready to relay requests from the clients to the PDR.

13.5. Tracing the entire process from client to PDR and back

- The client initiates the process by calling one of the functions exposed by PerspectivesProxy.
- PerspectivesProxy delegates the request to its Channel, including the callback function provided by the client or a callback function it has constructed itself to fulfill the promise returned by PerspectivesProxy.
- The Channel send function constructs a Correlation Identifier, stores the callback and sends a serialised version of the request, including the Channel ID and the Correlation Identifier, through the Messaging API.
- The SharedWorker receives that serialised request. It creates its own callback function that encapsulates both the ChannelID and the Correlation Identifier. It calls the `emit` function provided by the PDR, sending its self-constructed callback function along.
- The PDR interprets the serialised request and constructs an answer. It calls the callback function, which in turn serialises the answer and sends it through the Messaging API (using the Channel ID in the response to select the right MessagePort).
- The SharedWorkerChannel (on a client side), listening on its MessagePort, receives the result. Using the Correlation Identifier included in the result, it selects the original callback function and applies it to the result. The callback function fulfills a promise or does something to React

State, updating the screen.

Chapter 14. Modifying State

14.1. State in Perspectives

Perspectives describes part of the world in terms of contexts, roles and their properties. But we do not consider such a description to be universally valid or useful. Hence we limit access to its parts by *perspectives* of user roles. Only those participating in a context have access to it and the tacit assumption is that they will know how to interpret its representations (I use *description* and *representation* as synonyms in this text).

The Perspectives universe is not God-given. Rather, the participants build it piece by piece. There are at least two reasons to do so: to cover larger part of the world by a description and because the world itself changes, so the description has to follow.

Anyhow, it is useful to think about the *state* of a Perspectives representation. This state consists, obviously, of the state of its parts. The state of a context instance is determined by its role instances; that of a role instance by its filler(s) and the roles it fills; and the values of a property type for a particular role instance could be seen as property state. However, for practical reasons we collapse role- and property state together into role state.

Given a particular set of Perspectives types (context-, role- and property types), the number of states a certain description of the world in terms of those types can assume can be very large indeed (if the number of role instances is not limited, the number of states is infinite). Therefore, rather than thinking in terms of individual states, it is useful to think in terms of *state collections* (some authors speak of micro- and macrostates).

So when is a state member of a state collection? It turns out that for role state, this is governed by a *proposition*; a sentence in propositional logic. Or, in less fanciful words: when a set of equalities or comparisons of properties, combined with ‘and’ and ‘or’, is true. In Perspectives terms, this is a *property query* with a Boolean value.

For context state we use a sentence in predicate logic. That is, we apply quantification, such as ‘for all’ and ‘exists’ to roles and contexts. Nevertheless, in Perspectives terms, this is again a property query.

Now from this point on we will use ‘state’ instead of the more correct ‘state collection’ and we will say that it is defined by a *state query*. We also associate a *state label* with the query and use it to identify that state.

14.1.1. The use of states

What good is the notion of state? It turns out there are three good uses we can put states to:

- Sometimes we want to be *notified* if a context or role enters a particular state. A prime role state example concerns *presence*. We say an end user is *present in a context instance* if he or she has opened that context on screen (that is, if there is a presentation of his/her perspective on that context instance on screen). Think of a Chat. We want to be notified whenever our conversational partners ‘enter the room’, so to say.

Notification is a user interface event.

- There may be things that we want to happen automatically whenever a context or role enters or leaves a state. These are precisely the 'bots' we make part of our Perspectives models. We can think of bots in terms of *rules* with a condition (left hand side) and action (right hand side); but we can also view that condition as a state query and think of the action as something to be executed as the context enters that state

> As a matter of fact, thinking in terms of state uncovers something that has eluded us until now: that we might have use for a rule variant that fires when its condition switches from true to false, instead of when it switches from false to true. This corresponds, obviously, to leaving and entering a state.].

- Last, but not least, we may want to model that a perspective only holds in a given state. An example: in a medical context, some information is only useful when the patient is male (so we want to suppress certain form fields for females).
  We can think of this as user interface state and changes.

14.1.2. Role state versus context state

A modeller might want to specify that a particular user is notified when a new instance of another role is created. At first sight we'd think this is about *context* state, as that is determined by its constituent parts. However, it turns out not to be possible to write a first order logic sentence that captures the notion of 'a new role instance'. This is because the 'newness' is relative to a previous state only: we say an instance is new when it was not in existence earlier. But to recognise that situation, we'd have to write an expression that *accesses a previous state*. However, a Perspectives description of the world does not capture the flow of time; it is a timeless description. We can update that description to reflect a change in the world, but the updating itself is outside of the logical domain. A state condition cannot see it. That would require a whole level of description, and correspondingly a new mechanism to realise it.

However, we can think of this phenomenon in terms of *role state*. That is, if we interpret the coming-into-being of a role instance as a state transition that we can act upon. In other words, we can specify that our user must be notified in the *on entry* section of the role *root state*.

Another example of role state would be an invitation situation, where a Boolean property indicates rejection of the invitation. Again, notifying a user of such rejection can only be expressed in terms of state of the role instance, in this case not the entry of the root state but of some substate.

14.1.2.1. Automatic actions on Role state transitions

We want to be able to prescribe certain automatic actions to be carried out when a Role enters or exits a particular state. These automatic actions are *assignments* that will change state, possibly leading to new transitions. But what are these actions *on*?

For context states, we may have the `origin` variable we can use in statements. This variable will be bound to the current object set and is defined when the state transition is described in the lexical context of an expression that gives a role (we have two such lexical contexts: within a role definition and within the perspective on expression). We *may have* object; for in the onEntry and onExit expressions written in a top level state definition, no object is available and it is an error to refer to object in assignment statements.

For role states, we can always refer to the `origin` variable. It is bound to the role instance whose state changed.

Moreover, unless stated otherwise, a property assignment statement always changes the property values of the role instance whose state changed; i.e. the instance bound to object. So these two statements are equivalent:

```
PropertyType =+ 10
PropertyType =+ 10 for origin
```

Finally, we can always use the variable `currentcontext` in our expressions.

14.1.3. How to use state in models

14.1.3.1. Defining state

We introduce into the written form of the Perspectives Language (PL) a new construct:

```
state <identifier> = <booleanQuery>
```

We can add a notification like this:

```
state <identifier> = <booleanQuery>
  on entry
    notify "Entering state X."
```

This will cause the system (i.e. the combination of the Perspectives Distributed Runtime (PDR) and the end user application, MyContexts) to bring this state change to the end users' attention (the string may contain queries that must evaluate to string values).

This governs the first use of states: notification. No more is needed to bring state changes to the end users attention.

14.1.3.2. Using state to make things happen

A state change is an event and we can use it to make more things happen:

```
on entry of <state name>
  do
    <assignment>+
```

and

```
on exit of <state name>
  do
    <assignment>+
```

where, obviously, <state name> must refer to a defined state.

Such declarations are part of a user perspective. Automatic actions are only carried out on behalf of a user!

14.1.3.3. State in user perspectives

Finally, we use state in the specification of a user perspective:

```
perspective on: <RoleExpression>
  in state <state name>
    only Consult
  in state <another state>
    except Delete
    action <identifier>
      <assignment>+
```

This shows how some verbs are available to the user in just some states. It also illustrates an *action* available in just a single state. An action is a series of assignments that can be triggered as a whole by the end user. So, to prevent misunderstanding, an action is never carried out automatically, in contrast to rules and on-entry and on-exit assignments.

14.1.4. Alternative modelling

It may be useful to devise another syntax for state and perspectives. Traditionally, states are modelled as syntactical units, with parts specifying entry- and exit actions. In Perspectives we might have states as containers within contexts, defining some roles in some states and not in others. There are problems to be solved, like unifying roles that occur in two or more states but with different perspectives, for example. We consider this to be future extensions.

14.1.5. How to make it work

14.1.5.1. Representing state in instances

State holds for particular instances. How to represent it? First, we must ask ourselves whether state should be persistently stored. Should state be recomputed on each new session, or should it survive the end of a session?

Ending a session does itself not change the state of a context or role or property as we have defined it here. Hence, there is no *need* to recompute it on session start. Because we want to be able to present the user with a list of notifications that have a certain duration (a notification can be *valid* for some time) and the user can switch off his computer in the meantime, it would mean we would have to recompute state for all context instances on startup. That is clearly undesirable. Hence, state must be persisted.

At first sight, we have two opportunities to represent (and persist) context instance state:

1. as an external property (holding a list of strings representing the state types);
2. as a new member of the context representation.

When we represent state as properties, it will be automatically shared between those who play a role in the context (assuming every user role will have a perspective on the state of the context). Is that what we want? Let’s explore some examples.

Consider a medical examination related to a serious disease, having a physician, a laboratory technician and a patient. Suppose a blood test is involved. At some point, the test results are available and the physician should interpret them. The physician should be notified of this state, but the patient should only receive a notification after the interpretation has been added to the test results.

Consider a financial transaction system involving two business parties and an intermediate party. The latter should perform fraud checks. The situation is modelled such that some broad checks are performed automatically on behalf of the intermediate party. When alarm bells go off, manual intervention is required before further action is taken. Obviously, the alarm bells should not ring for the two business parties *before* the human audit.

We conclude that indiscriminately sharing state would *leak information* that we’ve carefully kept away from some roles, using perspectives. Notice we’re not talking about actual notification, as we can choose to not notify some user roles of some state changes. However, these changes would be sent to their computer and this opens up, in principle, a way for the receiver to get access to it.

In other words: state should not be shared among participants; each should recompute state given the information available according to his perspective (in other words, users do not have an implicit perspective on the state condition).

This analysis allows us to decide on state representation in terms of a new internal member of the context instance representation, rather than as external properties.

We add to the context instance representation an Array of the current states that instance is in (and do a similar thing to role representation).

14.1.5.2. Working with Properties and Verbs

We provide an API function that returns, for a given role instance, an Array of Property-Verb combinations given the state(s) of the role and context and the type of the role the owning user plays in the context. As with other queries, we support the functional reactive programming pattern for these functions. This means that on state change, the user interface program is notified by an invocation of the callback that it provided on requesting the Property-Verb combinations.

This makes it very easy to adapt our user interfaces automatically to changing state, as the visual representation of each Perspective is built on Properties and Verbs.

The underlying mechanism is the same as for ordinary queries: based on dependencies. However, the computation of Property-Verb combinations depends on the states of a role and its context. Hence we record a new type of dependency, the ‘state-dependency’.

When that state changes (see below) we record the correlation identifier of the API request for the Property-Verb combinations in the current Transaction in Perspectives State. On subsequently running that transaction, we look up the corresponding effects and apply them (recomputing the combinations using the new states and sending them to the client).

14.1.5.3. Applying the inverted-query pattern to state queries

But how does state change? As a state definition consists of a boolean query, we can invert it and thereby make sure that relevant assignments lead to re-evaluation of such queries (just as we do with the previous Bot Action implementation). Actually, this is a two-phase mechanism. On changing some context, role or property, we follow inverted queries to the contexts (or roles) where they are state queries and record these in the current Transaction in Perspectives State.

Then, when we run that Transaction, we re-evaluate the state queries for each context or role that is affected. Whenever a state query evaluates to true, but the associated state label is not in the current states of the context or role instance, we add the label. Conversely, we remove the label if the query evaluates to false. On doing so, we record the correlation identifiers whose computation depends on those states, in the current Transaction.

In a way, a state query is like a rule whose right hand side adds or removes a state label (and also executes entry- and exit automatic actions, see the next paragraph and also the last).

When we then later re-evaluate queries that came in through the API, the relevant state-dependent requests are re-computed.

14.1.5.4. Automatic actions on entering and exiting states

When we re-evaluate a state query and add a label (or conversely remove it), we also look up all entry automatic actions for the newly added state (or the exit actions when it was removed instead) for the role played by the owning user. These will be executed, triggering state change that may lead to a new round of evaluation of state queries.

14.1.5.5. Notification

We want to notify the user about some roles and contexts when they enter (or exit) designated states. Being in a ‘notified state’ is, in some cases, a phenomenon that should persist for some time (see next paragraph). For that reason we record those roles and contexts in specific role types in sys:PerspectivesSystem (thus, this becomes Perspectives State and survives individual sessions).

If the modeller specified, say, state entry notification for state S of context type C, at level L, for user role U, an instance I of C that enters S when the owning user is in role U will be added to the role ContextNotification of MySystem, with property Level having value L.

In other words: we keep a list of contexts annotated with notification level (and another for roles). A client program can request these role instances through normal API calls; so when I is added to ContextNotification, the client will be updated.

It is up to the end user program to determine how to actually alert the end user. It may throw up a screen alert, for example. Handling notifications is part of the framework provided by MyContexts; it is not the responsibility of the screen programmer of a particular app (model).

14.1.5.6. Notification life cycle

What happens to a notification when it has been shown to the end user? For some notifications, just showing it once may be good enough. This may suffice for notifying the end user that a chat partner has entered a context. This means that the PDR does not remember contexts that entered the triggering state; after the end user program has received updates, they are discarded.

But for others it might be better to keep them in a list the end user can choose to inspect, until he actively dismisses them. This may be appropriate for reminders to reply to an email; indeed, the very idea of a to-do list is modelled this way. Such notifications should survive the end of a session with MyContexts.

So, for notifications we have two dimensions:

- How urgent a notification is brought to the end users' attention;
- Whether it is dismissed automatically, or by hand (or after some time, etc.).

We must research whether these two dimensions can be collapsed into a single set of categories, or need separate representation.

14.2. State Change

A computation in Perspectives proceeds in part by changing state (the other part is: consulting state). State changes in two ways:

- by creating Context and Role instances (always attaching the latter to the former)
- and by changing them.

Changes are limited to:

- adding role instances to contexts, or removing them
- binding or unbinding roles;
- changing property values.

State is saved in Couchdb. State is also partly saved in memory: this we call the *cache*.

Because multiple users have access to the same contexts, roles and property values (though two users *never* share exactly the same contexts and roles), changes must be *synchronised.*

In this text we document the relation between changing state and caching, saving and syncing that state change.

14.2.1. Model files

A model consists of the description of many types. Types are part of the state of a computation. Models are saved as model files. They are cached as well. In fact, there is no real technical difference between a model and a context or role with respect to caching and saving. We do not synchronise model files, however. Models are published on a server. We have a mechanism of *subscription* that allows a PDR to stay up to date w.r.t. models. As publishing is a conscious act by a modeller, model updates are not spread in real time as the modeller changes his model.

14.2.2. Cache, database and synchronisation: mechanisms

We briefly characterise these mechanisms.

14.2.2.1. Cache

State is cached in MonadPerspectives. This is a Monad based on ReaderT and Aff. Through ReaderT we have access to an AVar. This AVar holds a record with fields. We modify the contents of the AVar in order to change state. This arrangement allows us to modify state asynchronously.

14.2.2.2. Database

Couchdb stores JSON documents. Each Context or Role instance is serialised as a JSON document. Instances are stored in a database in Couchdb whose name is constructed from the unique string that identifies a user, thus allowing a single Couchdb instance to host data for many users. An important facet of Couchdb is its synchronisation mechanism. This in turn depends on versioning. In order to be able to change a document in Couchdb, one must send the current *revision string* along to the http request. To speed up the process, we keep the revision string in cache for documents we've retrieved from Couchdb before.

14.2.2.3. Synchronisation

Synchronisation is a two-step process:

1. the PDR of a user who changed state compiles a *Transaction* of such changes and sends them to relevant other users (in fact, each user receives a Transaction tailored to his perspectives);
2. the PDR of these other users processes the Transaction. This involves checking whether the sender was authorised to perform the changes. If so, it loads the changed entities in cache (if they weren't there before) and applies the changes. Finally it saves them to its own Couchdb instance.

14.2.3. The dynamics of state change

Contexts and roles that are created, are immediately cached. However, we do not always immediately save them to Couchdb. This is because sometimes, after initial creation, resources have to be changed. Think of a context instance: we prefer to wait until all its roles have been created, before we save them. This is because we may yet encounter an error while working out the roles. Instead of retracting prematurely saved roles, we wait with saving them until the entire process has been finished without errors.

For similar reasons we do not immediately dispatch a Transaction as soon as a change has been cached or saved.

14.2.3.1. Creating and modifying resources

There are no more than two modules where instances are created:

- Perspectives.BasicConstructors, with the functions
  - constructContext and
  - constructAnotherRol
- Perspectives.LoadCRL

There is a single module through whose functions all changes to resources run:

- Perspectives.Assignment.Update

In turn, all these functions are used in two distinct places:

- Perspectives.Api
- Perspectives.Actions

The first module channels resource creation and modification that are the initiative of the end user. The second module realises the automatic execution of such actions by bots.

The consequence is that all caching, saving and synchronising starts in the above three modules.

14.2.3.2. Saving and syncing resources

The module Perspectives.SaveUserData contains five functions to save cached Context- and Role instances and to synchronise them:

- saveContextInstance (to used in tandem with constructContext)
- saveRoleInstance (idem, constructAnotherRol)
- removeContextInstance
- removeRoleInstance (to be used in tandem with removeRol)
- removeAllRoleInstances (to be used in tandem with deleteRol and setRol)

14.3. Execution Model

The Perspectives Language is partly declarative. It lets the modeller describe contexts, roles, properties, perspectives and more, on the type level. But it also contains *assignment statements* to update *instances* of those types, to track a changing world. Assignments are always carried out by an end user, either by direct manipulation through the user interface (e.g. by dropping a role onto another, signalling the PDR that the former should fill the latter), or by executing *actions*, which are pre-packaged sequences of statements. But there is yet another mechanism responsible for executing assignment statements and that is actions carried out automatically on behalf of some user when the state of a context or role changes.

More specifically, when a resource enters or leaves a state, actions may be carried out automatically. This raises some questions, a.o.:

1. Does state change statement by statement, during action execution?
2. If a resource moves in a new state S because of the execution of a statement in a sequence, when exactly will statements to be executed on entering S be carried out?
3. How does resource creation and deletion relate to state change?
4. When a context and its roles are deleted, in what order are on exit statements executed?

In this text we give answers by describing our design decisions. Together, they give the *execution model* of the Perspectives Language.

14.3.1. Statements execute in order of appearance

Consider the following action:

```
action A
  create role SomeRole
  SomeProperty = 10 for SomeRole
```

where SomeRole is functional and SomeProperty is Numeric. Assuming the context in which this takes place has no instance of SomeRole yet, this will work perfectly, because the statements are executed *in order of appearance*. So an instance of SomeRole is created; then, on the next line, the (minimal) query SomeRole returns this instance and it will acquire value 10 for its property SomeProperty.

Notice that each statement is executed in an environment that has been changed by the statements above it, in the program text.

14.3.2. States associated with resource creation and deletion

Each context or role has at least one type. The modeller may add Aspects to a type, thereby adding more types. When a resource is created, it immediately afterwards enters the *root state* associated with each type.

A type’s root state may be filled in by embedding an on entry or on exit clause directly in its body:

```
thing SomeRole
  on entry
    notify SomeUser
      “An instance of SomeRole has been created!”
```

In contrast, when a resource is deleted, it *first* leaves all states it is in (its *active states*). Only then is it actually removed. More on this topic below.

14.3.3. Staged evaluation of state changes

Let’s suppose that SomeRole had been defined like this:

```
thing SomeRole
  on entry
    do
      AnotherProperty = 20
```

And let’s change action A as follows:

```
action A
  create role SomeRole
  SomeProperty = SomeRole >> AnotherProperty + 10 for SomeRole
```

This will compile, but after executing A, SomeProperty will not have a value. This is because execution of statements proceeds in *stages*. Let’s work this out for our example:

- Stage 1 sees execution of
  - the create role statement (but *not* its on entry clause!)
  - the property assignment.
- Stage 2 sees execution of the on entry automatic action of the newly created SomeRole instance.

> The argument query `SomeRole >> AnotherProperty` of the property assignment in the action consequently finds no value, hence the entire addition expression has no value.

In other words: when a resource changes state during an execution stage, *the effects of this state change will only occur in the next stage.* That is, automatic actions due to state changes are postponed till all statements scheduled for the current stage have been executed.

Why this design? Our example is so small that it is easy to form a picture in our mind of how the on entry action would happen, before executing the property assignment. But things get very complicated quickly. Actions carried out automatically may trigger more automatic actions, recursively. The code describing this may be scattered all over a model. In no time at all it is very difficult to piece together what is happening where, when and why.

We’ve chosen this model because it allows the modeller to form a clear picture of what will *actually happen locally*, that is, the direct consequences for the state of the resources affected by his statements.

This is not to say that understanding automatic actions is always easy.

Another way of looking at this execution model is that state changes of various resources happen in parallel. If a statement sequence triggers state change with automatic actions in two other resources, these will execute (as if) in parallel. This means one should never count on a particular order of their execution!

14.3.4. Removing a role instance

Roles form the connections between contexts. A role can be filled by another role (usually a role from another context) and a role can fill many other roles. These connections are bi-directional in the sense that a query can traverse them

- both from a role to its filler, and
- from a filler to the role(s) it fills.

Think of these connections as pointers in and out of the role instance. It is convenient to name these two types of pointers:

- a role can have many *fills* pointers;
- but it has only one *filledBy* pointer.

When a role is removed, *we only detach the pointers towards the role instance.* We need not remove the outward pointers; we're going to throw the role away, anyway. So, for any role instance to be removed,

- we remove the filledBy pointer from all roles that are filled by it, and
- we remove the fills pointer from its filler.

A user is endowed with a set of perspectives that are qualified by role- and property-verbs, including verbs that affect roles:

- Remove
- Delete
- RemoveFiller
- RemovePropertyValue

> A user can only remove roles she has a perspective on with verbs Remove or Delete. Notice that Remove and Delete effectively imply RemovePropertyValue and RemoveFiller.

14.3.4.1. State evaluation

Before a role is removed, it exits all its active states. The deepest nested active state(s) are exited first, meaning their on exit actions are executed first. The actual algorithm is formulated the other way round:

1. For each state:
   a. exit active substates;
   b. execute the on exit action.

And this starts with all root states of the role. *All these actions are executed in the same stage!*

Subsequent automatic actions due to state changes caused by these on exit actions are *all postponed to the next stage.*

Furthermore, the automatic actions are execute *before the pointers into the role are removed.* This means that the statements are executed on the structure as it exists before removal. This is important, because it allows the modeller to modify remote parts of the web of roles and contexts from such an on exit action.

14.3.4.2. Synchronization

There may be peers in roles that have a perspective on the role instance that is removed. They should be informed. We compute these peers *using the network prior to the removal of the resource!* This should be obvious: we find peers by following outgoing links. When the resource is destroyed, there are no outgoing links left.

14.3.5. Removing a context instance

A context is embedded in the network of contexts and roles through the connections of its roles. In order to remove a context, we can simply

- remove the incoming links (fills and filledBy) of all of its roles
- and then throw away the context and all its roles.

In other words: the internal structure of the context does not need to be torn down to remove the context (but see below for role state evaluation).

A user is endowed with a set of perspectives that are qualified by role- and property-verbs, including verbs that allow her to remove information:

- Remove
- RemoveWithContext

> RemoveWithContext, RemoveWithContextLocally, DeleteWithContext, DeleteWithContextLocally are not yet implemented in InPlace v.0.12.0. We will introduce them to indicate whether the perspective allows removing (or deleting) the bound context, too.

- RemoveWithContextLocally
- Delete
- DeleteWithContext
- DeleteWithContextLocally
- RemoveFiller
- RemovePropertyValue

In order to initiate removing a context, a user must have a perspective on a contextrole filled with that context, with the verb RemoveWithContext, RemoveWithContextLocally, DeleteWithContext or

DeleteWithContextLocally. These are the only verbs that allow removing a context.

Notice, that, in effect, RemoveWithContext implies Remove, RemoveFiller and RemovePropertyValue. DeleteWithContext implies Delete, RemoveFiller and RemovePropertyValue. However: this does not mean that a user with a RemoveWithContext perspective therefore has the right to remove individual roles, for example.

The verbs that end in Locally permit the user to remove or delete their own copy of a context, but peers are not required to do the same. However, they must annotate the user roles with the removing peer such that they are no longer involved in the synchronisation process.

14.3.5.1. State evaluation

Before a context is removed, it exits all its active states. This happens in exactly the same way as for roles: deepest nested active states are exited first and all actions on exit are executed in the same stage.

Consequently, *all statements are executed while the context is still fully intact*. For each statement, the modeller can ‘reach out’ of the context and change things there.

Can he change the context that is about to be removed? He could, actually; and this may affect the states that are subsequently exited. Another reason to modify a resource that is about to disappear would be to enable conditions for *statements that follow*. The modeller is strongly advised against this (we may add, in the future, a compiler check that warns against this), because it makes removing a context less transparent. The acting user removes the context based on what she can perceive of it; if it is modified on the fly, she actually removes it in a different state from what she thought.

