Programming Robotic Agents: 
A Teleo-Reactive Multi-Tasking Approach

K. L. Clark
School of ITEE, The University of Queensland
P. J. Robinson
School of ITEE, The University of Queensland

November 25, 2014
Preface
This book shows the reader how to construct multi-threaded robotic agent software systems using two high level logic based programming languages, TeleoR and QuLog. It focuses on how an agent, controlling one or more independently useable robotic resources, for example several robotic arms, can concurrently achieve the goals of multiple tasks.

Its TeleoR programmed tasks use QuLog to do fast inference querying of the agent’s BeliefStore of dynamic facts. They comprise: rapidly changing sense data, more slowly changing told information, and co-ordination facts. Co-ordination facts are updated by the tasks themselves in order to avoid interfering with one another and to fairly share their use of the robotic resources. The manipulation is not visible in the code for each task. It is provided by the TeleoR translator guided by programmer declarations of the names of resources and atomicity of certain sub-tasks.

The agent is able to flexibly work with other agents and humans, automatically taking advantage of any help whilst rapidly recovering from any hindrance or setbacks due to exogenous events.

The book is self contained and is suitable as a supplementary text for a robotics or agents course, where material from the book has been used. It serves as an introduction to mid level inference based robotic agent programming for anyone interested in this exciting new area.

There is free open source software linked with the book downloadable from [105]. It can be installed on any Linux/Unix or Windows based device, or under OS X. On an Android phone it has been used to control a LEGO® MINDSTORMS™ robot. To use with any robotic hardware just three interface procedures need to be written. Examples are provided for MINDSTORMS™ robots and ROS (Robot Operating System) [100] in the downloadable software.

The TeleoR and QuLog languages

A TeleoR program comprises ordered guard → action rules clustered into parameterised procedures which can be invoked, sometimes recursively, as a sub-task action of a rule. Other rules have one or more primitive actions for the robotic resources, to be executed in parallel, as the rule action. The language is a typed and higher order (procedures can be passed as arguments to procedures) extension of Nilsson’s Teleo-Reactive robotic agent language [91], which we shall call TR.

TeleoR rule guards are queries to the agent’s BeliefStore that access the dynamic beliefs using rules written in the declarative subset of QuLog.
This is a typed higher order fully integrated logic and functional language
(relations and functions can be passed around as arguments and returned
as values and function expressions can be given as arguments of relation
queries). QuLog also has an imperative subset of action rules that use action
primitives and which sit on top of the declarative subset.

Typically, initially called TeleoR procedures have rule guards that use
inference and mostly call TeleoR procedures as sub-task actions. These
sub-task procedures then typically use shallower inference, also call other
procedures, but may directly execute resource actions. Eventually, procedures
are called that directly query the dynamic beliefs, particularly the
percept facts, and their rules have resource actions.

The use of QuLog mediated querying of the dynamic facts enables a pro-
grammer to effortlessly implement the transition between reason based and
sensor reactive behaviour. The interface a sequence of intermediary pro-
cedure calls actions that progressively do less reasoning (albeit fast reasoning)
and more direct querying of the percept facts.

The language is introduced in stages with each new feature motivated
by a robotic programming application that is more easily programmed, or
programmed in a more semantically clean way, using the feature. A taste of
TeleoR programming is given in the short Prologue.

The progressive introduction to the language is interleaved with a parallel
progressive development of its formal operational semantics and a reference
implementation entirely in QuLog. The implementation makes use of QuLog’s
action procedures and action primitives which comprise its imperative sub-
set. It also uses the declarative subset of relation definitions.

What is assumed and what might you learn?

The book assumes some knowledge of symbolic logic for the semantics
chapters. We give pointers to introductions the symbolic logic some of which
are free web resources. If you are averse to logic or mathematical notation
in general, the very formal sections of these chapters can be skipped.

It also assumes some programming experience. However TeleoR pro-
gramming is very different from other forms of programming, so prepare to
shed preconceptions.

Programming of robotic agents  The most important thing you will
learn is how to program the tasks of robotic agents using Nilsson’s elegant
and high level TR programs, as extended with types and the extra forms
of actions, rules and procedures of TeleoR. All are extensions that we have
found useful or necessary to significantly increase the range of application of the Teleo-Reactive concept, particularly for concurrent multi-tasking.

**Using logical rules for information representation** You will learn about rule based representation of information using QuLog and similar inference based languages such as Prolog [107] and Datalog [62]. You will also get a small taste of higher order programming. We illustrate this using of a higher order TeleoR procedure - a robotic action plan - in which a key sub-task behaviour is given as a TeleoR procedure argument when the higher order procedure is called.

**State transition semantics for an event driven language** When you develop a new programming language, or significantly extend an existing one as we have done, there are two ways you can precisely specify the behaviour of its programs. You can either give an unambiguous formal definition called an operational semantics, or give and explain the key features of a well engineered reference implementation. We do both.

By reading the three chapters that give the operational semantics of a TeleoR procedure call, you will learn about using a mathematical abstraction to define the state of a single, then a multi-thread computation of a program, and about the need to give a precise definition of the next computation state for the different actions or events that can trigger a state transition. For most programming languages the triggers are program operations. For TeleoR programs it is always an event, a BeliefStore update or a time reached event.

**How to implement multi-threaded robotic agents** By reading the three chapters describing the reference implementation of a multi-threaded TeleoR + QuLog programmed robotic agent with its task execution conforming with the operation semantics, you will learn how to implement a generic multi-threaded Teleo-Reactive Robotic agent.

We use QuLog for our reference implementation. Instead of QuLog, Java or Scala [110] could be used with the Java based Iris [62] Datalog [23] Reasoner for its BeliefStore. Iris is not typed, but has many type checking primitives.
Acknowledgements

The first acknowledgement has to go to Nils Nilsson. Without his novel and elegant concept of Teleo-Reactive procedures, and particularly his triple-tower architecture paper [91] which added inference from a BeliefStore, this book would not have been written. We also much appreciate his encouragement to complete the book and his helpful comments on drafts.

Secondly, we acknowledge the support and contributions of colleagues Ian Hayes and Brijesh Dongol of the University of Queensland, with whom we had very useful discussions regarding the operational semantics of our extra forms of rules. Ian actually suggested the until rule form, as a clean way of programming a safety critical systems test case. However his until rule had a proposed operational semantics that was difficult to implement. We considerably weakened its semantics, and added the while rule form with a complementary, more technically, a dual semantics. Both rule forms are independently useful. The combination while_until rule almost corresponds to Ian’s proposed until rule.

With Peter Robinson they developed an alternative semantics for variable free TR procedures using time intervals [46]. That work and the collaboration with Keith Clark was partly funded by the Australian Research Council. It facilitated much of the work of the book. So thank you ARC.

For their comments on draft chapters of the book we would also like to thank: Pedro Sanchez Palma and colleagues, John Staples, Kostas Stathis, Adam Stacy, Maarten van Emden, Robert Kowalski, Krysia Broda, Andrew Davison of PSU Thailand, Naranker Dulay, Marek Sergot and Frank McCabe.

Finally, an acknowledgement from Keith Clark of the unknowing assistance of a friend’s dog, Amber, a Bernese Mountain dog. Walking alone with a slow and aged Amber on Wimbledon Common facilitated relaxed and fruitful thinking about the scope and content of the book.
# Contents

**Preface**

1 Prologue: A Taste of the Book

1.1 What is a Robotic Agent? 

1.2 What is a Teleo-Reactive Agent Program? 

1.3 Key Features of our Teleo-Reactive Robotic Agents 

1.4 Example Applications

2 Origins of TeleoR and QuLog and Book Overview

2.1 Multi-Threaded Communicating Agents 

2.2 Teleo-Reactive Programs for Sense/Query/Act Robotic Agents 

2.3 Application Driven Extensions of TR Programs 

2.4 Task Atomic Procedures for Multi-Tasking 

2.5 Typed and Moded QuLog for the Agent’s BeliefStore 

2.6 Higher Order TeleoR Procedures 

2.7 TeleoR in stages

3 Introduction to Teleo-Reactive Programming

3.1 Structure of an Agent Program 

3.2 A Bottle Collecting Agent Controlling a Mobile Robot 

3.2.1 Sensor Percepts 

3.2.2 Robot actions 

3.2.3 The bottle collector’s TeleoR task program 

3.3 How the Program Behaves 

3.4 Recovering from Setbacks 

3.4.1 Repeating the bottle collection 

3.5 Universal Procedures, TeleoR Program Development and Correctness
10.6.2 while/until with two minimum duration constraints 200
10.7 Communicating robotic agent bottle collection 201
10.7.1 Agent specific handle_message 202
10.7.2 Agent specific handle_percepts 203
10.7.3 The communicating_collect_bottles procedures 204
10.8 Remembering Key Past Beliefs 210
10.8.1 Example use for remembering about possible collisions 213

11 Operational Semantics of Single Task Full TeleoR 215
11.1 Alternative definitions of fire, continue and refire for standard rules 216
11.2 Semantics of while Rules 218
11.2.1 K while C ⊢ A 218
11.2.2 K while C min Dc ⊢ A 220
11.3 Semantics of until Rules 220
11.3.1 K until U ⊢ A 221
11.3.2 K until U min Du ⊢ A 222
11.4 Semantics of while/until Rules 223
11.4.1 K while C min Dc until U min Du ⊢ A 223
11.5 Action updates for continued rules 225
11.6 New Evaluation State Specification 225
11.6.1 Updated formal conditions characterising a valid evaluation state 227
11.7 New State Transition Specification 228
11.7.1 State transition rule 229

12 Implementation of Single Task Agent Programmed Using Full TeleoR 233
12.1 Attached QuLog Actions 234
12.2 Handling Time 234
12.3 Timed Sequence Actions 234
12.4 while Rules 234
12.4.1 while rules with a time condition 234
12.5 until Rules 234
12.5.1 until rules with a time condition 234
12.6 while/until Rules 234
12.6.1 while/until rules with time conditions 234
12.7 Triggering a Stack Update on a Time Event 234
12.8 Changes to the Run-time System 234
12.9 Letting a TeleoR Task Terminate 234
## 13 Multi-Tasking with a Single Resource

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>13.1 Handling Task Requests</td>
<td>237</td>
</tr>
<tr>
<td>13.2 A Two Block Tower Building Scenario</td>
<td>240</td>
</tr>
<tr>
<td>13.3 Procedure Declarations for Multi-Tasking</td>
<td>241</td>
</tr>
<tr>
<td>13.3.1 Task atomic procedures</td>
<td>241</td>
</tr>
<tr>
<td>13.3.2 Allowed Start Procedures</td>
<td>242</td>
</tr>
<tr>
<td>13.3.3 Initial task atomic calls</td>
<td>243</td>
</tr>
<tr>
<td>13.4 Example Use of Coordination Facts</td>
<td>244</td>
</tr>
<tr>
<td>13.5 Changing the Granularity of the Task Interleaving</td>
<td>246</td>
</tr>
<tr>
<td>13.5.1 No interleaving for a task</td>
<td>248</td>
</tr>
</tbody>
</table>

## 14 Multi-Tasking with Multiple Resources

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>14.1 Task Queueing Behaviour with Multiple Resources</td>
<td>251</td>
</tr>
<tr>
<td>14.2 Declarations of Resources and Resource Use</td>
<td>252</td>
</tr>
<tr>
<td>14.3 Resources Used by a Primitive Action</td>
<td>253</td>
</tr>
<tr>
<td>14.4 Dynamic Resource Use and Resource Parameters of Procedures</td>
<td>254</td>
</tr>
<tr>
<td>14.4.1 Resources that must be acquired by resource parameterised procedures</td>
<td>256</td>
</tr>
<tr>
<td>14.4.2 Check that only the acquired resources will be used</td>
<td>257</td>
</tr>
<tr>
<td>14.4.3 Local checks regarding resource parameter use</td>
<td>259</td>
</tr>
<tr>
<td>14.4.4 Guard selection of which acquired resource to use</td>
<td>261</td>
</tr>
<tr>
<td>14.5 Parallel and Interleaved Block Tower Building with Two Arms and Three Tables</td>
<td>262</td>
</tr>
<tr>
<td>14.5.1 New percept facts and new defined percepts</td>
<td>264</td>
</tr>
<tr>
<td>14.5.2 Parallel use of the arms and shared table contention</td>
<td>265</td>
</tr>
<tr>
<td>14.5.3 Allocating arm resources to building tasks</td>
<td>268</td>
</tr>
<tr>
<td>14.6 A Multi-Resource makeTower Program</td>
<td>268</td>
</tr>
<tr>
<td>14.6.1 New generic block move procedure - move_to_loc</td>
<td>268</td>
</tr>
<tr>
<td>14.6.2 New multi-resource makeTower procedure</td>
<td>270</td>
</tr>
<tr>
<td>14.6.3 Behaviour of the multi-resource makeTower</td>
<td>273</td>
</tr>
<tr>
<td>14.7 Efficiency Gains from Co-operation by Tasks</td>
<td>274</td>
</tr>
<tr>
<td>14.7.1 Co-operation rules</td>
<td>279</td>
</tr>
<tr>
<td>14.7.2 An unpile_C procedure for co-operative unpiling</td>
<td>279</td>
</tr>
<tr>
<td>14.7.3 Smart self help by tasks</td>
<td>281</td>
</tr>
<tr>
<td>14.7.4 Should tasks always do a co-operative unpile?</td>
<td>282</td>
</tr>
<tr>
<td>14.7.5 Task Specific Co-operation vs Generic Co-ordination</td>
<td>282</td>
</tr>
<tr>
<td>14.8 Two Arm Tower Building with an Agent Controlling Each Arm</td>
<td>283</td>
</tr>
<tr>
<td>14.9 Co-operative Robot Bottle Collection as Two Tasks for One Agent</td>
<td>285</td>
</tr>
</tbody>
</table>
15 Multi-Tasking Semantics and Implementation 289
15.1 Evaluation state for a concurrent task 290
15.2 Early Acquisition of Needed Resources 294
  15.2.1 Controlling the order that tasks respond to percept updates 294
  15.2.2 Running and waiting tasks vs runnable and suspended threads 294
  15.2.3 Waiting tasks seeing slightly different BeliefStore states 294
15.3 Detecting when a Task is Inside an Initial Task Atomic Call 294
15.4 Special Translation of Task Atomic Procedures 295
  15.4.1 Call stacks of waiting tasks 295
  15.4.2 The extra tr_rule clause for a task atomic procedure 295
15.5 Querying and Updating the Coordination Facts 295
15.6 tr_rule Clauses for () Action Rules 295
15.7 Updating FiredRules of Waiting Tasks 295
15.8 Suspending the Call Stack Updating 295
15.9 New Tasks being Forked 295
15.10TR rules querying the coordination facts 295

16 Related Agent Systems 297

17 Epilogue: Future TeleoR Robotic Agents 299
17.1 Multiple Calls to TeleoR Procedures as Rule Actions 300
17.2 New Rules with Different refire Semantics 301
17.3 Optional Learning Threads 302
17.4 Using Task Priority and Cost of Execution to Select Runnable Tasks 303
17.5 QuLogP, probabilistic guard inference 303
17.6 Tasks Viewed as BDI Intentions 304
  17.6.1 Option generation rules for a goal 304
  17.6.2 Triggering a task on a significant BeliefStore update 305
  17.6.3 Task priority as priority of triggering event 307
  17.6.4 New start task for the evaluator thread 307
  17.6.5 Recovering from selected option failure 309
17.7 TeleoR Plans 310
  17.7.1 TeleoR plans and task atomic procedures 312
17.8 Sequential, Time-limited Action Procedures 313
  17.8.1 Example action sequence procedure with unconditional actions 314
  17.8.2 An action sequence with conditional actions 315
17.8.3 Syntax of timed sequence procedures ............... 317
17.8.4 Informal operational semantics of timed action proce-
dures .................................................. 318
17.8.5 Minimum execution time .......................... 319

18 Appendix 1: The Formal Semantics in a Nutshell 321
18.1 Semantics for Single Task using Standard TeleoR ........ 322
  18.1.1 Evaluation state for a standard TeleoR procedure call 323
  18.1.2 Initial Evaluation State .......................... 325
  18.1.3 State transition ................................. 325
18.2 Semantics for Single Task using Full TeleoR .............. 326
  18.2.1 Reformulation of previous fire, continue and refire .... 326
  18.2.2 Evaluation state ................................. 328
  18.2.3 State transition ................................. 332
18.3 Semantics for a Task when Multi-Tasking with Multiple Re-
sources .................................................. 332
  18.3.1 Evaluation state as a set of modified task states .... 332
  18.3.2 fire, continue and refire aware of resources ....... 332
  18.3.3 Multi-task transition rule ......................... 332

19 Appendix 2: Syntax Specification of TeleoR and QuLog 333
Chapter 1

Prologue: A Taste of the Book
1.1 What is a Robotic Agent?

This book is about artificial agents, specifically *software agents* [15]. For us, a *Robotic Agent* is a special kind of software process that emulates a simplified version of how we behave when interleaving the execution of several tasks using: reasoning, communication and flexible co-operation with others, and resources (like cars, food processors and computers) which may have to be alternated amongst the tasks.

For a robotic agent these resources are electronically controllable devices, like mobile robots, robotic arms or remotely controllable electronic devices, *Things*, connected to the Internet [6]. In place of our sense organs it has electronic sensors such as cameras with image-processing software, pressure sensors on the grippers of robotic arms, bump and sonar sensors on a mobile robot, or movement and open door sensors in a house. It has a rule based reasoning capability used to determine what action to execute next, for each task.

This action selection uses rules that link high-level interpretations of sense data with high level actions. Low-level sense data interpretation, particularly image processing algorithms, and device control algorithms such as those needed to move a jointed arm, are assumed as primitive perceptions and actions. They are at the sub-conscious level in humans. These important issues are treated in other books, such as [39],[16]. Our robotic agents use the rules to make conscious and purposeful decisions regarding which (possibly programmed) device control action to execute.

*Teleo-Reactive* robotic agents have their task behaviour controlled by a program written in *TeleoR*, an extension of Nilsson’s *Teleo-Reactive* [91] robotic agent programming language, which we shall refer to as *TR*. Task procedures execute primitive actions and invoke sub-procedures that achieve sub-goals of the task goal.

*Teleo* means to *bring to an end* or to achieve a goal. In the case of *TR* and *TeleoR* it is the goal of a task or one of its sub-tasks.

*Reactive* means that task execution continually monitors, via received sense data and communicated information, the effects of robotic resource actions or changes to its environment brought about by exogenous events, and reacts in a timely fashion to this information to change or modify its current action, or to abort a sub-task.

Programming tasks in *TeleoR* rather than *TR* enables us to program goal directed task behaviours for a multi-tasking robotic agent with:
The formal specification and techniques for implementing the TeleoR programming language, and its use to program the behaviour of a sensor monitoring, communicating and reasoning multi-task robotic agent, are the main topics of this book.

The TeleoR robotic agent process may be located on one of its robotic resources. This is often the case when there is just one resource which is moveable, like a mobile robot, with all sensors on board. The agent is then its on-board ‘controller’. For us this is just a special case robotic agent/robotic resources combination.

1.2 What is a Teleo-Reactive Agent Program?

A Teleo-Reactive program offers a simple, elegant and intuitive way of both programming and specifying robust, opportunistic, goal directed robotic agent behaviour. To see this goal directed elastic behaviour, download Nilsen’s Java application simulating a robot arm building a tower of blocks [92]. You can help or hinder by moving blocks around as it is building a block tower, which you give it as a task goal. The agent will immediately respond appropriately. In particular you can co-operatively build the block tower with the arm agent if you choose. Teleo-Reactive programs are well suited to programming robotic agents for human/robotic resource co-operative work.

We will give you a flavour of the TeleoR task programming language by giving the simplest procedure of a program that we fully discuss in Chapter 3. The full program comprises a set of TeleoR procedures, each of which comprises a sequence of

\[ \text{guard} \rightarrow \text{action} \]
rules where the *guard* is a query to the robotic agent’s *BeliefStore*. This contains rapidly changing *percept* facts recording current sensor data, as well as optional fixed rules and facts to aid the application specific *interpretation* or *understanding* of the percepts. Different *TeleoR* procedures are the task and sub-task behaviours.

The robotic agent has just one task executing a top level *TeleoR* procedure with a goal to deliver a bottle to a delivery area. It achieves this goal using sub-tasks that: get next to a seen bottle by approaching it, get hold of it by closing its open gripper, deliver it by getting next to the drop by approaching, then releasing the bottle by opening the gripper.

Below is the general purpose *TeleoR* approach procedure comprising just two action rules for getting the bottle collecting robot to approach something *Th* at a certain forward speed *FS*, with a correctional rate of turn *TS* if it wanders off target. It is used where *Th* is *bottle* or *drop*, where *drop* is a delivery area for a collected bottle. *Th,FS,TS,Dir* are variables representing any value of the correct type. We follow the Prolog convention and use alphanumeric strings beginning with an upper case letter or an underscore _ as variables.

\[
\text{thing::= bottle | drop} \quad \% \text{Type definition for thing type}
\]
\[
\text{approach:(thing,num,num)\~\rangle} \quad \% \text{Type declaration for approach procedure}
\]
\[
\text{approach(Th,FS,TS)\{}
\]
\[
\quad \text{see(Th,centre) \~\rangle move(FS)} \quad \text{Here \% \~\rangle may be read as do}
\]
\[
\quad \text{see(Th,Dir) \~\rangle move(FS),turn(Dir,TS)\}}
\]

Both rules have primitive actions. Where there is more than one they are to be executed in parallel. How the parallel action is achieved is resource dependent. If our robot has steering wheels as well as drive wheels the *move* and *turn* actions would be independent control actions for the robot. If it has just independently controllable drive wheels, the combination of a *move* and a *turn* would be mapped into different turn speeds for the drive wheels by robot interface code.

Suppose the procedure is invoked by a call \texttt{approach(bottle,1.5,0.1)} by the bottle collecting task, because a bottle has been sighted. Its rules become

\[
\text{see(bottle,centre) \~\rangle move(1.5)}
\]
\[
\text{see(bottle,Dir) \~\rangle move(1.5),turn(Dir,0.1)}
\]

Action rule guards, *BeliefStore* queries before the \~\rangle, of a procedure call are always checked starting at the top rule. The *first* rule found with a guard query to the *BeliefStore* that *succeeds* - is inferable generating bindings
for any variables - becomes the call’s *fired rule* and the rule’s instantiated action is executed. If the latest percept about the seen bottle, added to the *BeliefStore* within the last second, reports it is more or less in the centre of the camera image, the top rule is fired. Its action is to start or continue moving forward at a speed of 1.5.

Now suppose that new sense data from the camera results in an update of the agent’s *BeliefStore* so that *see(bottle, left)* is now inferable, but not *see(bottle, centre)*. As soon as that happens, the second rule will be fired.