None of the modifications to the context or its roles itself, make a material difference for peers: they throw away the entire context. Of course, modifications *outside* of the context that is removed, will be communicated to the peers.

14.3.5.2. Embedded role state evaluation

What about the roles embedded in the context? They are removed, too, so they should exit their active states as well. The question is: do we exit role states *before*, or *after* context states? We choose to exit them **before** exiting the context states.

14.3.5.3. Synchronization

Removing a context is a very powerful operation. All peers are required to remove their local version of the context, too, completely. Even if it holds more information than it did for the acting user, they should still remove it entirely. Synchronization is therefore simple, because it consists of a single delta that instructs the receiver to completely remove the context.

Consequently, we do not need to collect deltas when we actually remove the context, detaching it from its surroundings. All these deltas are superseded by the powerful instruction to remove the context, making them redundant.

> We may ask ourselves: can there be a peer with a role that fills a role of the removed context, without having a representation of that context? The answer is no, because *each reference must be locally resolvable*. That is, the ‘fills’ link of that peer must point to a role that is present in his installation – and hence the context is present, too. The same holds for a link in the other direction.

Which peers that should be informed about removing the context? This is the union, over all roles, of the peers that should be informed when the **incoming pointers** to the role are removed.

On collecting those users, we *should* ***not*** *already remove the pointers*. Otherwise, for each consecutive role instance, the computation is carried out on a diminished context representation. A simple example shows why that is a problem.

Consider a context with two user roles, filled by different peers of the acting user (who removes the context). Clearly both peers have to be informed that they no longer fill a role in the context after its removal. However, were we to remove the roles one by one, it is obvious that

- the removal of the first peer can be communicated to the second peer (who is still in the context)
- but the removal of the second peer would never be known to the first peer (who is, after all, no longer present in the context by this time).

So, in effect, we first run a kind of *simulation* of removal of the context:

- first we collect peers that should be informed when incoming pointers to the contexts’ roles would be removed;
- then we exit, for each role instance, its active role states;
- finally we exit all active context states.

Only then

- do we send the context removal delta to the collected peers;
- and we finally actually severe incoming pointers and remove the resources.

14.3.5.4. Synchronization may need *passing on*

The user that removes a context, may not have a perspective on all users in that context. As a consequence, he cannot inform all those concerned about its demise. This means that we require the synchronization mechanism of *passing on*. This is that some users receive the delta not from its originator, but via other users in the context.

14.3.6. Refining understanding of resource removal

Reconsider these three important rules of the execution model:

1. Statements are executed in order of appearance in the model source text;

2. Statements in an on exit clause are executed *before* the resource is actually removed from the structure of contexts and roles.
3. When a resource changes state during an execution stage, *the effects of this state change will only occur in the next stage.*

These three rules are not compatible, as we will illustrate with this example:

```
thing SomeRole
  on entry
    do
      remove role currentcontext >> AnotherRole
      create role YetAnotherRole in currentcontext
thing AnotherRole
  on exit
    do
      create role TheThirdRole in currentcontext
```

Clearly, rule 1 dictates that the AnotherRole instance must be gone by the time that the instance of YetAnotherRole is created. However, rule 3 says that the action on exit of the instance of AnotherRole can only be executed in the next phase (that is, after both statements have been executed). And rule 2 states that these actions must be completed while the AnotherRole instance is still there.

14.3.6.1. Solution: monotonic inference first

We solve this problem by, in effect postponing the actual removal of resources until the very last moment. This means that, seen per on entry or on exit clause, *removal statements always come last*. So the first part of our example is equivalent to (and should be written as):

```
thing SomeRole
  on entry
    do
      create role YetAnotherRole in currentcontext
      remove role currentcontext >> AnotherRole
```

Removal comes last. This holds recursively, for ‘nested’ automatic actions. As a consequence, execution of automatic actions is divided in two steps:

I. First, all additions to the structure are made, recursively, all the while postponing any removal encountered, while yet executing all on exit clauses of resources that are to be removed (and, obviously, any actions on entry as well);

II. Then, in one fell sweep, all resources marked for removal are detached from the network.

This may, of course, trigger fresh state changes in some resources, so then the entire process begins again.

Complicated though this may seem, it actually has a desirable characteristic: to understand the execution of automatic actions in a model, you can try to understand the *additions* independently from the *removals*. Both can be understood as *monotonic inferences* from the state of all resources. This is good, because it means we can analyse what happens in terms of ordinary mathematical logic.

In other words: you can use logical inference to determine from a given overall state:

- what will be added to the structure, and, independently,
- what will be taken away from the structure;

Then consider the new state that arises when new things are first added and then some others are removed.

> Actually, it does not matter whether we first add and then remove the results, or the other way round. This is because the system is robust enough not to fail if we try to add a role instance to a context that does not exist. But it's certainly more elegant and efficient to first add and then remove.

14.4. Creating and deleting contexts

What user roles have the right to create a context instance? Perspectives grants its users their powers exclusively through perspectives on roles. In order to dole out rights for creating contexts, we introduce a verb specifically for creating contexts, to be bound to a context role in an embedding context: CreateAndFill. A user role A with a perspective on context role C that includes this verb, can create a new instance of C and fill it with the external role in a new instance of the possible binding of C – all in one go.

So we see that we do not have the notion of a right to create an instance of a context *as such*; rather, this right is scoped to a particular context *in which* an embedded context may be created. And there may be multiple occasions to create the same context type, as embedded in various other contexts.

14.4.1. Local and remote

In the simplest case, a user role has a CreateAndFill perspective on a local context role. It allows them to add a role instance (bound to a context instance) in the context that they belong to.

However, with the existence of Calculated roles, a user role has access to 'remote' roles, too. As long as there is a path leading from the user role's context to a role, the modeller can grant a perspective on that role. And this perspective may include the verb CreateAndBindContext.

However, in order to create a context somewhere else, that 'somewhere else' must be well specified. Let's first examine the way the modeller instructs the system to do so, on behalf of some user.

> We're not just talking about creating, but also filling; in the following, I will not tediously repeat 'create and fill' but just write 'create', meaning 'create and fill a new context role'.]

14.4.1.1. Create a remote context automatically on behalf of a user

First, we create a Calculated role that consists of a remote context role (without losing generality, we can assume for the sake of simplicity that roles A and C are functional):

```
context: Remote = A >> binding >> context >> C
```

Instances of Remote actually are instances of C, in some other context.

```
in state <SomeState>
  on entry
    do for User
      create context X bound to C in A >> binding >> context
```

This causes a new instance of role C to be created in the context identified by `A >> binding >> context`, on entering state <SomeState> (for an otherwise unspecified resource). It is filled by the external role of a new instance of context type X.

Of interest is the clause in `create context X bound to C` in `create context X bound to C in A >> binding >> context`. It says *in what context instance* the new context role is created. Notice that it equals the definition of Remote, except for the last step. So the clause identifies a particular context instance (remember we created Remote to be a functional Calculated role) and creates a new context role in that instance.

Obviously, creating a context locally is simply:

```
create context X bound to C
```

We just leave out the in clause to create in the local context. The general syntax is:

```
create context ContextType bound to RoleType in <contextExpression>
```

14.4.1.2. Create a context, fill an existing role

A variant of the use case given in the previous paragraph would be the situation where we already have a role instance to fill the new context instance with. We then use a variant of the operator:

```
create context_ ContextType bound to <roleExpression>
```

Notice that

- we do not need to provide a context: the role instances selected have a context;
- we can never create an unbound context (see below for a definition) in this way.

14.4.1.3. How an end user creates contexts

Can an end user create a context remotely, too? Let's explore what that would mean. But before we do so, I'll show how an end user creates a context *locally*.

Through some user interface – let's assume it to be a GUI – the user has navigated to some context instance. In that context instance they play a particular role and with that role comes a screen, providing perspectives on the context instance. Returning to our example, we assume the context to be the one that has context role C. Now in order to create a new context in C, the end user instructs the core system, through its API, to execute create context x bound to c on the current context, with type X for role C. That's it.

In contrast, assume the end users' current context is the one that holds the role Remote. Now, the context holding role C is remote. For the end user to create a new context instance in that remote context, *he has to identify that remote context in some way.* This means that his GUI must offer him a means to do so.

That may be quite easy: for example, just consider a conventional order-detail screen, where both order and detail are represented as contexts. The end user can simply point to a detail and (for example) push a button in it to create a new context in it (say, a delivery sub context; so C – a role in the detail context - is bound to a new delivery context). The core API receives, along with the instruction create context, the (remote) context representing an order detail and the role type (delivery) to create a new instance in.

14.4.1.4. Non-functional Calculated roles

What if role C were not functional? The in clause in the bot's rule would now potentially identify many contexts. If not modified, the rule would create new contexts and *fill all of them*. It is up to the modeller to decide if that is desired behaviour. To avoid it, he might, for example, filter away from the retrieved contexts those that already have an instance of B.

14.4.2. Unbound contexts

Consider the role Chats:

```
context Chats = callExternal cdb:RoleInstances( "model:SimpleChat$Chat$External" )
returns: Chat$External
```

Its roles are computed by executing a query on the database, requesting role instances of a particular type, in this case the external role of the Chat context. Can we create a new instance for such a role?

Yes we can, but we cannot create role instances to fill them with. There simply is no Enumerated role to fill. Nevertheless, we can retrieve the instances of the role. The matter how a role is represented (enumerated within a context, or by a query on a database) is an implementation detail rather than a conceptual issue. So Chats is a bona fide context role.

If we make the core system execute create context on such a role, it will just create a context instance (including its external role)

```
create context Chat in Chats
```

Of interest is the question: how does the core system recognise this situation? How does it decide that the created context cannot be bound? The crucial property of a Calculated context role that identifies it as one in which contexts cannot be bound, is that its computation *consists of a call to a database query*.

There is no consequence for the modeler. He can use the same expressions for bound and unbound embedded contexts. It is the core system that differentiates. This is as it should be, as it is more an implementation detail than a conceptual issue.

14.4.3. Removing and deleting contexts

It follows that we cannot explicitly delete a context: we have no way of specifying the right to do so. However, we can provide a user role with a perspective on a context role that includes the DeleteWithContext or RemoveWithContext verb (remove does away with a single, specific role instance; Delete removes them all (within a context).

The PDR does not cascade delete resources (otherwise than removing the roles of a context with the context). This is because the true test whether a context can be removed is if there is still a user with a perspective on some part of the context. This is a resource intensive test.

14.4.3.1. RemoveWithContext

With remove, we can pick and remove a single (context) role instance. Because we identify the role instances we want to remove, we don't have to identify their contexts:

```
remove context <role expression>
```

Notice the context keyword that follows remove. Also note that this not only removes the context, but also the context role that is filled with the contexts' external role.

The following would only remove the context role that is filled with (the external role of) the context:

```
remove role <role expression>
```

Furthermore, the former expression requires a perspective with RemoveWithContext; the latter just requires Remove.

14.4.3.2. Delete

Deleting removes all instances of a role type.

```
delete role <role type>
```

This will remove all instances of the given role type from the current context.

```
delete role <role type> from <context expression>
```

This will remove all instances of the given role type from the context(s) retrieved by <context expression> (a path query). **It is an error** if the type of the role instances is not defined in the type of the contexts!

As with removing a single context, we can delete all contexts that fill a particular context role:

```
delete context <role type> from <context expression>
```

Again, all instances of the <role type> (which must be of kind context role) are removed as well.

14.4.3.3. Removing a context is computed per bubble

Consider this: the core system serving a particular end user has removed a context *from the bubble of that user*. Should it communicate all Delta's generated by this process to peers with a perspective on that context role?

No! It must, however, communicate a Delta that says that the context has been removed. Each peer's core works out what to remove from that Delta. This means, for example, that a user without a perspective on a particular role in the removed context, will make a user with that perspective remove that role as well.

14.4.3.4. Removing unbound contexts

The operational semantics of deleting is different for Database Query Roles. We stipulate that an external role instance and its context are removed permanently from the end users bubble (he part of the Perspectives Data Universe that is accessible to him). whenever it is being removed from a Database Query Role (and when the removing user has the verb RemoveWithContext or DeleteWithContext for that calculated role).

The consequence of this is that when this occurs, no DBQ Role based on the same type will show the instance any more.

14.4.4. Synchronisation: delta representation

In this more technical chapter we describe how the various operations should be represented in terms of deltas, to synchronise changes with peers. We describe the change to the users' bubble in terms of assignment statements in bot rules.

Notice that we only create deltas for single role- and context instances, even if, for example, we remove all instances of a role in a context. This is because we record the deltas with the role instance representations themselves. Furthermore, not all users may have the same instances for a particular role type in a context instance (this may arise as a user role has a perspective on a remote role that is defined with a filter).

14.4.4.1. Create a bound context

```
create context <context type> bound to <role type> in <context expression>
```

Represent this with:

- For each context instance produced by <context expression>, a UniverseRoleDelta
  - whose id identifies that context instance
  - where roleType equals <role type>
  - and where the new role instance is in roleInstances
  - and deltaType is ConstructEmptyRole.
- Pair each UniverseRoleDelta with a ContextDelta.
- For each newly created context instance a UniverseContextDelta, where
  - the id identifies the new context instance
  - the contextType is the context type
  - the deltaType is UniverseContextDeltaType.
- For the external role of each newly created context instance, create a UniverseRoleDelta whose deltaType is ConstructExternalRole.
- For each pair of UniverseRoleDelta's (that of the binder and the bound external role) a RoleBindingDelta
  - whose id identifies the new role instance in the context;
  - whose binding identifies the new external role.

14.4.4.2. Delete a bound context

```
delete <role type> from <context expression>
```

<role type> is a context role. Represent this, for each context instance, for each role instance of <role type> in that context instance:

- A UniverseRoleDelta:
  - whose id identifies that context instance
  - where roleType equals <role type>
  - and where the role instance is in roleInstances
  - and deltaType is RemoveRoleInstance.

Notice that we do not generate ContextDelta's, nor UniverseContextDelta's. The receiver checks whether any binders of the external role of the context remain. If not, he'll remove it.

14.4.4.3. Remove a bound context

```
remove <role expression>
```

We handle this in exactly the same way as for deleting a role type (with UniverseRoleDelta's) but we limit ourselves to the role instances identified by <role expression> and their bound contexts.

14.4.4.4. Create an unbound context

```
create context <context type> bound to <role type> in <context expression>
```

Represent this with:

- For each newly created context instance a UniverseContextDelta, where
  - the id identifies the new context instance
  - the contextType is the context type
  - the deltaType is UniverseContextDeltaType.
- For the external role of each newly created context instance, create a UniverseRoleDelta whose deltaType is ConstructExternalRole.

Notice that, in contrast to creating a bound context, we do not create a UniverseRoleDelta for the context role, nor a ContextDelta to represent its connection to the context instance, nor a RoleBindingDelta to represent the binding between context role and external role.

14.4.4.5. Remove an unbound context

```
remove <role expression>
```

The sender establishes that we handle an unbound role by finding that it is an external role and there are no binders. If there are binders, deleting an instance from a DBQ Role is a no-op.

It then represents this with a UniverseRoleDelta:

- whose (context) id is the context of the external role
- where roleType equals the type of the instances
- and where the role instance is in roleInstances
- and deltaType is RemoveUnboundExternalRoleInstance.

The receiver determines that we handle an unbound role from the deltaType. If the instance has no binders (on his side), he removes the external role and its context.

14.4.4.6. Delete an unbound context

```
delete <role type> from <context expression>
```

The sender recognises the fact that we want to remove an unbound context from the role type. However, we now want to remove *every* unbound context from the given contexts. So we generate a UniverseRoleDelta like above, but only for every role instance that can be retrieved from the database that has no binders. As long as it has binders, deleting an instance from a DBQ Role is a no-op.

Chapter 15. Synchronization revisited

15.1. Synchronisation

Because the Perspectives Runtime is a *distributed* system, we have to find a way to make state changes initiated by a user available to peers who share a perspective on the changed entity. We call that process *synchronisation*. In this text we explore some technical issues.

Users are the only source of state changes. A user can initiate changes through some client program that will communicate his intentions to the PDR. This client-PDR communication does not concern us here, but the Perspectives Application Program Interface (API, the module Perspectives.Api) is a good starting point for our exploration. Why? Because here we find top level functions that can be invoked by the user to change state.

In principle, if we could make the PDR of peers invoke the very same functions with the same arguments, they would change *their* state in the same way and we would have synchronised their local representation of Perspectives State. We might call this a *remote procedure call* mechanism (RPC).

Of course, not every change a user makes is relevant for all his peers. A peer should only be informed if he has a perspective on the changed entity. The PDR finds out by consulting the model and reflecting that on the instances and users at hand.

The bulk of this text is devoted to this RPC mechanism.

It is noteworthy that there is another starting point for our exploration and that is the language in which a modeller can write actions to be executed on state change. By writing such actions, the modeller *automates* some user actions (in a specific role). In principle, exactly the same functions for changing state can be used in automatic actions as in user actions. So we could take the module that implements the semantics of the language keywords for actions as a second starting point for our exploration: Perspectives.Actions.

There also is *another* mechanism by which PDRs can synchronise state and that is by sending an invitation by mail (or another manual transport mechanism). An invitation is the JSON serialisation of a number of contexts and roles. By dropping such a document on the Perspectives client program screen, the user instructs his PDR to add these contexts and roles to his local cache and store. This synchronises state with respect to the invitation between sender and receiver. This mechanism is of no importance to this text.

> This mechanism is to be used by users who are not yet connected through Perspectives. It is the mechanism by which they become peers in at least a single context

Finally, we mention a last mechanism that is supported by the PDR by which state can be changed and that is loading and parsing a file with text written in the Context Role Language (CRL). Such a text, not surprisingly, describes instances of roles and contexts. Loading such a file contributes these instances directly to the cache and database. However, the end user has no means of calling the relevant functions. They exist mostly for testing purposes and for modellers. A model describes types, but must be complemented by a small number of instances (among them a context instance that describes the model; it is used by repositories to present an inventory of available models).

We will now turn our attention to the question: what is in a remote procedure call?

15.1.1. What procedures to call?

It turns out that all functions that lead to state change are contained in just three modules:

- Perspectives.Instances.Builders
- Perspectives.Assignment.Update
- Perspectives.SaveUserData

> Perspectives is written in Purescript, a functional language. So strictly speaking we just have pure functions. Nevertheless, there are mechanisms to handle side effects like storing information in a database. In this text we will use the word ‘function’ too, when such side effects are sorted, not bothering to distinguish them from statements and procedures.

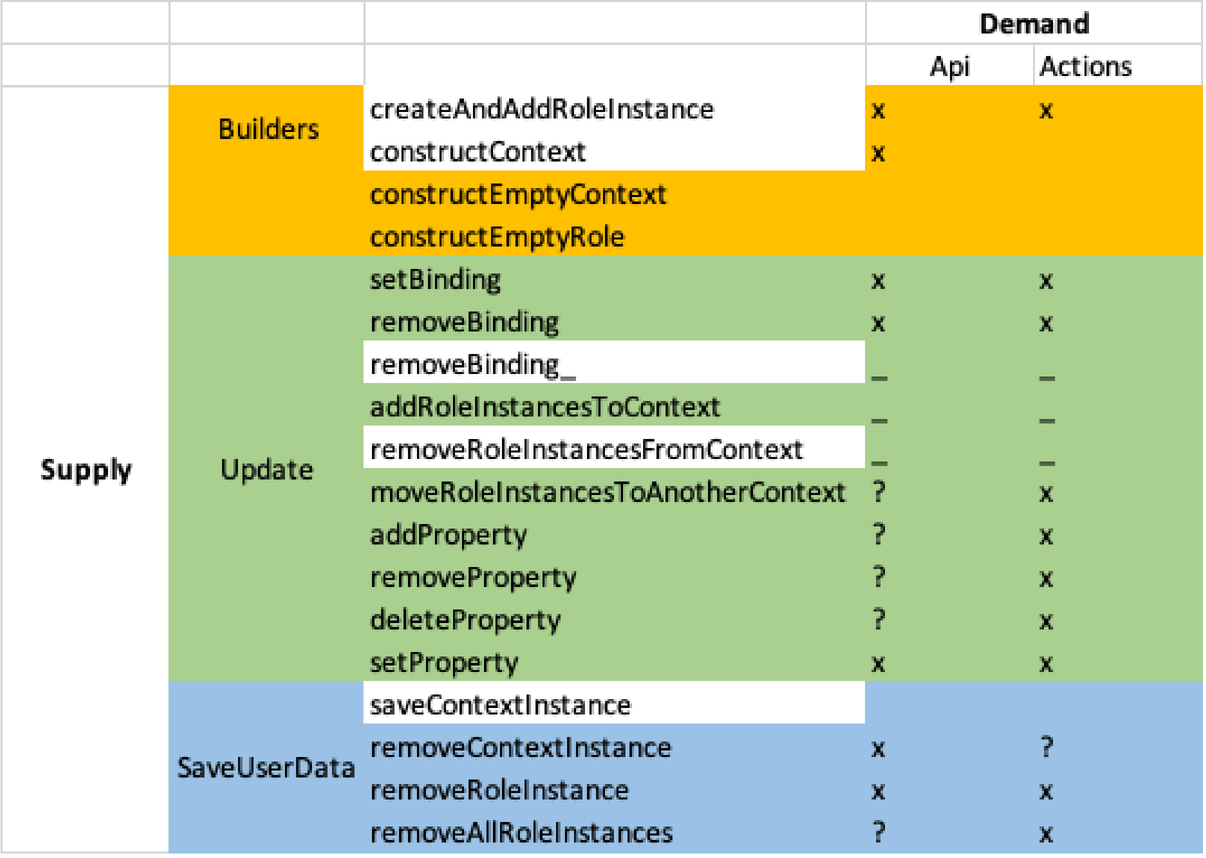
However, some of these functions call others in the same set of modules. Obviously we want to restrict the RPC mechanism to ‘top level’ functions, to minimise the number of calls. We’ll call the RPC functions DeltaFunctions, in honour of the fact that they should add a Delta to a Transaction (Transactions are the unit of shipping synchronisation information between PDRs). A Delta mostly is the ‘serialised’ application of a DeltaFunction to some arguments, but contains a little more information that need not concern us here (such as ordering information and a list of users for whom the Delta is important).

So all functions in these three modules are candidate DeltaFunctions. We have two criteria for a real DeltaFunction:

1. It should be used in either the Api or the Actions module, and then either
   a. directly, meaning that it is the implementation of the request sent by the user through the client program, or:
   b. indirectly, meaning that it is used in the implementation of handling such a request.
2. It should not have a Context- or Role JSON serialization as argument.

To explain the latter requirement: the client program can send a JSON fragment to the API when it wants to construct a context or role instance. However, we don’t want to include such fragments in Deltas because we’d have to specialise them for each peer. As we already have a mechanism in place to select relevant peers per elementary state change, we stick to Deltas with just such elementary changes.

Figure 1 shows what state changing functions are available and whether they are used in the Api or the Actions module.



			Demand	
			Api	Actions
Supply	Builders	createAndAddRoleInstance	x	x
		constructContext	x	
		constructEmptyContext		
		constructEmptyRole		
	Update	setBinding	x	x
		removeBinding	x	x
		removeBinding_	_	_
		addRoleInstancesToContext	_	_
		removeRoleInstancesFromContext	_	_
		moveRoleInstancesToAnotherContext	?	x
		addProperty	?	x
		removeProperty	?	x
		deleteProperty	?	x
		setProperty	x	x
	SaveUserData	saveContextInstance		
		removeContextInstance	x	?
		removeRoleInstance	x	x
		removeAllRoleInstances	?	x

Figure 1 State change function supplying- and demanding modules. An x marks the usage of a function in the current implementation; a ? indicates that it will be used in the final implementation. Functions with a colored (non-white) background are DeltaFunctions.

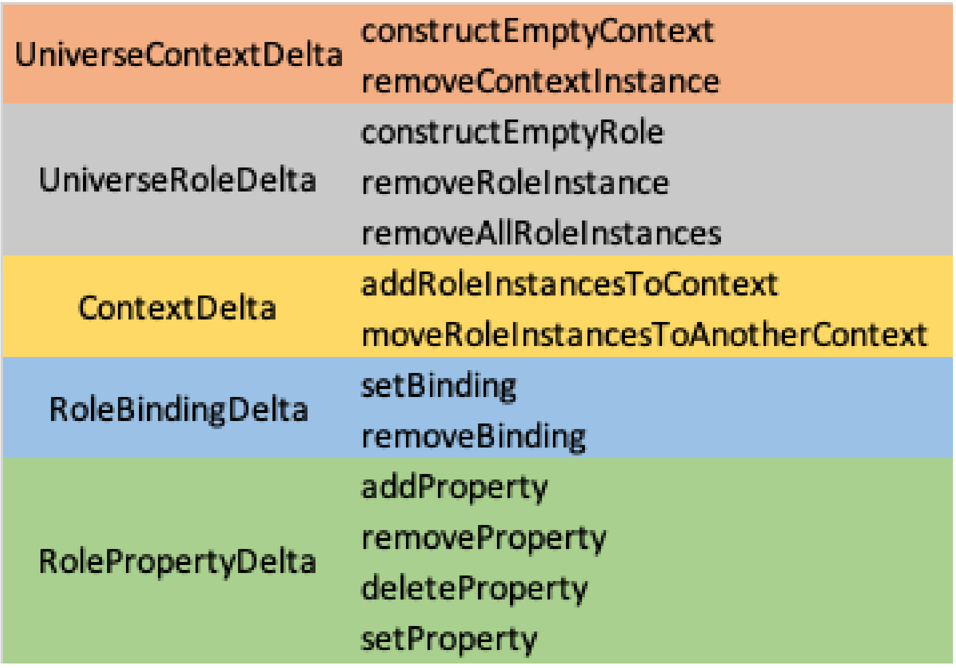
15.1.2. Types of Deltas

We group the DeltaFunctions in five categories, each described by a type of Delta. Some Delta members have a Maybe type if at least one function in the corresponding category has no parameter that binds that member while others do (this allows us to minimize the number of Delta types). Figure 2 gives an overview.

The ‘Universe’ Deltas deal with creation and annihilation of context- and role instances.

The ContextDelta deals with adding or removing role instances to or from a context instance.

The RoleBindingDelta and RolePropertyDelta deal with the two ways to change a role instance.



UniverseContextDelta	constructEmptyContext removeContextInstance
UniverseRoleDelta	constructEmptyRole removeRoleInstance removeAllRoleInstances
ContextDelta	addRoleInstancesToContext moveRoleInstancesToAnotherContext
RoleBindingDelta	setBinding removeBinding
RolePropertyDelta	addProperty removeProperty deleteProperty setProperty

Figure 2. Delta types and Delta functions.

15.1.3. Executing Deltas

On receiving a Delta, the PDR executes the function it specifies, applying it to values of the Delta members according to a fixed member-parameter mapping worked out in the source code.

Deltas are shipped in Transactions. It is important to execute Deltas in the same order as in which the DeltaFunctions were executed by the sender of the Transaction. One cannot, for example, add a Role instance to a Context instance before both are created! This order is preserved in the member sequenceNumber that is in each Delta (we cannot simply compile an ordered list of Deltas, because Purescript collections have to be homogeneous).

The DeltaFunctions are monadic. Their monad is MonadPerspectivesTransaction. Applying runMonadPerspectivesTransaction to the application of a DeltaFunction will actually effect the state change. It will, too, trigger state changes and update queries.

The sending PDR runs a value in MonadPerspectivesTransaction such that Transactions are actually distributed to peers. However, when a Transaction is run in the receiving PDR, state changes are triggered and queries updated, but no Transactions are distributed.

15.1.4. Declarative versus procedural synchronization

Synchronization by remote procedure call is not a very declarative mechanism. We rely on re-executing the original instructions in another PDR to reconstruct the relevant parts of state. Instead, we could try to send a full description of changed data.

While this would be of some interest, it would also be much more verbose. Also, it cannot be complete. This is because users have different perspectives. As a consequence, a particular role instance may be the binding of another role for some users but not for others. So we cannot put that information in a Delta. It would be *indiscrete*, as a user would send information on binder roles to a peer who did not even know of their existence; and it would be *incomplete*, as the user has no way of knowing the entire set of binders for a particular role *according to the perspective of a peer*.

> We have two use cases for this kind of delta. First, such Deltas could be used for an undo mechanism (particularly for deleting entities). Second, a collection of such Deltas would be a valuable base for machine learning or user statistics.]

Interestingly, peers that share a perspective on a role always have the same set of role instances for a particular context instance. We have no ‘conditional’ perspective, where the ability to see a role instance depends on that role’s property values or binding.

> To prevent misunderstanding: we do have calculated roles, that allow filtering based on property values and binding. The set of instances for such a calculated role might be different for each peer. However, the base enumerated set of unfiltered instances is always the same for each peer.

15.1.5. Design considerations per DeltaFunction

15.1.5.1. CreateEmptyRole

Conceptually we want to create a UniverseRoleDelta in the function createEmptyRole. We should compute the users that should receive this delta.

There are several reasons why we don’t want to compute these users when we construct the empty role. For one, if we construct multiple instances (in constructContext), we don’t want to recompute the users for each of them (the result would always be the same). Second, they are the same users as that should receive the ContextDelta that describes how the new role instance(s) should be added to its context instance. But we need but a single ContextDelta for all instances of a role, so we don’t want to create this delta in createEmptyRole.

So instead we compute the users in a function addRoleInstancesToContext and we have that function add the deltas to the transaction. As createAndAddRoleInstance and constructContext are the only functions to call createEmptyRole and both call addRoleInstancesToContext, we have covered all cases.

15.1.5.2. RemoveRoleInstance, removeAllRoleInstances

The DeltaFunction removeRoleInstance (Perspectives.SaveUserData) should add a ContextDelta to the current Transaction and so should removeAllRoleInstances. We’ve refactored the common functionality into an update function removeRoleInstancesFromContext (module Perspectives.Assignment.Update). It is this function that actually adds a ContextDelta, and a UniverseRoleDelta.

Roles are removed, too, when the containing context instance is removed. However, we don’t need – even don’t want – ContextDeltas or UniverseRoleDeltas in this case. The receiving PDR can work out by itself what role instances to remove.

On the other hand, we do need queries to be updated in all cases, taking into consideration the binders of the roles that were removed. We share the implementation of this process with the function addCorrelationIdentifiersToTransactie.

Similarly, we want state changes to be triggered, so we want to know all contexts that have a bot role with a perspective on the removed role.

Finally, for synchronization we need all users that should receive the Deltas. These are users in the same contexts that we needed to trigger state changes. It turns out that it is more efficient to compute contexts and users in a single process. This is implemented in the function usersWithPerspectiveOnRoleInstance. It returns the users and adds the contexts to the underlying Transaction.

15.2. In depth treatment of synchronisation

Consider a context with several thing roles and a couple of user roles. Now assume those user roles have exactly the same perspectives. Anytime one of the peers makes a change, the same transactions (PDR's exchange *transactions* that consists of *deltas*, the elementary changes to structural elements such as contexts and roles) are sent by his PDR to all his peers. This is the base case for synchronisation.

But user roles are defined by their perspectives, so, really, our context has but a single user role. This is a degenerate case indeed: a context with a single user type could be served from a single store, as all participants have access to exactly the same information.

> This may not be obvious at first sight, as roles might have different fillers or properties. Perspectives may be valid in some states only, where state can be defined in terms of fillers, but we explicitly stated that all perspectives were equal. Roles might have different properties, but because all user roles have the same perspectives, these properties evidently play no role in them.

Obviously, the general case is more complicated. It turns out that there are two important ways in which contexts differ when it comes to synchronisation:

1. By having user roles have different perspectives on the non-user roles;
2. By having user roles have different perspectives on other user roles.

15.2.1. Different perspectives on non-user roles

When a given non-user role R falls into the perspective of some user roles but not that of others, a change to that role should be distributed to just the first group of users. This is the base case of differential synchronisation. We have the PDR send changes just to those users with a perspective on the changed object.

15.2.1.1. An aside: trusting transactions

A PDR receives transactions from peers. These transactions reach it through a messaging service (in essence, each user has a *message box* on some server that others may send transactions to).

Anticipating on malicious agents, we ask ourselves the (rhetorical) question: should we trust each incoming transaction? The answer is no, as it is quite possible that the address of an inbox will be stolen. Consequently, a *denial of service* attack could be mounted on that PDR. We can mitigate the gravity of such an attack by abiding by the simple rule that we accept transactions *from peers only*.

This may seem to implicate that perspectives on peers should always symmetrical (run both ways). This is not so. A *peer* is not the same as another user role in the same context. Peers are instances of the User role in the PerspectivesSystem context. Consequently, perspectives need not be symmetric for PDR's to accept transactions coming from the peers behind them.

15.2.2. Perspectives on users: the *user graph*

In the most symmetrical case, every user type has a perspective on every other user type. This makes synchronisation easy. All we have to do for a given change is to calculate which peer types have a perspective on the structural element that has been changed, and then send the change to their instances.

> This may be a context (to which roles may have been added), a role (that may have gotten a filler) or a role one of whose property values has been changed. Finally, we can have the situation that a new context instance has been created.

15.2.2.1. Hidden peers

But we can think of many realistic cases where peers do not all 'see' each other. For example:

- In a scientific peer review, the reviewers are hidden to the authors;
- In an examination, a second examinator may be hidden to the student;
- A controller may have access to financial facets of sales contexts, without clients needing to see that controller;
- A shop has many clients but these generally are not aware of each other.

In each of these situations, disclosing the identity of the hidden roles to the other peers may be considered a breach of privacy. Of course each participant can be aware that *others may be involved in the context*, by reflecting on the model. But they can never recover the *identity* of those others.

15.2.2.2. The User Graph

Consider a graph where the user roles form the nodes and the perspectives on user roles the edges. Incomplete graphs will reveal challenges to synchronisation, as we will see below. In fact, a first and immediate problem will be recognised when the graph consists of two unconnected subgraphs: in such a case, there are really two contexts, with peers that will never know of each other's existence.

Why is that a problem? Because the PDR of a user is only able to communicate deltas to peers it knows about. So if a user in one subgraph makes a change, none of the users in the other subgraph will ever know about it – even if they have perspectives that would allow them to perceive the changed element. See Figure 1 for a simple example. We call this situation *incomplete synchronisation*.

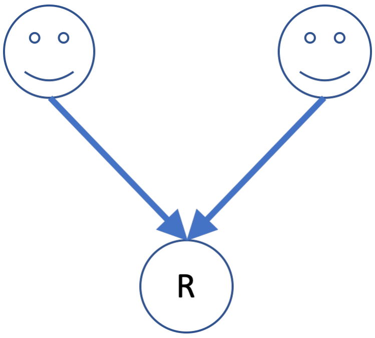


Figure 1. The users are unconnected by perspectives, hence they form two separated subgraphs of the user graph. They both have a perspective on non-use role R. But a change to R made by either user will never be communicated to the other user.

The user graph shows us how users connect, gives us paths between user roles. We will use those paths to spread changes beyond the immediate circle of peers a given user can see.

15.2.3. A synchronisation principle: *passing on*

Consider the user graph (with a non-user role R) in Figure 2. When the left user modifies R, he can send the change to the middle user, but not to the right user. But the right user has a perspective on R, too, so should receive the deltas. As we do not wish to add a perspective from left to right, we need another principle in synchronisation. This we call *passing on*. The middle user should pass the deltas concerning R, on to the right user.

> Remark: a user graph is defined on the type level. However, in the examples we will occasionally treat the graph as if the nodes were instances rather than types. To avoid the tedious distinction between a user and his PDR, we will take the liberty of writing that a user sends a change, while meaning his PDR does so.

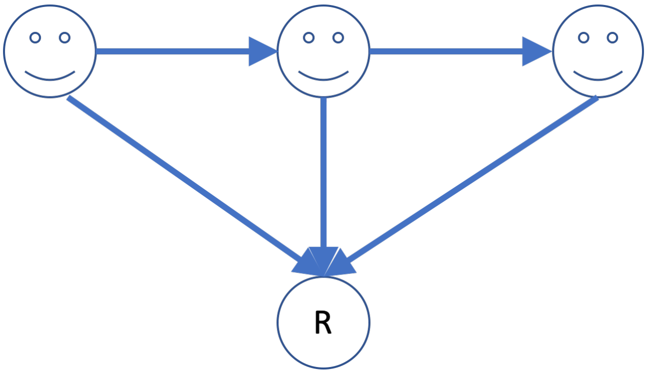


Figure 2. The user graph (top three nodes) is not fully connected. The left user is not directly connected to the right user (also, all arrows are unidirectional). Changes made to R by the left user should be passed on by the middle user to the right user.

15.2.4. Computing users to pass on to

The situation in Figure 2 is easy to comprehend. This is less so, however, in Figure 3. We can explore the graph by building a table with user roles on the rows and columns, and the paths between them in the cells (paths are expressed as user role nodes to pass on the way).

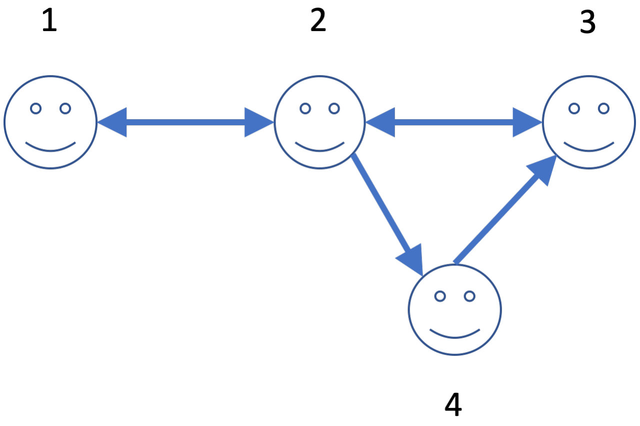


Figure 3 A more complicated user graph. We've omitted the non-user role R. However, all user roles have a perspective on it.

Table 1. Table 1 The paths in Figure 3 (expressed in terms of user role type nodes to pass through) from user role to user role.

	1	**2**	**3**	**4**
1			2	2
2				
3	2			2
4	3, 2	3		

Table 1 shows us that we have to pass through node 2 in order to reach node 3 from 1. This can easily be verified in the graph. Less obvious is the path from 4 to 1: it is through 3 and 2.

Here is how we use the paths table to create deltas that are passed on:

- We start with the user role authoring the change. This we use to index a row in the table.
- We then determine the destination of the delta (according to the perspectives of the peers). This we use to index a column in the table.
- We then add to the delta the user types we find in the cell we've located.

In many cases above, there are no intermediate user types. But if an instance of user type 1 adds an instance of a role R, we create a delta, add user types 3 and 4 to it and send it off to all instances of user type 2 that we know about.

An instance of user type 2, upon receiving that delta, notes that it contains user types (3 and 4). It looks up all instances of those types and sends the delta to them, after removing user types 3 and 4. Obviously, the instance of type 2 also modifies its local store by adding the new instance of R!

An instance of user type 3 or type 4 receives the delta and just modifies its local store.

This is a general principle: it will work in every conceivable situation. We will explore more examples below. It depends, obviously, on the user role graph and the paths table.

15.2.4.1. How to handle multiple paths

It may happen that there are more than one way to connect two user nodes. In fact, Figure 3 contains a few examples. We can reach node 3 directly from node 2, but also via node 4, for example.

This is an easy case, as the first path is shorter than the second. We prefer shorter paths.

But we may have situations where two paths of equal length exist. In such cases it seems tempting to choose one at random. However, we deal with the type level here. It may well be that on the instance level one path may exist while the other is absent. So we should use both paths.

15.2.4.2. Optimization

Let’s reconsider the example we worked out above where an instance of user type 1 adds an instance of a role R that can be seen by roles 2 and 3 as well. What if there are *multiple* instances of user role type 2? If we send the delta to all instances, all will send it on to instances of user role type 3. This doubles the load on those instances while it achieves nothing extra. In fact, only one of them needs to do the task.

In general it is not easy to coordinate work among the peers in a distributed system, but here we are in luck. The instance of role 1 that created the change can act as coordinator by

- Creating an instance of the delta *with* user role type 3
- And creating an instance of the delta *without it*.

It then sends the first delta to *just one instance of user role 2*, and the second delta to all others. Thus, it burdens just a single peer with the task to pass the information on to instances of user role type 3.

It may send the *pass on* instruction to multiple instances of role 2 to increase the chance that it is sent on promptly (not all peers will be online, the more receive it the sooner it will arrive at its final destination).

15.2.5. Adding new peers

A special case of synchronization arises when a user adds a new peer to a context. In such cases, the entire context *as perceived according to the perspectives of the new peer* should be sent to that new peer (and, obviously, the fact that a peer has been added, should be sent to other peers).

15.2.5.1. An example

In Figure 4 we have a context instance C, a role instance 1 and a role instance R. Now 1, having a perspective on a user role type 2, adds an instance of 2 to the context(n)ote how we jump opportunistically between the type- and instance level). Obviously, the instance of 2, being new in the context, has no prior information about it. So the burden falls upon 1 to send all deltas necessary to recreate C and R locally, to 2. In the figure we've shaded the structural parts that are to be sent to 2.

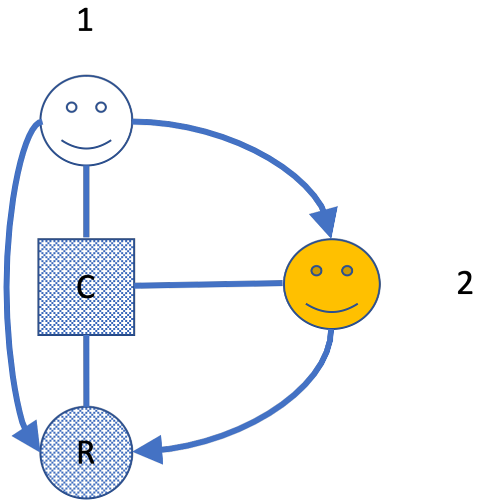


Figure 4. The yellow circle represents a peer added to the context C by user 1. Both peers have a perspective on role R.

This works by handling one by one all perspectives the new peer has on the context. Handling here means: run the queries that provide the objects of the perspectives while jotting down all context and role instances that are passed by the query evaluator. For example: for 2 to arrive at role instance R, the evaluator passes C. In a similar vein all property values are collected. For each of these structural elements, *the original deltas that created them* are sent to 2.

15.2.5.2. When adding a peer brings in even more peers

Consider a webshop in Perspectives. The visitor, deciding to browse the virtual shelves, creates a shopping basket (a context instance). It will contain himself, as (prospective) client, so he can see inside it, but otherwise it will be empty. He will then proceed to add articles to it in some way (presumably by dragging roles representing those articles over it).

Finally, he decides to really buy these articles. Only now will he reveal his identity to the shop, *by dragging a representation of a sales person onto the basket.*

This situation is covered by the example in the previous paragraph, where user role type 1 is the client, R represents the articles and 2 the sales person.

Let's increase the complexity of the case by involving the webshop's financial controller. This role must have access to all shopping baskets to retrieve the financial information from it. We assume the sales person and the controller are roles in the webshop context.

Now, the controller does not need to be an enumerated role in the basket context. We can easily compute this role once the sales person has been added to the basket: we just follow the path from the sales person role in the basket to its filler as employee in the shop, to the shop context itself and then to the controller role.

Obviously, the PDR of the client should disclose some information to the financial controller. But how? It has no access to the web shop context. See Figure 5.

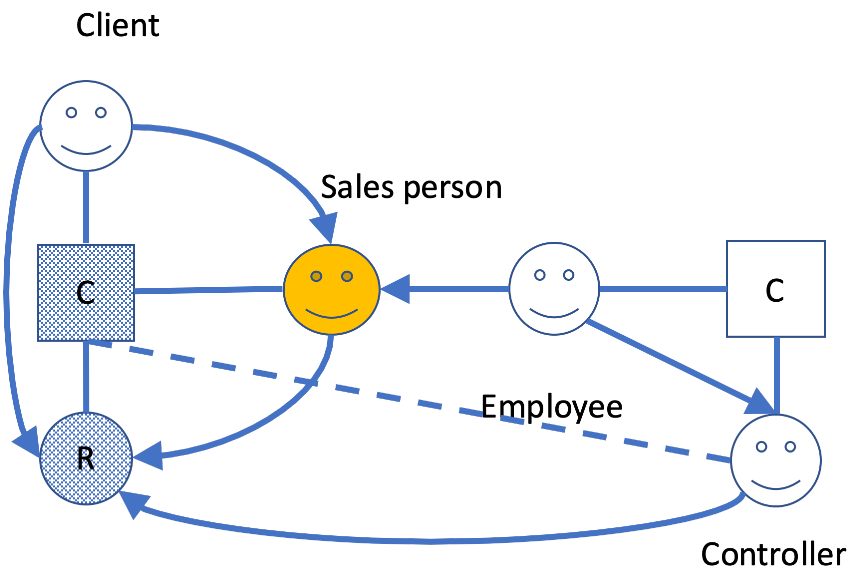


Figure 5. The webshop. The sales person is added to the basket © containing the articles ®. The Controller is a calculated role in the basket (represented by the dotted line). NOTE: this model is not final: see Figure 7.

As a matter of fact, this situation can be handled by the same mechanism of *passing on* we've defined above. In short: the Client is aware of the Controller role and includes it in the deltas it sends to the Sales Person. Thus, the Sales person (or rather the PDR for the peer that ultimately fills Employee) passes these deltas on to the Controller. The Employee has all information to compute the Controller.

But there is a catch we've glossed over. How does Client know he cannot address Controller directly, and how does he know he can reach Controller through Sales person?

This would happen if the user graph were like Figure 6. But Sales person does not have a perspective on Controller; Employee does.

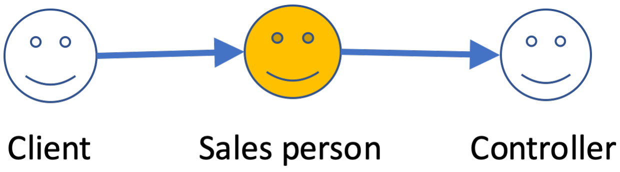


Figure 6. The user graph we would like to derive from Figure 5.

It takes some reasoning to derive Figure 6 from Figure 5. We start with the observation that Controller is a calculated role in the basket ©. We then should note that the user graph represents *connections between PDR installations*. This is because there is a one-to-one correspondence between User roles of PerspectivesSystem and a PDR – intentionally, PerspectivesSystem *represents the PDR*. So because Sales person is filled by Employee (and Employee is filled, in the end, by User) we may include the perspective of Employee on Controller in our user graph.

We will not implement this reasoning. Instead, we restrict ourselves to perspectives on roles in the context. Hence, we require a perspective of Sales person on Controller (as a calculated role of Basket). See Figure 7.

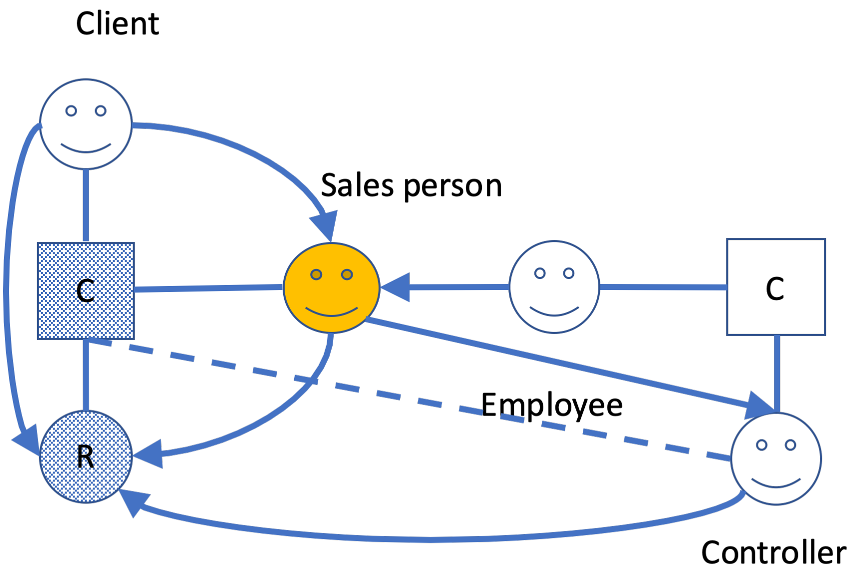


Figure 7. An improved version of Figure 5. The perspective of Sales person to Controller has been added (and we've omitted the perspective of Employee on Controller for cosmetic reasons; there may be reasons to have that perspective that fall outside this example).

It is straightforward to derive the user graph in Figure 6 from this model. Should the model lack this perspective, the system should warn that synchronization will be incomplete.

15.2.5.3. When multiple peers must contribute

What if the new peer has a perspective on something the user who adds him cannot see? Think about a soccer club with multiple youth teams. A parent enlists her daughter and the clubs' administrator assigns her to a team. The teams' trainer then provides her with the training schedule etc. We can model this as shown in Figure 8.