This will continue the move forward but with a parallel slight turn to the left, causing the appropriate control signals to be sent to the robot. This is to re-achieve the guard of the rule above. As soon as that happens the first rule will be fired again, to have the robot approach in a straight line again.

In the above procedure the rule actions directly controlled the motors of the mobile robot resource. More generally, the action of a *TeleoR* rule is either one or more resource actions, with multiple actions executed in parallel\(^1\), or the rule action is a *single* call to a *TeleoR* procedure, including a *recursive* call.

The above *approach* call was the result of firing of the middle rule of

\[
\text{near(bottle,\_)} \rightarrow \text{approach(bottle,0.5,0.2)} \\
\text{see(bottle,\_)} \rightarrow \text{approach(bottle,1.5,0.1)} \\
\text{true} \rightarrow \text{turn(left,0.5)}
\]

which are the last three rules of the parent call *get.next.to(bottle)*. The ‘\_’ underscore as a parameter means any direction, we do not need to know which. The last rule is the default rule of the calling procedure that is used if no bottle is in view, or if it goes out of view when the robot is approaching a bottle. If that happens the call to *approach* is abruptly terminated.

That is another peculiarity of *TR* and *TeleoR* procedures. The calling procedure remains *active* as do all the ancestor procedure calls. The parent or some other ancestor procedure call will terminate the most recent procedure call action, as well as any intermediary procedure calls, when it fires a different rule. There is no *return* or *exit* action in a Teleo-Reactive procedure. They persist unless explicitly terminated by an ancestor call.

When the *approach* call is terminated on firing of the above default rule in the parent call, the *move* action of the robot will be terminated and a *turn* to the left action started, or modified if the robot was already turning slightly having fired the second *approach* rule. The robot is made to turn

\(^1\)In Chapter 10 we shall see that time limited actions can also be executed in sequence.
to the left, on the spot, at a speed of 0.5. The two rules above the default rule invoke the approach procedure with different speeds, depending upon how far away a seen bottle is when it comes back into view.

1.3 Key Features of our Teleo-Reactive Robotic Agents

We have not said how sense data or messages are handled by our robotic agent. They are handled by separate threads inside the agent both of which are asynchronously updating the agent’s BeliefStore.

A multi-threaded communicating teleo-reactive robotic agent:

- Is a multi-threaded software process with a particular updatable state. This is a BeliefStore of changing facts and fixed facts and fixed relation and function defining rules.
- Robustly and opportunistically executes multiple tasks with compatible goals as separate threads. Tasks alternate the use of one or more robotic resources situated in and acting on an environment external to the agent - the robotic aspect.
- Is able to frequently harvest sense data about the state of the environment, or the resources, using sensors in the environment which may be in or on the robotic resources, or quite separate. This data is used to immediately update its rapidly changing percept beliefs, by the agent’s percept handling thread.
- Receives messages in a special incoming message handling thread - its public interface - possibly updating its BeliefStore with new beliefs extracted from the message. The communicated beliefs can affect task behaviour just as much as new percept beliefs.
- Has tasks that might update the BeliefStore, perhaps to record use of resources or progress towards a goal.
- Has tasks that might send messages to other agents or humans. The messages may inform or query. If they query, answers are sent back to the message handling thread.

---

2 By thread we mean a distinct computational activity that alternates execution time slices on a computer’s processors with all the other process threads. The time slices may be as short as one or two milliseconds. We can think of threads as executing in parallel.
- Has tasks that react quickly to changes in the BeliefStore relevant to the task - the reactive aspect - immediately changing their resource using actions if need be, and aborting and starting sub-tasks.

**Atomic transactions and task atomic procedures**

Each thread that updates the BeliefStore must leave it in a consistent state for other threads. This is done by having manipulations of the BeliefStore be done by a thread as an atomic transaction - a sequence of queries and updates that will lock access to the BeliefStore until completed. The updates to the percept beliefs by the percept handler are done as an atomic transaction so that the task threads can query a consistent set of percept beliefs for the latest environment state.

The alternation of resource use between concurrently executing tasks is done using the concept of task atomic procedures that have stable sub-goals. A task atomic procedure uses one or more resources and a task must wait before entering such a procedure until it can acquire the exclusive use of these resources for the execution of the procedure. When the procedure is terminated the resources may be released because the procedure will either have achieved its stable sub-goal or the sub-goal will have been abandoned. A stable sub-goal is a goal that will not be undone by other tasks using the released resources.

Task atomic procedures constitute our most important extension to Nils-son’s TR language. Our TeleoR compiler and run-time support code are such that the tasks do fair resource use co-ordination themselves using the shared BeliefStore in the manner of an inference enhanced Linda tuple space [22]. They inform one another about current resource use, resources they now need, and their position in a wait queue for these needed resources. Needed resources can change whilst a task is waiting due to new beliefs causing a change of fired rules and the need to enter a quite different task atomic procedure.

Ensuring non-overlapping fair access to resources whilst avoiding deadlock - a situation where no task can get access to the resources it needs - is one of the challenges of concurrent programming [11], for which various solutions have been offered [47].

Our solution for TeleoR programmed tasks just requires the TeleoR programmer to make task atomic, and related task_initial assertions about
key procedures. The translator does the rest.

The agent’s tasks will have opportunistic, robust, non-interfering and occasionally co-operative behaviour by being programmed using extensions to Nilsson’s Teleo-Reactive agent programming language as described in [91]. How this is done is the main topic of this book, hence its title.

Why Inference Querying and Multi-threading?

It is not controversial to assume an inference capability as a requirement for an agent process [15]. It is an implicit assumption of the two proposed software agent lingua francas [73][49][50]. The lingua francas are forms of messages that can be used to allow agents implemented by different organisations to communicate.

Requiring a multi-threaded architecture is somewhat controversial. However, a major benefit is that optional capabilities such as SLAM [104] (Simultaneous Localisation And Mapping), or abductive [67][43] reasoning, can be added as extra threads with no re-write of the programs for other threads. SLAM might be used to construct a partial map of the environment when one of the agent’s resources is mobile with attached sensors. Abductive reasoning might be used to add beliefs that ‘explain’ unexpected acquired beliefs, This is discussed further in the Epilogue.

Another major benefit of multi-threading is that it allows the tasks to co-ordinate their use of overlapping sets of resources themselves, with no need for a central resource allocator within the agent. The novelty in our approach is that it allows rule based programming of sensing, reasoning, multi-tasking robotic agents, each robustly and opportunistically working towards the goals of several tasks. The agents interleave the use of several robotic resources between the tasks to achieve the task goals. At the same time they communicate and co-ordinate their actions with other agents and humans by sending and receiving information that gets absorbed into their BeliefStores to affect subsequent behaviour.

1.4 Example Applications

- One agent, controlling one robotic resource for one or more tasks, fairly alternating the resource use between the tasks. For example an agent controlling robotic arm involved in several interleaved construction tasks.
- Several agents, each controlling their own single robotic resource for
one task, each just using sense data from resource independent sensors to co-ordinate their resource use to achieve an implicit shared performance goal. For example an automated production process involving several independently useable controllable devices like presses and robot arms. This could be elaborated to communicating agents using both communicated information and sense data to coordinate their actions and to have changeable explicitly given joint goals for flexible production process use.

• Several agents, each controlling their own mobile robot resource, using sense data and communication to co-ordinate their robot’s actions to efficiently achieve individual or joint goals. For example, several agents controlling mobile robots in the same space in find, collect and deliver tasks that may have a delivery total that is a joint goal. The agents tell one another when their robot has made a new delivery allowing each to have a shared belief about the current joint total. Their top level task procedure will have reached its goal when the joint recorded total is reached. Another example is a airborne small drone mission with a controlling agent for each drone with the agents communicating key perceptions to one another. The also receive information updates from a human mission co-ordinator that may significantly change their co-ordinated task behaviours.

• The above application multi-robot collect and deliver scenario but with one agent controlling all the robots. The agent co-ordinates the robots’ actions using percepts from the sensors of all robots. There is no communication.

• One agent controlling several robotic resources for multiple independent tasks with significant parallel use of resources and co-operation between the tasks whenever possible. For example, one agent controlling two or more robot arms in multiple fetch and construction tasks requiring co-ordinated and parallel use of both arms.

• Several WiFi communicating TeleoR agents which are the distributed intelligence of an ambient intelligence [38][108] system assisting an elderly person in their smart home [37]. The agents each have an extra reasoning thread that participates in a distributed abductive reasoning algorithm [78] to agree hypotheses about the person’s behaviour and wellbeing. They give aural advice, partially control devices to assist the person, and occasionally send out alert messages to relatives and doctors.
Several non-communicating agents controlling animats with *multiple tasks/goals* and *dynamically* acquirable resources in a computer game. They get sense data from the game engine [55].

For ambient intelligence applications the harvesting of sensor readings and dispatching of actuator action messages has been made easier by the MQTT [63] *publish/subscribe* [7] protocol for connecting *Things* to sensor monitoring and actuator controlling processes. It will be eased further when *Things* have high level interfaces describing their capabilities [6][9].

We have developed test applications for the examples of the first five of the above possible application areas. We have mostly tested them with interactive simulations written in Python, so that we could help or hinder to test the elasticity of the TeleoR task control programs. We shall give and explain the programs for all but the second, the implicit joint performance goal application for an industrial process.

Ambient intelligence and game programming are application areas we have as yet only briefly explored [19].
Chapter 2

Origins of TeleoR and QuLog
and Book Overview
This background and description of the book contents assumes that the short Prologue has been read, as that gives a bird’s eye view of the contents of the book. If you are keen to see more of the program fragment of the Prologue you may skip now to Chapter 3. You can come back to this chapter afterwards, or even after you have just part read Chapter 3.

We start by giving a one page summary of our prior use of Qu-Prolog [31], a multi-threaded Prolog system, for programming multi-threaded communicating and reasoning agent applications. This is what lead us into Teleo-Reactive Robotic Agent programming. Moreover both TeleoR and QuLog are currently implemented by translating them into Qu-Prolog, and many of their features derive from Qu-Prolog. In particular, the message communication, thread forking, atomic action sequences and wait queries of the action procedures of QuLog, described in Chapter 8, derive from Qu-Prolog. These powerful high level control features were added to Qu-Prolog to facilitate the implementation of multi-threaded message communicating software agents. Multi-threaded TeleoR+QuLog programmed robotic agents inherit characteristics from both Nilsson’s TR single task non-communicating robotic agents, and Qu-Prolog multi-threaded communicating software agents.

The reference implementation for TeleoR software agents of Chapters 9, 12 and 15 is described using QuLog. QuLog started as a purely declarative logic programming language for ‘programming’ a TeleoR agent’s BeliefStore. It evolved into a major re-design and transformation of Qu-Prolog into a modern type safe higher order language that integrates logic and functional programming, with an imperative layer on top. The imperative layer comprises the QuLog action procedures.

After the overview of Qu-Prolog and its use for communicating agent applications, we describe the different class of robotic agent applications that Nilsson’s TR programs address. We give a brief account of the origins of the TR concept. This is followed by an overview of the extensions we made to TR action and rule forms for facilitating and extending the range of single task robotic agent applications.

The elaboration allows a robotic agent task thread to concurrently send messages and update the agent’s BeliefStore, as part of an extended action of a task rule. This architecture is depicted in Figure 2.3 and will be discussed in more detail shortly\(^1\). The extra forms of actions and rules also provide enhanced control over task execution requiring more complex translation

\(^1\)A very early implementation of this communicating three thread architecture in Qu-Prolog was used for several years to teach both robotic and communicating agent concepts at Imperial College.
of the TeleoR action rules and run-time support. The implementation of the extended TeleoR was developed in tandem with its formal operational semantics, to the benefit of both.

Our most recent implemented extensions to TeleoR were to facilitate the programming of multi-tasking robotic agents. First, we extended the language so that we could have multiple tasks fairly sharing just one robotic resource, with interleaved and non-interfering execution of the tasks. By non-interfering we mean that no task would undo sub-goals achieved by another task. We then added the concept of resources and resource arguments to actions and procedures to allow concurrent multi-tasking using multiple robotic resources, with parallel use of the resources. We managed to do this whilst maintaining the fair sharing of resources and non-interfering behaviour of the tasks. The implementation of translator for multi-tasking TeleoR, and its supporting multi-threaded run-time architecture, were also beneficially developed in tandem with its formal operational semantics. This is an interleaving activity for language designers and implementers that we can thoroughly recommend.

2.1 Multi-Threaded Communicating Agents

For quite a few years we used the multi-threaded Qu-Prolog for building multi-agent applications [30][32] in which actions were essentially communications to inform, answer, or to request information or services. It was the programming language of a multi-agent systems course at Imperial College. Qu-Prolog’s message handling is modelled on that of Erlang [4], as implemented in the April [80] agent implementation language. The use of April and Qu-Prolog for multi-threaded communicating agent applications anticipated the much more recent work on using and extending Erlang for this purpose [115][119].

Qu-Prolog was originally developed as a single thread Prolog, enriched in order to support the implementation of sound theorem provers such as Ergo [118]. It was then extended to make it a language suitable for implementing message communicating multi-threaded reasoning software agents [31]. To this end we added time-shared threads that could either communicate with one another by manipulating the shared dynamic data base of asserted and retracted facts, as in the Linda [22] co-ordination model, or by asynchronous communication of messages, as in Actors [60] and Erlang [4]. Messages can be used to communicate between threads in different Qu-Prolog processes using an intermediary communications server.
Recently we changed the syntax-supported communication of messages between threads of different Qu-Prolog processes to use our Pedro communications server [106]. Pedro supports both publish/subscribe [7] and peer-to-peer addressed communication. Addressed communication enables threads in different Qu-Prolog agent processes, on different hosts, to asynchronously directly communicate by exchanging Prolog term messages. The messages can contain variables but are copied so there is no sharing of variables between different Qu-Prolog processes, or even between different threads in the same process. However it does allow the easy transmission of messages that are queries containing variables. These variables are usually given bindings if the query is accepted by the receiving agent process and successfully answered. The fully or partly instantiated query is then sent in an answer. It therefore does not matter if variables in the dispatched query are replaced by new variables on receipt. All that matters is that multiple occurrences of a variable become multiple occurrences of its replacing variable. This is what happens. We have an example of this sort of agent querying in Section 10.8.

Email style addresses identify the different agent processes. The messages are automatically converted to and from ASCII strings for transmission. The publish/subscribe capability of Pedro allows it to take over some of the role of a KQML match maker [72], or FIPA Directory Service [51], as illustrated in [106]. APIs exist for linking Pedro to Python, Java and C, C++.

All these features have made Qu-Prolog highly suited to implementing multi-threaded communicating and reasoning software agents. The agents can also communicate, via Pedro, with processes implemented in other programming languages, using symbolic messages. Such hybrid applications are used either for visualisation of the agent interactions, for example [33] and [53], both of which used visualisations implemented in Tcl/Tk of both the fleeting inter-agent message communication and key components of the changing BeliefStore state of each agent. Pedro may also be used for linking with an already implemented service process, perhaps a web application accessible using Python or Java. These features of the Qu-Prolog + Pedro combination that facilitate the building of hybrid distributed applications carry over to the use of TeleoR+QuLog+Pedro for building distributed applications comprising reasoning multi-threaded software and robotic agents, and special purpose processes, all interacting using symbolic messages. The messages are dispatched as addressed communications, or as notifications to be routed to any agent or process with a current covering subscription on the Pedro server to which they are all connected.
2.2 Teleo-Reactive Programs for Sense/Query/Act Robotic Agents

There is another class of agent applications, for example robotics, where, as depicted in Figure 2.1, the agent is situated in an environment the state of which it partially observes through its sensors. The agent, in pursuit of a goal, reacts to these sensor readings by acting on the environment using robotic or actuator resources. These actions can be durative actions such as move forward at a certain speed. The agent detects the effect of its actions by frequent sensing of its environment using sensors that may be quite separate from or located on or in the resources. The environment can also be changed by exogenous events, for example the actions of other robotic agents or humans or nature, that might help or hinder. For this type of agent, fast and robust reaction to rapidly changing sensor readings, to determine a goal directed action response, is required. Some inference querying of the sense data might be required, but that should be focused, fast and task specific.

![Figure 2.1: Situated Agent](image)

Nilsson’s TR programs [89][91] are designed specifically for this class of agent applications. Moreover, particularly in the version as described in [91], they fitted very well with our multi-threaded Qu-Prolog. In this variant,
the results of the sensor readings are recorded as transient percept facts and
the TR rules have guards (firing conditions) that query these percept facts,
and other facts that can be derived from them using simple Prolog style
implication rules. The updatable facts and rules are the agent’s dynamic
deductive belief store (BeliefStore for short).

The actions of the TR rules are actions for the agent’s robotic resources,
or they are TR procedure calls. That is, they are either primitive actions of
the agent or they are TR programmed actions. The programmed actions are
sub-tasks. Each rule action is usually intended to achieve a goal, which is
the goal of the procedure call Q in which it appears, or a sub-goal of that
goal. The goal of Q is often the partially instantiated guard of its first rule,
but it could be the partially instantiated guard of a rule R of Q’s parent
procedure call P, where R is before the rule of P that called Q.

The resource actions executed in pursuit of these sub-goals are dependent
upon the perceived state of the environment, as recorded by the current state
of the agent’s BeliefStore. Each action is executed conditionally upon the
guard K of an action rule K \rightarrow A being inferable from the current state of
the BeliefStore.

Since TR procedures can be called as rule actions, allowing recursive
and mutually recursive procedures, and because TR rule guards do inference
querying of a dynamic BeliefStore of facts and rules, TR procedures are quite
different from the Subsumption Behaviours [20], [21] of Brooks. They have
in common the concept of a hierarchy of action determining rules with a
higher rule suppressing the behaviours of lower rules when the sense data
tests of this higher rule succeed. However a Subsumption Behaviour is not
wrapped up as a program procedure that can be directly invoked by a rule
of another behaviour. Subsidiary Subsumption Behaviours get invoked by a
spreading activation mechanism.
Figure 2.2: Shakey Robot & Co

**Historical Aside - Origins of Nilsson’s TR**

Interestingly, as mentioned in Nilsson’s “Quest for Artificial Intelligence” book [93], the TR concept seemed to have been crystallised during a visit to Brook’s robotics group at MIT during a sabbatical year starting in 1990. However, its roots were in the work done by Nilsson and colleagues at SRI in the early 1970s [?] on generating and flexibly executing action plans for a mobile robot Shakey. Shakey became famous because it was featured in Life magazine [42]. The control concepts for Shakey were influenced by Miller, Galanter, and Pribram’s TOTE control concept [82], and the homeostasis concept of
Ashby [5].

Shakey’s controlling agent had a *BeliefStore* of percept facts about objects, rooms, and doorways. These facts were initially provided by Shakey’s programmers, but they could be updated using image analysis routines specially developed for ‘seeing’ things of interest in the robot’s environment, using an on-robot camera. The environment comprised several rooms with connecting doors some of which might be closed or otherwise blocked, with one room containing moveable boxes of various shapes and colours.

With Shakey located in a particular room, its controlling agent was given a specific goal, say to block the door between two other rooms with a box from one of these other rooms. The plans were generated using the STRIPS [48] planner that made use of pre and post condition fact specifications for parameterised actions. The STRIPS generated plan was a specific plan to get the robot from the room it was in now to the room with the box to be used to block the door way followed by the actions needed to get behind and push the box to the door opening to be blocked. The plan was generated using initial beliefs about the state of the environment, e.g. fixed which doors connected which rooms and dynamic beliefs about which were open and not blocked. The initial dynamic beliefs about doors might be false.

This specific plan was then generalised to a plan with the same number of actions but with starting and finishing room names, and doorway and intermediary room names replaced by variables. The generalised plan was then represented as a triangular table with the right end of each row of the table labeled by a parameterised action, and the left hand side column by the necessary preconditions for a successive execution of the action of that row. So each row was not unlike a TR condition/action rule.

The generalised plan table could then sometimes be used to recover from a situation arising in the execution of the specific plan when all the pre-conditions of the next action were not satisfied. For example, the door to be used for moving to the next room of the specific plan may now be seen as being closed or blocked. This update of the agent’s *BeliefStore* was only possible when the door could be seen. In such circumstances, the rest of the generalised table plan could sometimes be re-instantiated to achieve the goal of being in some destination room using a different route through doors and intermediary rooms from where
the robot was now. This is a weak form of the capability of a TR procedure to fire different instances of its condition/action rules. If a different re-instantiation of the rest of the triangular table plan was not possible, STRIPS was re-used to re-plan to achieve the goal using the agent’s current updated beliefs.

The intermediary level primitive actions of a Shakey triangular table plan were programmed using numbered simple perceive/action rules with their condition often being a single percept test. These were called Markov tables and in the use of these tables the influence of the TOTE (Test, Operate, Test, Exit) and homeostasis concept is seen. Markov tables are also a clear ancestor of TR rule sequences. This paragraph from [93] describes Markov tables and their use to achieve robust execution of plan actions.

“In thinking about how to achieve this robustness, I was inspired both by Miller, Galanter, and Pribrams TOTE units and by the idea of homeostasis. (Recall that a TOTE unit for driving in a nail keeps pounding until the nail is completely driven in and that homeostatic systems take actions to return them to stability in the face of perceived environmental disturbances.) I wanted the mid-level programs to seek and execute that action that was both closest to achieving their goals and that could actually be executed in the current situation. If execution of that action produced a situation in which, as anticipated, an action even closer to achieving the goal could be executed, fine; the mid-level program was at least making progress. If not, or something unexpected caused a setback, some other action would be executed next to get back on track. Richard Duda and I developed a format, called Markov tables, for writing these intermediate-level programs having this keep-trying property.”

A crude characterisation of a TR program is that the initial procedures that have conditions requiring inference from dynamic beliefs comprise the condition/action rules of a generalised STRIPS plan, written in reverse order to the row entries of the plan. This allows opportunistic jumping forward in the plan. The plan actions become calls to intermediary level plans/procedures that eventually call procedures with perceive/action rules that are the descendant of the Shakey agent’s Markov tables.

Because of its natural fit with our multi-threaded logic programming approach to programming software agents, we were very happy when we came
across Nilsson’s Triple Tower paper [91]. We immediately set about embedding TR programs as a syntactic extension of Qu-Prolog. This embedding allowed us to program the reactive goal directed behaviour of an agent’s task thread by a set of TR procedures mixed with Qu-Prolog facts and rules giving the fixed component of the agent’s BeliefStore. The task thread, which executed TR procedures translated into Qu-Prolog programs, became one of three threads of a three-thread agent architecture as depicted in Figure 3.1.

The other two threads were programmed directly in Qu-Prolog, or a mixture of Qu-Prolog and C/C++.