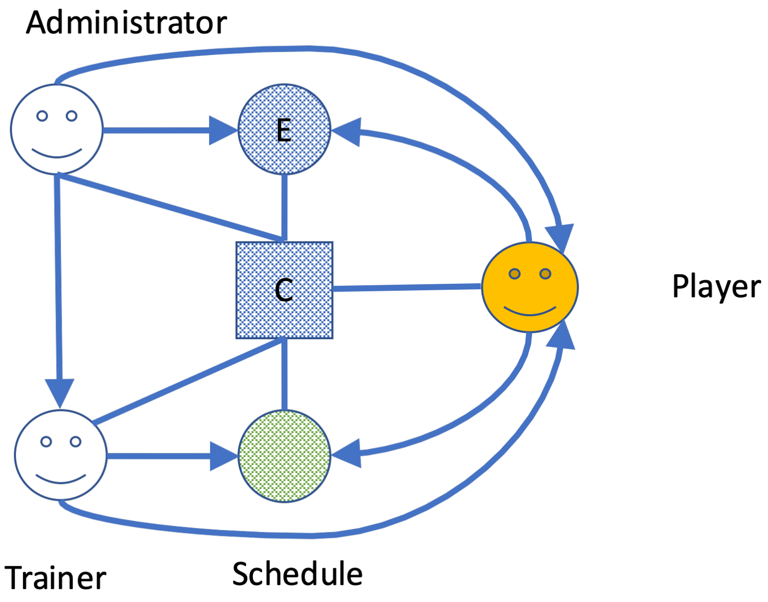


Figure 8. A soccer team © with external role (E) and a Schedule role. The Administrator adds the Player (yellow) to the team and sends the blue items to her. The trainer receives the new player and sends the schedule (green) to her.

The situation is like in Figure 3, but with an extra user role (Trainer) and an extra thing role (Schedule). As the Trainer has a perspective on Player, Administrator sends deltas to Trainer so he can update his local database with the new Player.

However, we can see in the picture that the Administrator cannot send the Schedule to the new Player: he has no perspective on it.

Instead, his PDR makes use of the connection of Trainer to Player. But, unlike in Figure 2, he cannot send a delta along that path – because Administrator has no information on the Schedule instance. However, his PDR can *reason* about Schedule instances (using the model). So instead of sending a concrete delta with details, he sends a *type level delta* to the Trainer. This type level delta contains both the Schedule role type and the path to Player from Trainer. We call this type of delta an *InformNewPeerDelta*.

Trainer, upon receiving a InformNewPeerDelta, verifies if he can find instances of Schedule in his database. He can and therefore creates a delta on the instance level and sends it to Player.

Several caveats here:

1. How does Administrator find out he has to instruct Trainer to send Schedule to Player?
2. Trainer should send the Schedule *only to the new player!*

Determining who should inform the new peer

Administrator should reason like this:

1. What roles does Player have a perspective on?
2. Which of those roles do I myself not have a perspective on? Let’s call them M_i.
3. What roles (other than Player) do have a perspective on M_i?
4. Do those roles have a perspective on Player (how do they connect to Player in the user graph)?

Obviously, as this reasoning is based purely on the model, the other roles could do it too. However, if we have the role that added the new peer, perform the reasoning, it can coordinate the work in the same way we’ve seen before. Trainer is likely a functional role, but suppose there were two Trainers, only one of them needs to inform the new Player.

Making sure only the *new* peer is informed

It requires the special InformNewPeerDelta to make other user role instances inform (just) a new peer. So in our example, only because Trainer receives a delta of this special kind, he informs just the new Player. The user path in such a delta will end in an *instance,* rather than a user role type.

15.2.5.4. Refinement: properties in a perspective

In the above, we’ve dealt with perspectives as if they only concern roles. However, usually a perspective pertains to properties as well. So instead of merely asking whether the new peer sees the same role, we should ask about properties, too.

Here it is useful to introduce the concept of the *difference between two perspectives*. Let’s conceive of a perspective as the combination of a role type and a set of properties: <R, props>.

We define P1 minus P2 as follows:

- If the role object of P1 is not equal to that of P2, the difference is P1: P1 – P2 = P1.
- If the role objects are equal, the difference is the a perspective with the same role combined with the difference of the sets of properties in the perspectives: <R, props1> – <R, props2> = <R, props1-props2>.

Let’s start by subtracting from a perspective of the new peer (let’s call that P1). For each perspective P2 of the user that adds the new peer (the *initiator*), we subtract it from P1. If the difference is P1, it is of no use to us. If the (property set of the) difference is empty, we can inform the new peer completely from the PDR of the initiator. If the difference is a non-empty set of properties, we’ll have to enlist the help of other peers.

Remember we can only make use of peers who are on a path between the initiator and the new peer!

For such a peer, we evaluate one by one his perspectives. From such a perspective we subtract the difference we obtained above. If the result is empty, we’re done. Otherwise we’ll have to take the remaining difference and try another peer, until we’re done.

The above algorithm ends with either a set of peers, each combined with a partial perspective to send to the new peer, or with the conclusion that synchronization cannot be complete, it does not give the best result in the sense that we’ve enlisted the help of more peers than needed. This can be fixed easily but is computationally more intensive.

But there is another issue. Consider this situation: the initiator can send the object of a perspective and some properties, but not all. Another peer need only send the rest of the properties. How do we do that?

We’d need to make clear *exactly what the peer doesn’t need to send*. That is not trivial. Properties may reside anywhere on the role telescope. The initiator may have sent part of that telescope, while the peer needs to send a further part, that bears the required properties (and those fillers require their contexts too!).

I see no easy way to communicate, to an enlisted peer, what he doesn’t need to send. So as it stands, we accept that enlisting a peer to complete a perspective for a new peer involves overhead for both that peer and the new peer (who’ll receive some role- and context information more than once (this is not a problem in the sense that handling a delta is an idempotent operation; there is no difference to the state if it is added more than once)).

Refinement of the notion of InformNewPeerDelta

In the light of this discussion, we can exactly define the – up till now informal – notion of a InformNewPeerDelta. It is the combination of three things:

- A user path, in terms of user role types;
- A final destination, in terms of a user role instance;

- A simplified perspective, as the combination of an object (to get really technical: the object is given in the form of a QueryFunctionDescription (the internal representation of a query path to a role)) and a set of property types.

15.2.5.5. Reaching out to peers along a user path to make them inform a new peer

Finally, a really complicated case that combines aspects of what we've seen above. With a stretch we extend our soccer example by including a Trainer Assistant, who is in the know of the Schedule, while Trainer himself has no access to it (unlikely, but just for the sake of the example assume it is so). Moreover, Administrator has no direct perspective on Trainer Assistant.

This requires Administrator to send an InformNewPeerDelta along a user path to Trainer Assistant. The path is via Trainer. See Figure 9.

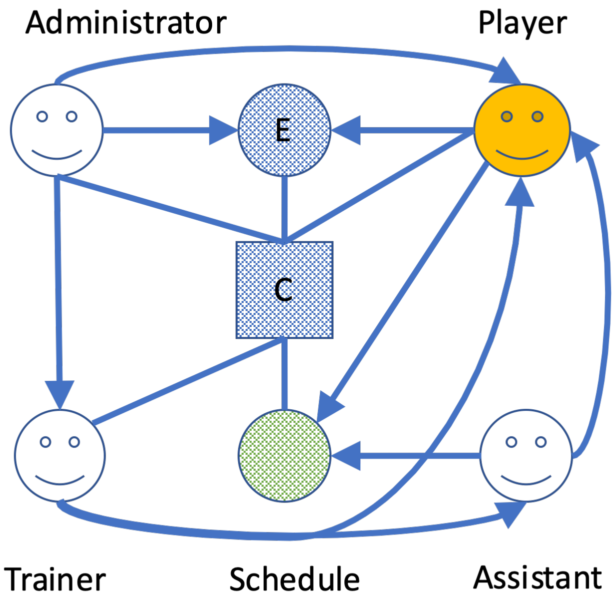


Figure 9. An extension of the situation given in Figure 8. The Trainer now has an Assistant. Trainer cannot see the Schedule, but Assistant can.

Administrator must instruct Assistant with an InformNewPeerDelta, so includes the path from himself to Assistant in it: (Trainer, Assistant) (see Figure 10). The new Player instance is included as the final destination. And the Assistant perspective on the Schedule is included, so Assistant knows what to send to Player. We assume the Assistant perspective on Schedule covers that of Player.

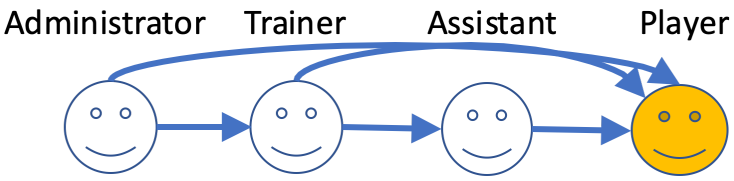


Figure 10. The user role graph for the model in Figure 9.

15.2.6. Implicit perspectives

In Figure 1 we presented two user roles with a perspective on the same role R, that did not see each other. We speculated that if one of those roles modified R, the other would never know. Thus, to synchronize properly, we must add a perspective from the modifying user to the other user. In Figure 11 we've reiterated that situation. We've drawn the context as well and we've indicated which of the users modifies the role R (by starting the perspective arrow with a closed circle). We've also drawn the *inverted query* that originates in R and leads to user role 2. This is the inversion of 2's perspective on R.

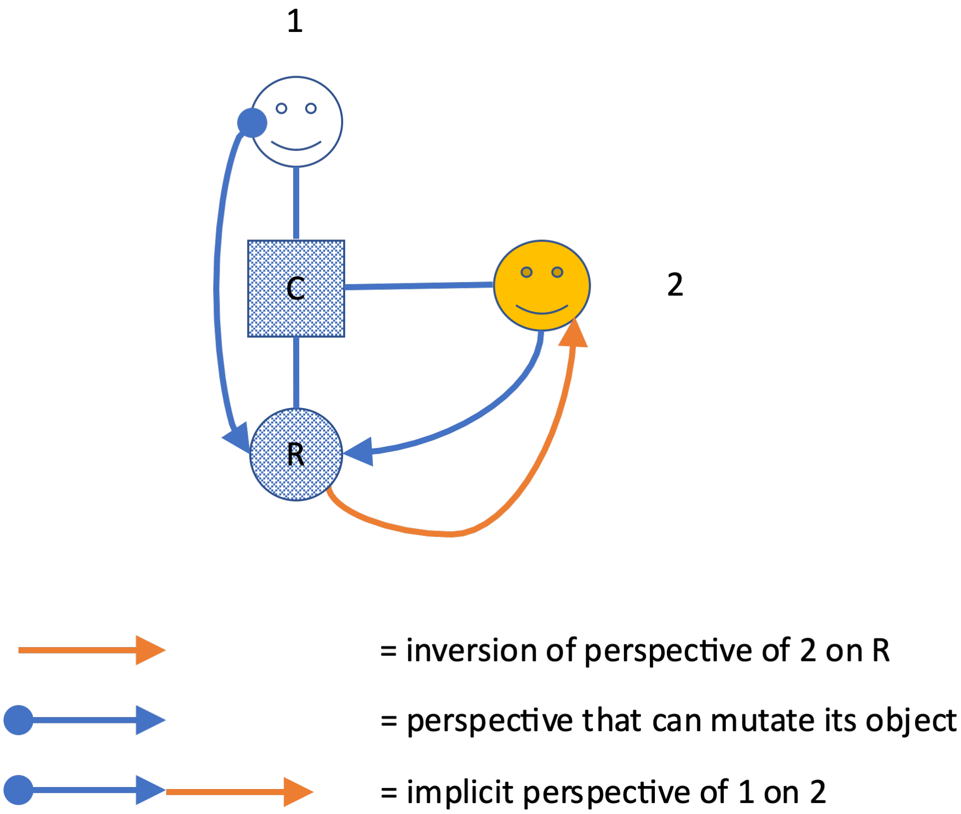

1
C
2
R
= inversion of perspective of 2 on R
= perspective that can mutate its object
= implicit perspective of 1 on 2

Figure 11. User role 1 can modify role R. User role 2 can 'see' role R. This implies that 1 should send his modifications of R to 2, hence has an implicit perspective on 2.

As shown in the figure, we might consider the combination of the modifying perspective on R and the inverted perspective from R to be an *implicit perspective of user role 1 on user role 2*. In other words, we interpret this situation as if there really **was** a perspective of 1 on 2.

Consider the impact of that interpretation. It would mean that as soon as an instance of 2 was added to context C by some role 3 (not drawn), 3 would send that instance to 1, because 1 has a perspective on role type 2.

Similar reasoning applies to the inverse situation that arises as an instance of 1 is added to C. This new peer would receive the user role instance of 2.

In other words, under this interpretation of inverse perspectives, there is no need for explicit perspectives on user roles for the good of synchronization.

However. It would constitute a security breach as we've discussed in the paragraph *Hidden peers*. In the example of the peer-reviewed article, the identity of the Authors would be disclosed to the Reviewers, and the other way round (as their comments on the Article would be directly sent by their PDR's to the Authors'). We may be able to limit the disclosed information to just that needed to send transactions (omitting the identities and other personal information). However, the address information that must be exchanged could be a real giveaway.

For this reason we will not interpret inverted perspectives as implicit user perspectives.

15.2.7. Perspectives over model boundaries

What happens if we give a user role a perspective on a role in a context that is contained in an *imported model?* We can easily do so by either adding that role as a calculated role to our local context, or by specifying the object of the perspective with a query.

As a consequence, modifications made to the object of the perspective should be sent to our user role. But, and this is a **big** but, there may very well be modifying roles in the *original model*. They should have a perspective on our user role in order to be able to send their deltas to it – but this would reverse the dependency relation between the two models!

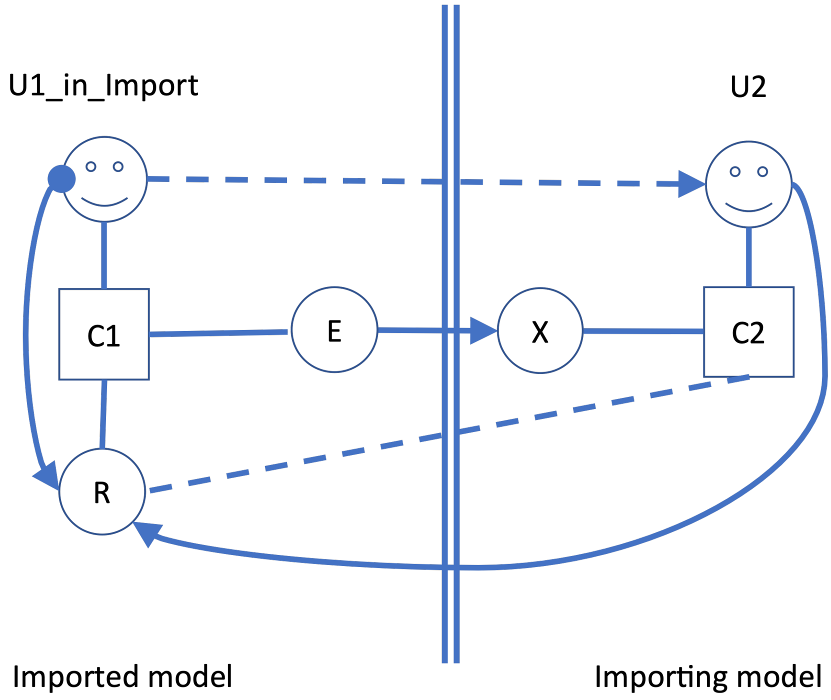


Figure 12. Crossing the border between an importing model (right) and its import. The dotted line between context C2 and role R indicates that R functions as a calculated role in C2. The dashed perspective line from role U1_in_Import to role U2 represents the perspective that is necessary for U2 to receive updates to R made by U1_in_Import.

Figure 12 shows us a model of this situation. On the left we find the model that is imported; on the right the model that imports. Now it is important to realize what happens when the *importing model* is created. It may refer back to contexts and roles in the import, as is shown with role R, that is added to context C2 as a *calculated* role. The query path that calculates it runs from C2 to X, to E, to C1 and then to R. This is a perfectly normal query.

The synchronization problem arises from the fact that user role U1_in_Import has a modifying perspective on R. When it modifies R, it should send deltas to U2 – but it can have no knowledge of U, as the imported model was written before the importing model!

We cannot even adapt the imported model, because it would then need to refer to a role in a model that imports it. This cannot be: the import relation between two models runs one way only.

15.2.7.1. Solution

We can solve this by adding perspectives *in runtime* to the imported model. This is not as crazy as it may seem; as a matter of fact, we routinely add inverted queries to imports as importing models are added to a PDR installation.

It is important, however, to understand that this only solves the problem when *both the PDR installation whose user fills U1_in_Import, and the PDR installation whose user fills U2, have the importing model*.

To make things easier, when you look at Figure 12, identify the part to the left of the double line with one PDR, the part on the right with another.

In other words, both peers must have the importing model. Otherwise either deltas will not even be sent (when the importing model is absent on the left), or no destination for them will be found (when the importing model is absent on the right).

15.2.8. An alternative way to *pass on*

In the chapter “A synchronisation principle: *passing on*” I describe in conceptual terms an algorithm to make sure a particular delta reaches all concerned peers (peers that have the modified structural element in one of their perspectives). Here I present an alternative algorithm that may be easier to implement.

15.2.8.1. Overview

Let’s start with the user who makes a change (the *Initiator*). He will send a transaction consisting of deltas. To implement the algorithm, these deltas must be augmented with two elements:

- The *user types* with a perspective on the structural element that has been changed (AllUsersWithAPerspective)
- The difference between that set and those types that the Initiator has a perspective on (ToBeInformed). These are types whose instances the Initiator cannot reach.

Now consider a peer receiving such a transaction. He will carry out the change described by each delta, so his database is in sync with that of the sender. Next, he’ll compute all user types he has a perspective on and

- Compute its union with ToBeInformed and
- Subtract it from ToBeInformed, to form RemainingToBeInformed.

For the first set, he computes the user instances known to him. For them he creates a personal transaction and adds this particular delta to it, *after replacing the element* ToBeInformed *with* RemainingToBeInformed.

When all incoming deltas have been handled in this way, he will send the personal transactions that have been built in the process to their respective recipients.

This algorithm will make sure that no user receives the same delta twice and it guarantees that all users receive the deltas they need.

15.2.8.2. Some implementation details

Computing the set of user types connected to a particular user type

This algorithm requires all users to compute the users they have a perspective on, quite often. It seems worthwhile to compute this graph once during the processing of the model source file and save it with the DomeinFile. We compute this graph for *the entire model*, rather than per context, because when we deal with a particular change, users outside the context may have an (implicit) perspective on it.

We start with the perspectives stated explicitly in the model source text. These form the base of the UserGraph.

Calculated User Rule. For the next step, realize that some user roles having a perspective may be calculated. We compute the extension (in terms of Enumerated Roles) for such a user role and add them to the graph. We then connect all these new roles to the same targets as the original Calculated User Role.

Inverted Calculated User Rule. The user role that a perspective is on, may be calculated, too. Like above, we add to the graph all Enumerated Roles that form the extension of the Calculated target and we connect the user having the perspective with all of them.

Filler Rule. An Enumerated User Role U1 that is filled with another User Role F1, gives rise to even more connections. If U1 has a perspective on U2, its filler F1 should be connected in the UserGraph, too. It’s not that the filler has the same perspective; but we know that F1 ‘can see’ U2.

Inverted Filler Rule. An Enumerated User Role F1 that has a perspective on another user role U2, will be connected in the UserGraph with an arrow. But then a user role U1 that is filled by F1 may be connected to U2 as well. After all, U1 and F1 represent the same user (or installation) and the UserGraph shows connections between users.

Computing user types for a delta

For any given delta, we have to compute all user types with a perspective on that type. Notice that because a user role may have a perspective on a calculated role, not only the end results of such calculations are in scope for him, but all intermediate steps are so, too. How to compute them?

This is where inverted queries come to the rescue. We have established, while processing a model source text, an inverted query on each element a user can ‘see’ by force of a perspective. So, starting with the type of a structural element, we can read off all user types that have it in scope.

These user types are contained in the newtype InvertedQuery. We use this type to compute users instances that should receive a particular delta. To implement this algorithm, we should add some code to collect the user type from all inverted queries at a particular role- or property type and then add them to the delta’s constructed for the modification.

Optimization by a refinement

What if there are more than one user instances for a particular type? All of them would carry out the above algorithm, duplicating work and moreover burdening peers with double work too, if they have something to pass on.

We can avoid that by having the Initiator select a single instance of each type at random and only send him a delta with a non-empty ToBeInformed. The others receive a delta with an empty ToBeInformed. Subsequently, each peer that receives the delta does the same. This guarantees that for each user type, just a single instance will pass the delta on to types that could not have been reached before.

15.2.9. Connecting users

It turns out that a user graph that consists of unconnected subgraphs certainly means synchronization cannot be complete, but it is not the other way round. In other words, this test is not strong enough. Have a look at Figure 13. There are no unconnected subgraphs, but only changes made by Client will be communicated to Sales Person and Controller.

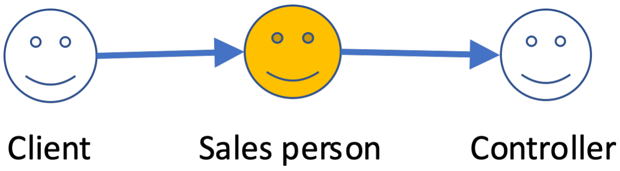


Figure 13. A UserGraph that does not allow synchronization of changes made by Controller.

A better test is to check whether, from a node representing a user with a modifying perspective, all other nodes representing users can be reached. This can be done with a simple depth first search (while preventing loops).

We still want to project the graph for a given mutation, but on top of that we need a list of nodes that represent a user with a modifying perspective.

15.2.9.1. A special case: modifying a property on the role graph of a user role

Consider Figure 14. This user graph allows for complete synchronization, as the left user is the only one to modify R *and* has a perspective on the only other user who, in turn, has a (consulting) perspective on R.

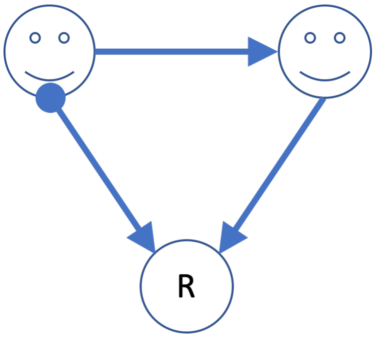


Figure 14. The left user has a modifying perspective on R, while the right user has a consulting perspective.

What about when it is the other user itself that is being modified? Figure 15 depicts this situation, with an added twist.

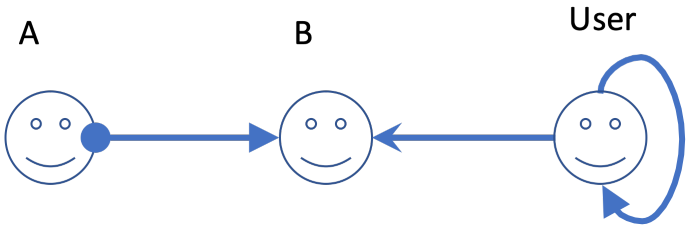


Figure 15. User A modifies a property on user B. B is filled by the system role User, having a self-perspective.

Now suppose that a modifies a property P on User – by virtue of having a perspective on B (this is allowed, as a perspective includes all fillers of the role, recursively). Clearly, because User has a self-perspectives (and we make it include P), he should be informed. The user graph projection for P is given in Figure 16. The synchronization checker will report a problem: changes made by A to P will not arrive at User! This is because a has no expliciet perspective on User.

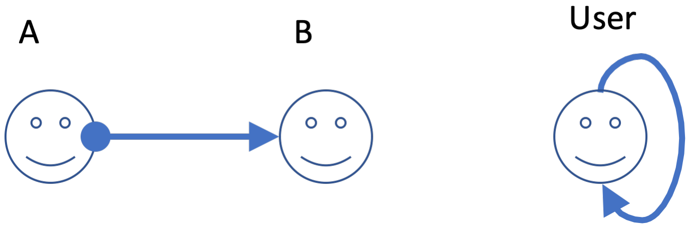


Figure 16. User graph for Figure 15. There is no path from A to Role “User”!

However, a user role with a perspective on a calculated object needs access to all entities along the query path. We say this user has an *implicit perspective* on those entities. This is worked out by establishing an inverted query on each entity along a query path, for the user having the calculated perspective.

We consider the property P on User to be a *calculated property on B*. Consequently, there will be an inverted query on P saying that when it changes, A should be informed.

In other words, A has an implicit perspective on User. This is not just theory but works in practice: User makes sure that A has all information about him that his (A’s) perspective allows – due to the inverted query on P.

So the simple connection check between A and User gives us a false negative. Should we include implicit perspectives when we build the user graph? It turns out that this is not necessary (and that is a good thing, because recursively adding perspectives to all fillers of a user role would make the graph grow very big).

Instead, we can simply skip the connection check for projections on a user role U and its properties, *for a modifying user M and U*. The reasoning goes like this:

1. We should check whether M can reach U;
2. We do check because M modifies U;

3. Hence M can reach U.

We should still check *other* user roles with a perspective on U. M may or may not have a perspective on them.

15.3. Synchronizing subnetworks due to role filling

In the text *Perspectives on Bindings* we've concerned ourselves mainly with changing values of Properties of a Role. Another relevant change is that some user adds a binding to a Role. While we have (in version 0.4.0) a mechanism in place that informs peers of such an isolated change, we yet lack a mechanism to provide them with the necessary deltas to ensure they have access to the role graph of which that binding is the root. This text describes that mechanism.