One of these Qu-Prolog programmed threads was the sensor interface of the agent frequently receiving batches of sensor readings to atomically add as new percept facts to the agent’s BeliefStore. The other was a message receiving thread able to accept information supplied by humans in relatively infrequently received tell or untell messages. The information in each message was also atomically added to the BeliefStore as facts. The message receiving thread was the public interface of the TR agent. Messages were sent to it using an email style address such as bottle_collector@texel03.doc.ic.ac.uk via a Pedro communications server. texel03.doc.ic.ac.uk is the host on which the agent process is executing, with a Pedro registered name bottle_collector.

We explored the use of such single task TR agents to control both simulated and real mobile robots, and simulated non-mobile robotic resources such as robot arms, conveyor belts and presses. The simulations were written in Python and had interactive visualisations allowing us to explore the elastic capabilities of TR robotic agents and their potential use for human and robotic agent co-operative work.

With colleagues Ian Hayes and Brijesh Dongol at the University of Queensland we looked at the use of TR procedures for programming classic test cases in safety critical systems and industrial automation. One was a control problem involving two conveyor belts, a press and two connected robot arms, all five robotic resources being controlled by their own TR robotic agent running as a separate Qu-Prolog process. The actions of the TR agents are co-ordinated and triggered by specific sense data. Industrial process control was one of the possible application areas for TR robotic agents we mentioned in the Prologue.

2.3 Application Driven Extensions of TR Programs

Partly as a result of our test applications, but also because of our embedding in Qu-Prolog, we extended Nilsson’s TR language of [91]. We added:
• concurrent execution of primitive actions

• the optional attaching of a *BeliefStore* update, or a Qu-Prolog message send action, or a call to a Qu-Prolog behavioural program to any action rule

• **while, until** and combined **while/until** rules that change the normal conditions under which other action rules of a procedure call can be fired, when one of these rules has been fired.

• timed action sequences allowing cyclic execution of a sequence of time limited durative actions, including procedure calls.

• **wait/repeat** actions allowing the re-starting of the primitive actions of a fired rule a small number of times at specified intervals when there has not been the expected observable result.

**while/until** rules are the most general form of rule. The other forms, including the original rules, are just special case uses of **while/until** rules. This extended TR language is the single task subset of *TeleoR*.

The linked Qu-Prolog goal allowed atomic updates of the agent’s *BeliefStore* to be executed when the primitive actions determined by the firing of the rule are executed. This enabled an agent to remember past key actions and perceptions. It also allowed agent to agent communication and agent to human communication as well as the human to agent communication we already had. This enabled *TeleoR* agent’s to inform and query one another, and for query answers to be returned. All messages were sent to the message handling thread inside each agent using the portal architecture of Figure 2.3. Information transferred by a message, perhaps the answer to a query message sent out by the task thread, was simply remembered as one or more facts to the *BeliefStore*. When added, such facts could immediately affect the behaviour of the task thread causing sub-tasks to be terminated and new ones to be launched. This allowed us to program robotic agents with goals that were achieved by a combination of robotic resource use by several robotic agents that kept each other informed about progress towards a joint goal.

These extensions significantly expanded the range of single task robotic agent applications we could program. They also made the programming of already covered applications easier, or more transparent, with a cleaner semantics. We tested all of the extensions with an application involving two identical communicating robotic agents with a given joint goal, co-ordinating their actions using communication as well as sense data from their mobile
robot resource, which each separately controls. Each agent keeps the other informed regarding progress towards their joint goal. The robotic agents know when the joint goal is reached because of the communication, and would not know without the communication. This application is fully explained in Section 10.7.

Figure 2.3: Three Thread Communicating TeleoR Agent Architecture

Along with extending our initial implementation of TR programs we started developing the formal operational semantics. This turned out to be an essential concurrent activity as we found it necessary to alternatively revise both the language extensions, their semantics and their implementation. One fundamental revision was regarding the semantics of until rules. This had been suggested to us by a colleague at the University of Queensland, Ian Hayes, but with a strong semantics that was difficult to implement; we did not succeed. That led to weakening of its semantics and the adoption of the dual rule form, the while rule. Defining the semantics also considerably helped with issues regarding optimising the implementation. It gave us
a clear description of the behaviour we wanted to optimise. It also exposed an ambiguity in the informal operational semantics given by Nilsson in [91].

There are echoes in the until and while rules of TeleoR of the input inhibition concept of Brook’s Subsumption Behaviours [20], [21] as both our new forms of rules temporarily prevent the guard evaluations of other rules of the procedure call. Subsumption Behaviours were not our motivation for adding the new rule forms. The motivation was to enable the programming of certain behaviours using behaviour transparent programs with a cleanly defined semantics. But it is interesting that this motivation resulted in a control mechanism similar to inhibition.

To summarise, our single task agent architecture has three threads as depicted in Fig 2.3:

- a percepts thread that executes a simple program that gets new sensor data at specific intervals, or just waits for new sensor data to become available, depending on the application. It then converts this data to percept facts using an application-specific set of predicates (a simple ontology [2]), and updates the agent’s BeliefStore with the new percept facts using an application-specific BeliefStore percepts update program that may choose to remember old percepts or add other facts. It may also do some application-specific reasoning to infer and add to the BeliefStore certain rule defined percepts.

- a messages thread that waits for messages from other agents and people. It also may update the BeliefStore using an application-specific program for dealing with a single message.

- an evaluator thread executing some start procedure call of a TeleoR program with $K \rightarrow A$ conditional action rules grouped into procedures. Here $K$ is a query to the agent’s BeliefStore, and $A$ is a set of robotic resource actions that can concurrently update the BeliefStore and send messages, or it is a call to a TeleoR procedure.

The evaluator converts primitive actions into control messages that do discrete actions, or stop, modify or start durative actions taking into account the last set of primitive actions it determined should be being executed. They are sent to a simulator, or to a real application via a robotic resource interface that could be implemented in C/C++ or Java, and be running as a separate process.

23
2.3.1 Extending with extra learning or reasoning threads

In the Prologue in Section 1.3 we mentioned that a major advantage of a multi-threaded architecture was that extra capabilities could be added to the agent by simply adding extra threads that did atomic querying and updates of the BeliefStore. Such an extended architecture is depicted in Figure 2.4. It has a SLAM thread perhaps constructing a topological map of the environment [27] and an adductive reasoning [67] [43] thread perhaps engaged in distributed adductive reasoning with other agents [78].

![Figure 2.4: Five Thread Communicating TeleoR Agent Architecture](image)

2.4 Task Atomic Procedures for Multi-Tasking

Our most radical extension of TR is the concept of task.atomic procedures, and the auxiliary concept of task.start procedures, to allow multi-tasking by our TeleoR robotic agents. We started with an extension to our single task
agent architecture in which the **evaluator** thread became a task forking thread, forking tasks sent as **requests** to the agent’s **messages** from other agents or humans. These were all tasks that used and must share a single resource and a new task is forked only if compatible with existing tasks.

This was tested by changing the block tower building program of [91], reproduced in Section 6.2, so that it could safely interleave the building of several towers using different blocks and one arm. The change required was the adding of two **task_atomic** procedure assertions and one **task_start** procedure assertion. The former caused a quite different compilation of the procedures. We tested this with a visualised simulation similar to the Java simulation of the one tower builder accessible from [92]. We could help or hinder the agent controlling the simulated arm as it interleaved its two or three tower building tasks, as the simulation was interactive. The agent always responded rapidly and appropriately. The natural generalisation,

![Figure 2.5: Concurrent Multi-Tasking TeIoR Agent Architecture](image)

concurrently implemented with the elaboration of the semantics we had developed for a single **TeIoR** task, was to allow multi-task programming using multiple independent robotic resources that could be used in parallel. This required us to add a resource use co-ordination policy to the opera-
tional semantics that could be implemented without the need for a resource allocation thread. Another goal was to allow as much parallel use of the resources as possible, consistent with a ‘fair’ allocation of resources so that there was no starvation of any task of the resources it needed, and there was no possibility of deadlock.

Again there was useful symbiosis between the development of the implementation and the semantic specification. The resource use co-ordination policy is implemented as a concurrent algorithm using atomic transactions on the BeliefStore. Its specification is much more abstract and makes no mention of BeliefStore manipulation. Figure 2.5 depicts this architecture, which we shall discuss again when we compare it with Nilsson’s Triple Tower architecture of [91] in Chapter 6. We will then use a slightly different figure.

We tested this multi-tasking architecture by re-programming Nilsson’s tower building program so that it could be used by an agent interleaving the building of multiple tower blocks distributed over three tables using two robot arms. A short video showing the two arm controlling agent in action, using a visualised simulation in Python and its PyQt GUI extension, is accessible from [34].

This application can be viewed as a simplification of a robotic arms assisted product assembly line in which a two arm controlling agent and, perhaps several people, co-operate flexibly on multiple assembly tasks.

2.5 Typed and Moded QuLog for the Agent’s BeliefStore

Quite recently we decided we needed to ensure that TeleoR rule guards, which were just Qu-Prolog queries, did not generate run-time errors due to incorrect use of arithmetic primitives (e.g. not supplying both argument values as numbers for an inequality test), and that robotic resource actions were fully instantiated and correctly typed when sent out to an external device or simulator.

We first added type and mode declarations for relations with numbers, atoms (aka symbols), strings or lists of these types of values for relation definitions and for the primitive arithmetic relations. Type checking the TeleoR procedures and BeliefStore clauses using these declarations guaranteed freedom from run-time errors and type correctness, and guaranteed full instantiation with the correct types of values of robotic actions. We next allowed program defined types, particularly enumerated and range types to ensure that any symbolic and integer argument values of the robotic actions
were more precisely constrained to required values by source program type checking rather than runtime checks. But so as not to sacrifice too much of the ease of programming in untyped Prolog, we did sub-type checking rather than strict type checking.

As a syntactic convenience, and at the request of colleagues, we then added function rules and the use of function calls, arithmetic and set expressions as relation call arguments. After adding function rules the next step was to make QuLog higher order so that relation and functions could given as arguments to relations and functions providing they satisfied a special sub-type relation to the required higher order argument type. This resulted in an expressive typed and moded higher order logic+functional programming language for ‘programming the agent’s BeliefStore. This is the declarative subset of QuLog.

The syntactic richness of QuLog as compared with Prolog means the BeliefStore rules and TeleoR rule guards of our agent programs can be less verbose and much more transparent than the Prolog equivalent. Unfortunately this expressive power is not illustrated in the relatively simple BeliefStore’s of the example robotic agents of this book, kept simple for pedagogical purposes.

The final development of QuLog was the addition of action rules defining action procedures that can call functions and query relations, but not vice versa. They are the imperative upper layer of QuLog. The action rules are what allow us to program the entire TeleoR+QuLog agent architecture in QuLog. They also enable us to allow a combination of an atomic transaction on the agent’s BeliefStore and message send actions to be executed concurrently with robotic resource actions. This is done by linking a call to a QuLog action procedure with the normal action of TeleoR rule. This was the very first extension we made to Nilsson’s TR rules, except that was to allow a linked call to a Qu-Prolog imperative program.

### 2.6 Higher Order TeleoR Procedures

Because QuLog is higher order, it was easy enough for us to make TeleoR higher order. So, a TeleoR procedure can be called passing in relations, functions, QuLog action procedures or TeleoR procedures. Using this feature enables one to define a TeleoR procedure that is a behaviour ‘plan’ specialised by passing in sub-procedures to be called as rule actions and relations to be called in rule guards. We have a simple example of a higher order TeleoR procedure in Section 3.8.
Higher order TeleoR is the icing on the cake that we get for free because we have higher order QuLog and its sub-type checking algorithm for higher order types.

2.7 TeleoR in stages

Two simulated robotic agent application domains are used throughout the book to exemplify TeleoR+QuLog robotic agent programming. One is agent control of one or more mobile robots used in various bottle collection scenarios. All are developments of the example program of Chapter 3 used to introduce TeleoR programming. The variations include: one agent collecting bottles one at a time indefinitely, reacting only to sense data (Chapter 3); two communicating robotic agents controlling their own robots using simple inference and communication (Chapter 10); one agent using concurrent multi-tasking to control two bottle collecting robots with no need for communication (Chapter 14). In the second example, the robotic agents exchange messages to avoid collision with minimal divergence from their preferred path, and to know when they have reached a joint bottle collection target. (This is a simplification of an automated warehouse fetch application using automated fork lift mobile devices.) In the last example the single agent receives sense data from both robots allowing more intelligent collision avoidance (perhaps a better way to orchestrate the warehouse fetching using several mobile devices).

The other application domain is block tower building and re-configuring, much used as an AI test application. Our example programs range from a slight modification of Nilsson’s program of [91], given in our TeleoR syntax (Chapter 6); via the sleight-of-hand variant that allows fair interleaving of the building of several towers by one agent using one arm (Chapter 13); to the significant elaboration of that program to allow one agent to multi-task the building of any number of towers using $N$ robotic arms and $N+1$ tables (Chapter 14, a simplification of an industrial arm assisted component assembly task); to a discussion of the more complex structure of the program that would be needed if each arm was controlled by a separate agent.

Interleaved with the progressive development of these programs, the requirements of which are used motivate the TeleoR language features not in TR, we develop and present its operational semantics and its reference implementation.

The semantics chapters assume familiarity with the notations of predicate logic and set expressions, and their meaning. Predicate logic is fully
covered in Chapter 3 of [116]. Predicate logic and set expressions are covered in Chapter 1 and Appendix B1 of [94]. Predicate logic syntax and semantics are introduced gradually in Chapters 1 to 4 of [112]. The last two are downloadable from the web without charge. All sources use the same quantifiers, $\forall$ for ‘for all’, $\exists$ for ‘there exists’, but differ with respect to the logical connectives. We use $\land$ for ‘and’, $\lor$ for ‘or’, $\Rightarrow$ for ‘implies’ and $\Leftarrow$ for ‘implied by’ or ‘if’.

Readers familiar with BDI agent architectures such as AgentSpeak [102] and Jason [14] may be wondering why there has been no mention of them in this background chapter. They are not forgotten or ignored. They exemplify another approach to programming agents that fits well with a logic programming approach. We have plans to move our TeleoR + QuLog multi-thread architecture in that direction. It will allow our robotic agents to be asked to achieve a goal, as an alternative to being asked to execute a specific TeleoR procedure call. This will use QuLog rules for an option relation that will give a suitable TeleoR procedure call to use for a requested goal, taking into account current beliefs. A goal is a single call query that should eventually become inferable from the dynamic beliefs in the agent’s BeliefStore after a selected option TeleoR procedure call has achieved its goal.

The same option rules will allow is to add achieve Goal actions as allowed TeleoR rule actions. In addition, by catching the failure of a selected TeleoR procedure call option, and remembering it has failed, recovery from its failure can be handled by the standard TeleoR recovery-from-adversity response.
Chapter 3

Introduction to
Teleo-Reactive Programming
Hopefully the Prologue with its short examples of TeleoR procedures has whetted your appetite to discover more about Teleo-Reactive programming. If you skipped Chapter 2 to start this chapter we recommend you go back to Chapter 2 immediately after reading this chapter, or even half way through.

Figure 3.1: Simple Three Thread TeleoR Agent Architecture

We discuss the behaviour of this first control program in some detail so that you will get a good understanding of the elastic properties of TeleoR procedures, and their very different behaviour compared with called procedures in other programming languages. We also introduce Nilsson’s regression and completeness properties for his TR procedures, and their relationship to program correctness. These properties apply unchanged to the standard TeleoR procedures used in this chapter. We finish with an introduction to the use of QuLog relation definitions to give an agent-specific interpretation of quite low level sensor percept facts.

The agent architecture we assume is as depicted in Figure 3.1. It com-
prises just three threads. The role of the percepts handling thread is to convert an incoming batch of sensor data into an atomic update of the percept facts of the BeliefStore. The role of the message handling thread is to respond to a single message. If the message is an information message the response is usually will be an atomic update of the belief BeliefStore. If it is an enquiry message the response is usually an atomic query of the BeliefStore and the sending of a response.

The third thread evaluates a call to a TeleoR procedure querying the BeliefStore immediately after each time it has been updated by either the percepts or messages threads. The evaluator responds to the update by checking that if all previously fired rule instances in the current called procedures remain the fired rule instances of each call. If not it fires new rules to determine new robotic resource actions, resulting in a change or modification of current action.

An agent is implemented by configuring a generic agent shell with a QuLog+TeleoR agent program. The incremental development and debugging of a fully configured agent is covered in Chapters 4 and 5, and documentation with the software. The software includes the agent shell.

3.1 Structure of an Agent Program

An agent program to be used with the agent shell minimally comprises a set of TeleoR procedures with declarations of the percept facts. It usually also contains QuLog relation and function definitions to augment the querying of the dynamic facts of the agent’s BeliefStore - the rapidly changing percepts facts and more slowly changing told facts. We give an example of such QuLog definitions at the end of this chapter. The TeleoR procedures and relation and function definitions define application specific task behaviour. In addition the program can contain QuLog action procedures for modifying the default behaviour of the three agent threads regarding how percepts are updated, how messages are handled and how control actions are communicated.

The predicates for the dynamic percept and belief facts have to be declared as percept and belief predicates in the program. No facts for the percept predicates can be given in the program file but we can have initial facts for the belief predicates. We can also have rules for some of the declared percept predicates. These are percepts that are not directly linked to sensors but are one level removed. The rules for these percept predicates are used by the generic percepts handler to infer all their instances when a
new set sensor percepts arrives to infer and then add all their instances to
the BeliefStore. We shall see that doing this can significantly improve the
efficiency of TeleoR task execution.

The program file may also contain one or more of six application spe-
cific QuLog action procedures. Three are used by the percept handler to get
interpreted sensor data as a list of percept terms, to use the list to up-
date the percept facts in the agent’s BeliefStore, then to check the updated
BeliefStore for consistency, perhaps adding or removing beliefs. Another is
used by the message handler to deal with each individual message the agent
receives. The last two are used by the TeleoR evaluator. If any of these
optional action procedures is not defined there is a default behaviour by
the generic TeleoR agent supplied with the TeleoR+QuLog system. These
default programs assume that the control actions and incoming sense data
will be communicated as lists of QuLog terms. They also assume that the
agent is interacting with an environment simulator, or an interface process
to the robotic resources and the sensors, with communication using our
Pedro communications server [106]. What these default programs do, and
the communication protocol they assume that must be supported by the
sensors/robotic resources interface process, is summarised in Chapter 5.

When the agent is to get its percepts and send its control actions using
either a MQTT [63] or a ROS [100] interface, the agent shell needs to be
configured by defining several of these optional procedures. Examples are
supplied with the TeleoR agent software.

The entire software package for developing and running multi-threaded
TeleoR agents comprises the following open source software:

- The QuLog interpreter that can be used to develop and debug the rules
  for an agent’s BeliefStore as described in Chapter 4.

- The TeleoR extension of the interpreter which has the configurable
  agent shell. This extension supports the incremental development,
testing and debugging of combined QuLog and TeleoR programs, as
described in Chapter 5.

- A compiler for generating stand-alone agents.

- The Pedro communications server [106] with its own documentation.

- Example QuLog, TeleoR, and Python simulation programs.

- HTML and PDF Documentation Files for QuLog and TeleoR.
3.2 A Bottle Collecting Agent Controlling a Mobile Robot

The robot is to be used to find, get hold of, and deliver an empty drink bottle of known size and colour to a drop area of known but different colour. The bottle and the drop area are on the floor. They have colours different from walls and other distant things that the robot might see. We assume that the space around the drop area in which bottles will be found is obstacle free.

The robot is mobile, able to independently turn its wheels as it moves forward. It has a forward facing camera with simple image processing software. It has a gripper with sensors that may be used to grasp a bottle that the robot is next to, and to then (mostly) keep hold of the bottle while it is delivered. The grasping may not always succeed, and the bottle may slip out of the gripper. Figure 3.2 depicts the environment.

![Figure 3.2: Bottle Collecting Task](image)

3.2.1 Sensor Percepts

By fairly simple image processing we can get sensor percepts
see(Th,Dir) near(Th,Dir) next_to(Th,Dir) close_to(Th,Dir)

where Th is bottle or drop and Dir is left, right or centre sent to the controlling robotic agent. These alternative possible values for the Th and Dir arguments will be formalised as enumerated types in the program. An enumerated type gives alternative symbol, more formally atom, values for instances of the type.

The difference between the four percept predicates is determined by the size of the area of Th’s colour in the camera image, and the direction is determined by the position of the centroid of this area in the camera image. centre is returned not when this is dead centre but within a small distance either side of dead centre equivalent to being in a direction + or - 10 degrees from the forward pointing direction of the camera and robot. left is returned when the middle of the coloured area is somewhere between -10 and -25, and right when it is somewhere between 10 and 25.

By calibration near(Th,Dir) is a generated percept from the camera image processing software only if Th is within a metre. This should be generated in addition to see(Th,Dir) which also holds.

close_to(Th,Dir) is generated when the robot is within 10 cms of Th, in addition to see(Th,Dir).

next_to(bottle,Dir) is generated only if the robot is within bottle grasping distance. next_to(drop,Dir) is generated only if the robot is within the drop area.

Using the touch sensors on the gripper, we get either of the percepts

gripper_open holding

but not both at the same time, sent to the robotic agent.

3.2.2 Robot actions

The robot has durative actions

move(Fs) turn(Dir,Ts)

where Fs and Ts are a forward speed and a turning speed respectively. It has discrete (aka ballistic) actions

open_gripper close_gripper

with names that tell us what they do. If move and turn are executed together the effect is to move the robot in an arc swerving in the Dir direction. These actions may be used by the robotic agent to control the mobile robot.

In addition there is the empty action which we denote using the empty tuple (). Nilsson used nil for this.

Durative actions are actions that continue until stopped and which may be modifiable whilst executing. For example, with move the speed argument can be changed modifying the forward speed of the robot.

36
Discrete actions are actions that cannot be modified before they naturally terminate after a short time. We assume a robot or a robotic resource will normally continue with a discrete action until it terminates and will ignore all other action instructions until it does terminate. If our robot was such that the gripper actions were prematurely terminable they need to be declared as durative.

We assume that there is a bottle within visual distance of the robot when its controlling TeleoR robotic agent starts executing the `collect_bottle` procedure given below. Before this happens the agent’s BeliefStore will have been updated with the latest percepts from the sensor interface program. We assume the robot starts with its gripper open. Its task is to find and deliver the bottle to the drop area. The goal of the task will have been achieved when the robot is next to the drop, next to a bottle, with its gripper open, so that the bottle may be picked up by a human and placed in a bin.

As mentioned in the previous chapter, following the Prolog convention all alphanumeric strings beginning with an upper case letter, or an underscore _, are variables unless singly quoted. In contrast, all sequences of letters, numbers or underscore _, that begin with a lower case letter, and any singly quoted sequence of any characters, are names of things. They can be used as names of TeleoR procedures, relation and function names and as constants, also called atoms.

### 3.2.3 The bottle collector’s TeleoR task program

```prolog
% Definition of enumerated type, dir
dir::= left | centre | right

% Another enumerated type, thing
thing::= bottle | drop

% The changing percept predicates + their argument types
percept holding, gripper_open, next_to:(thing,dir),
  close_to:(thing,dir), near:(thing,dir), see:(thing,dir)

% The discrete actions
discrete open_gripper, close_gripper

durative move:(num), turn:(dir,num)

% Top level procedure
collect_bottle{
  next_to(drop,_) & next_to(bottle,_) & gripper_open \(\Rightarrow\) () % Goal achieved, do nothing
  holding \(\Rightarrow\) deliver_bottle % holding sub-goal achieved, try for top
  true \(\Rightarrow\) get_bottle % Try to achieve holding sub-goal
}
```

37
get_bottle{
    holding ~> () % get_bottle goal achieved, do nothing
    next_to(bottle,centre) & gripper_open ~> close_gripper
        % above action should achieve holding
    gripper_open ~> get_next_to(bottle) % Try for next_to(bottle,centre)
    true ~> open_gripper % Action should achieve gripper_open
}

deliver_bottle{
    % Will only be active whilst holding inferable
    next_to(drop,_ ) & gripper_open ~> () % Goal achieved
    next_to(drop,_ ) ~> open_gripper % Try to achieve gripper_open
    true ~> get_next_to(drop) % Try to achieve next_to(drop,Dir), any Dir
}

get_next_to:(thing) ~> % Proc. type declaration, one argument type thing
get_next_to(Th){
    % to be used when Th is bottle or drop
    Th=bottle & next_to(bottle,centre) ~> () % Goal achieved for bottle
    Th=drop & next_to(drop,_) ~> () % Goal achieved for drop
    Th=bottle & next_to(bottle,Dir) ~> turn(Dir,0.2)
        % Turn slowly towards centre of bottle to achieve next_to(bottle,centre)
    close_to(Th,_) ~> approach(Th,0.2,0.2)
        % Now very near to Th, slow right down
    near(Th,_) ~> approach(Th,0.5,0.2) % near(Th,_) achieved, approach
        % more slowly to achieve close_to(Th,Dir)
    see(Th,_) ~> approach(Th,1.5,0.1) % see(Th,_) achieved, approach
        % Th quickly to achieve near(Th,)
    true ~> turn(left,0.5) % Th not in sight, turn hoping to see it
}

approach:(thing,num,num) ~> % Three arguments, a thing and 2 numbers
approach(Th,Fs,Ts){
    % Only active whilst see(Th,_) holds
    see(Th,centre) ~> move(Fs) % whilst see(Th,centre), move forward
    see(Th,Dir) ~> move(Fs),turn(Dir,Ts) % else swerve in Dir direction
}

Each new statement of the program and each new procedure definition begins at
the left end of a new line. The rules for each procedure are sandwiched between
{ ... } braces. The separation between the different rules of a procedure is implicit
in that after the ~> of each rule there is either a single procedure call or a comma
separated set of actions. When there is no following comma we have reached the
end of a rule. Each TeleoR rule must begin on a new line indented by at least one
space. A rule can be written over more than one line providing each extra line is also indented by at least one space.

The first three rules of get_next_to explicitly treat the two cases of the use of the procedure, (Th is bottle or drop). We could have used two procedures. One for getting the robot next to the centre of a bottle, the other for getting it next to the drop. However, they would have the last four rules in common and this procedure shows that there can be argument dependent goals (the different tests of the \( \O \) action rules). In Section 3.8, we shall give a higher order version of the procedure in which both a goal test relation, and the approach procedure to use, are arguments. It has a simpler structure.

Notice the type definitions (the ::= statements) at the beginning of the program. The TeleoR + QuLog type system will be covered in a little more detail in Chapter 4. For now suffice it to say that for this program it constrains dir values to atoms left, centre, right and thing values to atoms bottle, drop. It tells us for the percepts next_to, close, near, see which argument has which type of value. It also tells us the types of the arguments to the procedures get_next_to, approach, and of the durative actions move and turn.

When giving argument types we always separate the name of the code being typed from the tuple of argument types using ‘:’ with optional spaces before and after the ‘:’. The TeleoR translator and QuLog compiler check that all these type declarations are consistent with all uses in the program - it type checks the program. When a relation or procedure is called, given arguments do not have to be exactly of the declared type, they can be of a sub-type. For example an integer can be given where the argument type is num. The flexibility of being able to give sub-type values, and to return sub-type values from relation and function calls is discussed further in Chapter 4.

Finally a note about the use of the \(~>\) operator after the type declarations for the TeleoR procedures that have arguments. It tells us each is a type declaration for a TeleoR procedure. We use \(~>\) as this is the arrow operator of a TeleoR rule. When the procedure is defined in the same program file it need not be given. However all the type declarations may be given at the beginning of the program file along with the type definitions. In that case it is very useful to know that the declaration is for a TeleoR procedure.

When a type declaration is for a QuLog relation a postfix <= is used, for a QuLog action procedure a postfix \(\sim>>\) is used. Both can be dropped for relations and action procedures defined in the same file. The postfix <= does not need to be given when giving the type for a percept or belief. These are always relations.

### 3.3 How the Program Behaves

The goal of collect_bottle is to have the robot next to the drop, next to a bottle, gripper open. When collect_bottle is invoked the guards of the first two rules of the procedure will fail. The default last rule will be used, to get a bottle.

The goal of the get_bottle procedure is to have the robot holding a bottle, to
be tested by the gripper sensors interface returning the percept holding. A bottle is the only thing it will try to close its gripper on. The guards of the second or third rules of this procedure might now be inferable, in which case the first with an inferable guard will be fired and its action will be executed. But more likely its third or fourth rules will have to be fired. Initially it will be the third rule as we have assumed the robot starts with its gripper open. This calls \texttt{get\_next\_to(bottle)} so that the robot can achieve one of the sub-goals \texttt{next\_to(bottle,centre)} needed for it to attempt to get hold of a bottle using its \texttt{close\_gripper} second rule.

When \texttt{get\_next\_to(bottle)} is called, the procedure will evaluate its now partly instantiated rule guards (Th replaced by \texttt{bottle}), against the latest state of the \textit{BeliefStore}. The goal of the procedure is slightly different depending on the purpose, and the alternative goals are given as two () action rules. For getting next to the bottle, the goal is to get next to the centre of the bottle. For the drop, the goal is just to get next to the drop, whether the robot is pointing to the left, right or centre of the drop.

Condition \texttt{next\_to(bottle,centre)} of the first rule will not be inferable, else the rule that called the procedure would not have been fired by \texttt{get\_bottle}. The key condition of the third rule, \texttt{next\_to(bottle,Dir)} where Dir must be left or right, might be inferable. More likely the instantiated guards of the fourth, fifth or sixth rules will be the first rule guards to be inferable. If so, depending on how far away the seen bottle is, the robot will approach it very slowly, slowly or quickly. It switches to slower approaches as it gets nearer.

The last default rule is used when no bottle is in sight, it has the robot turn to the left looking for a bottle. On our assumption there is a bottle somewhere within view in the robot’s environment, one of

\texttt{see(bottle,Dir), near(bottle,Dir), next\_to(bottle,Dir)}

will eventually be received, depending on how far away the seen bottle is. Most likely it will be either \texttt{see(bottle,Dir)} or \texttt{near(bottle,Dir)}. Either the fifth or sixth rules will be fired. The robot will move towards the seen bottle slowing down as it gets closer.

Eventually a percept \texttt{next\_to(bottle,Dir)}, for some Dir, should be returned, if the bottle is not moved. The \texttt{bottle} goal of the called procedure has been achieved if Dir=centre. If not, say Dir=left, its third rule will be fired, turning the robot slowly to the left towards the centre of the bottle. The rule guards of the call will be re-tested when the \texttt{next\_to(bottle,left)} fact is replaced by one with a different Dir value. The slow turn action of the rule should soon result in the percept \texttt{next\_to(bottle,centre)} being received, when the bottle’s image is more or less in the centre of the camera picture. The turn action will be stopped. The goal for \texttt{get\_next\_to(bottle)} will be achieved, allowing the procedure to fire its first () action rule.

But even before \texttt{get\_next\_to(bottle)} gets to fire this goal achieved rule, \texttt{get\_bottle} will have re-tested its rule guards triggered by the addition of the latest \texttt{next\_to} fact to the \textit{BeliefStore}. It will have re-tested the rules when the
percept next_to(bottle, left) was added with no change of the fired rule. The second re-test when next_to(bottle, centre) is added will cause a switch to firing the second rule of this parent procedure call, and a termination of the get_next_to call. This is because we have assumed the robot started with its gripper open, so the second rule of get_bottle will be fired to close the gripper in an attempt to get hold of the bottle. The call to get_next_to(bottle) will be terminated before it can respond to the new percepts and fire its first () action rule.

Let us suppose that this attempt to get hold of the bottle is successful, and the holding percept is received from the gripper touch sensors on the next percepts update when it also no longer gets the percept gripper_open. Again, before get_bottle can fire its initial goal achieved rule with the () action, the guards of rules of the collect_bottle call will be re-tested. Its second rule will be fired, causing the get_bottle procedure to be terminated, and the procedure deliver_bottle to be called.

3.4 Recovering from Setbacks

deliver_bottle will remain active only while holding remains inferable, which in this case is while the holding percept is in the BeliefStore. Its goal is to deliver the held bottle to the drop area. Should the robot accidentally drop the bottle on route to the drop area, the next percepts update will not include holding, and so the holding condition for deliver_bottle will no longer be inferable. collect_bottle will re-test its rules because the fact has been deleted from the BeliefStore, terminating the deliver_bottle call and firing its default rule, invoking get_bottle again.

The dropped bottle will probably still be in view and near, so get_bottle should have less to do. One difference is that it must first fire its last default rule to re-open the gripper before it tries to close it around the bottle again. This is because its second and third rule both have a gripper_open condition which will not be in the BeliefStore after the bottle has been dropped. This belief must be re-achieved by firing the fourth rule of get_bottle and getting back the sensor percept gripper_open from the touch sensors when the gripper has been successfully opened. If the gripper is stuck, the percept will never arrive and the control program and the robot will suspend. This is a fault situation from which it would be useful to have a recovery mechanism, or at least a fault report generated. One of the extra forms of actions of TeleoR discussed in Section 10.3 allows for an attempt at a recovery action and, if that fails, generates a fault report.

Let us suppose that the gripper does open and the percept gripper_open, say paired with near(bottle, right), arrives within a second or two. Suppose the last actions of deliver_bottle were move(0.5), turn(left, 0.2), because it was at that time approaching the drop slowly. get_bottle will fire its third rule calling get_next_to(bottle) again, which will fire its fifth rule with action approach(bottle, 0.5, 0.2). All this will happen within a few milliseconds of the arrival of the new percepts. Because the bottle is on the right side
of the robot’s field of view the second approach rule will be fired with actions move(0.5), turn(right,0.2). There will be a smooth transition to the two new actions. move(0.5) will continue unchanged but turn(left,0.2) will modified to turn(right,0.2) causing the robot to now start swerving slowly to the right towards the dropped bottle, rather than to the left towards the drop.

Let us now suppose that eventually the percept next_to(bottle,centre) is received but the second attempt to get hold of the bottle by re-closing the gripper fails. After the next percepts update neither holding nor gripper_open will be inferable. This indicates failure of the close_gripper action to achieve its holding goal.¹

next_to(bottle,centre) will still be in the BeliefStore, if the robot is still next to the centre of the bottle. get_bottle will again fire its fourth rule to re-open the gripper. On the next percepts update if gripper_open is included, its second rule will be fired again in a second attempt to get hold of the bottle. Suppose this again fails, but this time the bottle is pushed away from the robot. After the next percepts update next_to(bottle,centre) will no longer in the BeliefStore. get_bottle will again have to fire its fourth rule to get the gripper open again, and then its third rule to get the robot back to being next to the centre of the bottle. It will then try again to get hold of the bottle by closing the gripper. This cycle will continue until holding is returned with some percepts update.

There is another way that the close_gripper action can fail to achieve the holding sub-goal. The gripper just might not close, or it might start to close then get stuck so that the holding percept is not returned on the next percepts update and neither is gripper_open. If the latter, the program will again recover because gripper_open will no longer be inferable and the fourth rule will be used to re-open the gripper and try again to get hold of the bottle. However, if the gripper does not even start to close, as it stands our get_bottle procedure will just hang. It will get percepts signalling gripper_open still holds, so it will continue to want to have the second rule as its fired rule. It will not re-attempt to close its gripper because it can only re-fire a rule if a percepts update results in a different action, which cannot happen in this case. This is a fault similar to the one mentioned above should the gripper fail to open. Both will be discussed again in Section 10.3.

We leave the reader to understand what deliver_bottle does from the comments in the procedure. It is quite straightforward.

3.4.1 Repeating the bottle collection

When it has released the bottle in the drop area the robot will sit there looking at the delivered bottle doing nothing. That is what the () action means. If a human removes the bottle, the goal of the collect_bottle procedure is no longer

¹ This is based on the assumption that the rate of sampling the sensors is not so fast that the agent will perceive that its gripper is no longer open before it gets the holding percept. If that should happen our poor robot would oscillate for ever opening and closing the gripper but never giving it time to get hold of the bottle. We would have to slow down the rate of sampling.
achieved and the procedure will begin to execute actions again, looking for, getting hold of and delivering another bottle if there is one within site as the robot turns around at the drop. If there are no other bottles it will just keep turning to the left, having fired the default rule of \texttt{get\_next\_to(bottle)}, hoping that a bottle will eventually be put down on the floor and be in sight. There is an issue about battery re-charging that we will deal with in section 10.5.

Later we will allow the action of the first rule of the top level procedure of some agent program to have an action that does not maintain the task goal. We shall find it useful to have an action that does not maintain the guard condition of the first rule but instead facilitates the re-achieving of that guard for repetitive task behaviour.

### 3.5 Universal Procedures, TelegoR Program Development and Correctness

All procedures satisfy what Nilsson calls the \textit{regression property}. The first rule of each procedure has a guard \( K_1 \) and the empty action \( () \), or another action that will not undo \( K_1 \) once achieved (such as the first rule of the \textit{approach} procedure). \( K_1 \) is the goal of the procedure. Every other rule is such that when the guard \( K_j \) of the rule is inferable, and no guard of an earlier rule is inferable, the action \( A_j \) of the rule \textit{should} result in the guard \( K_i, i < j \), of some earlier rule becoming inferable from some future state of the \textit{BeliefStore} updated with new percepts. In the case that \( A_j \) is a durative action or \textit{TelegoR} call, it \textit{should eventually} result in the guard \( K_j \) of the earlier rule becoming inferable. \( K_j \) is the \textit{regression} of \( K_i \) through \( A_j \). This is the goal seeking behaviour of the procedure.

For example, if

\[
\text{next\_to(bottle,centre) \& gripper\_open}
\]

is inferable and \texttt{close\_gripper} is executed then \texttt{holding} should become inferable, Similarly,

\[
\text{next\_to(bottle,\_) \& next\_to(drop,\_) \& gripper\_open}
\]

should eventually become inferable when \texttt{holding} is inferable and \texttt{deliver} is called. That this is the case can be checked by showing that the goal of \texttt{deliver}, the test of its first rule, implies the above on the assumption that \texttt{holding} will be true immediately before the goal of the procedure is achieved. That this will be the case follows from the assumption that the \texttt{deliver} procedure will be active only if the guard \texttt{holding} is inferable. So informally (or formally) demonstrating that each procedure satisfies the regression property is a proof of \textit{partial correctness} of the program, and that when a rule is fired some progress should be made towards the goal of the initial procedure call by the action of the rule.

Whether we will always be able to make progress is another matter. It requires that the procedures satisfy another condition - always having a rule that can be fired
whilst the procedure is active. For all the procedures except approach this is trivially the case as all are have default last rules with guard true. The approach(Th,...) procedure always has a rule that can be fired on the assumption that while it is active the robot can see Th, i.e. it is only called from inside get_next_to. This is therefore not a procedure that can be used as a start TeleoR procedure. It is purely an auxiliary procedure with see(Th,_) being inferable its condition for use. This will be the case when called from either rule in get_next_to, as see is implied by near. So this is an informal argument, that providing a bottle will be in view when the robot bottle collector starts, and it is not purposely and repeatedly thwarted by having the bottle moved, and the primitive actions do not fail, a bottle will be collected and delivered to the drop area.

Procedures satisfying the regression property and such that there is always a rule that can be fired are universal procedures for the goal of the procedure. With assumptions about non-failure of primitive actions and not being repeatedly thwarted, and that any sub-procedures they call are universal, they are totally correct procedures for their goals.

3.5.1 Elasticity of universal procedures

Universal TeleoR and TR procedures immediately respond to opportunities and automatically recover from setbacks. The procedure get_next_to(Th) will automatically jump to the highest rule with an inferable condition after the turn action. If, when the robot is approaching a bottle, some other agent (say a human) moves the bottle to be near to the robot, it will immediately switch to executing the approach procedure with a slower move speed. Conversely, if that other agent then moves the bottle to be out of sight of the robot, the procedure will automatically drop to firing the last default rule should no other bottle be in sight.

We have also discussed how the program will cope with actions and procedure calls that do not achieve their intended goal, e.g. close_gripper not achieving the intended goal holding and the deliver_bottle procedure failing to deliver a bottle because it drops it on the way.

3.5.2 Regression methodology for developing a TR program

The desired regression + completeness property suggests a methodology for writing TR programs. We start with the goal G environment situation that we want the procedure P1 to achieve, assuming that environment situation I initially holds and that some environment property H will hold throughout the execution of a call to P1. I and H may be just true, and will usually be true when we start developing a program.

For our bottle collection program we may have taken I and H to be the assumptions that: the environment is a flat open space, there is at least one bottle close enough to the robot so that it can be seen by the robots camera if the robot is orientated towards the bottle, and that there are no non-bottles that can be obstructions, and that the drop is a painted patch on the floor that can be seen...
from the position of every bottle. \( G \) could be the goal that there is a bottle in the drop area with the robot next to it with its grippers open.

We must be able to reduce \( G \) to a query \( K_1 \) to percepts from our available sensors such that \( K_1 \) and \( H \) imply \( G \). We can add definitions of auxiliary relations such as \( \text{at} \) to allow succinct expression of \( K_1 \). However, some of these definitions may need to be recursive, meaning we could not formulate \( K_1 \) without using them. If \( G \) cannot be reduced to such a query that together with \( H \) implies \( G \), we may need auxiliary sensors, or better sense data analysis (better visual processing) to give us richer percept facts.

We then work backwards from \( K_1 \) trying to find a condition \( K_2 \) 'close' to \( K_1 \) such that there is a set of robotic actions, or an existing TR procedure call, or a call to a new TR procedure \( P_2 \) that we can construct using the regression methodology. The goal for \( P_2 \) is \( K_1 \) and we can assume that \( H \& K_2 \& \text{not} \ K_1 \) holds throughout the call \( P_2 \). Whatever action \( A_1 \) we find, it must be such that \( K_2 \) is the regression of \( K_1 \) through \( A_1 \). The extra condition \( \text{not} \ K_1 \) can be assumed because if the second rule is fired it must be the case that the evaluator has failed to infer \( K_2 \). Providing our we have complete information about the relations of the BeliefStore, and the if\( f \) completion of the relation definitions and current percept facts is, consistent, not \( K_1 \) follows by the negation as failure [29] meta-rule. The if\( f \) completion of the BeliefStore will be consistent providing it is structured [3], in effect that there are no recursive definitions that go through a \( \text{not} \). In fact, the negation as failure rule is assumed when we interpret \( \text{not} \) as logical negation in a relation rule of the BeliefStore, or in a guard of a TR rule.

We now repeat this step to find a \( K_3 \) 'close' to \( K_2 \) or a possible alternative' to \( K_2 \) 'close' to \( K_1 \), and an action \( A_3 \). If \( K_3 \) is 'close' to \( K_2 \) we are extending one goal regression branch. If it is an 'alternative' to \( K_2 \) the we are staring another regression branch.

We may again have to construct another procedure \( P_3 \) a call to which will give us the required regression from either \( K_2 \) or \( K_1 \). If the regression is from \( K_2 \) (the \( P_3 \) call should eventually achieve \( K_2 \), we can assume that \( H \& K_3 \& \text{not} \ K_1 \& \text{not} \ K_2 \) holds throughout the call to \( P_3 \). If the regression is from \( K_1 \), we can only assume this condition holds providing we make the new rule with guard \( K_3 \) the third rule. If we put it as the second rule it is the weaker condition \( H \& K_3 \& \text{not} \ K_1 \) that holds through the call to \( P_3 \). But the stronger condition \( H \& K_3 \& \text{not} \ K_1 \& \text{not} \ K_3 \) will now hold throughout the call to \( P_2 \), which is now the action of the third rule. This may enable us to simplify the procedure \( P_2 \). At least it must be re-considered.

We continue this regression development until we have a sequence of rules for \( P_1 \) that 'covers' all the environment situations that might occur in some transformation of situation \( I \) to situation \( G \) assuming \( H \) holds throughout. That is, where \( P_1 \) has a sequence of \( n \) rules with guards \( K_1 \) to \( K_n \), the disjunction of the alternatives,

\[
K_1 \quad K_2 \& \text{not} \ K_1 \quad \ldots \quad K_n \& \text{not} \ K_{n-1} \& \ldots \& \text{not} \ K_1
\]

is always true. Often the simpler disjunction of \( K_1 \), \( K_2 \),...,\( K_n \) is always true.
As Nilsson says on page 8 of [90]

*The backward-from-the-goal approach to writing T-R programs makes them relatively easy to write and understand, as experience has shown.*

[58] describes a software tool for helping with the construction of TR programs for autonomic systems.

### 3.5.3 Learning and planner generation of standard TeleoR programs

This regression methodology is very like backwards from the goal planning with recursive sub-planning. The use of a modified STRIPS [48] planner for generating TR programs has been explored in [13], [12], coupled with learning of STRIPS like descriptions of the resource actions, both discrete and durative, using supervised learning.

Another approach is to learn a simple TeleoR program, one in which the guards only query the percepts and all the actions are resource actions, by mimicking human control of the robotic resources to achieve some goal state. [90] describes this approach and several other approaches for learning TR programs, giving experimental results.

[18] describes an alternative approach to developing TR programs to achieve a goal. Their approach also deals with programs for co-operating communicating agents, such as those that can be programmed in TeleoR.

Other approaches to learning similar goal seeking reactive programs are explored in [26]. [58] describes a software tool for helping with the construction of TR programs for autonomic systems.