Before we do so, it is important to notice the large differences between binding a role and the other modifications, to build some intuition for the magnitude of this operation.

15.3.1. Binding: the way to build large structures

Adding roles and properties will build up isolated contexts. However large these may be, their structure is simple. Only by creating bindings between roles can we build more complicated structures. In general, the data structures that we can create are *graphs* of contexts and roles.

User roles have perspectives. For the users, they make roles in and around their context *visible*. The modeller may define quite long paths through the graph, that give a user playing a specific role the right to view roles in far-off contexts. These paths are defined over *types*.

When in runtime a graph consisting of *instances* is built, more and more roles become visible to the end users. Now imagine two as yet unconnected subgraphs of instances and a user role U in one of them. The other subgraph has been created by other uses and is not yet available to U. Let's assume that for users of that type, a path is defined that runs through both subgraphs. The moment that the right binding connects both subgraphs, user U should suddenly be able to 'see' into the other subgraph.

> One may wonder: what is the graph that they are subgraphs of? In our example we assume that part of it is available to one user, while another part is available to others. It helps to image a very large graph – we call it the Perspectives Universe – that contains all data kept by all users of Perspectives. There is no single place (no computer) where this graph exists in its entirety. Each user sees part of it: what he sees, falls within his *horizon*.

This must translate into a whole set of deltas to be shipped to U, so that his PDR can construct that formerly invisible subgraph.

15.3.1.1. This algorithm is suboptimal

We send deltas for the entire subgraph that should be visible to peer P. However, P might already have access to parts of that graph – indeed, maybe to the entire graph. So the set of deltas we prepare may contain too much information.

On receiving the Transaction, P will have to handle this: detecting a UniverseRoleDelta for a Role instance he already knows, he will simply not create a new instance. The other deltas are treated similarly.

While this is not optimal, there is no quick check to find out whether P already has access to parts of the graph. We might run the inverted queries stored with the type of a node in the graph to find all peers having access to it. If P is one of them, we don't have to send a new UniverseRoleDelta. However, this might be a very expensive computation. Indeed, when we bind a user role to a graph that bottoms out at the 'own' user, that query would return all peers with access to the own users' properties – potentially many hundreds or even thousands of them!

15.3.2. General approach

Consider, again, the simplest case of a query that computes a role set: a series of steps (a path). It will be applied to a context instance. Its first step will take it to a role instance. At that point, we can invert that first step to bring us back to the context of origin; we can also run the rest of the query and it will bring us the results *as computed from that role instance*. Notice, that the first step may have landed us on *multiple* instances of a Role!

We can repeat that for each successive step in the query: invert the steps we've taken so far, to take us back to the context of origin; run the rest of the query to get (part of) the result set.

Notice, that the steps back always form a path. That is true, even, if the original query is a tree! No matter how high up in its branches we are, the way back is always a path.

To return to the problem at hand, suppose that some user U has two subnetworks within his horizon. Another user, P, has only one of these within his horizon; the other is hidden. P has a perspective on a calculated role that will fetch him roles from the hidden part, but its query execution halts at a particular node that lacks a binding. That binding would, as it were, bridge the two networks for P.

Now U constructs that binding. Surely, U should now send him that formerly hidden network.

We can envision the two parts of the query at the node R that received the new binding. One part is the inversion that leads us back to the context of origin of the query. Here is P, who has a perspective on that particular calculated role. The other part is the rest of the original query. This is how we find out what to send to P:

1. Apply the inverse query to R, to find the context of origin and with that the user P (we store, with an inverted query, the user role types that have a perspective on the query whose inversion it is).
2. Apply the rest of the query to R, to find all contexts and roles that it visits. These we must send to P.

Intuitively, we have 'kinked' the query at R. The left part we invert: it takes us backwards to the context of origin. The right part we keep as it is: it takes us forwards to the query results.

So here is the outline of our approach:

1. In compile time, we do not merely invert queries at each step; we *kink them*, keeping both the inverted backwards part and the remaining forwards part. These we store with the role types (just as we have described for inverted queries in *Query Inversion over Model Boundaries* and in *Perspectives on Bindings*).
2. In run time, we do some extra work for RoleDeltas, to find the extra nodes we should send to some users. This involves running the remaining forwards part of the kinked query.

15.3.3. Finding nodes to be sent from query assumptions

When we run a query, we obtain a role set as result (or a set of values in case of a Property query). However, the query may have visited many intermediary roles that should be sent to P as well. To find them, we can re-use the existing *dependency tracking mechanism* we deploy to keep up to date the results of queries sent in by the client through the API (a client program requests a query to be executed and expects to be notified when the results change due to changes to the underlying network of context- and role instances).

The dependency tracking works by accumulating *assumptions*. Each query step adds an assumption.

Step	Assumption constructor	Assumption parts	Query triggered by:
role R	RoleAssumption	the context instance + R	Any change to instances of R
external	External	the context instance	Never (external role is fixed)
binding	Binding	role instance	Changes to the binding of the role instance
binder R	Binder	the role instance + R	Changes to the binders R of the role instance
property P	Property	the role instance + P	Changes to the values for P of the role instance
context	Context	the role instance	When role instances are moved to another context.
me	Me	the context instance + maybe the role instance	When a user role is added to the context that is ultimately bound by the user of the system.

From these assumptions, we can derive the role and context instances that should be sent to P.

For a RoleAssumption, we create a UniverseContextDelta, a UniverseRoleDelta and a ContextDelta. These instruct P to create an empty context, an empty role and to connect the two.

We do the same for the Me, Binding, Binder, Property and Context instance.

For the External assumption we do nothing, because a query with a kink by construction never starts with the external step. So whenever external is applied, a context step will have been applied before (as it is the only way to get to a context). That step already adds all deltas that we could wish to add for an external step.

15.3.3.1. This algorithm is suboptimal

We currently (version v0.5.0) handle each assumption in isolation from the rest. That causes a lot of deltas to be generated twice. For example, a query where a context step is followed by a role step will create the ContextDelta twice. Only one will be added to the transaction, but we could avoid the double work if we take the *order* of assumptions into account and keep an eye on the history of assumptions we've handled. This is for future optimisation.

15.3.4. Property set

User P will have a perspective on the Calculated Role that allows him to see the value of some Properties. We create Deltas for these Properties by obtaining, for each such Property, its value from each of the role instances in the set that results from applying the forwards part of the query. Remember that we run that forwards part from the node that U just added a binding to. Getting a Property's value from a role instance generates assumptions, too, so this ties in nicely with the general approach outlined above.

Now, what if the original query actually *ends* at the role that U added a binding to? We then obtain the property values from that node itself. As a justification: the remaining forwards part of the query is empty: we could construe that as the identity function, that, applied to the role instance, yields itself, so it forms by itself the result set of the remainder of the query.

15.4. Aspects require refinement of Inverted Queries

The text "Query Inversion" introduces the concept of an Inverted Query. Briefly, this is a mechanism to ensure synchronisation. A perspective grants a user access to entities (roles, contexts, including bindings and property values). On the type level, we associate inverted queries with, say, a Property type that will help us find the User role instances that should be informed about a modification of the Properties' value on a given role instance.

Aspects are nothing else but Context- and Role types that are *added* to other types, to enrich them with Role types (when adding a Context as Aspect) or Property types (when adding a Role as Aspect). We will call the property P of a role R an *Aspect Property* if the lexical context of P is not R, but a role AR of some other context AC. It helps to prefix the local property name with its lexical role context to recognise it as an Aspect Property: so AR$P is an Aspect property of R, while R$Q is a *local* property of R.

In this text I identify a problem that arises during synchronization using Inverted Queries for Aspect Properties and Aspect Roles.

15.4.1. The problem in detail

Consider Figure 1. It shows a somewhat contrived model for a family and the birthdays of father and mother, respectively (all contexts). Both birthday contexts use an Aspect Party that has a role Present with a property Price. The role Present Father uses Present as Aspect Role and so does

Present Mother. The result is that both Birthday Mother and BirthDay Father have a Price property on their Present roles.

Furthermore, the Father role of Family is a calculated role in Birthday Mother and has a perspective on Present Mother. Notice the cross-over: Father has access to the Price property of the Present Mother role; Mother has access to the Price property of the Present Father role.

Consequently, this perspective is inverted and attached to the Price property, leading to Birthday Mother, naming (the Calculated) Father role as one that should be informed when Price changes. Symmetrically, we have an inverted query from the Price property to Birthday Father.

But remember that Father is calculated. So when we detect a Price change in Present Mother, we first find an instance of Birthday Mother and then calculate Father.

Now look at the red lines from Price to Father and Price to Mother. They consist of many hops from entity to entity and this reflects the query path that connects the property to the user role that should be informed when the property changes (consisting of two parts: the *inverted* path from price to Birthday Mother, and the *normal* path from that context to the Father role of Family). Obviously, when the price of mother's present changes (maybe a child sets it), father should know (but mother should not!).

We can now state the problem precisely. The *two* inverted queries are *both* stored with the definition of Property Price. When the actual property value of the actual role instance of Present Mother changes, we look up the inverted queries stored with Price and execute them to find the role instances that should be informed. So we find two queries. But, by coincidence, both queries will give a result when applied to either the price of Present Mother, or to the price of Present Father. Hence, we'll find that both Father and Mother will be informed when the price of either present changes – clearly an unwanted situation and not the one that was modelled.

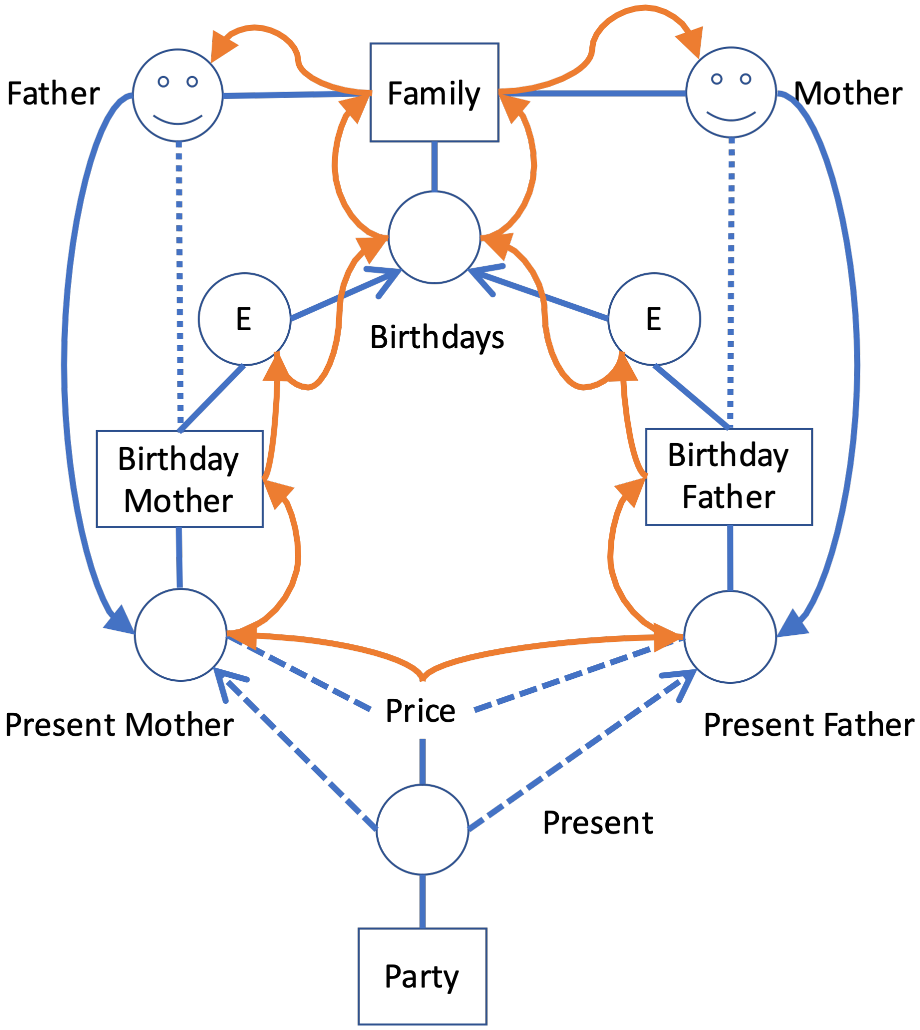


Figure 1. This model contains a synchronization problem for Price. The red lines are paths to follow when Price changes; the lines with long dashes represent an Aspect relation. The lines with short dashes represent the fact that the roles are calculated with respect to their contexts. So: Father is an Enumerated role of Family, and a Calculated role of Birthday Mother, while Price is an Aspect Property of Present Mother (and Present Father, too).

15.4.2. Making sense of this situation

We should consider the property Price *as added to one role*, Present Mother, to be different from the property Price *as added to another* (Present Father). Think of them as templates, where their incorporation into other role types stamps them into something new. Price *as used in Present Father* is a different property from Price *as used in Present Mother*. Changing either of these different properties should not be mixed up with changing the other.

A similar thing holds for role Aspects. In fact, at the type level, we should identify roles with the combination of a context- and a role type: RoleInContext.

This is important when we analyze and use *paths through type space*. A query, and consequently an inverted query, is such a path. We should describe such paths in terms of Context types and RoleInContexts, rather than Context types and EnumeratedRoleTypes.

When a user makes a change to the structure of roles and contexts, she adds or removes a role to a context or fills (or clears) a role. We describe such a change in terms of a *Delta* that identifies either a context- and role instance, or two role instances. As a role instance representation contains a direct reference to a context instance, we can immediately derive a RoleInContext type combination from a role instance. Hence, we can identify with certainty two points of a path through type space. With those points in hand, we can find inverted queries that run through them and use those to find the peers that need to be informed.

We really should use two points rather than one. The Delta describes a path of length two; we want to make sure we only follow (inverted) queries that incorporate that entire segment. Were we to use just one of the two points, we would include all relevant queries, but would err on the other side by including paths we don't want to follow.

15.4.3. Solution

All this means that the implementation up till version v0.12.0 wass incorrect.

15.4.3.1. Separating groups of inverted queries conceptually

Inverted queries starting on a property

We should be able to distinguish between Inverted Queries on a Property type for the various EnumeratedRole types that they are added to. Currently, Inverted Queries are stored in an Array in the representation of Property. This we will change to an Object, where the keys represent EnumeratedRoleTypes.

Inverted queries starting on a context

When a new role instance is added to a context, we must follow queries of which that segment from context to new role instance is a part. The relevant Inverted Queries are stored on the Context type in the invertedQueries collection (there is no need to make this name more discriminative, as only one step (the role step) leads away from contexts). A ContextDelta gives us both the context instance and the role instance. In principle, we need to identify the type of the role instance with a RoleInContext; but, unsurprisingly, its Context type part will by construction be the type of the Context instance. Hence, we can properly recognize InvertedQueries fitting our Delta by the EnumeratedRoleType alone, in the collection of inverted queries on the appropriate Context type.

A role type may have more than one instance in a context instance. Because we do only need to evaluate the new (or removed) segment, we do actually not apply the inverted query from the context instance, but from the new (or deleted) role instance. This is somewhat confusing. We treat these queries as if they start with the context step, while they actually do not.

Inverted queries starting on a role

Let's reconsider the various Inverted Queries that are stored with an EnumeratedRoleType. We categorize them according to their first step, that determines whether the path leads to the role's context, its filler, or the roles it fills.

For queries that start with a context step, the relevant queries in the model are those that start on the RoleInContext that we derive from the ContextDelta. But as we store the InvertedQueries on the EnumeratedRoleType, we can separate out the relevant inverted queries with the Context type alone, similar to the role step discussed above.

Inverted queries between two roles

The fills step and the filledBy step move between two role instances. We can derive a RoleInContext from both and that means we can find the relevant inverted queries by the *combination* of these two RolInContexts.

15.4.3.2. Representation of Queries

We represent queries with a QueryFunctionDescription that gives us the domain, range, function, and some meta-properties and the argument expressions that supply values to be bound to function parameters. Domain and range are constructed as an Abstract Data Type (ADT) of a base type and role domains are based on EnumeratedRoleTypes. We see now that they must be based on RoleInContext.

15.4.3.3. Runtime indexing

Let’s illustrate the above with some examples.

Consider an Aspect Role Driver, to be added to both a Car and a Train context type. Suppose both contexts have other user roles that have a perspective on the Driver. This would establish inverted queries for both contexts on the Driver role. Clearly, we must be able to distinguish the queries for the Car context type from those for the Train context type. In other words, we have a RoleInContext Train Driver and a RoleInContext Car Driver.

Runtime indexing: context step

In runtime, how do we index? We should subdivide inverted queries that depart from a role instance with the context step according to the RoleInContext that is a combination of the EnumeratedRoleType and the ContextType that is reached. However, by construction, the role type will be the type that we store the queries on. So we can leave it out and just use the ContextType to subdivide the inverted queries.

That is, we can choose the right subcollection of the contextInvertedQueries on the Driver role by using either the Train or the Car ContextType.

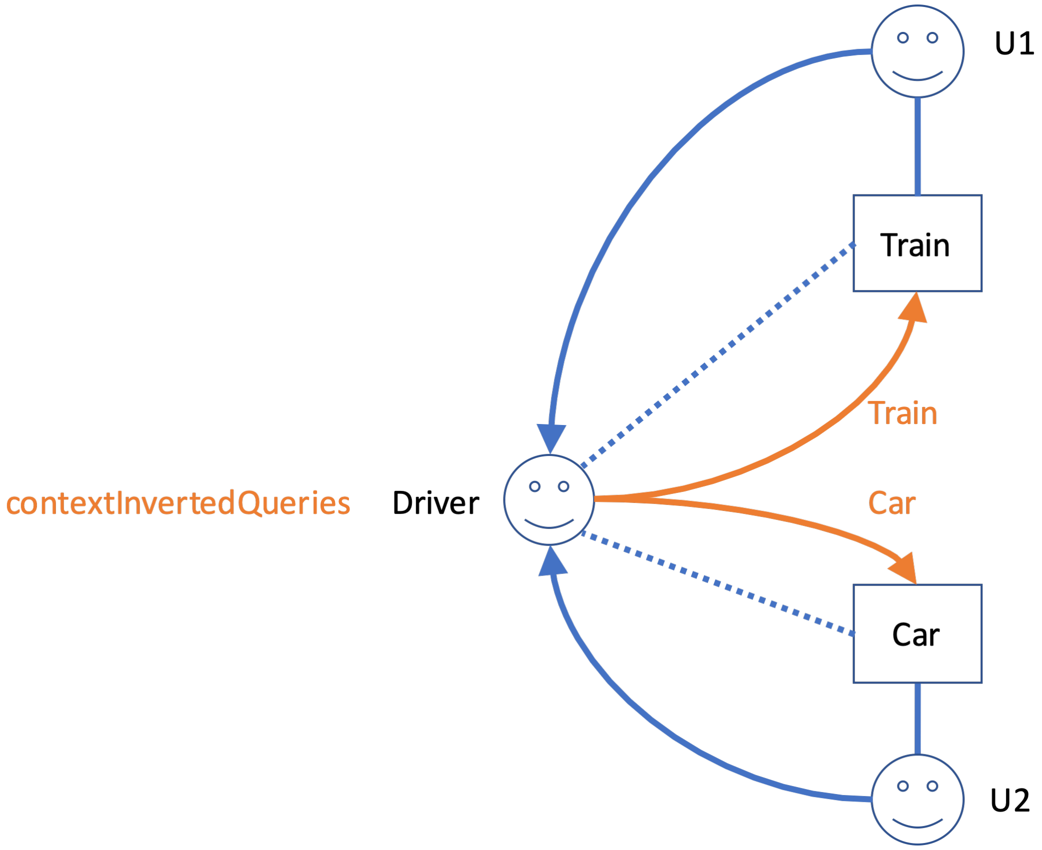


Figure 2. The inverted queries stored in contextInvertedQueries (orange lines) in the Driver Role type should be indexed by the context type of their endpoints, or, equivalently, the context type of the role type of departure.

Runtime indexing: role step

Inverted queries that, conceptually at least, start with the role step, are stored with a Context type. They are subdivided according to the EnumeratedRoleType that they lead to (as explained above).

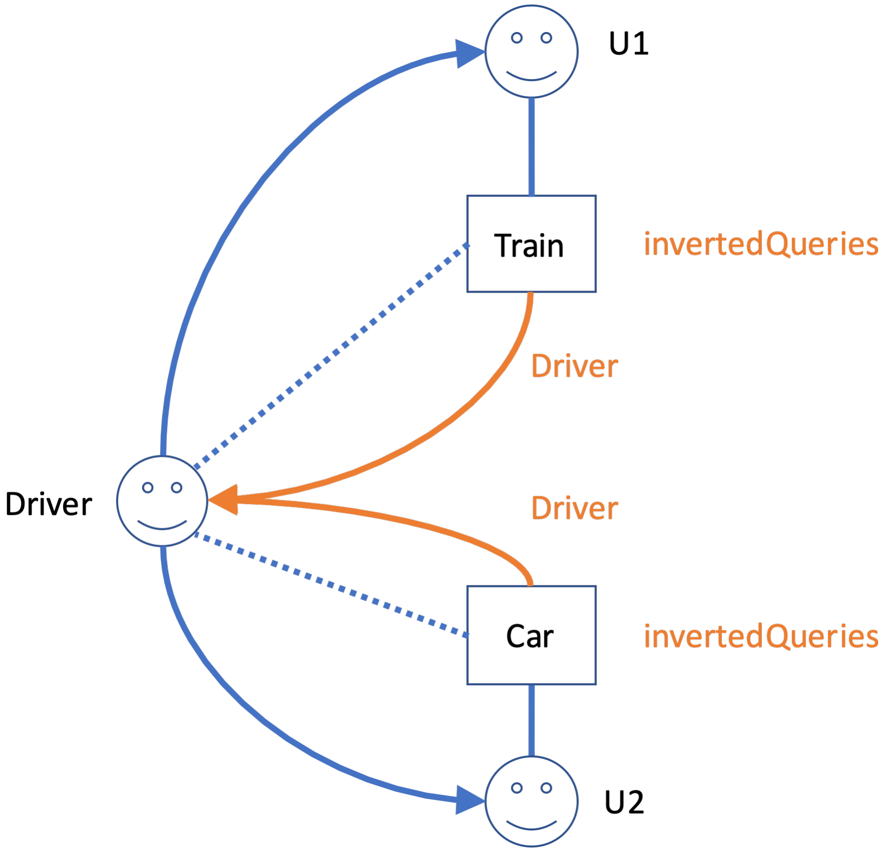


Figure 3 In this figure, the perspectives are the other way round (this is a contrived example, as Aspect roles would not have perspectives on roles in contexts to which they are added. However, it serves to illustrate the principle. We see that inverted queries, stored with the Context type, should be indexed with just the EnumeratedRoleType they lead to.

Runtime indexing: filler step

Inverted queries that depart from a role instance with the filler step, are stored in the filledByInvertedQueries collection on the EnumeratedRole type. What subcollections should we distinguish? Remember that we must make sure that we only apply inverted queries that contain the segment that is described by the Delta. This is now defined by *two* role instances, or, in other words, *two* RoleInContext instances.

As the step has a direction (from Filled to Filler), we will combine (apply) the RoleInContext instances in that order. Notice that, by construction, the first of these will always have the type of the EnumeratedRole that the collection is stored in. So we could do with just its ContextType. This is not true for the second RoleInContext instance: either of its components may vary freely. Thus, we have three keys, applied in this order, to find the right subcollection of inverted queries:

1. The ContextType of the Filled role in the Delta, or, in type time, of the Domain of the QueryFunctionDescription;
2. The ContextType of the Filler role in the Delta, or, in type time, the ContextType of the Range of the QueryFunctionDescription;
3. The type of the Filler role in the Delta, or, in type time, the EnumeratedRoleType of the Range of the QueryFunctionDescription;

For the implementation we have a number of representation choices, ranging from three nested

Objects to a single Object with a key that is the combination of the string representation of the three keys, to a Map with a key constructed of the three types.

Runtime indexing: fills step

Inverted queries that depart from a role instance with the fills step, are stored in the fillsInvertedQueries collection on the EnumeratedRole type.

The reasoning is symmetrical to that for the filler step. Again, we must use two RoleInContext indices; again, we can ignore the EnumeratedRoleType of the first of these. However, the direction is inverted. So we have:

1. The ContextType of the Filler role in the Delta, or, in type time, of the Domain of the QueryFunctionDescription;
2. The ContextType of the Filled role in the Delta, or, in type time, the ContextType of the Range of the QueryFunctionDescription;
3. The type of the Filled role in the Delta, or, in type time, the EnumeratedRoleType of the Range of the QueryFunctionDescription;

Again, like with the role step, the fills step has cardinality greater than one (it can fill many other roles). For this reason we shorten the inverted query and the runtime applies it to the filled role – not to the filler role.