### 3.5.4 Verification of TeleoR programs

Whether TR programs are constructed using techniques such as those given above or are hand-coded, it is important to be able to prove properties of such programs. In [46], an interval-based real-time temporal logic is used both to specify the properties of actions and to specify the semantics of TR programs in a rely-guarantee style. The rely conditions give constraints on the environment and the guarantee condition says what the program will do if the rely conditions are satisfied.

One of the main results of the paper is the Progression Theorem which, informally, says that, as long as the environment satisfies the rely conditions, then the goal of program will be achieved within a given time. The example program used in the paper is a can collector that our bottle collector program is based on. Essentially, for this program, the Progression Theorem says that as long as there is a can on the table and the environment does not move or remove the can, then the robot will deliver a can to the depot within a given time. The time is computed as the longest time required to find a can, get to a can and then deliver it to the
depot. This calculation is based on the size of the table, the position of the depot and the specification of the actions.

In the appendix of the paper, Prolog code is given for a tool that takes a TR program and the goal to be achieved and applies the Progression Theorem repeatedly, generating and simplifying a collection of formulae that can be used to construct the necessary rely conditions. It was felt that this tool was necessary as manually applying the Progression Theorem to non-trivial TR programs is very error-prone.

3.6 QuLog for Defined Percepts

The guards of the TeleoR rules are not restricted to percepts directly linked to the sensors of the agent. They can use predicates defined by a set of facts and simple rules in our typed logic based language QuLog, more fully described in Chapter 4. These rules for the defined percepts directly, or indirectly via other defined percepts, depend upon the sensor percepts.

For example, the predicates next_to, close_bottle, near and see can either be sensor percept predicates, or defined in terms of a more low level sensor percept predicate see_patch(Col,Size,Dir). This gives the colour Col of a dense patch of percepts in the camera image, a measure of its size, Size, and the relative direction Dir, which is left, right or centre, of the seen patch in the image on the forward facing camera. The higher level percepts would then be defined by implication rules such as those below. For historical reasons rules and facts defining relations are also called clauses, a term we shall sometimes use.

```
col::= green | blue | red | yellow | brown
% new enumerated type of the possible patch colours
percept see_patch:(colour,num,dir)

next_to: (?thing,?dir)<= % Declaration for relation, signalled by the <=
next_to(Th,Dir) <=
    see_patch(Col,Size,Dir) & colour(Th,Col) &
    next_to_size(Th,NTs) & Size>NTs

close_to: (?thing,?dir)<=
close_to(Th,Dir) <=
    see_patch(Col,Size,Dir) & colour(Th,Col) &
    close_to_size(Th,CS) & Size>CS

near_to: (?thing,?dir)<=
near_to(Th,Dir) <=
    see_patch(Col,Size,Dir) & colour(Th,Col) &
    near_to_size(Th,NS) & Size>NS
```

47
see: (?thing, ?dir)<=  
see(Th, Dir) <=  
   see_patch(Col, Size, Dir) & colour(Th, Col) &  
   see_size(Th, SS) & Size > SS

colour: (?thing, ?col)<=  
colour(bottle, green)  
colour(drop, blue)

next_to_size: (?thing, ?num)  
next_to_size(bottle, 200)  
next_to_size(drop, 450)  
   ...  
see_size: (?thing, ?num)  
see_size(bottle, 50)  
see_size(drop, 110)  
.

The <= in a clause should be read as ‘if’. The clauses are implicitly quantified with respect to all their variables - the names beginning with an upper case letter. The types in the declarations for all but the percept relations are annotated with a prefix ?. This indicates that the relations can be use both to test or to find instances. Percept types do not need to be annotated as they can always be use to test or find instances.

Note that the rules are given over multiple lines. This can be done because the extra lines are indented. New rules, type definitions and declarations, and TeleoR procedures must always start at the extreme left of a new line. Indenting means a continuation of the most recent syntactic ‘thing’ that started at the extreme left of some line above.

The rules can be used each time there is a guard condition for a relation defined by rules to reduce the condition to a query to the lower level see_patch percept. However, it is better to immediately infer and place in the BeliefStore all the inferable facts for these still quite low level percept ‘interpretation’ predicates. This can be done in the agent’s percept handling thread immediately after it has updated the BeliefStore with new see_patch percepts.

In Chapter 9 we shall see that this anticipatory inferring and caching of these agent-specific interpretation facts will also help to improve the efficiency of our TeleoR procedure evaluation by saving some unnecessary re-testing of rule guards.

2Readers familiar with Prolog will notice there is no full stop at the end of each clause. This can be given, followed by a space, tab or newline character, but is ignored. The beginning of the next clause at the left end of a fresh line is what terminates the preceding one.
We refer to them as agent-specific because, for another TeleoR control program, we might well have rules that interpret a green patch of colour of a certain size as something quite different from a bottle. We might want to interpret it as an obstacle to be avoided.

To indicate that we want these rules to be used to infer and remember all instances of the relations they define immediately after each BeliefStore percepts update, we give the rules a percept type declaration instead of a normal relation type declaration. There are restrictions on the form of rules that can be used to define these auxiliary percept relations that we shall explain in Section 9.2.4.

If we want to allow the colour facts to be updated as a result of a message sent to the agent whilst it is executing the collection program, we give it a belief type declaration.

To allow remembering inference of instances of the vision perceptual relations see, near_to etc., and to allow updating of the colour facts to allow changing the colour of the bottles that are collected, we would add the following type definition and declarations to the program.

```
col::= green | blue | red | yellow | brown

percept see_patch:(colour,num,dir), holding, gripper_open
    next_to:(thing,dir), close_to:(thing,dir),
    near_to:(thing,dir), see:(thing,dir)

belief colour:(thing,col), next_to_size:(thing,num),
    close_to_size:(thing,num), near_to_size:(thing,num),
    see_size:(thing,num)

% Rules for next_to, close_to, near_to, see
```

As with percept declarations we do not need to annotate the argument types of belief declarations. They are always defined by a sequence of changeable facts and so are always test or find for each argument.

In general, an agent’s BeliefStore will contain a whole hierarchy of rules, progressively defining higher level concepts in terms of lower level concepts, but with most ultimately rooted in a set of directly perceived percepts such as see_patch. Some facts may contain fixed information about the environment and the resources, such as the connectivity of rooms in a building, the normal action capabilities of resources, and locations of fixed resources and sensors. Mostly it will be better to use the rules for the higher level defined relations to infer instances of the relations they define as an when used in a guard evaluation. Such relations are not declared to be percept relations.
Making use of an extra relation definition, we could also replace the guard of the first rule of \texttt{collect bottle} by the single condition \texttt{delivered}, using the rule

```
percept delivered
delivered <=
    next_to(drop,_) & next_to(bottle,_) & gripper_open
```

This is also declared to be a \texttt{percept} relation so that the rule is used just once in the percepts handler and either the fact \texttt{delivered} is added to the BeliefStore, or it is not. Absence of the fact means that the guard of the first rule of \texttt{collect bottle} will not be inferable.

Using another defined percept relation we can also replace the first two rules of \texttt{get next to} by one rule.

```
3.6.1 Using a function definition

We slow our robot down in stages at particular distances from its target. Another approach, especially when the robot is getting close, is to slow it down linearly depending on its distance from the target so that as it reaches the target is slows down to almost zero forward speed. This is particularly useful when the robot is approaching a bottle to prevent the bottle being pushed away by the forward momentum of the robot.

A straightforward way to do this is to use a function \texttt{speed:(num)->num} that uses the distance from the target to compute the forward speed. Using such a function we can change the fourth rule of \texttt{get next to} making use of a modified \texttt{close to} relation that returns the size of the colour patch of the seen thing, as well as its relative direction.

```
close_to(Th,_,Size) ~> approach(Th,speed(Th,Dist),0.2)
```
Now the close_to rule of the get_next_to TeleoR procedure will be re-fired each time the percepts are updated in such a way that the computed distance to Th has changed. The types of function arguments do not need to be annotated as they must always be given, like the arguments of TeleoR procedures.

The speed function returns a speed of 0.25 when the robot just becomes close to the approached bottle, when the size of its colour blob is 150. This linearly reduces to 0.05 when the robot is next to the bottle. At that point the next_to rule of the get_next_to procedure will fire, stopping the forward motion of the robot. We do not need to alter the correctional turn speed between being close and being next to the target. The speed when approaching a drop does not change until the robot is next to the drop.

3.6.2 Updating an agent’s dynamic beliefs

If we want the robot to collect blue bottles as well as green bottles, we do not need to re-program the robot’s controlling agent. We can just send the agent a tell message containing colour(bottle,blue) whilst it is controlling the robot collecting green bottles. If we have anticipated this, the agent will add colour(bottle,blue) as a new fact belief to its BeliefStore, and immediately afterwards will perceive blue patches of a certain size as blue bottles that are near in a particular direction. This is assuming that the different coloured bottles are the same size.

Handling such a message is the role of the agent-specific handle_message QuLog action procedure that was mentioned in Section 3.1. Suppose we want to be able to send messages to the bottle collector that result in new facts being added and old facts removed for its dynamic colour beliefs, and for any other relation declared as a belief relation. Giving a handle_message. QuLog action procedure is one of the agent shell configuration options. If defined, it will be called by the agent’s message handler each time a message is received passing in the message term received and the Pedro handle of the agent or other Pedro connected process that sent the message. This Pedro handle is always of the form Nm@HostName where Nm is the Pedro registered names of the agent or process and Hostname is a single quoted domain name such as ‘zeus.itee.uq.edu.au’ or a single quoted IP address. Any reply to a received message must be sent to messages:Nm@Host, the message handler of the sender.
Our handle_message procedure can be defined using three action rules.

```
handle_message_:(term?,process_handle)~>>
handle_message_(rem(colour(Th,C)),Nm@Host)::thing(Th)&col(C) ~>>
   remember colour(Th,C);
   remembered(colour(Th,C)) to Nm@Host
   % adds colour(Th,C) if not in the BeliefStore, sends a response

handle_message_(unrem(colour(Th,C)),Nm@Host)::thing(Th)&col(C) ~>>
   forget colour(Th,C);
   forgotten(colour(Th,C)) to messages:Nm@Host
   % removes colour(Th,C) if in the BeliefStore, sends a response
handle_message_(_,_)     % Ignore all other messages
```

The `::` can be read as *such that* and the `~>>` as *do*. If the condition between the `::` and the `~>>` of a QuLog action rule is inferable then *no later rule should be tried*. It is a *commit* test for a QuLog action rule similar to the guard of a TeleoR rule.

For a call to a QuLog action procedure its rules are tested in the order given. If the call *unifies* with the head of some rule - technically there is a substitution for the variables of the rule head, and maybe also for variables in arguments of the call, that makes the rule head and the call identical - and, if present, the commit test of the rule is inferable, the call is evaluated using *just* that rule. If there is no commit test success of the unification is sufficient for commitment to the rule.

`term` is the most general type of a data value. All other data types such as `int`, `num`, `string` and any enumerated type, is a sub-type of `term`. Received messages always have to be handled as type `term` values with use of the QuLog runtime type checking primitives to make sure the received message is of a type that we want to handle. `term` has a postfix `?` mode annotation in the type declaration to indicate that the `term` argument may be a message term containing variables and that it may still contain variables after the call to `handle_message` terminates.

Notice that we have used a relation name `colour` as the functor of the term `colour(Th,C)`, as an argument of the `rem` and `unrem` terms that are received message patterns. As in Prolog, this overloading of relation names as data constructors is allowed. However, in QuLog this transition between use as a data constructor, and use as a relation name, must always be done in a type safe way.

For example, if the first two rules did not have the runtime type tests `thing(Th)` & `col(C)`, they would be rejected by the QuLog compiler because the primitives `remember` and `forget` must be known to be adding and trying to removing type correct facts and `rem(colour(Th,C))` is a data term that has come from another agent or process. This means that the values given to Th and C, by the match with the call to `handle_message` from the generic message handler of the agent, are not
guaranteed to be of the correct type. They may, indeed, be variables. thing and col, both enumerated type names, are here being used as type checking primitives. All defined types can be used as runtime type checking primitives in this way.

**remember** and **forget** are QuLog action primitives for updating the belief facts in the BeliefStore. **remember** maps a ground term that ‘names’ a type correct belief fact into a new fact of the BeliefStore. It adds it after all the existing facts, but only if it is not already in the BeliefStore. **forget** searches for a fact for the belief relation ‘named’ by the functor of its argument term \( B \), that matches \( B \). It searches the facts in before/after order. The first one it finds that matches, is removed. If there is no matching fact it does nothing. \( B \) need not be ground but where arguments are given they must be type correct for the belief relation. **remember** and **forget**, and other BeliefStore updaters are more fully described in Section 8.4. They are also used by the percepts handler for updating percept facts.

This **handle_message_** procedure enables the agent to be told to collect bottles of colour \( C \) as well as any other coloured bottles it is already collecting. It can also be told to ignore, from now on, bottles of a colour that it might have been collecting. For example, if it is sent **unrem**(colour(bottle,green)), it will stop collecting green bottles.

### 3.7 Standard TeleoR Procedure Syntax

Now that we have looked at several standard TeleoR procedures, we can give a more general description of their syntax. We use a modification of the syntax to that Nilsson gave in [91] for TR procedures. One difference is that when a procedure has arguments we need a separate type declaration which may require the support of some type definitions. In addition we put \{\ldots\} brackets around the sequence of rules of each procedure. We also use the character combination \( <> \) as the rule operator instead of Nilsson’s \( ! \) which we use for function rules in QuLog. We use \& to separate conditions in the rule guard as that is what is used in QuLog.

The overall form of a TeleoR standard procedure is

\[
p: (t_1, \ldots, t_k) <> \{ \\
\quad K_1 <> A_1 \\
\quad \& \\
\quad \vdots \\
\quad \& \\
\quad K_n <> A_n \\
\} 
\]

\( p \) must be an unquoted atom. It must be different from the name of: any other procedure, any discrete or durative action of the program, any percept or dynamic belief, any BeliefStore defined predicate, any QuLog action procedure name. When the procedure source code appears in the same program file as the type declaration the postfix \( <> \) on the declaration need not be given.
The $t_i$ are the types of the procedure’s $k$ parameters. Each $t_i$ may be a primitive type such as `int` (integer), `num` (any number), `atom` (an unquoted sequence of alphanumeric characters beginning with a lower case letter, or a singly quoted sequence of any characters or spaces), or a program defined type such as `thing`, or a type expression telling us the required argument types for program code. This could be the type expression of another `TeleoR` procedure that might be given to this procedure to use for a procedure call action. That a `TeleoR` procedure can be passed code when called is a difference from TR procedures.

### 3.7.1 `TeleoR` procedures versus `QuLog` action procedures

The `TeleoR` procedures use the agent’s robotic resources to do things in the environment of the robotic agent. The `QuLog` action procedures do internal agent things, such as: `BeliefStore` updates, parameterisations with other agents (in order to get updates for the `BeliefStore` or to update the `BeliefStores` of other agents), and spawning of new agent threads. None of these `QuLog` actions directly use what we view as robotic resources. `QuLog` actions are things that can indirectly affect subsequent use of robotic resources.

When we discuss extensions of standard `TeleoR` rule actions in Chapter 10, we shall see that a call to a `QuLog` action procedure can be executed concurrently with robotic resource actions. An agent can be updating its beliefs, sending messages to other agents, executing several robotic resource actions, all at the same time.

### 3.7.2 `TeleoR` rule guards - restricted `QuLog` queries

Each guard $K_i$ is a restricted `QuLog` query comprising a single condition or `not` (negated) condition, or an `&` conjunction of conditions and `not` conditions, with restrictions on the `not` conditions. These restrictions are to ensure that any variable a negated condition contains, other than a procedure parameter and the ‘don’t care’ anonymous variable $\_\_$ will have a value when the guard is checked by the `QuLog` query evaluator. The restrictions will be explained in Section 4 when we describe `QuLog`, as will the guard query evaluation process. We shall see in that chapter that we can indirectly include quite complex conditions in guards using `BeliefStore` implication rules.

The guard can also use a restricted set of pre-defined predicates such as unification `=` and expression value comparisons, such as `<` and `>`. All the allowed primitives are described in the `QuLog` documentation with the `QuLog` and `TeleoR` agent software from [105].

If a local variable only appears once in the guard and does not appear in the action, it is good practice to use an anonymous variable such as $\_\_$. Remember, each occurrence of $\_\_$ stands for a different but unnamed variable.
3.7.3 Rule actions of standard TeleoR procedures

Each action $A_i$ is either a single call to a TeleoR procedure, or it comprises one or more primitive actions, separated by commas. ( ) is used to indicate doing nothing. The primitive actions are to be executed in parallel.

Allowing parallel execution of primitive actions is an extension of TR procedures as described in [89],[91], although parallel primitive actions and even concurrent TR procedure calls are allowed in an earlier research report [88]. The primitive actions are either discrete (aka ballistic) or they are durative. Discrete actions naturally terminate usually after a short time, for example grasp or open for grippers, or beep for a horn. They are also called ballistic because they cannot be prematurely terminated once started. A robotic resource can execute compatible discrete and durative actions concurrently. If it is important not to execute another action until a discrete action has terminated then we had best have a percept that will indicate that this has happened, such as gripper_open, before attempting another action. If we had a robot with a gripper for which the close and open actions were terminable before they naturally finished, then these would be durative actions even though short lived.

Durative actions may naturally terminate, for example a pickup action for a robot arm as will be used in the block tower building program of Chapter 6. Often they are open ended and continue unless explicitly stopped, or modified. For example, move(1.5) being modified to move(0.5) when a robot gets close to the thing it is approaching, or stopped and replaced by a turn action if it loses sight of the thing.

This is the key characteristic of durative actions. They can be stopped at any time even if they would eventually terminate. If parameterised, they can be modified whilst executing. The ability to modify durative actions allows for a smooth transition from one set of primitive actions to another. A durative primitive action is terminated only if a rule is fired which has primitive actions that neither include the durative action nor a parameter modification of the action.

3.8 A Higher Order TeleoR Procedure

To illustrate the use of higher order TeleoR we will show how the procedure for getting the robot next to either a bottle or the drop may be generalised so that its approach activity, which is slightly different for a bottle or the drop, can be give as a procedure argument. This allows a different approach method to be used for a bottle and the drop. Also, by defining an auxiliary reached relation, we can use one goal achieved rule instead of two. The current get_next_to procedure is
The higher order version is

```
get_next_to: (thing, (thing,num,num)~>)
get_next_to(Th,Approach){
    at(Th) ~> ()
    close_to(Th,_,Size) ~> Approach(Th,speed(Th,Size),0.2)
    near(Th,_) ~> Approach(Th,0.5,0.2)
    see(Th,_) ~> Approach(Th,1.5,0.1)
    true ~> turn(left,0.5)
}
```

Notice that the procedure parameter `Approach` is used as if it were a program defined `TeleoR` procedure. This is indeed what it will be when the procedure is called. It will be called passing in the `name` of a program defined `TeleoR` procedure that takes a `thing` and two number values as arguments. The type expression for this higher order arguments is annotated with `~>` to indicate that it is a `TeleoR` procedure. Its this annotation that tells us that it is a 'code' argument, not a data arguments. Without the `~>`, `(thing,num,num)` would indicate that the third argument was a 3-tuple of data values.

It is easy to remember what the annotation we must attach to the argument type tuple of a code argument as it is always the operator of the rules that define that type of 'code'. So, `<=` annotates the argument type tuple for a relation argument and a type expression of the form `(t_1,\ldots,t_k)`->`t` is used for a function type. As examples, `(int,atom)`< = is the type expression for a relation over pairs of integers and atoms, that can be used to generate or test value pairs, and `(num,num)`->`num` is the type expression for a function from a pair of numbers to a number. When the code argument has no arguments we use the empty tuple () for its argument type tuple.
For approaching the drop in the `deliver_bottle` procedure, the `Approach` argument is just the original `approach` procedure. The default rule of `deliver` becomes

```
true ~> get_next_to(drop, approach)
```

For approaching a seen bottle in `get_bottle`, we need to change its third TeleoR rule and define a new `approach_centre` procedure.

```
gripper_open ~> get_next_to(bottle, approach_centre)

approach_centre: (thing, num, num)
approach_centre(Th, Fs, Ts) {
  next_to(Th, centre) ~> ()
  next_to(Th, Dir) ~> turn(Dir, 0, 2)  % Dir must be left or right
  true ~> approach(Th, Fs, Ts)  % Default is to call the standard approach
}
```

When the robot is being controlled by a call to the `approach_centre` procedure to move very slowly towards a bottle, as soon as it is next to the bottle the second rule will fire, unless the bottle is in the centre of the robot’s field of view. The second rule will turn the robot so that it faces the bottle head on. Before, this turning was done in the `get_next_to` procedure. If the robot happens to reach the bottle and have it in its central field of view, the parent call `get_next_to(bottle, goal, approach_centre)` will fire its first rule, terminating the `approach_centre` call. The default behaviour of the new procedure is the original `approach` behaviour to get the robot next to `Th`.
Chapter 4

*BeliefStore* Facts and Rules in Declarative QuLog
In Chapter 6 we will give a standard TeleoR program which is a modification of Nilsson’s block tower building program of [91]. As the program makes essential use of BeliefStore rules, even recursive rules, we start by describing the use of the QuLog interpreter to develop and test a deductive BeliefStore independently of its use by the TeleoR program. We explain how it evaluates interpreter entered queries and TeleoR rule guards.

QuLog has similarities with Prolog, but it also has very important differences. Because we want to guarantee that all TeleoR guard evaluations generated correctly typed values for the action of each rule, a relation used in the BeliefStore must be typed and moded. By moded we mean that we declare which arguments must be given for a relation query and which may have values generated by a use of its defining rules. So we also explain in more detail the type system and the mode annotations for relation type declarations.

Step by step we will develop and test the type definitions and BeliefStore facts and rules for our tower building program. The definitions use lists so we will also introduce some of the list processing features of QuLog such as pattern match decomposition, list and set comprehension expressions. Sets are aggregates of ground values that contain no duplicates. The can be amalgamated, intersected and the difference between two sets can be found. Lists can be mapped to sets and vice versa. They are different generic types.

QuLog is a fully integrated higher order logic and functional programming language in which arithmetic and functional expressions can be used as arguments of relation conditions, which we shall also sometimes refer to as calls. We will also briefly introduce these features even though they are not needed for the block tower building program.

This introduction to the declarative subset of QuLog is more than sufficient to understand all its example uses in this book. For a more complete description of QuLog, which also shows how it can be used for engineering multi-threaded communicating agent applications, see the book [35].

4.1 Getting the types right

The robotic arm and the blocks that must be re-configured as particular block towers are depicted in Figure 4.1. We assume that there is a separate camera that has a side view of the table as in the figure, with image processing software. This software can detect that the arm is holding block numbered 7, that blocks 9, 1 and 2 are on the table, that block 6 is on top of block 1 etc. on and holding are the percept predicates.

There are three towers in the figure. The list [8, 5, 4, 9] are labels of the blocks of the tallest tower. [3, 6, 1] and [2] are also towers. The concept of a tower needs to be defined using the adjacency relation on. One task we might want our tower builder to do is reconfigure the blocks to that [4, 6, 2, 7] is a tower. This will involve un-piling blocks on top of blocks 6 and 4 in order to pick them up.

Unless we are absolutely confident we can correctly define tower and any aux-
iliary relations we may need, we should first test separately our definitions using the QuLog interpreter. We will then load into the interpreter just the relation and function definitions we will be using together, with a set of on facts representing different configurations of blocks. When we use these relation definitions with our TeleoR program these on facts will not be in the program file. They will be supplied and repeatedly updated by the agent’s percept handler.

We could represent the above state of the world as the facts

```
holding(7)
on(8,5)
on(5,4)
on(4,9)
on(9,table)
on(3,6)
on(6,1)
on(1,table)
on(2,table)
```

This set of facts could be included in a test file with our relation definitions.
If you are familiar with Prolog you will notice there are no terminating fullstops. In QuLog and TeLeoR the termination of a fact, rule or procedure is indicated by the lack of indentation (at least one space or tab) on a newline of text. All facts, rules and procedures must start at the extreme left end of a new line. They can be continued over several lines providing the text on each continuation line is indented slightly to the right. This idea we borrowed from Python. Unlike Python the indentation does not need to be a fixed number of spaces, a single space will do.

For our bottle collector program we defined the thing, dir and colour types, and declared the types of its percepts, actions and TeLeoR procedures using these defined types. Defining and declaring types for our BeliefStore relations by the tower builder program is a good starting point.

In the figure the blocks are labelled with the numbers 1 to 9. We can use this range of numbers as our block type. Only block things can be an argument to holding. However, although on may only have a block as its first argument, its second argument may be a block, or table. To capture this alternative we can use a union (or disjunctive) type. We can define a tab type comprising just the atom table.

As in Prolog, an alphanumeric sequence that begins with a lower case letter and may contain underscores, or a singly quoted sequence of any characters including spaces, e.g. ‘Keith L. Clark’, is an atom. Integers are possibly signed sequences of digits and numbers have the usual decimal or e notation. Strings are sequences of doubly quoted characters, e.g. "Robotic Agent Programming.". They are represented a packed sequences of byte codes and are not the same as atoms.

We can then define a loc as either a block or a tab. These decisions are captured in the type definitions and declarations below.

```
block::= 1..9 % block identified by an integer between 1 and 9
tab::= table % enumerated type just containing atom table
loc::= block || tab % union type for where a block can be
belief holding:(block), on:(block,loc) % holding a unary relation with a block argument
% on a binary relation with block and loc arguments
```

holding and on have been declared as beliefs not percepts as we have facts for them in the program file. We could have given them normal type declarations but by declaring them as beliefs we can update them in the interpreter, as we shall show in Section 4.5. In the final QuLog + TeLeoR agent program file they will be re-declared as percepts and there will be no sample facts.

A file containing the above type definitions, declarations and the on and holding facts can now be loaded into the QuLog interpreter to check. Suppose the file is called towerBS.qlg, where .qlg is the extension for a QuLog or combined QuLog+TeLeoR
program file. After starting up the QuLog interpreter as per the documentation for the OS being used, we ‘consult’ the file using the action

```
| ?? consult towerBS.<return>   % full stop followed by a <return>
success
```

A <fullstop><return>, or <return><return>, signals the end of a query or command in the interpreter. Before it is given the entered query or command can be spread over several lines. success or fail is what QuLog displays after an action input. The | ?? is the QuLog prompt for a new query or action command.

If we have any syntax errors or we have the types wrong, helpful syntax errors will be given. The program file needs to be edited and consulted again.

### 4.2 Defining and testing the rule defined relations

Let us now address the issue of defining what a tower is. We will assume that we are given a list of block identifiers - i.e. integers in the range 1 to 9, and our QuLog definition has to test whether that list is already configured a tower. We want `tower([3,6,1])` to be inferable but not `tower([6,3,1])` and not `tower([6,1])` from our sample facts.

`[3,6,1]` is a tower because each adjacent pair on the list, i.e. the pairs 3, 6 and 6, 1 are such that we have a fact that tells us the first of the pair is immediately on top of the second, `on(3,6)` and `on(6,1)`, *and* there is no fact telling us there is a block on top of the first block on the list, 3. `[6,3,1]` fails on two counts. There is a fact telling us that we have a block on top of 6, `on(3,6)`, and no fact telling us 6 is on top of 6. `[6,1]` is not a tower because we do have a fact telling us that there is a block on top of 6, 6 is not clear.

What we need to define `tower` with argument type `[block]`, which is the QuLog type for a list of block values, are two concepts. The concept of a stack of blocks, where each adjacent pair B1, B2 on a given list of blocks is such that we have a fact `on(B1,B2)`, and the concept of a clear block B, which is a block for which we do not have a fact `on(_,B)` telling us there is another block, we do not need to know its identity, hence the _, which is on top of B.

In QuLog the most straightforward way to define stack is recursively. Suppose we have a list of blocks of the form `[B1,B2,..Blocks]` where Blocks may be the empty list of blocks [], or it could contain one or more blocks `[B3,..]`. If we know that the list `[B2,..Blocks]` was a stack, then we could infer that `[B1,B2,..Blocks]` was a stack if `on(B1,B2)`. This gives us our recursive rule. The simplest stack is a list of one block `[B]` such that we have `on(B,table)`. That is we have a stack of one block sitting directly on the table.

We need two QuLog rules to capture this recursive definition for testing a given list of block labels.
For Prolog programmers, the list patterns can be written using | instead of , , , e.g. [B1,B2|Blocks].

We can add these two relation rules, for historical reasons often called clauses, to the towerBs.qlg file and consult the file again. We can now test this definition by entering various stack queries.

| ?? stack([3,6,1]).  
  
  yes

| ?? stack([6,1]).  
  
  yes

| ?? stack([6,3,1]).  
  
  no

| ?? stack([B1,6,B3]).  
  Mode Error: the variables [B3, B1] occur unbound in input arguments in the condition stack([B1, 6, B3]) (since stack:! [block])

The first two queries show we have a correct definition for stack for testing given lists of block values. The mode error given in response to the last query reminds us the clauses can only be used for testing. This means that the argument must be a list that contains no variables.

### 4.2.1 Ground and template terms

- Terms containing no variables are *ground* terms.

- Terms that may contain variables are *template* terms.

If we wanted to use it to find the stacks as well as to test, and as in the case of the above query, to check if 6 is the centre block of a stack of 3 blocks, *and* to find the blocks that are above and below 6, we need to
change the type declaration for \texttt{stack}, dropping the \texttt{!}. In fact we also need to slightly change the second clause.

If we just change the type declaration to \texttt{stack: ([block])<=}, when we consult the program file we will get

$$\text{Type Error: argument Block2 might not of be type block}$$

in the body condition

$$\text{stack([Block2,..Blocks]) (stack:[block])}$$

of relation clause

$$\text{stack([Block1, Block2,..Blocks]) <=}$$

$$\text{on(Block1, Block2) & stack([Block2,..Blocks])}$$

Why do we get this? It is because, for ease of representation of the perceptual information we have allowed the second argument of \texttt{on} to be \texttt{table}. For the test only use the \texttt{QuLog} mode checker knows that \texttt{B2} will be given and be a block identifying integer. So the \texttt{on} condition, which will be checked first, is only used for testing. But if the list of blocks is not given, or is only partially given, and \texttt{B2} is an unbound variable when the clause is used, then the \texttt{on} condition when evaluated may return \texttt{table} as a binding for \texttt{B2}, which is not of \texttt{block} type hence we may be passing into the recursive call a list that begins with a non-block value.

Fortunately, this error can be remedied by filtering out any rogue value with a run-time type test inserted immediately after the \texttt{on} condition. For test or generate flexibility, we need

\begin{verbatim}
stack: (?[block])<=
  % ? annotation means arg. may be a variable or a template list,
  % but will be grounded by a call
stack([Block]) <= on(Block,table)
stack([Block1,Block2,..Blocks]) <=
  on(Block1,Block2) & type(Block2,block) &
  stack([Block2,..Blocks])
\end{verbatim}

Now we can find or test lists of blocks that are configured as stacks.
| ?? stack([B1,6,B3]). B1=5:block B3=1:block 
| ?? stack([B1,B2,B3]). B1=5:block B2=4:block B3=9:block ...

| ... % ... separates from the next answer, output by QuLog

Note that each answer is annotated with its type.

The default for a relation query in QuLog is that up to 5 answers are displayed in response to a query. After 5 answers have been displayed we can enter a . (two fullstops) to see the next answer. Or we can enter .. n, where n is a positive integer, to see the next n answers.

As an alternative to having to do the type check in the above definition, we could use a belief `on_table` instead of an `on(.,table)` belief. We could then use

```
belief on_table:(block) 
on_table(9) 
on_table(1) 
on_table(2) 

stack:([block])<= stack([Block]) <= on_table(Block) 
stack([Block1,Block2,...Blocks]) <= on(Block1,Block2) & stack([Block2,...Blocks])
```

However, we have not used this representation of block configurations for two reasons. Firstly, we wanted to introduce the idea of union types and subtype testing. Secondly, by combining a table and a block into the concept of a location we can use just one block moving procedure rather than two - one for destination a block and one for destination the table. This will be particularly useful for our generalisation of the TeleoR program in Chapter 14 when we use two arms and have three tables.
4.3 Displaying the program

At any point you can see all the loaded relation and function definitions, preceded by their type declaration, by entering the command

| ?? show.

You can just see the clauses for a particular relation, such as stack, or rules for a particular function, by entering

| ?? show stack.

\[
\begin{align*}
\text{stack} : (\text{?}[\text{block}]) & \leq \\
\text{stack}([\text{Block}]) & \leq \text{on}([\text{Block}, 0]) \\
\text{stack}([\text{Block1}, \text{Block2}|\text{Blocks}]) & \leq \\
& \quad \text{on}([\text{Block1}, \text{Block2}]) \& \text{type}([\text{Block2}, \text{block}]) \& \\
& \quad \text{stack}([\text{Block2}|\text{Blocks}])
\end{align*}
\]

Notice that it retains the variable names of the source program.

You can even enter show st instead of show stack. The QuLog interpreter will extend this partial name to each of the names of relations or functions with loaded definitions and display them preceded by their type declaration. This is useful if you have used a long name.

It is also useful to see just the type declarations and the type definitions, or a particular type declaration.

| ?? types.

\[
\begin{align*}
\text{block} & ::= 0 .. 9 \\
\text{tab} & ::= \text{table} \\
\text{loc} & ::= \text{block} || \text{tab} \\
\text{holding} & ::= (\text{block}) \\
\text{on} & ::= (\text{block}, \text{block}) \\
\text{stack} & ::= (\text{?}[\text{block}])
\end{align*}
\]

| ?? type st.

\[
\begin{align*}
\text{stack} & ::= (\text{?}[\text{block}])
\end{align*}
\]

4.4 Building Upon Defined and Tested Definitions

Now that we have stack recursively defined and tested we can define tower, the configuration that we want our TeleoR program to achieve. A tower is a stack \([B, .. Bs]\) such that there is no block on top of \(B\). A first definition, for checking only, could be
which we add to the QuLog program file. If no mode annotation is given on a type in a declaration ! is assumed as default. It will be added if we show the type.

The first condition is a negated condition read as ”not there exists something on Block”. As with the rule for stack, when this rule is used to test if a given list of block integers is configured as a tower the not condition will be evaluated first to check that there is not other block recorded as being on Block in our BeliefStore. That is, there is an attempt to show that there is such a block on Block. If that succeeds the negated condition fails, but if the attempt to find such a block fails, the negated condition succeeds.

This way of confirming that negated test conditions hold is called negation as failure. An interpretation of the use of such an inference rule to infer negated conditions as proof from a set of if and only if definitions and inequalities that tell us that different names for things denote different things, e.g. $2 \neq 3$ etc., is given by Clark [29]. It is the only way that negated conditions are checked in QuLog. A constraint on its use is that by the time the condition is checked, in the left to right order of evaluating conditions in queries and rule bodies, all but underscore _ variables must be bound to terms containing no variables, to ground terms. In the above rule this will be the case as it is moded as a test only rule. Two testing queries are

| ?? | tower([3,6,1]). |
| yes |

| ?? | tower([6,1]). |
| no |

Here the yes response should be read as can infer, the no response as cannot infer. They are the same as true and false only if we assume we have complete information about the defined relations and that different names denote different things, as mentioned above.

For our use in guards of the TeleoR program for controlling a robot arm to build block towers we shall only need test only use of our tower and stack clauses. However, let us see what we need to do to the tower definition if we wanted to find lists of blocks configured as towers according to the current set of on facts.
Just changing the mode to test or generate by changing the ! to ? on the type declaration will result in a mode error telling us that we have a negated condition not on(_,Block) where Block might be an unbound variable. One way to avoid this is to but it at the end of the rule

\[
\begin{align*}
tower:(?\text{[block]}) & \\
tower([\text{Block,..Blocks}]) & \leq \\
& \text{stack([Block,..Blocks]) & not on(_,Block)}
\end{align*}
\]

Note that we have not given the \(\leq\) indication in the type declaration that \(tower\) is a relation. Such an abbreviated type declaration can be given for relations defined in the same program file as the declaration. We shall use this style of type declaration from now on.

When used to find or to instantiate a template list this uses the \(\text{stack}\) clauses to find candidate towers that are stacks and then rejects those that are sub-stacks having another block on top of the stack top \(\text{Block}\). The trouble is that this is inefficient for testing if a given list of blocks is a tower. Having the critical negated condition be tested first immediately rules out all lists of blocks where the first block is not covered by another.

Fortunately, there is another solution that involves a cheap extra test or generate condition as the first condition of the original clause. Where a program defined type is a range type such as \(\text{block}\), or an enumerated type, such as the \(\text{thing}\) and \(\text{colour}\) types of the bottle collector program, or a union of such types, we can use a QuLog primitive \(\text{isa}(V,T)\). \(V\) is usually a variable. \(T\) must be the name of a range or enumerated type. If \(V\) has a value when the \(\text{isa}\) is evaluated, it checks if the value belongs to type \(T\). If \(V\) does not have a value, it will generate each of the possible values of type \(T\) in turn, as possible solutions to the \(\text{isa}\) condition. For example,

```haskell
| ?? 8?isa(N,block).  \% The 8 ? prefix is request to see 8 answers if poss.
N=0:block
...
N=1:block
.
N=7:block
.
N=8:block
...
N=9:block
\% \.. is request for another 8 if there are 8 more
```

69
A definition of \textit{tower} that is good for testing or generating is

\begin{verbatim}
tower:([?block])                      % test or generate use
tower([Block,..Blocks]) <=
  isa(Block,block) & not on(_,Block) & stack([Block,..Blocks])
\end{verbatim}

We could use \texttt{isa(Block2,block)} instead of test only \texttt{block(Block2)} of the \texttt{stack} definition. We \textit{cannot} use \texttt{block(Block)} instead of \texttt{isa(Block,block)} in the above clause as \texttt{block(Block)} is not capable of generating candidate bindings for \texttt{Block} when this is not given.

### 4.5 Emulating a percepts update

Suppose we would like to check what lists of blocks will be inferable after an action to put the held block 7 on top of the currently clear block 2. One way is to edit our program file and to re-consult it. However, because we have declared \texttt{holding} and \texttt{on} as belief predicates, we can do the update as a command when prompted to enter a query or command.

\begin{verbatim}
| ?? forget holding(_); remember on(7,2).
success
or
| ?? replace holding(8) by on(8,2).
success
\end{verbatim}

will do the update for us. Note that we have separated the update commands of the first query using \texttt{;}. This is the connective that must be used in \texttt{QuLog} action rules. It is optional in an interpreter query command. The above \texttt{BeliefStore} updaters, and others, are more fully described in Section 8.4.

If we did this update \texttt{tower([7,2])} would be inferable.

### 4.6 Query evaluation and backtracking

We have already mentioned that queries and rules have their conditions/calls evaluated left to right. But where a condition has multiple solutions, as with \texttt{isa(Block,block)}, all its solutions are not generated at once. The \texttt{QuLog} evaluator finds the first solution binding for \texttt{Block}, which will be \texttt{Block=1}, then passes this on to the rest of the conditions in the rule body, which become
not on(_,1) & stack([1,..Blocks])

For the facts given in Section 4.1, the test not on(_,1) will fail, as we have the fact on(6,1).

At this point the next solution, Block=2 to the isa condition will be found. Going back to a previously evaluated condition to find the next solution, if there is one, is called backtracking. Left to right evaluation of conditions with backtracking, to re-try and find another solution to a previously solved condition, is how QuLog finds solutions to queries and evaluates clause bodies.

In this case the next solution binding Block=2 is a block that has no other block covering it. It is also directly on the table as we have the fact on(2,table). The conclusion is that [2] is a tower.

To find more towers using the facts of Section 4.1, the QuLog evaluator will first see if there are more solution bindings for Blocks in the last condition stack([1,..Blocks]). As there are none, it will skip the not test and look for the next solution to isa(Block,block). This will give the candidate binding Block=3. As this does not have a covering block, that [3,6,1] is a tower will be inferred.

Finally, after two more backtracking retries of the isa condition, the last solution tower([8,5,4,9]) will be found.

4.7 Watching the evaluation of one or more calls

The QuLog support documentation describes how to get a log of the evaluation of calls to one or more relations at two levels of detail. It is very useful for learning about how QuLog evaluates queries and hence the guards of TeleoR rules.

There is a watch Rel command that will display all calls to the relation Rel, log whether they succeed or fail giving the solution instantiation, and log when the call is retried on backtracking with details of subsequent success or failure.

The next level is watchC Rel. This also gives the instantiation of the clause for Rel currently being used to try to find a solution to a Rel call.

Any number of relations can be watched at the same time but doing just one or two at one time is the most useful. We give an example watchC log in Section 4.9.
4.8 Mode annotations

No mode annotation or a prefix `!` indicates that an argument must be given as a variable free data value - a ground term - when the relation is called - that the argument is input only. That this will be the case, for all calls to the relation in a QuLog+TeleoR program file, is checked by the translator as it is type checking the program. If there is a call for which this might not be the case a mode error will be signalled and the translation terminated.

The argument types for TeleoR procedures and QuLog functions are not mode annotated as they are all implicitly input only mode. Each call must be such that every argument is given as a fully instantiated value - a ground value. Each must also be of the specified type, or be a sub-type of the specified type. For example, if the specified type is atomic, meaning an atom, a string, a nat, int or a num value, any one of these values can be given as argument in a call. nat is the type of the non-negative integers. If it is necessary to distinguish in the rules which sub-type is actually given in a call, a run-time type test such as `type(N,nat)` must be used. A ground argument term of a sub-type of the declared type of an argument of a relation can always be given when the relation is called.

Argument types for a QuLog action procedure can be mode annotated using the same modes as for relations. An unannotated type for an action procedure type declaration is also assumed to be an input only argument.

There are three more mode annotations that can be used for QuLog relations and action procedures. The moded type `?t` means that in the call the argument may be an unbound variable but that this will be given a ground value of type `t` if the call succeeds. In addition, if `t` is a structured type like a list, a template term of type `t` (or a sub-type of type `t`), can be given as the call argument, as in the query call `stack([B1,B2,B3])`. After the call the argument term will be a ground value of type `t`. For the `stack` call all three variables will be bound to block numbers. `?` signals test or grounding generate use. We did not need to annotate the argument types for percept facts because an implicit `?` mode annotation is assumed. However, we could have prefixed their argument types with `?`.

The other two modes will not be used for relations of our agents’ Belief-Stores. These are a postfix `?` and a prefix `??`. A postfix `?`, as in `[int]??`, is a relaxation of the `?` mode that does not require a call to the relation to generate a ground value for the given argument, it may be left as a template term, even as an unbound variable if that was given in the call. An example is the relation `app` for appending two lists of any type of value to produce a list of the same type of values. It can append and split template lists. We
give its definition in the Section 4.9.

The last mode is ?? which can be given without a type or as a prefix to the type term, the type of any QuLog data term. It means that the given argument will not be changed in any way by a successful call. As an example, type:(??term,typeE(T)) is the type of type relation. typeE(T) is a system meta-type with value the type expression for a type T. So an instance of this parameterised type, of meta-type typeE([int]), is the type expression [int]. The term argument is prefixed ?? as it will not be altered by the type test, even if, as allowed, it is a template term containing variables. So, type([3,X,9],[nat]) will succeed.

4.9 List and string processing

The QuLog definition for the relation app(L1,L2,L3), which holds when L3 is L1 with all the members of the list L2 inserted after the last member of L1, e.g. app([1,2],[3,4],[1,2,3,4]) holds, is

```
app: ([T], [T], ?[T]) | (?[T], ?[T], [T]) | ([T]?, [T]?, [T]?)
% Multiple mode declarations
app([],L,L)
app([U|L1],L2,[U|L3]) <= app(L1,L2,L3)
```

T is a type variable telling us that the relation is polymorphic, able to be used with any type of list arguments.

The first moded type tells us that when the first two arguments are ground lists of T data values, and the last is a variable or template list term containing only values of type T, then the last argument will be a fully ground list of T values if the call succeeds. The second tells us that if the last argument is a ground list of T values, and the first two are variables or template list terms containing only values of type T, that they will be ground if the call succeeds. The last covers the case when each argument may be a template term. Its tells us that even after the call succeeds the arguments may still be template terms even though some of their variables have been bound to ground values.

Here are example uses.