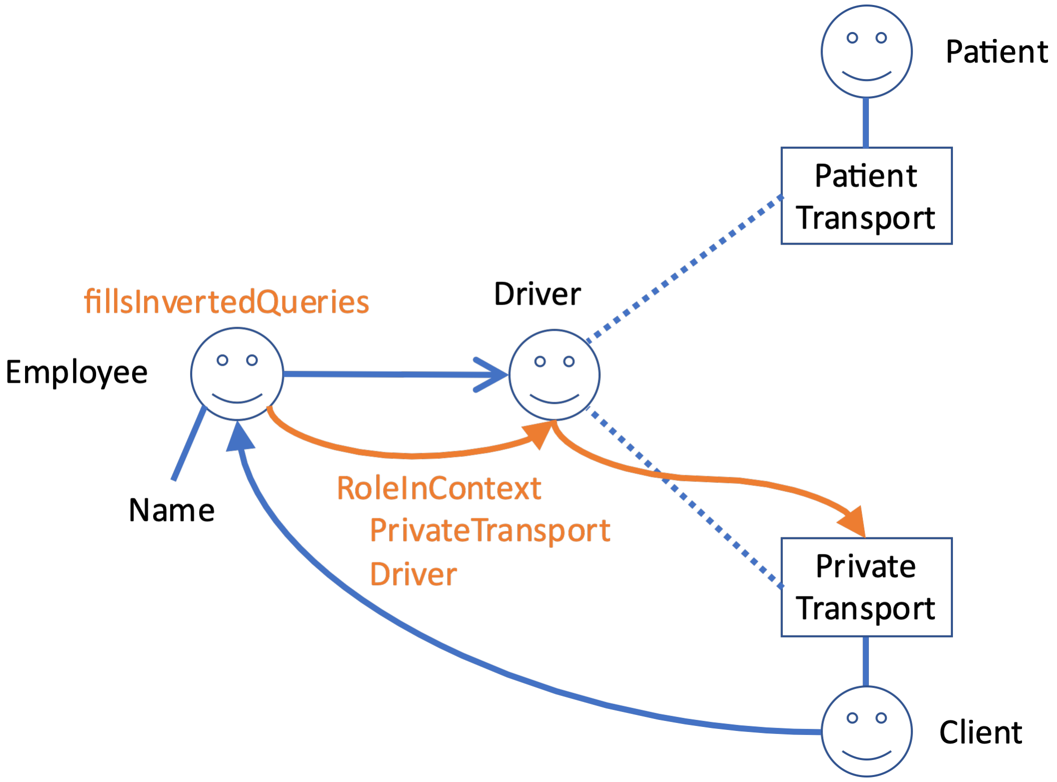


Figure 4. The Employee (of, say, a taxi company) fulfills the Driver role in both transport of severely ill people and for privatel rides. Now suppose (for the sake of the argument) that the Client of such a private ride has a perspective on the Driver, including his name, but the Patient does not. The inverted query that is stored in the fillsInvertedQueries collection of Employee actually starts on the Driver role instance (because the binder step has cardinality greater than 1). But clearly, we should index this collection of inverted queries with RoleInContext PrivateTransport Driver, order

to prevent the system from informing Patients when the Driver role is filled. Actually, we should also use the Context Type of Employee to cover the situation where Employee was used as Aspect as well.

15.4.3.4. Compile time indexing

In compile time, we invert the *description* of a query. We then cut the resulting path(s) at all steps and store those ‘cuts’ in collections of inverted queries on the various types (Contexts, Roles, Properties). On storing them, we must use the same keys we use when we retrieve them. How do we compute these keys in compile time?

> Actually, it is more like ‘kinking’ the query path at all steps. So a query with n steps results in n-1 kinked queries. Each consists of two parts: a part going backwards (against the direction of the original query) and forwards (so it is the rest of the original query from that point). However, in this text I just call them ‘cuts’ and you can think of it of all subpaths of the inverted query leading to its end.

Let’s step back for a moment and contemplate our predicament. What we need to do is to add the cuts to the DomeinFile in such a way that we can find the relevant ones when we have a Delta on our hands. The strategy to follow depends on the Delta and the first step of the inverted query.

Keys for the Value2Role step

A Property query calculates, for a given role, the values of a particular PropertyType. When we invert that query it starts with the values. However, values are not stored as indexable entities in the PDR. Instead, we skip the first step of the inverted query and start with the role instance. Hence we can derive from a RolePropertyDelta on the instance level a ‘PropertyInRole’ element on the type level: the combination of a EnumeratedRoleType and a EnumeratedPropertyType. When we store a particular cut on a Property, we use that as a key to file it under. Actually, the EnumeratedPropertyType part will not discriminate anything, because it will always be equal to the type of the Property we’ve stored the cuts in. So we can make do with just the EnumeratedRoleType part.

It enables us to distinguish a Property in its lexical context (let’s say A) from that same Property in some Aspect role context (say, B). Suppose that for B we have a CalculatedProperty that computes the average of the Property’s values. Now, when that value set changes on an instance of B, we should recompute the average. But imagine that the value of the property changes on an instance of A. If we had not filed the cut under the key B, we were doomed to recompute the cut for A’s instance, too. As a consequence, we might end up informing some peers that have nothing to do with that change.

In compile time, we have a QueryFunctionDescription whose Range is an ADT RoleInContext (it is the inversion of a PropertyGetter QueryFunctionDescription, whose *domain* is an ADT RoleInContext. From that range we take all EnumeratedRoleTypes and store the cut on the Property under each of those as key. This we do for all EnumeratedPropertyTypes we find in the domain of the QueryFunctionDescription.

Keys for the role step

When the inverted query approaches a role from the other direction (from the context, using a role step), we have a similar story. The QueryFunctionDescription has a Range that is an ADT RoleInContext. Since the domain can be a complex ADT holding many ContextTypes, the range can be a complex ADT holding many RoleInContexts.

For each of these RoleInContexts, we store the cut on the ContextType under the EnumeratedRoleType.

Keys for the context step

When we traverse the connection between role and context in that direction, we have a QueryFunctionDescription with a Range whose value is an ADT ContextType. It's Domain is an ADT RoleInContext. For each of the RoleInContexts we find in that Domain, we store the cut in its EnumeratedRole, under its ContextType.

Keys for fills and filledBy steps

A RoleBindingDelta describes a particular *segment* of the role- and context instances network. Moving into type space, we can project that instance segment on a particular *segment of RoleInContext* nodes. We'll call that pair the *TypeLevelSegment* and its first RoleInContext its *start* and the last its *end*.

Our task, then, runtime, is to find for a TypeLevelSegment the relevant cuts: the TypeLevelSegment is our *key*, on a conceptual level. So how do we find those cuts in the DomeinFile? How should we add cuts to the DomeinFile so we know the key will return the ones we look for?

A QueryFunctionDescription holds a domain and a range. These are *abstract datatypes* of RoleInContext. The simplest possible case would be a simple type (each consisting of a single RoleInContext) for both domain and range: then the queries' domain and range form a single TypeLevelSegment. Mapping the TypeLevelSegment derived from the RoleBindingDelta is easy in that simple case. The general case, however, is that a pair of ADT's represent multiple TypeLevelSegments.

What if the domain of a particular cut consists of a SUM of two RoleInContexts? A moment's reflection learns us that if the start of a particular TypeLevelSegment is either of these two RoleInContexts, this cut should be evaluated (meaning that when we modify an *instance* whose type is the start of the TypeLevelSegment, we should evaluate the cut). After all, the elements in a SUM represent alternatives. So now we can derive *two* TypeLevelSegments from the QueryFunctionDescription. Should our RoleBindingDelta map to either of them, we must evaluate the inverted query.

A PRODUCT of two RoleInContexts represents not alternatives, but composition. We should consider the product of two role instances to be a single role instance with combined properties. Again, if the start of the TypeLevelSegment we derive from the RoleBindingDelta is one of the RoleInContexts of the domain of the inverted query, we should evaluate that query (changing part of the composition is as good as changing the whole).

So we see, that as long as the start of the TypeLevelSegment derived from the RoleBindingDelta occurs somewhere in the abstract datatype of RoleInContexts that describes the domain of the inverted query, we should pick that query and re-evaluate it.

Or, formulated differently, for each RoleInContext occurring in the domain of the QueryFunctionDescription, we can construct a key from it with the RoleInContext of the range. The query should be stored under each key.

But wait. The *range* of the QueryFunctionDescription can be a SUM or PRODUCT, as well. Then what?

For the various RoleInContexts in the range, we can tell a story similar to what we said above about the domain. It’s symmetrical. Given a single RoleInContext in the domain and several in the range, we should create a key with each of the latter combined with the first.

A problem arises when *both* domain and range have multiple RoleInContexts. Should we form the full Cartesian product of the RoleInContexts in both? It turns out we should not (see the example below): we’ll generate far more keys than will ever turn up in runtime. Now this does not lead to semantically wrong results, but it is an efficiency issue not to be snuffed at (quadratic in complexity).

There is, however, a way to generate just the required keys. For this we re-run (during this process of storing inverted queries in the DomeinFile) the process of mapping the domain to the range when we compile a fills or filledBy step.

For the filledBy step this consists of looking up, for each RoleInContext in the Domain (of the cut), its binding (an ADT RoleInContext). Then, if a target context type has been specified in the query step, we replace the context in the RoleInContexts of that binding with the specified context (It requires us to store that target context in the QueryFunctionDescription).

For the fills step, the situation is somewhat simpler. The range of a fills step is, by construction, always a simple RoleInContext ADT whose EnumeratedRoleType is completely specified by the fills’ step first parameter. If no context type is given (as a second parameter), we just use the RoleInContext as we find it as the endpoint and combine it with the RoleInContexts we find in the domain. Otherwise, we replace the context type in the end RoleInContext.

To sum up: we can construct, for each cut of an inverted query starting with the fills or filledBy step, a number of keys (TypeLevelSegments) we should store the cut under. Each TypeLevelSegment consists of two RoleInContexts; each RoleInContext consists of a ContextType and an EnumeratedRoleType.

We want to store the cuts on an EnumeratedRole. By convention we store it on the EnumeratedRole whose type is in the first (domain) RoleInContext of the compound key. We obviously don’t need that first EnumeratedRoleType in the key that we store the cuts under. So we end up with a key compounded from the ContextType of the first RoleInContext and the two parts of the second RoleInContext.

15.4.3.5. Summary: sets of inverted queries

The table below gives, for several types, the subdivided sets of inverted queries that are stored on them and how to construct the keys to index them (in runtime) and to store them (in compile/type time).

Type	Member holding inverted queries	Key construction in runtime	Key construction in type time
Property	invertedQueries	EnumeratedRoleType	RoleType in Range
Context	invertedQueries	EnumeratedRoleType	RoleType in Range
EnumeratedRole	contextQueries	ContextType	ContextType in Range
	filledByInverted Queries	ContextType Filled – ContextType Filler – EnumeratedRoleType filler	ContextType Domain, ContextType Range, RoleType Range
	fillsInvertedQueries	ContextType Filler – ContextType Filled – EnumeratedRoleType filled	ContextType Domain, ContextType Range, RoleType Range

15.4.4. Comparison to Aspect Perspectives

In the discussion so far, we've restricted ourselves to the use of Aspects as the object role of a Perspective. However, we can have user roles in Aspect Contexts, too. Suppose we had added to Party a user role Giver, with a perspective on Present and its Price. We could then cast Father as a specialization of Giver (and Mother too).

> Actually, we cannot do that in this model, as Father and Mother are calculated in the Birthday contexts and Calculated Roles cannot have Aspects. But the reasoning would apply for Enumerated Roles.]. Now there will be *a single* inverted query on Price. The perspective of Giver would have to be *contextualized* to Birthday Mother, which means that its object would be changed to Present Mother.

In short, the perspective of Giver is adapted and *added to Father's perspectives*. Only afterwards will the perspective's query be inverted. Consequently, it will be indexed by the Present Mother Enumerated role when it is stored with Price. Symmetrically, the same happens to Mother's perspectives and the inversion of the new perspective object will be indexed with the Present Father Enumerated role.

Price will then bear *three* inverted queries:

1. one indexed with Present (the Aspect object role in Party) for Giver (the Aspect subject role in Party),
2. one indexed with Present Mother, for the Calculated Father in Birthday Mother and
3. one indexed with Present Father, for the Calculated Mother in Birthday Father.

When the Price property of Present Mother changes, only one of these three queries will be followed to a context. This will turn up with an instance of Birthday Mother and we will then calculate Father in it and make sure the Price delta is sent to him.

15.4.5. Keys for aspects

We've shown how to construct keys in runtime from the types of the instances. The most complicated keys are derived from RoleBindingDeltas, consisting of two ContextTypes and one RoleType. But each of these types can be composed of Aspects. What are the consequences for the keys that should be constructed?

Each instance has a collection of types. A key is a triplet; should we construct all triplets from the type collections? At least we know that all relevant keys are in this product, but there are constraints on the combinations:

1. Aspect Roles and Aspect Contexts of a Role- and ContextType pair cannot be freely combined. We must treat Role- and ContextTypes as fixed pairs, where the RoleType is the lexically embedded in precisely one ContextType.
2. The possible fillers of a role are constrained by the possible fillers of its Aspects. If we say that Driver is constrained to be filled by a Human, we cannot have a role ArmouredCarDriver that uses Driver as Aspect but should be filled by a Robot (unless we let Robot have Human as Aspect).

As a consequence, on constructing triplet keys for queries starting on the filledBy step we can work as follows:

1. We start with the Filled role type;
2. We add to that all its Aspects;
3. For each of these we construct a key that consists of:
   a. Its context type;
   b. The type of its Filler;
   c. The type of the lexical context of that Filler.

So, from a single RoleBindingDelta, we construct as many triplet keys for the filledBy step as there are types of the filled role instance.

For the fills step we follow the same procedure (working from the Filled role type), but we construct the key by leaving out the type of the Filler role instance instead of that of the Filled role instance.

15.4.5.1. Handling complex filler types

But we're not done yet. A role may have a complex filler type, expressed as an ADT RoleInContext. Let's consider the role Parent to be filled by either Father or Mother, so its filler type is SUM Father Mother. Clearly, we can derive *two* keys from that filler type.

So our final algorithm is:

1. Start with the Filled role type;

2. We add to that all its Aspects, forming the set of FilledTypes;
3. For each element FilledType of FilledTypes we construct a number of keys, by
    a. Listing all of the EnumeratedRoleTypes in FilledType’s Filler type;
    b. for each of those Filler types we construct for the filledBy step a single key from
        i. FilledType’s context type;
        ii. Filler;
        iii. The type of the lexical context of Filler.
    c. And this is how we construct keys for the fills step instead. Take
        i. FilledType’s context type;
        ii. FilledType;
        iii. The type of the lexical context of Filler.

15.5. Who authors?

Changes to Perspectives State (or, equivalently, the Perspectives Universe) come into being by executing functions in one of these modules:

- the Perspectives.Assignment.Update module,
- the Perspectives.Instances.Builders module or
- the Perspectives.SaveUserData module.

All these functions return results in (monads based on) the monad MonadPerspectivesTransaction.

To accept changes coming from another user, they have to be authorized. We achieve this by sending deltas that are signed by the other user, and that explicitly represent (contain) the roles in which users made their changes.

In this text I treat the question: how do we compute the role that is authorized to make the changes, in the modules listed above? The answer consists of two parts, because changes are either instigated either directly by the user, mediated by the GUI, through the API; or are executed automatically on state change on his (the user’s) behalf.

15.5.1. Automatic actions

Automatic action on state change is given for a particular user. For example:

```
state Unconnected = exists UnconnectedIndexedContext
  on entry
    do for <User>
      bind object to IndexedContexts
```

As the example shows, we must stipulate the user role that we execute an action on behalf of. This source code translates to an Action and the user (i.e. the type) shows up as the Subject member of that Action.

In the module Perspectives.Actions we compile automatic actions to executable code. Part of that code is a function that computes a Unit result in MonadPerspectivesTransaction, a so-called Updater:

```
type Updater s = s -> MonadPerspectivesTransaction Unit
```

This Updater executes the effect. Now, just prior to executing this RHS, we add the subject of the action to the state of MonadPerspectivesTransaction. This state is a Transaction: it contains a member authoringRole and we bind the subject to it.

As soon as the RHS has finished, we restore the previous value of authoringRole (if any) in the Transaction. In other words, we *wrap* the Updater in a state that holds the authoring user role. State changes may trigger automatic actions; this mechanism makes sure that all functions in the aforementioned three modules can just take the author role right from state.

15.5.2. API calls

The end user can only change the Perspectives Universe through the Graphical User Interface. Now this interface consists of screens that are built from parts that represent a particular user role's perspective on a context. When the user navigates to another context, just prior to displaying such a part, the GUI code asks the PDR for the role that the user plays in that context.

With each and every call that changes Perspectives State, the GUI returns that role. It is the authoring role. In the API, we put that role in the Transaction in which the change is computed, thus ensuring that all functions in the three modules mentioned above are informed correctly about the authoring role[1].

15.6. Query Inversion over Model Boundaries

We use the technique of query inversion for a number of reasons (see the texts *Perspectives across context boundaries*, *Query Inversion*, and *Perspectives on Bindings*). This text presents yet another facet of this technique.

A model is a collection of types. Models build on other models, for example because a Role dom2:B gives dom1:B as its possible binding). We can envision the Perspectives type space as an unbroken network of types. However, this network is segmented in parts that constitute the models.

Now a query, as a path through type space, may cross boundaries between models. This is not a problem: a query from model X that extends into model Y merely means that X depends on Y.

A problem arises, however, when we invert that query. The inversion starts in Y! And the modeller of X may well not be the author of Y; he has no jurisdiction in the domain Y.

So we may have numerous loose ends on our hands. If we leave them dangling, many things go wrong: for example, changes in instances whose types are in Y will not be propagated to users that are defined in X.

This text describes a solution.

15.6.1. Alternatives

There are actually three viable alternative approaches to solving this problem. All depend on storing the loose ends with the model.

1. Whenever we change something, we might check all models to see if one or more loose ends apply to that change.
2. Alternatively, when we start a session, we may run through all models and distribute the loose ends over the other models, in memory.
3. For a more durable solution we would apply loose ends to all models *on taking a model in use.* Notice that we would store the enriched models locally, in the users' own model database.

From 1 to 3, the solutions cost less time for the end user. However, 3 requires us to undo the changes if the user ditches a model. Being able to do so would also give us the opportunity to update a model.

We've implemented the last solution.

15.6.2. Design

The first question is: how do we represent the loose ends? We select a representation for swift application in end user time. This will require more model compile time, but that is ok.

We organise the loose ends by model, first, so we can apply a whole bunch of them efficiently to a given model.

Next, we observe that we have to distribute the inverted queries over five collections:

- onContextDelta_context
- onContextDelta_role
- onRoleDelta_binding
- onRoleDelta_binder
- onPropertyDelta.

We therefore create a Sum data type that reflects those collections:

```
data InvertedQueryCollection = OnContextDelta_context (Array IQuery) |
  OnContextDelta_role (Array IQuery) |
  OnRoleDelta_binding (Array IQuery) |
  OnRoleDelta_binder (Array IQuery) |
  OnPropertyDelta (Array IQuery)
```

Here, the IQuery type is a Tuple, really, of the string representation of the type that the InvertedQuery should be attached to, and the InvertedQuery itself.

```
type IQuery = Tuple String InvertedQuery
```

Finally we can put these together in a data type for the loose ends:

```
type LooseEnds = Object InvertedQueryCollection
```

and here the indices are the (string) names of the models that the loose ends should be distributed over.

These data types allow us to iterate, first, over the models that should be updated; then we do, per IQuery, a case analysis to decide what member of the Role (or Property) we should add to; and finally we find the actual type to modify from the first element of the IQuery itself.

15.7. Optimizing Inverted Query Application

In this text we describe an optimization of the functionality that applies inverted queries (see: Query Inversion). We apply inverted queries for two reasons:

1. To find resources whose state must be evaluated after a modification to the represented state;
2. To find users who must receive a Transaction with the said modification.

The optimization we describe does not change the functionality.

15.7.1. Problem statement

Referring to Figure 1, we see that onRoleDelta_binder of r2 holds two inverted queries. As the first step of the inverted query is skipped because of the cardinality of the binder operation, those two are:

```
binder r5 >> binding context        I

context                             II
```

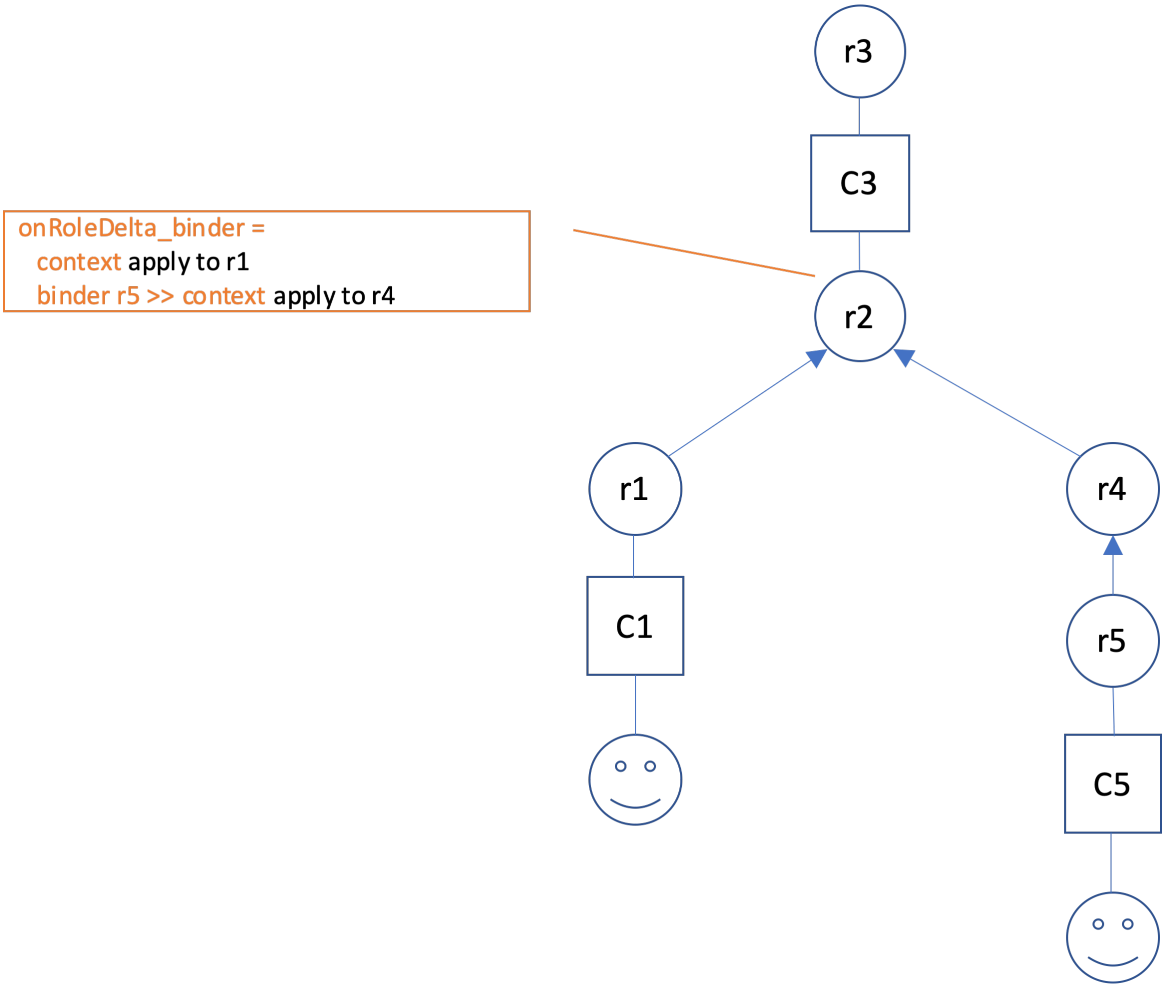
Now consider a RoleDelta with binder (id) equal to (an instance of) r1 and binding equal to (an instance of) r2. It is obvious that only inverted query II should be applied (to r1) (the new path, having the connection between r1 and r2, will never lead to c5).

The query evaluation mechanism currently is implemented in such a way that applying I to r1 will give no results, so the semantics is preserved. We can safely skip this step, however.

15.7.2. Solution

We will solve the problem by storing queries in onRoleDelta_binder not as an Array, but as a map indexed by the type of the range of the original binder step (the step that is actually removed from the inverted query before it is stored in onRoleDelta_binder).

Runtime, we then use the type of the binder in the delta to index the queries in onRoleDelta_binder so we only apply the right inverted queries.



onRoleDelta_binder =
context apply to r1
binder r5 >> context apply to r4

Figure 1. Two inverted queries stored with onRoleDelta_binder in r2.