```
|?? app([1,3],[-7,4,5],L).
L=[1,3,-7,4,5] : [int]  % Type expression of for a list of integers
```

73
| ?? app(L1,L2,[1,2,3]).
| L1=[] : [nat]
| L2=[1,2,3] : [nat]
| ...
| L1=[1] : [nat]
| L2=[2,3] : [nat]
| ...
| L1=[1,2] : [nat]
| ...
| L1=[1,2,3] : [nat]
| L3=[] : [nat]

| ?? app([X1,2],[X2,4,..L2],[1,X3,3,..L3]).
| X1=1 : nat
| X2=3 : nat
| L2=L2 : [Ty1]
| X3=2 : nat
| L3=[4,..L2] : [(nat||Ty1)]

For the last query the first argument has become the ground term [1,2],
the second argument the template term [3,4,..L2] and the last argument
the template term [1,2,3,4,..L2]. The type of the binding [4,..L2] is
given as a list if values from the union of nat (the number 4), and the type
T of the elements of L2.

You can see the log of the last query evaluation as follows

| ?? watchC app.
| yes

| ?? app([X1,2],[X2,4,..L2],[1,X3,3,..L3]).
| 1:app([X1, 2], [X2, 4,..L2], [1, X3, 3,..L3])
| Call 1 unifies clause 2 input U_0 = 1 L1_0 = [2] L2_0 = [X2, 4,..L2]
| L3_0 = [X3, 3,..L3]
| output X1 = 1
| Clause body is:
|   app([2], [X2, 4,..L2], [X3, 3,..L3])
| 2:app([2], [X2, 4,..L2], [X3, 3,..L3])
| Call 2 unifies clause 2 input U_1 = 2 L1_1 = [] L2_1 = [X2, 4,..L2]
| L3_1 = [3,..L3]
| output X3 = 2
| Clause body is:
|   app([], [X2, 4,..L2], [3,..L3])
| 3:app([], [X2, 4,..L2], [3,..L3])

74
Call 3 unifies clause 1 input \( L_0 = [3, 4, \ldots L_2] \)
output \( L_3 = [4, \ldots L_2] \) \( X_2 = 3 \)

No clause body

3: \( \text{app}([], [3, 4, \ldots L_2], [3, 4, \ldots L_2]) \) succeeded
2: \( \text{app}([2], [3, 4, \ldots L_2], [2, 3, 4, \ldots L_2]) \) succeeded
1: \( \text{app}([1, 2], [3, 4, \ldots L_2], [1, 2, 3, 4, \ldots L_2]) \) succeeded

\( X_1 = 1 : \text{nat} \)
\( X_2 = 3 : \text{nat} \)
\( L_2 = L_2 : [\text{nat}] \)
\( X_3 = 2 : \text{nat} \)
\( L_3 = [4, \ldots L_2] : [\text{nat}] \)

The \( .0, .1 \) and \( .2 \) extensions of the variable names of the program clauses indicate different uses of the clause with new variables for each use.

There is a QuLog primitive \texttt{append} with exactly the same type declaration and clause structure.

4.9.1 The \( <> \) operator and list pattern matching

The \texttt{append} primitive must be used if we want to manipulate template lists. However, our \texttt{BeliefStore} inference support for \texttt{TeleoR} guard evaluation, for all examples we have looked at so far, only needs to manipulate ground lists with the occasional generation of a ground instance of a list template term.

For this more usual form of list manipulation QuLog has a primitive function \( <> \) used as an infix operator. Its type declaration is

\[
<:([T],[T]) \rightarrow [T]
\]

Our first use of the \texttt{app} relation for appending two lists \([1,3], [-7,4,5]\) is achieved with the expression \([1,3]<>[-7,4,5]. \) This can be entered in the QuLog interpreter as an expression query

\[
| ?? [1,3]<>[-7,4,5].
[1,3,-7,4,5]:[\text{int}]
\]

The type of an expression value is given after the value. In this case list of \texttt{int} type.

Another example use is the expression \( L_1<>[3]<>L_2 \) which will append ground list values of \( L_1 \) and \( L_2 \) inserting 3 between them. \( L_1 \) and \( L_2 \) do not have to be lists of integers or even numbers. If \( L_1 = [a,b] \) and \( L_2 = [4.5,c] \) the expression value is \([a,b,3,4.5,c]\) of union type \texttt{atom} || \texttt{int} || \texttt{num}. 75
We can use <> to recursively define a function that will reverse a ground list of values of any type.

\[
\begin{align*}
\text{rev} : & \{[T] \} \rightarrow [T] \\
\text{rev}([]) & \rightarrow [] \\
\text{rev}([Hd,..Tl]) & \rightarrow \text{rev}(Tl)<>\text{[Hd]}
\end{align*}
\]

<> can also be used in a list pattern expression for splitting lists. Our second example use of app can be achieved using the query

| ?? L1<>L2 =! [1,2,3]

with exactly the same four answers. To exclude the empty list answers, we can use

| ?? L1?L1\=[]<>L2?L2\=[] =! [1,2,3]

where the inequalities are constraints on the bindings that are allowed for the variables of the pattern.

=! is QuLog’s pattern match operator that will also evaluate its right hand argument which has to be a ground expression when the =! is evaluated. =! is a significant generalisation of Prolog’s is operator. For list splitting use, the left hand side can have any number of occurrences of list patterns separated by <>. 

Bf<>[3]<>Af =! [1,2,3,4,5]

will produce the bindings Bf=[1,2], Af=[4,5].


will append the three ground lists L1, L2 and L3. It will then find all the occurrences of 3 in the resulting list generating bindings for Bf and Af which are the sub-lists before and after each occurrence. For example,

% List bindings for L1, L2, L3 are given in the query

Bf=[1] : [nat]
Af=[4,2,3,9] : [nat]
...

76
We can add any single condition constraints to the splittings of a list achieved using $<>$ patterns. These are relation calls that appear after the list pattern with a separating ?. The constraints are checked left to right as candidate bindings for the variables of the pattern are found and when not satisfied will result in a different splitting being tested, if any remain.

For example,

$$\mid \ ? \ L1=\{1,-2\} \ & \ L2=\{4,6\} \ & \ L3=\{-3,9\} \ & \ (Bf\#Bf>1 \ <> \ [E]\?E<0 \ <> \ Af) \ !=! \ L1<>L2<>L3.\mid$$

$Bf=[1,-2,4,6] : [int]$
$E=-3 : nat$
$Af=[9] : [nat]$

# is the QuLog operator function for finding the length of a ground list or the length of a string. The first constraint $\#Bf>1$ therefore requires $Bf$ to be bound to a list of at least two members. The constraint $E<0$ means $[E]$ must be single element list containing a negative number.

### 4.9.2 The ++ operator and string pattern matching

Concatenation and splitting subject to constraints can be done with strings using the $++$ string concatenation operator. This allows quite complex string processing.

A simple example is the function $\text{words: (string)$\to$[string]}$ which will return a list of the ‘word’ sub-strings appearing in a given string where a word sub-string does not contain any space character, the string does not start or end with a space, and only one space separates each word. Its definition is

$$\text{words: (string)$\to$[string]}$$
$$\text{words(Str) :: Word ++ " " ++ RemStr !=! Str} \ -> \ [\text{Word},..\text{words(RemStr)}]$$
$$\text{words(Str)} \ -> \ [\text{Str}] \quad \% \text{Only used if first rule does not apply}$$

$$\mid \ ? \ ? \ \text{words("Hello world how are you").}$$
$$["Hello","world","how","are","you"] : [\text{string}]$$

The definition can be modified to allow leading and trailing spaces and any number of spaces between words.
4.10 List and set comprehension expressions

Like Prolog QuLog has both deductive data base and term manipulation capabilities. The interface between the two is provided by QuLog’s list comprehension expressions and the primitive in relation for accessing the members of a ground list one at a time. in is used as an infix operator.

| ?? X in [1,2,3] & X in [2,3,4] & 0 != X mod 2. |
| : nat |

X=2

This query finds the even numbers in both [1,2,3] and [2,3,4]. Here, != is being used as an arithmetic expression evaluator.

Going the other way we can create lists from relational queries using list comprehension expressions.

| ?? \{X | X in [1,-2,3] & X in [-2,3,4] & 0 != X mod 2\}. |
| : \{nat\} |

| [2] |

The difference between the use of [...] brackets and {...} braces is that the latter will create set which has no duplicates. A set is always displayed surrounded with braces and with its elements ordered.

| ?? \{B | isa(B,block) & clear(B)\}. |
| : \{block\} |

The type of the set is \{int\}.

in: (?T,[T]) is a QuLog primitive for accessing the members of a ground list one at a time from the beginning of the list. There is another primitive list membership relation, member:(T?,[T]?). that can be used for accessing members of, or instantiating, a non-ground list.

| ?? member(2, [1,X,3]). |
| : nat |

| X=2 |

| ?? member(2,L). |
| : [nat] |

L = [2,..A] : [nat]...

L = [A, 2,..B] : [nat]...

L = [A, B, 2,..C] : [nat]...
\[ L = [A, B, C, 2, \ldots D] : \text{[nat]} \]

\[ L = [A, B, C, D, 2, \ldots E] : \text{[nat]} \quad \% \text{infinite number of answers} \]

If we wanted to convert from a set to a list we apply the \textit{to list} function denoted symbolically as the \([\text{[]}]\) pair of square brackets.

\[
?? \{3, -2, 8, 3, -1, 8\} \\
\{-1, -2, 3, 8\} : \{\text{int}\}
\]

\[
?? \text{[[]]}(\{3, -2, 8, 3, -1, 8\}) \\
\{-1, -2, 3, 8\} : \{\text{int}\}
\]

\[
?? \text{[[]}((\text{[[]]}([3, -2, 8, 3, -1, 8]))) \\
\{-1, -2, 3, 8\} : \{\text{int}\}
\]

The last query has converted an unordered list with duplicate into an ordered list without duplicates. More generally we can use the double conversion to define a polymorphic \textit{order} function.

\[ \text{order: [T] -> [T]} \]
\[ \text{order}(L) -> \text{[[]]}(\{\text{[]}\}(L)) \]

4.10.1 Set operations

\text{QuLog} has functions

\[ \text{union, inter, diff} \]

all of type \((\{\text{T}\}, \{\text{T}\}) \rightarrow \{\text{T}\}\), for manipulating sets. They can be used as infix operators.

\[
?? \{2, 4, 8\} \text{ union } \{4, 9, 11\}. \\
\{2, 4, 8, 9\} : \{\text{nat}\}
\]

\[
?? \{2, 4, 8\} \text{ diff } \{4, 9, 11\}. \\
\{2, 8\} : \{\text{nat}\}
\]

\[
?? \{2, 4, 8\} \text{ inter } \{4, 9, 11\}. \\
\{4\} : \{\text{nat}\}
\]

The \text{QuLog} type checker will check that these operations are only used with set arguments.
4.11 Iterating over all solutions to a query using \texttt{forall}

We can wrap all solutions to a query as a list using the list comprehension operators. Sometimes all we want to do is check that all elements of such a list satisfy some test condition. We can do this after constructing the list using the higher order \texttt{testAll} relation.

\begin{verbatim}
\texttt{testAll:([T], (T)\leq) \ % arguments are a list of Ts and a relation over T}
\texttt{testAll([],\_)}
\texttt{testAll([Hd,..Tl],Test) \leq Test(Hd) \& testAll(Tl,Test)}
\end{verbatim}

\begin{verbatim}
even: int
even(N) \leq 0 =! N \bmod 2
\end{verbatim}

\begin{verbatim}
| ?? testAll([\#Bs \mid tower(Bs)], even).
no
\end{verbatim}

We get the answer \texttt{no} because, as depicted in Figure 4.1, there is one tower of length 3. \texttt{\#Bs} is the length of the list of block numbers.

There is a more direct way of doing this test that does not require the construction of the list \([4,3,1]\) of the tower lengths. We can use a \texttt{forall} with an explicit quantification on \texttt{Bs}.

\begin{verbatim}
| ?? forall Bs (tower(Bs) \Rightarrow even(\#Bs))
no
\end{verbatim}

4.12 Type/mode checking in \texttt{QuLog}

There are constraints regarding the use of both list comprehension expressions and \texttt{forall} regarding scope of variables, and which variables must have ground values before the \texttt{forall} is evaluated. These are explained in [35].

We have shown that a mixture of type declarations for relations coupled with run-time type checking allows for a little more flexibility of use of relation definitions. It allows us to use a relation that only accepts a sub-type of term type that could be passed to it providing we guard the call with a run-time sub-type test. We can also use a relation that would normally generate a super-type of the required type providing we follow that with a type test filter that rejects all but values of the required sub-type.
There is actually an extreme of QuLog programming in which there are no declared types for relations, just mode annotations for their arguments that do not have the default \( ? \) mode, and there are no type declarations for functions. The default type \( \text{term} \) is assumed which is the super-type of all data types (not code types). We can still have new type definitions and we can use the \( \text{type} \) and \( \text{typeG} \) runtime tests with both primitive types and program defined types. We then have much more use of run-time type tests, especially as type guards before a primitive relation is used.

When an argument has type \( \text{term} \) we can pass in an integer, a number, an atom, a string, a list of terms, a constructed term \( c(t_1, \ldots, t_k) \) of some program defined type, or a constructed term \( \text{any}(t_1, \ldots, t_m) \) where \( \text{any} \) is any atom and \( t_1, \ldots, t_m \) are any terms. \( \text{any} \) is an anonymous constructor - implicitly a constructor of \( \text{term} \) values not of any sub-type of \( \text{term} \). The only constraint is that different uses of \( \text{any} \) should have the same number of arguments.

As an example of the use of the \( \text{term} \) type, suppose we wanted to sum all the numbers that appear on a list of any terms. The relation could be defined as

```prolog
sumNumsOnList: [\text{term}], \text{?num}
sumNumsOnList([],0)
sumNumsOnList([N,\ldots,Tail],Sum) :: \text{type}(N,\text{num}) <=
    sumNumsOnList(Tail,TSum) & Sum =! TSum+N
sumNumsOnList([\_\ldots,Tail],Sum) <=
    sumNumsOnList(Tail,Sum)
```

Without the commit test \( \text{type}(N,\text{num}) \) in the second clause we would get a type error. A possible query to this relation is

```
| ?? sumNumsOnList(["apples", 4, tom, [1,2,k], g(a,4), -2, 9],Sum).
  Sum=11 : num
```

Here, since \( g \) is not a constructor of any defined type or a defined function, \( g(a,4) \) will have QuLog inferred type \( \text{term} \).

In the above case we cannot tighten the type declaration for the relation but in other cases, when a program has been written and tested, we will be able to replace some, even most, runtime type tests with type declarations.

The QuLog translator uses abstract interpretation [40], [41] to data flow check each clause body and function rule commit test carrying along the
current type and groundness status of each variable. To give you an idea of what this does here are some rules that it implements.

A variable that appears in head of a function rule or in an input argument position in a clause head, starts with a \(!t\) status. Otherwise, a variable that does not appear in the function rule or clause head, gets a \(!t\) status as soon as it appears in a \(?t\) position of a relation call. A variable \(V\) with a \(!t\) status gets this changed to \(!t'\) if, and immediately after, it appears in a type test \(\text{type}(V,t')\), where \(t'\) is a sub-type of \(t\). The type test acts as a filter that only allows sub-type \(t'\) values through, possibly causing backtracking to get a different binding for \(V\) of sub-type \(t'\).

A variable with a \(!t\) status it can appear in a later relation or function call in a position with type \(t\), or a super-type of \(t\), no matter the mode. If a variable appears in the head of a clause in an \(?t\) argument position this becomes its initial status. Such a variable \(U\) can then appear in calls in positions with \(t?\) moded type, but must eventually appear in a call \(C\) in a position with moded type \(?t\), or moded type \(?t'\), where \(t'\) is a sub-type of \(t\), or moded \(?t''\) where \(t''\) is a super-type of \(t\). In the former case the call must be preceded by a variable ignoring type test \(\text{type}(U,t')\). This will succeed if \(U\) is unbound, else it will check that \(U\) is bound to a term of type \(t'\). In the latter case, the call \(C\) must be immediately followed by a type \((U,t)\) test to filter out values of the required sub-type, in case a value for \(U\) might be generated by the call \(C\). Where a ground value for \(U\) was given in the call, checking of the call arguments will have ensured that this value is at most type \(t\), so this \(\text{type}(U,t)\) test will always succeed.
Chapter 5

Incremental development and testing of TeleoR programs
Chapter 6

Recursive TeleoR Procedures, Relation and Procedure Hierarchies
We will introduce the concept of rule and procedure hierarchies, as Nilsson did, by first giving his robot arm block tower building program from [91], which we have slightly modified.

- It makes essential use of BeliefStore rules as developed in Chapter 4.
- It is a very good example of the robustness and task specific ‘intelligence’ of Teleo-Reactive programs.
- It is an example of a TeleoR program that uses recursive and mutually recursive procedures.

You can interact with a downloadable Java program of a simulated robotic arm controlled by Nilsson’s program accessible from the website for TR programs [92]. You can help or hinder the arm by moving blocks with the mouse and it will immediately respond sensibly. It will take advantage of any help and recover from any hindrance.

If you help by moving the next block that it would be placing onto a partially built tower onto the tower, it will immediately move to pick up the next but one block, or suspend its activity if the tower is now complete. Similarly, if you remove a block from its partially or completely built tower, it will abandon whatever it was doing to pick up and replace the displaced block, uncovering it if need be. This robust behaviour is a consequence of the testing of the rule guards of all the active TeleoR procedures on each significant change in the simulated environment, with the guards of the rules of each parent TeleoR call always re-evaluated before those of its child call, to check if the child call should be aborted.

The program we give is also the program our single arm controlling multi-tasking agent, that interleaves the building of two or more towers, uses. We do not need to change the program for this, just add the task atomic and task start assertions we mentioned in Chapter 2.

We will need to significantly modify the program to be used by our multi-tasking agent using two robot arms and three tables to concurrently build several towers. However that program still has the same top level structure and recovery and opportunity grabbing properties as Nilsson’s original program.

6.1 The BeliefStore rules

We will use the type declarations of Section 4.1 and the test only definitions of stack and tower from Sections 4.2 and ??.

We need to add one more definition.
We mentioned at the end of Section 4.2.1 that we defined the \texttt{loc} type in order to make use of one block move \texttt{TeleoR} procedure that moves a block, from wherever it is, perhaps inside an existing tower, to be either onto the table or onto a \texttt{clear} block, one that is not covered already with a block. The block or table will be the destination \texttt{loc} of the block and the \texttt{TeleoR} procedure will check that the \texttt{loc} is clear. The upshot is that we need to define a \texttt{clear} test predicate for any \texttt{loc} value. Its definition is

\begin{verbatim}
clear:(loc) % Test only relation for a location
clear(table)
clear(Block) <= not holding(Block) & not on(_,Block)
\end{verbatim}

We have made a simplifying assumption that there is always a clear space on the table that will be found by a high level robotic arm action \texttt{putdown} executed when the arm is holding a block.

6.2 Block Tower Building Program

The top level \texttt{TeleoR} procedure of the tower building program, making use of the rules we gave in Section 4, is the recursive procedure

\begin{verbatim}
percept holding:(block), on:(block,loc)
makeTower:([block]) % argument a list of blocks to be configured as a tower
makeTower([Block,...Blocks]){
tower([Block,...Blocks]) ~> () % list of blocks is configured as a tower
 stack([Block,...Blocks]) ~> unpile(Block)
   % [Block,...Blocks] % will be a tower if Block is cleared
 Blocks=[] ~> move_to_loc(Block,table) % need to move Block to table
tower(Blocks) & Blocks=[TopBlock,...] ~> % move Block to be on top
 move_to_loc(Block,TopBlock) % of clear TopBlock
 true ~> makeTower(Blocks) % recursively configure Blocks as a tower
}
\end{verbatim}

which only uses defined percepts. The tower comprises blocks identified by a given list of integer block labels \texttt{[Block,...Blocks]}. \texttt{unpile}, \texttt{move_to_loc}
are auxiliary TeleoR procedures. Firing of the last rule will normally eventually result in the test of the fourth rule being inferable. Firing of the second, third or fourth rules will normally eventually result in the guard of the first rule being inferable.

The procedure `unpile(Block)` moves all the blocks on top of `Block` onto the table. Note the mutual recursion between `unpile` and `move_to_loc` with location the table. The assumption is that a block can always be put down somewhere on the table.

Nilsson’s program has two procedures, one for moving a block to the table and one for moving a block to be on another block. We have merged them into one using the concept of a location. Our rules are also in a slightly different order so that they satisfy the regression property. We have the rule that puts any other block that happens to be being held on the table as the last rule. The cost is that we have to have the explicit `not holding(_)` tests in three earlier rules.