15.7.2.1. Multiple types (Role Aspects)

In general, r4 may have multiple types. We should index the collection of inverted queries stored with (type) r2 with all types of (instance) r4.

> We jump somewhat opportunistically from instance to type with respect to the roles in this example. Now we mean to understand c4 as an instance, having multiple types.

[1] If no authoring role is provided by the API caller, we take it to be the System User.

Chapter 16. Extensions

16.1. External Function Interface

A Perspective model consists in part of calculated roles and properties. It can also include rules for bots: the combination of a condition and a series of statements that have a side effect. All these calculations consist of compositions of built-in functions such as binding or context.

In this text we describe a mechanism to make use of functions that are not built-in but added through modules. A *built-in* function keyword is recognised by the parser. In contrast, the name of an *external* function is not recognised. We need to use an explicit calling mechanism to signal the system it is a function.

Furthermore, the types of the arguments of a built-in function are compared to the requirements of its parameters. This is not true for functions that come from modules. The only check the parser/compiler system performs is on the *number* of arguments.

We will call these non-built-in functions *external.*

16.1.1. Outline of the design

16.1.1.1. Core and foreign modules

External functions can *belong to the core*, meaning they are compiled with the source code into the core program. These function are written in Purescript and thus are type checked by the Purescript compiler. It may seem confusing to call a function that belongs to the core program ‘external’. It may help to remember that such functions do not belong to the Perspectives Design Language; they are external in that sense.

An external function can also be *foreign*. A foreign function is *not* distributed with the core program but comes in a separate Javascript module. Such modules can be added to a particular PDR installation by the end user (through appropriate mechanisms).

A foreign module introduces a safety risk, as its code is executed in the same interpreting environment as the core code itself. Someone with enough determination and skills will be able to write code that can access all secrets and affect the working of the PDR. This means that we will have to provide a mechanism of trust to enable end users to take an informed decision with regard to such a module. However, that topic is not the subject of this text.

Needless to say that core modules do not introduce such a risk: the end user can put the same trust in them as in the PDR itself.

16.1.1.2. Functions that yield a result

A *Computed Role* depends on an external function, by definition. Actually, a Computed Role is just the call of such an external function. To underline that fact, the syntax for constructing a Computed Role uses the keyword callExternal:

```
context Modellen = callExternal cdb:Models() returns: Model$External
```

Here, cdb:Models is an ordinary prefixed Perspectives identifier that expands to model:Couchdb$Models.

The returns keyword is followed by a Perspectives Identifier that identifies the type of the return value of the external function. Obviously, the function must return role identifiers (in the case of a Computed Role).

> Please do not be confused by the $External part of the result type given here; it has nothing to do with the fact that Models is an external function.

Similarly, a *Computed Property* is an external function that yields a sequence of Values.

Once defined, a Computed Role or Property can be used in other calculations, just like any other Role or Property.

db:Models is a good example of an external function. It fetches the list of documents stored in the Models database of the Couchdb installation of the PDR.

It is also an example of a parameterless function. Functions with parameters need their arguments between parentheses. For a Computed Role, the argument expressions are interpreted relative to the context instance; for a Computed Property, the argument expressions are interpreted relative to the role instance.

External functions may be part of a composition. They may occur in exactly the same syntactical places as built-in functions.

16.1.1.3. Code that exerts an effect

In contrast to Computed Roles and Computed Properties, an *Effectful Statement* is the name we give to code that is executed purely for its side effect. The functional result of such code (if any) is ignored by Perspectives. Effectful statements can only be part of a model as the right hand side of bot rules.

A *core* Effectful statement can change the state of Perspectives: it may add or delete roles, contexts, properties, etc. A *foreign* Effectful statement cannot do so. It is executed outside the Perspectives engine. An example of a foreign Effectful statement might be code that sends an email.

> ‘Cannot’ is strictly not true. Such code is executed in the same javascript environment as the core code itself, as we’ve seen. But foreign Effectful statements are not *meant* to change Perspectives state.

For example:

```
callEffect mail:Send( Receiver >> EmailAddress, Message >> Text )
```

This (hypothetical) statement reads the property EmailAddress of the role Receiver, and the property Text of a role Message and actually sends that text to the receiver. In general, an Effectful statement is called with any number of argument values between parentheses, separated by commas. Arguments are ordinary Perspectives query expressions. Like all other expressions in a rule (apart from Property Assignment expressions: these are interpreted relative to the current object set), they are interpreted relative to the current context (the context of the bot).

16.1.2. Technical details

16.1.2.1. Core modules with external functions

We have several modules that are part of the PDR, that expose external functions to the modeller (*core* external functions). These modules have namespaces that are recognised by default. One of them is model:Couchdb, with the prefix cdb. It contains functions to access various aspects of storage of types and instances in Couchdb.

A qualified Perspective identifier does not comply with lexical rules for Purescript identifiers. For that reason we map the Perspective identifier we use in the model text for an external core function, to a function name as used in the core external modules. For example, model:Couchdb$Models is mapped to couchdb_Models. The mapping must be provided in the module that defines the external functions.

The QueryCompiler builds a function from a QueryFunctionDescription. This includes descriptions of external functions. So how does the QueryCompiler actually access the core modules that hold the functions themselves?

Even though these external functions are defined in Purescript modules that are compiled with the core, we need to retrieve them from a store and apply them to arguments in the QueryCompiler. Because of their variable number of parameters, we store them as HiddenFunctions in a specific store for ExternalCoreFunctions (defined in the module Perspectives.External.CoreFunctionsCache).

The compiler must generate a function call with a fixed number of arguments. Hence we record the number of arguments for external functions. Because we check the number of arguments in the DescriptionCompiler, the QueryCompiler can use an unsafe function to retrieve arguments by index from an array of computed values.

16.1.2.2. Foreign modules with external functions

If the QueryFunctionDescriptionCompiler encounters an expression with callExternal followed by a function name scoped to a *foreign* module, it constructs a description that is different from that constructed in case of a *core* module. The QueryCompiler uses that description as follows:

> The same holds for statements with callEffect.

1. It separates the module name from the function name
2. It then calls a function callForeignModuleFunction in Aff, that has three parameters:
    a. The first is bound to the module name

b. The second is bound to the function name

c. The third is bound to an array holding all arguments. Each argument is itself an Array of Strings.

The result of that function is ignored.

callForeignModuleFunction is a foreign import (a function imported by Purescript). It’s implementation is in Javascript. It essentially requires (‘require’ is Javascript lingo for loading a module) a module by the given name, obtains a function from it and applies it to the argument list.

It also throws an error in each of these situations:

1. The module is not found;
2. The function is not found;
3. The integer value of property nArgs of the function object is not equal to the length of the argument list.
4. An error is raised during execution of the function.

Retrieving a foreign module

We use exactly the same mechanism for foreign external modules as for the modules that hold screen definitions for models. Consequently, a foreign module is stored as an attachment to a model file (DomeinFile). These files are stored in Couchdb and are retrieved from Couchdb just like screen modules.

16.2. The Practical Guide to Writing an EFI module

External functions are introduced in the previous chapter External Function Interface. A quick recap: queries consist of compositions of built-in functions and external functions. External functions come in two flavours: core and foreign, where core functions are written in Purescript and are compiled and bundled with the PDR.

This text describes the intricacies of creating a new core external module and adding functions to one.

16.2.1. Create a Core External Module

A core external module is just a Purescript module. However, in order for it to be usable in the Perspectives Language, it should be described by a PL model. This model can be empty, as illustrated by the model for the external module model:Couchdb:

```
-- Copyright Joop Ringelberg and Cor Baars 2019
domain Couchdb
```

However, a model description instance is mandatory:

```
UserData
import model:System as sys
sys:Model usr:CouchdbModel
  extern $Name = "Couchdb External Functions"
  extern $Description = "A collection of purescript functions made available as
external core functions."
  extern $Url = http://127.0.0.1:5984/repository/model%3ACouchdb
```

The PDR parser/compiler system uses this model and its description to recognize names scoped to the model. Each core external module must be accounted for in the module Perspectives.External.CoreModules.

WARNING The implementation is in flux on this point, as we are moving to a new system of introducing model instances. This documentation will change in the near future.

16.2.2. Declare function names in the Core External Module

Name your external function as you wish. However, you should tie it (in your module) to its qualified name as follows:

```
externalFunctions :: Array (Tuple String HiddenFunctionDescription)

externalFunctions =
  {empty}[ Tuple "model:Couchdb$Models" \{func: unsafeCoerce models, nArgs: 0} ]
```

Points to notice:

1. There should be an identifier `externalFunctions` in your module, with the given type;
2. Each function should be declared by a `Tuple` having the *expanded* qualified name as its first member;
3. The second member of the `Tuple` contains a `HiddenFunctionDescription` and the number of parameters the function needs, minus one.

A HiddenFunctionDescription is just a type that tells the Purescript compiler to look away. The QueryCompiler will force the compiler to accept our functions as types as we have meant them.

16.2.3. The default argument

A query is a composition of functions. On executing a query, the entire composition is applied to either a Context instance (in the case of a Calculated Role) or a Role instance (in the case of a Calculated Property). Furthermore, each function in the composition receives as input value the output of the previously applied function.

All built-in query functions have just a single parameter. The modeller cannot apply a query explicitly, in the Perspectives Language; the query expression is just used as part of a Role or

Property definition:

```
thing SomeRole = AnotherRole >> binding
```

So we do not need syntax for applying a built-in functions to arguments. This is different for external functions, as we are free to define them with any number of parameters!

That is why we have the callExternal syntax:

```
callExternal cdb:Models () returns: Model$External
```

Notice the parentheses. *If your function needs more arguments than the default, list them here, comma-separated!*

To come back to declaring the number of arguments in your module: you should declare the number of *extra* arguments. By default, a single argument will be offered. Your function is expected to bind that to its *last* parameter.

The type of that argument depends on the syntactical location you deploy your function

- If it is used as a query step, the type is the result of the previous step;
- If it happens to be the first function in a Calculated Role definition, it will be a Context instance;
- The first step of a Calculated Property definition gets a Role instance;
- If it is the (first step of the) right hand side of an assignment, it is the resource the assignment is applied to.

16.2.4. The other arguments

It is not possible to type the parameters of your function, because every parameter will be bound to an array of strings by the QueryCompiler. Obviously, you are free to coerce these strings to whatever value you like, as long as you do so responsibly. Remember that Context- and Role instances are referred to by their string identifier and that all Values are shuffled around as their string representation.

16.2.5. CallEffect

Instead of calling an external function for its functional result, you are free to create and deploy a function for side effects. As an example:

```
callEffect cdb:AddModelToLocalStore( object >> binding >> Url )
```

Here we call a function that downloads a model and adds it to the users' local model database.

Like all external functions, functions called for their effect take the default argument. You will have to provide a parameter for it to be bound to; and like with the other functions, you should ignore it when declaring the number of arguments your effectful function takes.

callEffect constitutes in itself a complete assignment expression. That means that it should stand by itself on a line wherever assignment expressions are allowed:

- following a do for clause;
- following an action clause;
- In the body of a letA expression.

When callEffect is in an automatic Action, it will be executed on a state transition. If the state is defined on a role, the default parameter is a role instance. If the state is defined on a context, it will be a context instance.

When callEffect is in a user Action, it will be in a perspective on a role. The instance of that role is the value of the default parameter.

16.2.6. The type of the external functions

As shown above, we have two ways of calling functions that are defined in an External Core Module:

- callExternal
- callEffect

The former is a query expression, while the latter is a statement evaluated for its side effect. Consequently, the type of the purescript functions that are actually evaluated – i.e. the functions you write in your external puresecript module – depends this distinction.

A query expression, called with callExternal, should be a purescript function whose return value is in MonadPerspectivesQuery.

A statement, called with callEffect, should be a purescript function whose return value is in MonadPerspectivesTransaction.

Chapter 17. Various topics

17.1. The "mode" option on Fetch

Per March 2023 it became clear that the PDR suffered from an intermittent CORS problem. For no apparent reason, requests to perspectives.domains would be rejected.

At first it seemed to be an Apache problem. This is because the Chrome browser would report that the wrong certificate was sent along with a file requested from this domain (the inplace.dev certificate seemed to be attached). However, I've come to believe that Apache is not the source of the problem. Rather, the mistake stems from the fetch request made by Pouchdb: this does not set the "mode" option, making it effectively "no-cors" where it should have been "cors". According to the spec:

```
A request has an associated mode, which is "same-origin", "cors", "no-cors",
"navigate",
or "websocket". Unless stated otherwise, it is "no-cors".
(https://fetch.spec.whatwg.org/)
```

I have corrected this by using the 'fetch' option provided to the database creation of Pouchdb. It allows one to capture fetch and add options and headers etc. to the actual request. On the assumption that, for Perspectives, any request through Pouchdb will be a cors request because we don't store resources on mycontexts.com, every request is a cors request.

It seems as though FireFox doesn't suffer from the same problem. This might be because browser implementers have a certain freedom in implementing a default for the "mode" option:

```
Even though the default request mode is "no-cors", standards are highly discouraged
from using it for
new features. It is rather unsafe.
(https://fetch.spec.whatwg.org/)
```

It looks like Chrome defaults to "no-cors" and Firefox does not.

17.2. Internationalization (I18n) and Localization (L10n)

For obvious reasons, an end user should be able to select the language used in MyContexts. This is called *localization* (L10n). To prepare a program such that it can be localized, is called *internationalization* (L18n).

17.2.1. What texts should be translatable?

First of all we note that the immediately visible user interface parts of MyContexts hardly have any readable text at all, with the exception of:

- text in the tabs that are visible before the user has logged in (the login dialog, the create account dialog and the remove an account dialog);
- tooltips of the icons in the toolbar.

When an error condition occurs about which the end user should be informed, a modal text displays some text. These texts should be translatable, too.

We also have error conditions that lead to warnings addressed to the modeler or the core programmer. We will not translate these texts.

Last, but not least, models contain text that should be translatable:

- the names of types (especially roles and properties and to a lesser extent contexts);
- the names of actions;
- notifications for the end user.

17.2.2. Design decisions

I have decided to implement internationalization with the library i18next. This is possibly the best known and most used javascript I18n library. Even though there is an adaptation for the React framework available, I do not use it. As there is hardly any text in JSX, its services add little that is of use to Perspectives.

17.2.3. Dynamic loading of language resources

One typically doesn't want to load all available translations at once, but only on demand. I have implemented this by dynamically importing a language resource using an async function and the `import` function. This causes Webpack to split the output in multiple parts, one for each language resource. I prefer this above using the http backend because Webpack handles the splitting and distribution to the output directory and also the loading.

17.2.4. MyContexts and perspectives-react

Quite a few messages originate in React components delivered by `perspectives-react`. The corresponding modules import i18next on their own in order to have access to the translation function. However, it required some special attention for Webpack configuration in order to ensure that modules in the MyContexts package and modules in the perspectives-react package actually share the same i18next object.

Initialization of this object is done in MyContexts, in the constructor of the App. This is the outline of how it works:

- the `perspectives-react` Webpack configuration declares i18next to be **external**
- the `MyContexts` Webpack configuration declares an alias for i18next:

  ```
  "i18next": path.resolve(__dirname, "node_modules/i18next")
  ```

Furthermore, in `package.json` of perspectives-react I've declared i18next to be a peer-dependency.

17.2.5. Namespaces for MyContexts and perspectives-react

I wanted to be able to create language resource files in both projects. Accordingly, I put the translations in perspectives-react in a namespace **preact** and those in MyContexts in the namespace **mycontexts**.

To make sure that Webpack would split the language resources into seperate chunks for perspectives-react, I have included an async function that dynamically loads a language resource file.

However, this setup creates a runtime problem. This async function loads a resource that is rooted in a dependency, from the point of view of the MyContexts package. In other words, the chunks split off by Webpack for perspectives-react do not make it unassisted to the output directory that is filled by Webpack for MyContexts. To ensure that those language resources end up in the MyContexts output directories, I deploy the copy-webpack-plugin in the MyContexts Webpack configuration.

17.2.6. To add a language

Adding a language is simple:

- create the language resource files for both namespaces in the new language (copy an existing language and translate!);
- add the language to the i18next init function call in the file `i18next.js` in MyContexts.
- rebuild MyContexts.

17.3. The range type PFile

Role properties are said to be in a `Range`. We support some run of the mill ranges like string, boolean and number (respectively PString, PBool, PNumber). However, we also support PFile as a way to handle files that can be uploaded and downloaded from Perspectives screens.

The PFile type is declared as a Purescript String. However, the values of these strings can be parsed to the following newtype:

```
type PerspectivesFile =
  { name :: String                  -- The name associated with the file on creating
or uploading it. Use only client side.
  , mimeType :: MIME
  , database :: Maybe String        -- The database where the role instance is stored.
  , roleFileName :: Maybe String    -- The name of the role instance document.
  }
```

Properties that have range File **must be functional**.

We use PFile in a very limited way in expressions and statements:

- there is no literal syntax for a file. I.e., it is not possible to write a statement to assign a literal value to a property declared as File.
- we do support a createFile statement, however (see below)
- and we support two functions on PFile: `fileName` and `mimeType`. These allow us to filter sequences of files.

17.3.1. A pattern to restrict MIME types

In a model, one can add a pattern to a PFile definition:

```
  thing Thee
    property Beschrijving (File)
      pattern = "text/arc" "Only .arc files (Perspectives Language source files) are
allowed, so use `text//arc."
```

The pattern will be applied by the React component (see below) to the values the user enters for mime types.

17.3.2. Creating a file in an assignment statement

We have the following assignment statement to create a file:

```
create file <Name> as <MimeType> in <PropertyIdentifier> [for <RoleExpression>]
  <Contents>
```

The parameters Name, MIME and Contents expect a value that is written as a string, i.e. enclosed in double quotes. This is because each of them can contain characters that also function as identifier delimiters (such as a space). The name can be any string that conforms to the file system's conventions for file names. It includes an extension. The mime parameter should conform to the conventions described in MIME types on MDN. The optional contents are useful only for text MIME types.

RoleExpression is optional when the expression is applied in such a way that the `current object` is a role instance, which is the case

- in the state transition of a role (so, in the `on entry` or `on exit` clause of the role type that has the File property)
- in an action, when the `current object` is defined:
  - in the body of a `perspective on` clause (on the role type that has the File property);
  - in the role definition of the role type that has the File property.

Some examples:

```
-- With a RoleExpression (`origin` is defined as the instance of R due to the
perspective):

thing R
  property Model (File)

user X
  perspective on R
    action MakeFile
      create file "myModel.arc" as "text/arc" in Model for origin
        "domain model://perspectives.domains#MyModel"

-- RoleExpression can be left out in the definition of the role with the File
property:

thing R
  property Model (File)
  perspective of X
    action MakeFile
      create file "myModel.arc" as "text/arc" in Model
        "domain model://perspectives.domains#MyModel"

-- It can also be omitted in (an automatic action in) the state transition of the role
with the File property:

thing R
  property Model (File)
  on entry
    do for X
      create file "myModel.arc" as "text/arc" in Model
        "domain model://perspectives.domains#MyModel"
```

Obviously, each parameter can be supplied an argument value with an arbitrary String expression.

NOTE Creating files is particularly useful for text MIME types, such as text/html or text/arc (our unofficial type for PL source files).

17.3.3. Allowed MIME types

We apply a regular expression to candidate MIME strings:

```
^[^\./]+\/[^\./]+$
```

The regex is applied to the second argument of `createFile`. It is also applied to input entered in the MIME field in the PerspectivesFile React Component.

17.3.4. Allowed file names

We apply a regular expression to candidate filename strings:

```
^[^\./]+\.[^\./]+$
```

The regex is applied to the first argument of `createFile`. It is also applied to input entered in the fileName field in the PerspectivesFile React Component.

17.3.5. Selecting files based on their name or MIME type

We support two selector functions available in expressions on values of type File:

```
fileName :: File -> String
mimeType :: File -> String
```

This allows us to filter a sequence of files like this:

```
thing MyFiles (relational)
  property TheFile (File)

thing OnlyArcFiles = filter MyFiles with TheFile >> mimeType == "text/arc"
```

NOTE the selectors are not yet implemented.

17.3.6. How files are stored

File storage comes in two flavours: private and public. A user's PDR stores a file that he uploads privately, if there are no other roles with a perspective on the File Property. However, if there are peers that the file should be shared with, it is stored publicly - that is, if the user has a public file storage registered in Perspectives (do not confuse this with a public perspective, see below). If not, the file is stored privately but the File Property is **not** synchronized with peers.

This is because we cannot send large amounts of data through the channel that we use to synchronize with peers. Instead, we must send file attachments as **claim data**; references to the actual data.

There is one exception to the latter case and that is for a *public perspective*. By construction, role instances in a public perspective are accessible to all users and so are any files attached to a role instance.

What about a peer who has himself registered public file storage? Should we not share the file with her, too, like with a public perspective? But no. The public perspective case is special because *the modifying peer actually modifies the public resources, too*.

NOTE Currently, only private storage is supported - and public perspectives.

Couchdb supports file attachments on documents. We store the actual file associated with a Property of type File in an attachment *that has the same name as the Property*! So, in the example above, each instance of MyFiles would have an attachment named "TheFile", even though the individual files may have different names. This makes it possible to upload a file "cat.jpg" to one instance and "dog.jpg" to another, while both are stored as an attachment named "TheFile" with their respective role documents. The original file names are stored as part of the Property value.

The database name and the name of the role instance resource that holds the stored attachment, whether it be in a public store or a private one, are saved in the property value. They are two of the fields of the serialised JSON object stored as the property value.

The interface that actually accesses Pouchdb from the PDR has been changed in the course of implementing PFile, to preserve attachments on role resources. The main thrust of the change is that we now preserve the attachment info that Couchdb adds to the resource files; otherwise, on updating a role, the attachments will be discarded.

17.3.7. Handling files client side

Most of the interesting stuff pertaining to files goes on client side. We display a property of range File on a screen using the custom PerspectivesFile React Component. Outline:

- users can up- and download files through the PerspectivesFile interface. The component accesses the file attachments through the API of the PDR.
- users can create a new file instance through this interface by uploading or by providing a file name and mime type. It is not possible to create content through the interface; an empty attachment will be created.

For `image` MIME types, the component shows the image; for `text/arc` mime types, it shows the source text with syntax coloring.

17.3.8. The PerspectivesFile React Component

A quick specification. The component has four states:

- empty
- filled
- readonly
- editable

The **empty** state displays a name field and a MIME type field, both plain string types (but only values that match the regular expression mentioned above are accepted as MIME type). It also displays an upload icon button. When a name and MIME type are entered for the first time, the control creates and stores a new file. The state then becomes `filled`. In the empty state, the component also functions as a dropzone (one can drop a file on it).

- On tabbing into the control, by default, the cursor will be in the name field (other than in **filled** state, the control does not have to be unlocked for editing when **empty**)
- press `right arrow` to move to the MIME field. Changes to the name will be preserved

temporarily. Pressing `right arrow` again will focus on the upload icon button; pressing `right arrow` again moves the cursor back to the name field; changes to the MIME field are preserved temporarily.

- Press `space` when the focus is on the upload button to open the file selector dialog.
- Press `escape` to discard all changes.
- Press `enter` while in any field to actually save changes and to move the control to **filled** state.

NOTE The use of the left-arrow key is consistent with the way one can move through a table. However, as a consequence, one cannot move through *the text* that has been entered in the control with the left-arrow key.

The **filled** state shows the name and mime type (and neither is editable). In this state, the control is draggable if a url is available (the payload will be a standard HTML File object). It is also a dropzone for such objects.

NOTE the draggable interface has not yet been implemented.

- The control state can be moved from **filled** to **editable** by selecting it and then pressing `enter`. The cursor will then be in the name field. The MIME value cannot be edited.
- The download button can be selected if a url is available.
- Press `space` on the download button to activate it.

readonly is like **filled**, but without the possibility to move to **editable**. If there is an url in the property value, the end user will be able to download the file.

When **editable**, the control displays two buttons: one to download the file, one to upload it. Both can be activated by selecting and pressing `space`. The name of the file may be changed; its MIME type cannot. Move from button to button or field by pressing `left arrow`.

- Uploading a file will move the control back to **filled** state (after preserving changes).
- Pressing `enter` will preserve a change to the file name and move the control back to **filled** state.
- Pressing `escape` will discard changes and move the control back to **filled** state.

NOTE When the user has not yet changed the file name, pressing `enter` has no effect. Press `escape` to leave the control.

In all states, the download button is only enabled if the control has a value for the database for the file. This will be

- after creating a new file and
- after uploading a file
- when the property value coming through the PDR API contains the serialised structure that contains a database value.

17.3.9. Storing files through the PerspectivesProxy

We use the API function `setProperty` to store the file name and mime type. But to save the actual file, we have to call another function provided by the PerspectivesProxy: `saveFile`.

> **NOTE** Please observe that changes are only persisted after pressing `enter` or after dropping a file!

17.4. When the model ecosystem is broken

Obviously, some models are essential for MyContexts to function correctly. These are:

```
model://perspectives.domains#System
model://perspectives.domains#Couchdb
model://perspectives.domains#Serialise
model://perspectives.domains#Sensor
model://perspectives.domains#Utilities
```

However, these models themselves have been created using MyContexts, particularly their machine-readable versions (DomeinFiles). To be able to do so we need yet more models:

```
model://perspectives.domains#CouchdbManagement
model://perspectives.domains#BodiesWithAccounts
model://perspectives.domains#Parsing
```

Using CouchdbManagement, we can describe

- the CouchdbServer that represents the actual server with a CouchdbServer running on it
- the Repository perspectives.domains for the namespace `model://perspectives.domains/`
- the Manifests describing the models listed above
- the Versions that hold the source codes and can compile the DomeinFiles.

There are several ways in which the ecosystem can break down:

1. the MyContexts installation from which these descriptions have been published, has been lost, subdivided as
    a. the models and public instances are still available (on the actual server)
    b. just the models are still available
    c. no remote resources are left
2. the MyContexts installation is intact but the remote data is all lost.

17.4.1. MyContexts lost, models and public instances available

First, observe that a fresh MyContexts installation can be created without problem. We can also install CouchdbManagement and it will come complete with the CouchdbManagementApp context instance. We just need to re-create the local versions of the published resources, starting with the CouchdbServer itself.

Here we will assume it is possible to restore local resources from public ones, as described in Restoring from public resources. We can trigger the restoration process by bringing an instance of the `CouchdbManagementApp$CouchdbServers` role in a state by providing the property PublicFiller with the value that identifies the external role of the public version of the CouchdbServer instance. On entering that state, we execute a function CopyPublicResource, that will retrieve the external role, followed by the context and its roles, recursively until the entire remotely available structure has been restored locally.

17.4.2. MyContexts lost, just models are available

Restore the ecosystem as follows:

- make models available in the models_perspectives_domains database *without* version numbers. This is because the function `retrieveDomeinFile` will not find any manifests and thus will load versionless models.
- It is now possible to create a fresh MyContexts installation, but it will have no BaseRepository and thus no list of models.
- However, it should be possible to add CouchdbManagement to the installation. To this end, we've created a state of PerspectivesSystem that provides an action to User to do so.
- Once CouchdbManagement is available, create the CouchdbServer, the Repository, the Manifests and their Versions and add the source code of the models.

17.4.3. Everything is lost

Assuming there is a copy of the essential DomeinFiles,

- recreate the databases models_perspectives_domains and models_perspectives_domains_write (make sure they have no members, so are publicly accessible);
- replicate from the write database to the read database;
- replicate from the copy to the write database;
- follow the procedure given in the previous paragraph.

17.4.4. Public databases have been lost but the MyContexts installation is intact

We have as yet no procedure for this case. It should be rare; it is entirely possible to protect against this situation by creating regular backups at the system level of the Couchdb databases on https://perspectives.domains.

17.4.5. Precautions to be taken

In order to prepare for the eventualities given above, it is imperative to

- maintain a copy of the basic models in a safe place, ideally a CouchdbServer instance from which it is possible to restore them to the server at https://perspectives.domains.
- keep a copy the source code of the basic models, lest they be lost forever with the Versions;
- create backups at the system level of https://perspectives.domains, at the very least of the Couchdb installation.

17.5. Restoring from public resources

WARNING	Functionality described in this section has not yet been implemented.

It may happen that a privately held collection of resources gets lost. It is possible to restore the part of them that had been published. Essentially, the idea is that the end user visits the public version and then applies the restore operation to it. Restoring consists of executing all deltas found in the resource - they do constitute a complete log of its construction.

Of course, these deltas have to be executed in an order that makes each operation possible:

- first reconstruct the context, its external role and its properties;
- then the other roles and their properties.

Complications arise from automatic actions that can be executed when a resource gets into a particular state. We'll discuss that below.

17.5.1. Referenced roles and their contexts

Roles can be filled by other roles and these links constitute a network of contexts. Each time a role fill delta is executed, the algorithm doing so retrieves the referenced role from the public store, unless it can find a privately held version. Each time a context delta is executed, the algorithm fetches the context instance in a likewise manner, always preferring private versions.

In this way, the resources that were lost *grow back into the privately held network*.

17.5.2. Backlinks

Role fill relations are registered at both ends in the network of resources, in order to be able to traverse them in both ways: from filler to filled and vice versa. However, the role binding delta describing the fill operation *is stored just in the filled role*. The end user starts the restore operation from a context. It's external (context) role will be re-created from a UniverseRoleDelta and a ContextDelta. However, if that external role fills an embedded context role in another context, the RoleBindingDelta describing the fill operation *is not available in the external role*. In order to restore the roles that are filled by it, the algorithm has to traverse the registered backlinks and restore the filled roles.

17.5.3. Limitations

Notice that only the public version of a resource can be retrieved in this way. A user may well have had a more extensive perspective. Property values, role instances and indeed entire referenced contexts that were not published cannot be restored in this way.

What if the users' perspective was actually *smaller* than the public version? This is no problem, at least not conceptually. The resulting local resource will be larger than the original, lost version; but the user will just 'see' the roles and properties that fall within his perspective. A practical drawback might arise if, for example, the public version consumes much more memory than the original private one.

17.5.4. Automatic actions

On reconstructing a resource, it may attain a state that holds an automatic action for the user. Should we execute it? One might reason that it must have been executed before. Whether the user constructed the resource himself, or a peer did so, the users' PDR has performed the exact same operation that brings the resource in this state, before.

But is that really true? Consider the case of a state condition consisting of two parts:

- a boolean property that should be `true` and
- an expression that says that a role instance should not exist.

Now the order in which we restore the resources matters: if we first give the boolean property its `true` value, the condition will hold (as the PDR is built on the closed world assumption). The automatic action is executed. But should we first reconstruct the role instance, the condition will never hold and the action is not executed.

We don’t know what the historic sequence of events was. Or can we? Delta’s have a time stamp, so we might execute them in that order. That may save the day - but it need not. A resource does not carry its complete genesis history. If we have assigned a property a value multiple times, only the most recent delta is available.

Let’s try another line of reasoning. An automatic action is just that: an action that is carried out on behalf of a user. Instead of relying on that mechanism, on restoring resources, we might just leave it to the end user. Instead of executing actions automatically, we might leave the decision to do so to him.

Supposed, of course, that he can actually do so. This will not be the case in general. Let’s fantasize for a moment about a very deluxe version of this mechanism. It will make a record of all state transitions with automatic actions and present them to the end user, who can then decide, one by one, which of them to execute. It would involve presenting the model source text to the end user; it would require the end user to understand them. And also to understand the implications of actually re-executing an automatic action. This will not always be the case. Nor will this be a fail safe way to do everything right. During execution of an automatic action, values are constructed *based on state as it is at that moment*. Now, it may well be that different values will be constructed at the moment the list of actions to be curated is presented.

We can learn from this exercition that, yes, it is possible to restore lost private resources from public ones - but only up to a certain extent.

17.5.5. Triggering restoration

As things stand at the moment of writing, only public contexts can be visited. Whether one provides MyContexts with a public context identifier or a public role identifier, it will open the public context (a role cannot be published without its context).

NOTE Currently, one can only succesfully open a public context with either the context identifier or that of its external role (contextrole).

Consequently, the restore operation most naturally should be applied to a public context. We may do this by adding a tool in the bar at the top of the screen.

17.6. When resources cannot be stored or retrieved

The users' resources may be stored locally, or remotely. In the latter case, problems arise if no network connection is available, the user (no longer) has no sufficient authorization to access the remote storage, or the remote storage malfunctions. How should we deal with such problems?

NOTE The functionality in this section has not yet been implemented.

17.6.1. Recognise the various situations

First, we should be able to distinghuish the following cases:

- there is no netwerk (i.e. access to the internet is unavailable);
- the user has insufficient or no authorization to perform the storage operation;
- malfunctioning of the remote storage system (a rest category)

17.6.1.1. Being offline

This can be readily detected in the browser, as documented on MDN: https://developer.mozilla.org/en-US/docs/Web/API/Navigator/onLine. A caveat is in place, however. If `onLine` property is `false`, it is safe to assume that the system has no internet connection. However, when it is `true`, there is no guarantee that it actually has connection.

17.6.1.2. Authorization

The PDR implements 'just after the fact' authorization: it assumes it can access a resource and when that fails, tries to authenticate. Building on that, we consider the situation that accessing a resource fails *again* after authentication, to be an authorization issue **unless** it can be shown that the browser is offline.

Notice that we cannot distinghuish having *no authorization* from having *insufficient authorization*. So it may be the case that the user can identify herself at a specific endpoint, yet still not is provided access to a particular database.

17.6.1.3. Other cases

As this is a rest category, we'll treat all other situations in this category.

17.6.2. Options for handling these cases

In all cases, we will

- inform the end user;
- save a resource that could not be stored, in a local database (where we assume it will always be available). This database is called the `stash`.
- try to ignore the fact that we could not read a resource.

The success (or effect) of latter strategy depends on the situation. We have built-in error boundaries on getting resources (look for situations in the code where a ContextErrorBoundary or RolErrorBoundary is generated).

When the system comes online again (see the same MDN section on the `window.online` event), we'll try to save any resources put temporarily aside in the stash. Also, on system startup, we'll check the stash. Notice that resource identifiers contain information on where to put them or retrieve them from.

Notice that the PDR has cache that will keep these resources in memory. However, as this is a Least-Recently-Used case, there is no guarantee that the resource will remain in the cache.

We do not yet try to reload resources that cannot be read from a remote location from the stash.

17.6.2.1. Handling being offline

Having been notified that connection has been lost (or has not been established) the user should take appropriate action. This is a relatively harmless situation.

17.6.2.2. Handling Authorization problems

It may not be easy for the end user to handle this situation. However, there may be cases where (s)he recognises that (s)he could add credentials to the installation, for the given remote storage location. After adding them, restarting the system should be enough to have the resources put into the stash stored away in their intended remote location.

All in all, we will inform the end user about the remote endpoint for which no credentials are found.

17.6.2.3. Other cases

Again, it may be difficult for the end user to act. It all depends on the remote situation.

PART IV. ESSENTIAL MODELS

Chapter 18. CouchdbManagement

In previous chapters, we've given a lot of detail concerning

- Models
- Type and Resource Identifiers.

In the current chapter, we build on those concepts to describe how a particular Perspectives model in the namespace `perspectives.domains` is used in MyContexts to manage models on the internet.

18.1. Overview

The main concepts in this model are

- CouchdbServer
- Repository
- ModelManifest
- VersionedModelManifest.

A CouchdbServer instance describes a particular, existing server on the open internet. Its address is registered in the CouchdbServer instance. A Couchdb installation should be present on that server and the Perspectives user creating the CouchdbServer instance should have Admin credentials for this Couchdb installation. The credentials must be entered in MyContexts; they are added to the CouchdbServer$Admin role. For this CouchdbServer instance, a database is created with the standard name `cw_servers_and_repositories`.

CouchdbServer has a Visitor role that publishes to this database. Hence, a public version of the CouchdbServer instance is available at this publicly accessible URL. The Visitor has a perspective on PublicRepositories, a filtered version of the role Repositories that is filled with Repository instances. These instances have a public Visitor role, too; it publishes to the same `cw_servers_and_repositories` database. Consequently, a Visitor can navigate from a CouchdbServer to its (public) Repositories (there may be private Repositories, too) and his PDR retrieves all these instances from the same database.

Within a Repository, the Visitor has a perspective on a role Manifests filled with ModelManifest. However, Manifests are grouped by namespace. For example, both the System model and the CouchdbManagement model are in the `perspectives.domains` namespace. The PDR translates a model URI like `model://perspectives.domains#Systems` automatically to a URL that retrieves the compiled model (the DomeinFile). In doing so, it assumes that the DomeinFile is in a database whose name is derived from the namespace. In our example this will be the database `models_perspectives_domains`. It also derives a URL from the model URI that points to the models Manifest. It assumes this manifest is in the database `cw_perspectives_domains`.

Now we have the following situation:

- the Repository instance describing the namespace perspectives.domains is expected in the `cw_servers_and_repositories` database

- the CouchdbManagement and System ModelManifest files are expected in the `cw_perspectives_domains` database.

ModelManifest has a Visitor role, too. The instance that is associated with the namespace `perspectives.domains` publishes to the `cw_perspectives_domains` database. We now have everything in its place, because, for example, the ModelManifest describing the CouchdbManagement model is, indeed, stored in the database where the PDR expects it to be.

18.2. An aside: how does this work?

Consider for a moment an Admin role who adds a ModelManifest to the Repository for the `perspectives.domains` namespace. In his own PDR installation, the modified Repository instance, the new Manifests role instance and contexts and roles describing the new ModelManifest will be stored in his own local repositories. But because of the public roles, versions of these resources should be published as well. Let's look in detail to some of these resources:

- the new Manifests role is within scope of Repository$Visitor and thus will be published to `cw_servers_and_repositories`
- however, its filler, the external role of ModelManifest, is within scope of ModelManifest$Visitor and should be published to `cw_perspectives_domains`!

Consequently, the url of the Manifests role will be like this: `https://perspectives.domains/cw_servers_and_repositories/<guid1>`. But the url of the filler, the external role of ModelManifest, will be like this: `https://perspectives.domains/cw_perspectives_domains/<guid1>`. It appears that, on publishing, the PDR should look up the public url for each context type. If a context has a public role, the PDR should use it; otherwise the PDR just uses the public url of the publishing role. So, because the filler is a role of ModelManifest, the PDR uses its public URL to construct a pub: identifier.

Should it actually publish the external role itself (to the `cw_perspectives_domains` database)? It can only do so if the end user (or rather the Admin roles he plays) has the credentials that allow him to do so. To make that explicit: a Repository$Admin creating a new ModelManifest should have write access to

- the `cw_servers_and_repositories` database for the CouchdbServer;
- the `cw_perspectives_domains` database, or, in general, the database that holds the ModelManifest instances for the namespace described by the Repository.

We can actually arrange that in `model://perspectives.domains#CouchdbManagement`, and we should and have done so. So, under these circumstances: yes, the PDR should publish not only the new Manifests role instance, but also all role- and the context instances describing the new ModelManifest.

We consider this to be a model pattern. A user role with the perspectives to create a new context, where both the embedded context role and the new context have a public perspective, should have write access rights to both databases.

Now what if a user role has the perspective to create a new embedded context role *that is filled with an existing public role*? That is, its filler has the `pub:` scheme? In this case, the user role does not necessarily have the perspective to create such public roles (i.e. someone else made that role instance). However, *his own version of that embedded context role will also be filled with an identifier in the* `pub:` *scheme*. And on publishing, we just leave such references intact. All is well!

18.3. URL Rewriting Revisited

Consider the namespace `perspectives.domains`. It's Repository is associated with the database cw_perspectives_domains. In this database, we'll find the ModelManifests for all models in the `perspectives.domains` namespace, like model://`perspectives.domains#System`. The URL of that ModelManifest can be derived directly from the model URI: https://perspectives.domains/cw_perspectives_domains/perspectives_domains-System.json.

Now consider the namespace `joopringelberg.nl`. Let's suppose it's Repository, too, is hosted in the CouchdbServer at https://perspectives.domains. Consequently, there will be a database cw_joopringelberg_nl holding the ModelManifests for models in this namespace, like for example `model://joopringelberg.nl#MyModel`. The PDR will derive the following URL for the ModelManifest for that model: `https://joopringelberg.nl/cw_joopringelberg_nl/MyModel`. But now we have a problem, because the manifest isn't at joopringelberg.nl, it is at perspectives.domains!

We must solve this problem by rewriting the URL of the manifest on the server that hosts joopringelberg.nl. It should swap the authority of all ModelManifest urls with perspectives.domains. See this paragraph for how to configure Apache.

18.4. Naming instances

Because the PDR derives the ModelManifest instance URL from the model URI, care should be taken to explicitly name the ModelManifest instances. We can use the following syntax:

```
create context ModelManifest named <modelname-expression> bound to Manifests
```

where, obviously, <modelname-expression> should result in the name of the model, like MyModel or System for the examples in this chapter.

Similarly, we should carefully name instances of VersionedModelManifest. Instances of the other important types (CouchdbServer, Repository) need not be named.

NOTE the PDR does not yet construct VersionedModelManifest urls.

18.5. Configuring an Apache VirtualHost running Couchdb for a domain of perspectives models

In this paragraph we explore the issue of configuring Apache for a particular domain of Perspectives models. We assume the server runs Couchdb, which is available as an http server on localhost. Apache will forward requests for resources to this local couchdb server.

NOTE This is different from the situation where a domain is hosted on a server *that does not have a Couchdb installation*. The VirtualHost configuration on such a server must forward requests to another, external server. See the next paragraph for that.

As an example we'll use the namespace of the System model, `perspectives.domains` (like `model://perspectives.domains#System`). For this domain, we will configure a VirtualHost in Apache on the relevant server. The VirtualHost starts out like this:

```
<VirtualHost *:443>
    ServerName https://perspectives.domains
    ServerAlias *.perspectives.domains
</VirtualHost>
```

All requests to `https://perspectives.domains` will now be captured by this virtualhost (assuming, of course, that the DNS refers this domain name to our server).

NOTE this topic is related to Booting the Perspectives Universe

18.5.1. Forwarding requests to Couchdb

In principle, we just want to pass all requests in the `perspectives.domains` namespace to Couchdb. We can use ProxyPass for that. However, we have reason to except some requests (we'll come back to that in the next paragraph), so we use ProxyPassMatch.

```
  ProxyPassMatch "^(?!.*index\.html)(.*)$" "http://localhost:5984/$1"
  Header edit Set-Cookie (.*) "$1; Secure; Domain=perspectives.domains;
SameSite=None;"
```

Notice that we edit the cookie. Couchdb will return an `AuthSession` cookie when we request it at the `_session` endpoint, presenting our credentials. In order to make the browser keep it and return it with subsequent requests, we have to add some attributes:

- as we forward internally over http, we have to add the **Secure** attribute manually in Apache;
- Couchdb does not include the Domain attribute. It seems wise (but has not proven to be necessary) to include the domain;
- we absolutely **must** include the **SameSite** attribute; otherwise the browser will ignore the cookie.

18.5.2. Preflight requests

Given the fact that the PDR runs in the `mycontexts.com` domain and we request resources from the `perspectives.domains` domain, we have to deal with CORS. A problem to solve is that the browser sends a preflight request for .json resources, before it sends a PUT or POST request to store some information in the database on our server. However, Couchdb doesn't accept the OPTIONS method (which goes into the preflight request). So we have to have Apache handle that. We do so using url rewriting:

```
RewriteEngine On
RewriteCond %{REQUEST_METHOD} ^(OPTIONS)$
RewriteRule ^/.*$  index.html  [R=204]
```

Every request using the OPTIONS method is rewritten as a request to a standard resource `index.html` (its content is of no importance). Referring back to the previous paragraph, this explains the condition for ProxyPassMatch:

- we have Apache apply the url rewriting first;
- and only then do we forward requests to the local Couchdb - in principle any request, *except a request for index.html*!

We force the response code 204 (No content), described by *There is no content to send for this request, but the headers may be useful.*. So what headers do we return?

```
Header always set Access-Control-Allow-Credentials "true"
Header always set Access-Control-Allow-Origin https://mycontexts.com
Header always set Access-Control-Allow-Headers "content-type"
```

Notice that these should be read as server instructions to the browser as to what headers it (the server) will allow in requests from the browser. We should also inform the browser what methods are allowed, but here we need a refinement for the PDR. This is the subject of the next paragraph.

First, however, we have to make absolutely sure that we control these headers, by unsetting them first:

```
Header unset Access-Control-Allow-Credentials
Header unset Access-Control-Allow-Methods
Header unset Access-Control-Allow-Origin
Header unset Access-Control-Allow-Headers
```

18.5.3. Protecting read databases

Couchdb authorization is not ideally suited to our needs, as it does not distinghuish between reading and writing to a database. However, we want all users to be able to read models but certainly will not allow all users to modify them! We have solved this issue by creating a separate database to write to, for every database we need. By convention, the name of such a database will end in **_write**. Now, to actually prevent the browser from writing to a read database we have to be careful about the http methods we allow:

```
<If "(%{REQUEST_URI} =~ m#(cw_|models_).*/.+$# ) && (%{REQUEST_URI} !~
/.*(_write|_security|_design).*/)">
  Header always Set Access-Control-Allow-Methods "GET, OPTIONS"
</If>
<Else>
  Header always Set Access-Control-Allow-Methods "GET, PUT, POST, DELETE, OPTIONS"
</Else>
```

So while we usually allow all methods, under some conditions we only allow GET and OPTIONS:

- for good measures we restrict ourselves to perspectives databases. Their names, by convention, have to start with **cw_** or **models_**.
- Furthermore, we will allow PUT etc. on the database objects themselves - otherwise we cannot create or modify them! This is accomplished by requiring that the url not only contains a database name, but also a resource identifier. For example **cw_perspectives_domains**/<**some id**>.
- Then, from the set of urls that satisfy the above conditions, we *exclude* those to databases whose name ends in **_write**. These are, of course, exactly the databases that we allow writing to!
- Finally, we also exclude Couchdb endpoints like `_security` and `_design`. Each database has such an endpoint and it is used to set database membership.

18.5.4. Certificates

To complete the treatment, we should of course include references to the necessary certificate and key file for Apache to handle SSL.

```
  SSLEngine on
  SSLCertificateFile /path/to/certs/perspectives.domains-2023-certificate.crt
  SSLCertificateKeyFile /path/to/keys/perspectives.domains-2023-certificate.key
```

18.5.5. Putting it all together

In the right order, we now have:

```
<VirtualHost *:443>
  ServerName https://perspectives.domains
  ServerAlias *.perspectives.domains

  RewriteEngine On
  RewriteCond %{REQUEST_METHOD} ^(OPTIONS)$
  RewriteRule ^/.*$  index.html  [R=204]

  ProxyPassMatch "^(?!.*index\.html)(.*)$" "http://localhost:5984/$1"
  Header edit Set-Cookie (.*) "$1; Secure; Domain=perspectives.domains;
SameSite=None;"

  Header unset Access-Control-Allow-Credentials
  Header unset Access-Control-Allow-Methods
  Header unset Access-Control-Allow-Origin
  Header unset Access-Control-Allow-Headers

  Header always set Access-Control-Allow-Credentials "true"
  Header always set Access-Control-Allow-Origin https://mycontexts.com
  Header always set Access-Control-Allow-Headers "content-type"
  <If "(%{REQUEST_URI} =~ m#(cw_|models_).*/.+$# ) && (%{REQUEST_URI} !~
/.*(_write|_security|_design).*/)">
    Header always Set Access-Control-Allow-Methods "GET, OPTIONS"
  </If>
  <Else>
    Header always Set Access-Control-Allow-Methods "GET, PUT, POST, DELETE, OPTIONS"
  </Else>

  SSLEngine on
  SSLCertificateFile /path/to/certs/perspectives.domains-2023-certificate.crt
  SSLCertificateKeyFile /path/to/keys/perspectives.domains-2023-certificate.key
</VirtualHost>
```

Finally, notice that it is useful to add some more instructions like ServerAdmin, DocumentRoot, LogLevel, ErrorLog and CustomLog.

18.6. Configuring an Apache VirtualHost WITHOUT Couchdb for a domain of perspectives models

In contrast to the previous paragraph, we here describe a situation for an author with his own domain - we will take `joopringelberg.nl` as example - who wishes to publish models but does not want to install his own Couchdb server. Presuming he finds a provider willing to do that for him, he should then configure his own server to forward requests for Perspectives resources to this provider.

NOTE a requirement is that the author can configure his own virtual host on the server that the DNS refers the author’s domain to.

Assuming a virtual host configuration like this:

```
<VirtualHost *:443>
  ServerName joopringelberg.nl
  ServerAlias www.joopringelberg.nl
<VirtualHost>
```

that furthermore contains the necessary directives for SSL certificate and key files, we must add the following lines:

```
  SSLProxyEngine on
  ProxyPassMatch "^/(_session|.*_design|cw_|models_)(.*$)"
https://perspectives.domains/$1$2
  ProxyPassReverseCookieDomain perspectives.domains joopringelberg.nl
```

Note that we use mod_proxy; this should be enabled for Apache. First of all, we refer any requests for Perspectives resources to the server of the party that hosts Couchdb for us. Notice that we restrict this forwarding to requests to the well-known Perspectives databases and to the necessary Couchdb endpoints, respectively

- databases whose name starts with **cw_** or with **models_**;
- the server root endpoint **_session**
- the database endpoint **_design**

Next, we must edit the domain name attribute in the AuthSession cookies that are returned by Couchdb and have their domain name attribute set by the Apache server on perspectives.domains. We exchange it for our own domain name.

And that’s it! This is all it takes to forward the right requests to the server hosting Couchdb.