```lisp
(durative pickup:(block), putdown:(block,loc)

unpile:(block)
% recursively (via move_to_loc) moves blocks above Block to table
unpile(Block) {
  clear(Block) ~> ()   % the goal of the procedure is achieved
  on(OBlock,Block) ~> move_to_loc(OBlock,table)
}

move_to_loc:(block, loc) % moves Block from wherever it is to be on Loc
move_to_loc(Block,Loc) {
  on(Block,Loc) ~> ()   % the goal of the procedure is achieved
  holding(Block) & clear(Loc) ~> putdown(Block,Loc)
  not holding(_) & clear(Loc) & clear(Block) ~> pickup(Block)
  not holding(_) & clear(Loc) ~> unpile(Block)   % need to clear Block to pick it up
  not holding(_) ~> unpile(Loc)   % Loc must be an unclear block
  holding(B) ~> putdown(B,table)
}   % putdown any other block being held on the table
```
6.3 Brief Description of How the Program Behaves

Suppose the start call is \texttt{makeTower([2,4,1,8])} for which the goal of the task is to re-configure the identified blocks so that 8 lies directly on the table, 1 is on top of 8, 4 on top of 1, 2 on top of 4, with no block on top of 2.

If the agent is really lucky, the blocks are already configured as a tower and the first rule of \texttt{makeTower} can be fired. If it is just lucky, these blocks are on top of one another in sequence with 8 on the table, but there are blocks on top of 2. The second rule of \texttt{makeTower} then applies, and the blocks on top of 2 will be one by one moved onto the table. This is done by calling \texttt{unpile(2)}. If block \texttt{b1} is immediately on top of 2, \texttt{move_to_loc(b1,table)} is called. If \texttt{b1} is clear - no block on top - it is picked up and then put down on the table. However, if there is a block \texttt{b2} on top of \texttt{b1}, there is a call to \texttt{unpile(b1)} which in turn calls \texttt{move_to_loc(b2,table)}. If there happens to be a block \texttt{b3} on top of \texttt{b2}, \texttt{unpile(b2)} is called and then \texttt{move_to_loc(b3,table)}.

Suppose \texttt{b3} is clear. It will be picked up. Immediately its ancestor call \texttt{move_to_loc(b1,table)} will fire its 5\textsuperscript{th} rule (remember all ancestor calls remain active), which will put \texttt{b3} on the table, somewhere. The calls to \texttt{unpile(b1), move_to_loc(b2,table), unpile(b2), move_to_loc(b3,table)} will have been terminated. Since \texttt{b1} is still not clear, \texttt{unpile(b1)} then \texttt{move_to_loc(b2,table)} will be re-called. Now \texttt{b2} is clear so will be picked up. Again, immediately that happens ancestor call \texttt{move_to_loc(b1,table)} will fire its 5\textsuperscript{th} rule to put \texttt{b2} on the table making \texttt{b1} clear. It can now be moved to the table.

This yo-yoing up and down procedure calls is a little unsatisfactory but the re-calling of \texttt{unpile} and \texttt{move_to_loc} will be very fast. Even so, in section 13.5 we will give an alternative \texttt{move_to_loc} procedure in which the putting down of blocks \texttt{b3} and \texttt{b2} will be done inside the calls \texttt{move_to_loc(b3,table), move_to_loc(b2,table)} and not by an ancestor call.

Immediately after \texttt{b1} has been put down on the table, before the first goal achieved rule of \texttt{move_to_loc(b1,table)} can be fired, the first rule of the initial call \texttt{makeTower([2,4,1,8])} will fire. \texttt{[2,4,1,8]} will now be inferably a tower.

Let us now examine the other extreme, that no tail subsequence of the given list is already a \texttt{tower}, or a \texttt{stack} with other blocks on top. The
makeTower call recurses down using its 5th rule to a call makeTower([8]). As we have assumed no tail subsequence is a tower or stack, 8 must be buried inside some tower on the table. The 3rd rule of this last makeTower([8]) call will fire, calling move_to_loc(8,table). The behaviour of this is similar to the unpiling behaviour discussed above. All the blocks on top of 8 will be recursively moved to the table until 8 is clear, when it will be moved to the table.

Now the parent call makeTower([1,8]) can fire its 4th rule. This has the procedure call action move_to_loc(1,8). If need be 1 is made clear by a call to unpile(1). It is then picked up and put down on 8. Two more firings of rule 4, in the ancestor calls makeTower([4,1,8]) and makeTower([2,4,1,8]), possibly requiring calls to unpile if blocks 4 and 8 are not clear, will complete the tower resulting in the firing of rule 1 of the initial makeTower call.

The intermediary situations are where a tail sub-list of [2,4,1,8] is already a tower or is stack with blocks on top. In that case the initial makeTower call recurses until a recursive call can fire its rule 4 or rule 2. If it is rule 4, the tail say, [1,8], is a tower. The behaviour is then the recursion exit behaviour just discussed to move blocks 4 and 2 to complete this partial tower. If rule 2 is fired, the tail, again say [1,8], is a stack. It has all blocks covering 1 moved to the table as in our first scenario. The rule 4 of the makeTower([4,1,8]) call will then be fired followed by a rule 4 firing of makeTower([2,4,1,8]).

This is all assuming there is no interference or help. We encourage you to download and interact with Nilsson’s simulation of its use from [92], helping and interfering, to observe how the program then behaves. We will discuss this co-operative behaviour a little in Section 13.4, in the context of the program’s use by an agent interleaving the building of two towers which it is executing as two independent tasks.

6.4 Tower Architectures

When the sensor percepts and other dynamic beliefs are augmented by rule defined percepts and beliefs, there are two main ways in which the rules can be handled. One way is to use the rules in a forward inferring way to compute all the instances of the defined predicates when the dynamic beliefs change, with a truth maintenance system (TMS) revising these inferred facts, as need be, when the dynamic beliefs change again. In the second way, the rules are used as in Prolog in backward inferring fashion to check or to find instances of the defined predicates each time a guard that uses them is
evaluated to the point where they are used. This approach always accesses the latest state of the dynamic beliefs so the inference dependences that must be remembered as part of a truth maintenance system are not needed.

### 6.4.1 Triple Tower Architecture

In the triple tower agent architecture [91] depicted in Fig. 6.1 the first method is adopted. Each tower corresponds to a layering of progressively more abstract concepts. As we ascend the definitions tower we find definitions of higher level beliefs defined in terms of lower level ones. As we ascend the beliefs tower we find the inferred instances of these higher level beliefs. As we ascend the action tower we find higher level TR procedures with preconditions that query these higher level beliefs and which call lower level procedures that query the lower level beliefs. At the top of the action tower we have behavioural skills that require reasoning to determine how to
act, where the actions are typically complex activities defined by TR procedures. At the bottom we have behavioural skills that directly respond to changing dynamic beliefs, particularly sensor percept beliefs, with primitive actions.

The simulation of [92] shows all the facts in the agent’s belief tower as the configuration of blocks changes. For example, if you start with two blocks labelled a and b on the table you will see the facts

\[
\begin{align*}
on(a,\text{table}) & \quad on(b,\text{table}) \quad clear(a) \quad clear(b) \\
stack([a]) & \quad stack([b]) \quad tower([a]) \quad tower([b])
\end{align*}
\]

The clear, stack and tower facts are all inferred using the BeliefStore rules. The truth maintenance system keeps track of the percept facts, or absence of percept fact needed to infer each.

If you move block a to be on top of block b you will see

\[
\begin{align*}
on(b,\text{table}) & \quad on(a,b) \quad clear(a) \\
stack([b]) & \quad stack([a,b]) \quad tower([a,b])
\end{align*}
\]

clear(a) and stack([b]) were the only inferred facts not to have their dependencies undermined by the change of on percepts. stack([a,b]) and tower([a,b]) are newly inferred.
6.4.2 Double Tower Architecture

The backward inferring method whereby one or more instances of a defined predicate are inferred each time a condition using the predicate is accessed in a guard evaluation, with no remembering of the inferred instances, corresponds to a two tower architecture as depicted in Fig. 6.2. A hybrid approach would use some rules to infer all instances of the percept predicates they define when primitive percepts arrive, with the inferred instances remembered. We then re-infer instances of these defined percepts for each new batch of primitive percepts, deleting previously inferred instances that are no longer inferable and remembering new instances. Using this simple approach to truth maintenance, we do not need a dependency based system.

We further restrict the BeliefStore rules that can be used in this way to percepts that are neither directly or indirectly recursively defined, nor do they directly or indirectly depend upon other infer and remember predicates. They can depend upon other defined percepts, even recursively defined percepts, but the rules for these other percepts are not used when new percepts arrive to add further facts to the BeliefStore. This means we can infer the instances of these independently defined non-recursive predicates in any order. This simplifies the implementation of defined precept remembering inference even further.

As an example, the one rule definitions of next_to, close, near and see given in Section 3.6 are independent of one another and not recursive. They are ideal definitions to use for this rough and ready inferring and remembering on each percepts update that will provide fresh see_patch percepts.

We have adopted this restricted hybrid approach on efficiency grounds. We actually do backward chaining inference to generate all the instances of these dynamic rule defined percepts, remembering each instance as it is inferred. This amounts to a special form of tabling evaluation [117]. Normal tabling only remembers those instances that are inferred during a normal backwards inferring query evaluation. The implementation of this infer and remember inferencing, which is done by our agent’s percepts handler, is described in section 9.2.4.

6.5 Multiple Thread Architecture

For both architectures we can adopt a multi-threaded implementation. In the case of the Triple Tower, one thread handles the stream of sensor percept facts, the forward inferring inferencing and the associated truth maintenance
system. In the case of the Double Tower architecture, this thread just has to handle the stream of sensor percept facts. A second thread might handle a stream of messages from other agents. In both cases another thread handles the evaluation of the top level \texttt{TeleoR} procedure and any sub-procedures it calls. It generates the stream of primitive actions.

The advantage of having separate threads is that, as is depicted in Fig. 6.3, we can generalise the architecture to allow multiple \texttt{TeleoR} threads each accessing the same set of \texttt{TeleoR} procedures, querying the same deductive belief tower, but achieving different compatible goals, using and sharing independent resources. This is the multi-task agent architecture of our \texttt{TeleoR} robotic agents. In the figure the transparent arrows labelled with '?' depict the querying of the \texttt{BeliefStore} by the rule guards. The blue arrows from the task threads to the \texttt{BeliefStore} depict both the updating of the dynamic beliefs by \texttt{TeleoR} rule update actions, as explained in Section 12.1, and by translator generated updates of multi-tasking coordination facts, as will be explained in Section 15.5.
Chapter 7

Operational Semantics of Standard TeleoR
The formal part of this chapter, sections 7.5 and 7.6, can be skimmed on first reading, but a general understanding is needed when we discuss implementation issues in Chapter 9. At the very least you should read sections 7.1, 7.2 and 7.8. If you are not familiar with predicate logic or set expression notation, now is an appropriate time to dip into the sources of descriptions of this notation mentioned in Section ??.

We will begin by giving a control reading of a standard TeleoR (hence TR) procedure. When we introduce the new forms of rules in Chapter 10 we will give similar control readings for them.

We follow that with an informal operational semantics for a TeleoR procedure call illustrated by applying it to the call get_next_to(bottle) using the program of Chapter 3. We follow its evaluation through three BeliefStore states. This leads to the more precise formulation of the evaluation algorithm in Section 7.3. This is the basis of our reference implementation of Chapter 9. We then illustrate the use to the algorithm by continuing the example evaluation for two more BeliefStore updates.

We then consider the issue of the ambiguity in the original very informal operation semantics given for TR evaluation in [92]. It is whether or not a rule should be re-fired if another instance of its guard becomes the first inferable instance but the old instance is still inferable. For reasons we shall give, we decide on allowing a re-fire.

The formal semantics starts with Section 7.5 in which we give formal definitions of the inference relationships that must hold between the guard of some partially instantiated rule \( R \) of a procedure call \( P \) (partially instantiated because call arguments of \( P \) replace their occurrences in the rule), and the BeliefStore state for a ground instance \( GR \) of the rule to be: the new ground fired rule of a procedure call, continue to be the ground fired rule of the call on a BeliefStore update, be replaced by a different ground instance \( GR' \) of the same \( R \). The first event we shall call a firing of \( R \), the second a continued firing of \( R \), and the third event a re-firing of \( R \). These three definitions are key components of the transition semantics.

In Section 7.6 we begin by informally discussing what components characterise a state of evaluation of a TeleoR procedure call, which includes all its descendant calls and the last control messages that will have been sent to the robotic resource. We then formally define the state as a collection of logically expressed constraints that must be satisfied by a mathematical abstraction of the evaluation state. The transition rule that tells us when and how one static state transitions into another is then relatively simple.

We illustrate the formal semantics by giving the mathematical descriptions of three states in the evaluation of the call collect_bottle in Section
7.7. You might find it useful to dip into this section as you read Section 7.6.

We end the chapter with a discussion of the relationship between the formal and informal semantics and introduce the concept of conditions on the latest updates of the BeliefStore that must hold for there to be a change of fired rule in a procedure call, and hence by aggregation, for there to be a change of fired rule in any procedure call of the current evaluation state. This concept is important for the efficiency of the reference implementation of the operational semantics.

7.1 Informal Reading of a Standard Procedure

To remind you, a standard TeleoR procedure consists of a parameterised sequence of guarded rules

\[
p(X_1, \ldots, X_m)\{
    K_1 \rightarrow A_1
    
    \ldots
    
    K_n \rightarrow A_n
\}
\]

\(X_1, \ldots, X_m\) are the *global* variables of the rules. Variables in \(K_i\) and \(A_i\) that are not global are the *local* variables of the rule. A local variable appearing in \(A_i\) *must* be a local variable of \(K_i\).

Each \(A_i\) is either a template single call to a TeleoR procedure, or it is a template tuple of resource actions, which can be a mixture of durative actions (can be terminated and modified while executing) and discrete actions (cannot be terminated or modified while executing, usually for a short time). When instantiated by the evaluation of the guard \(K_i\) of its rule against the agent’s BeliefStore, all resource actions are executed in parallel using a robotic resource outside the agent.

\(K_i\) is implicitly existentially quantified with respect to its local variables not in \(A_i\), these are the ‘don’t care’ variables of \(K_i\). The whole rule is implicitly universally quantified with respect to the common local variables of \(K_i\) and \(A_i\), with \(K_i \rightarrow A_i\) read as *when_then_while* \(K_i\) *do* \(A_i*.

We can think of the sequence of all the rules as implicitly universally quantified with respect to the procedure parameters with an implicit *always-do-first-of* after this quantification.
approach(Th, FS, TS) \{  \% FS is forward speed, TS is turn speed
  see(Th, centre) \rightarrow move(FS)
  see(Th, Dir) \rightarrow move(FS), turn(Dir, TS)
\}

can be read as

$$\forall (Th, FS, TS) \text{ while doing } \text{approach}(Th, FS, TS)$$
$$\hspace{1cm} \text{always do } \text{first-of}$$
$$\hspace{1cm} \text{when then while } see(Th, \text{centre}) \text{ do } move(FS)$$
$$\forall Dir \text{ when then while } see(Th, Dir) \text{ do } move(FS), turn(Dir, TS)$$

deliver\_bottle\{}$
$$\hspace{1cm} \text{next to(drop, Dir)} \& \text{ gripper \_open} \rightarrow ()$$
$$\hspace{1cm} \text{next to(drop, Dir)} \rightarrow \text{ open\_gripper}$$
$$\hspace{1cm} \text{true} \rightarrow \text{ get\_next\_to(drop)}$$
\}$

can be read as

$$\text{while doing } \text{deliver\_bottle}$$
$$\hspace{1cm} \text{always do } \text{first-of}$$
$$\hspace{1cm} \text{when then while } \exists Dir \text{ next to(drop, Dir)} \& \text{ gripper \_open} \text{ do } ()$$
$$\hspace{1cm} \text{when then while } \exists Dir \text{ next to(drop, Dir)} \text{ do } \text{ open\_gripper}$$
$$\hspace{1cm} \text{when then while } \text{true} \text{ do } \text{ get\_next\_to(drop)}$$

Dir is univerally quantified using $\forall$ for the second clause of approach as it appears in the action of the rule. It is existentially quantified using $\exists$ for the first two rules of deliver\_bottle as it does not appear in the actions of the two rules. All that matters is that some value exists.

### 7.2 Informal Operational Semantics

When a TeRe procedure is called as an action $P = p(v_1, ..., v_m)$, its given arguments $v_1, ..., v_m$ are substituted for its parameters $X_1, ..., X_m$ throughout the rules resulting in a sequence of partially instantiated rules.
$K_1 \leadsto A_1$

where the bolding indicates the partial instantiation.

We use $eK_1$ to denote the existential quantification of $K_1$ with respect to its local variables not in $A_1$.

**Example** The procedure call $get\_next\_to(bottle)$ for the procedure

```plaintext
get_next_to(Th){
    Th=bottle & next_to(bottle,centre) $\leadsto ()$
    Th=drop & next_to(drop,_) $\leadsto ()$
    Th=bottle & next_to(bottle,Dir) $\leadsto $ turn(Dir,0.2)
    close_to(Th,_) $\leadsto $ approach(Th,0.2,0.2)
    near(Th,_) $\leadsto $ approach(Th,0.5,0.2)
    see(Th,_) $\leadsto $ approach(Th,1.5,0.1)
    true $\leadsto $ turn(left,0.5) }
```

of Section 3.2.3 gives us the following partially instantiated rules

```plaintext
bottle=bottle & next_to(bottle,centre) $\leadsto ()$
bottle=drop & next_to(drop,_) $\leadsto ()$
bottle=bottle & next_to(bottle,Dir) $\leadsto $ turn(Dir,0.2)
close_to(bottle,_) $\leadsto $ approach(bottle,0.2,0.2)
near(bottle,_) $\leadsto $ approach(bottle,0.5,0.2)
see(bottle,_) $\leadsto $ approach(bottle,1.5,0.1)
true $\leadsto $ turn(left,0.5)
```

and we have:

- $eK_1 = bottle = bottle & next_to(bottle,centre)$
- $eK_2 = \exists Dir$ bottle = drop & next_to(drop,Dir)
- $eK_3 = bottle = bottle & next_to(bottle,Dir)$
- $eK_4 = \exists Dir$ close_to(bottle,Dir)
- $eK_5 = \exists Dir$ near(bottle,Dir)
- $eK_6 = \exists Dir$ see(bottle,Dir)
- $eK_7 = true$

99
Note all occurrences of the anonymous variable \( ' \) are replaced by an existentially quantified \( \text{Dir} \).

The existentially quantified guards \( eK_1 \) to \( eK_n \) of the rules of \( P \) are then evaluated, one after the other. This is to determine the \emph{earliest} rule \( i \) such that \( eK_i \) has a successful evaluation against the current state of the agent’s \emph{BeliefStore}, i.e. which has a variable free (ground) instance \( eK_i\theta \) that can be inferred from the store.

\( \theta \) is a set of bindings for the free variables of \( eK_i \), if any, which are the variables in \( A_i \). There may be more than one such set of bindings, but the \( \theta \) returned will be the \emph{first} that the \emph{QuLog} query evaluation method returns.

The \( i^{\text{th}} \) rule instance \( K_i\theta \Rightarrow A_i\theta \) becomes the \emph{fired rule} of the procedure call, with \( A_i\theta \), the action to execute. If this is a \emph{TeleoR} procedure call \( Q \) the rules of \( Q \) are then checked in the same way and will normally result in the firing of a rule of \( Q \), which may also have a procedure call action. This cascading of procedure calls will continue until a rule is fired with resource actions.

If for some procedure call there is no guard query that succeeds, meaning that none has an inferable instance given the current state of \emph{BeliefStore}, this is a error condition and the whole \emph{TeleoR} evaluation terminates.

There is also the outside possibility that this cascading of procedure calls would not terminate for the current state of the \emph{BeliefStore}. An example is a call to the \emph{silly} procedure where \emph{SomeTest} fails for the current \emph{BeliefStore}

\begin{verbatim}
 silly{
   SomeTest \Rightarrow ()
   true \Rightarrow silly
}
\end{verbatim}

Recursion in \emph{TeleoR} procedures is not common and in any case nearly always has a small depth bound determined by the size of some argument of the call as in the recursive \emph{makeTower} of Section 6.2. We can therefore set an upper limit to the depth of \emph{TeleoR} procedure calls, which can be \emph{TeleoR} program specific and declared in the program file. In our implementation the default value for this call limit is 100. We therefore have another error condition, as well as failure to find a rule of a call that can be fired, which is that the call depth limit has been reached. It is an extremely rare error condition but also marks the end of a \emph{TeleoR} task evaluation.
7.2.1 Control actions

If the cascade of procedure calls does terminate it will end with the firing of a resource actions rule of the very last procedure call. If this is the first firing of a resource actions rule for a task, the last set of resource actions are taken to be the empty set \{\}. Otherwise, the last set of actions \textit{LActs} comprises all the actions of the most recent prior firing of a \texttt{Telescope} rule with resource actions.

Control actions are generated using \textit{LActs} and \textit{Acts}. Any durative action \textit{Dur} in \textit{LActs} not in \textit{Acts} is stopped - a control action \textit{stop(Dur)} is generated. Any durative action in both \textit{LActs} and \textit{Acts} is allowed to continue. No control action is generated. If a durative action \textit{Dur} in \textit{Acts} has the same action name but different arguments as an action \textit{LDur} in \textit{LActs}, a control action \textit{mod(LDur, Dur)} is generated. For any durative action \textit{Dur} in \textit{Acts} not in \textit{LActs} a \textit{start(Dur)} control action is generated. Finally, for any discrete action \textit{Dis} in \textit{Acts} a control action \textit{do(Dis)} is generated. The control actions are then executed.

\begin{example}

\textbf{Example continued} Suppose that \texttt{get.next.to(bottle)} is called for the first time as part of the evaluation of a task with start call \texttt{get.bottle}. This start call has fired its 3\textsuperscript{rd} rule because we have

\begin{center}
\textit{Belief Store} = \{\texttt{see(bottle, left), gripper.open}\}
\end{center}

The 6\textsuperscript{th} of the partially instantiated rules for the call \texttt{get.next.to(bottle)} becomes the fired rule of this procedure call with \(\theta = \{\}\), since the binding for \textit{Dir} is not retained, with action \textit{approach(bottle, 1.5, 0.1)}.

The \textit{approach} call has partially instantiated rules

\begin{center}
\begin{align*}
\texttt{see(bottle, centre)} &\sim \texttt{move(1.5)} \\
\texttt{see(bottle, Dir)} &\sim \texttt{move(1.5), turn(Dir, 0.1)}
\end{align*}
\end{center}

The second rule of this call is fired with \(\theta = \{\text{Dir = left}\}\). This is a rule firing with an instantiated guard

\begin{center}
\texttt{see(bottle, Dir)\{Dir = left\}} i.e. \texttt{see(bottle, left)}
\end{center}

and an instantiated tuple of resource actions
\end{example}
(move(1.5), turn(Dir, 0.1))\{Dir = left\}

giving the set of actions

\{move(1.5), turn(left, 0.1)\}

Because the get bottle task was started with the robot’s gripper open, there are no preceding actions, so these two durative actions will be started. Control actions \textit{start}(move(1.5)), \textit{start}(turn(left, 0.1)) are executed.

### 7.2.2 Response to a BeliefStore update

When the BeliefStore is next updated, the guards of the rules of the start procedure call are re-checked. The re-check is to see if the same rule should be allowed to continue as the fired rule of the call, and if not, if the same rule should be re-fired, or if a different rule should be fired, given the new BeliefStore state. The same \(i^{th}\) rule \(K_i \Rightarrow A_i\) should be allowed to continue as the fired rule if it is still the first rule with an inferable guard and the inference returns the same set of bindings \(\theta\) for \(eK_i\), as the first answer. The rule is re-fired if it is still the first rule with an inferable guard but the inference returns a different set of bindings \(\psi\) for \(eK_i\) as the first answer.

It is a continuation and not a re-fire if the re-inference of \(eK_i\) from the new BeliefStore, which in practice will be a QuLog evaluation of the query \(K_i\) without the existential quantifiers, returns different values for just the existentially quantified variables of \(eK_i\), the local variables of \(K_i\) not in \(A_i\). We do not care about the values of these variables, all that matters is that values exist such that \(eK_i \theta\) can be inferred.

In the above example, the fired rule of \textit{get next to(bottle)} is its \(6^{th}\) rule with existentially quantified guard \(\exists Dir \ see(bottle, Dir)\). This was inferable because \(see(bottle, left)\) was in the BeliefStore. \(Dir\) would have been given the binding \(left\) but that binding was discarded as \(Dir\) does not appear in the rule action. Suppose that the BeliefStore is updated again and now contains \(see(bottle, centre)\) but no \(near(bottle,..)\) fact. \(\exists Dir \ see(bottle, Dir)\) is still inferable and the fact that \(Dir\) will be given a different binding is ignored. There is no rule re-firing and the call \textit{approach(bottle, 1.5, 0.1)} will continue.

Finally, a new rule \(K_j \Rightarrow A_j, j \neq i\), is fired if the previous fired rule is
not continued and this $j^{th}$ rule is the first rule with an inferable guard from the new BeliefStore state.

Suppose the previously fired rule is continued and it has a procedure call action $Q$, the existentially quantified guards of $Q$ are now re-checked. This re-checking of the guards of all descendant procedures calls of the start call continues until there is a new rule firing, or the bottom-most procedure call is reached and its fired rule continues as this call’s fired rule. If the latter occurs, there are no new control actions generated and any previous durative actions continue unchanged.

If at any stage there is a different rule firing, or a re-firing, and the previously fired rule had an action that was a procedure call $Q'$, the call $Q'$ is immediately terminated. If it has descendant procedure calls they are also immediately terminated. A TeleoR procedure call is terminated by its parent call or some other ancestor call. It does not terminate itself.

If the new firing also has a procedure call action, the guards of the rules of this new call are tested in turn to find the first fired rule of this call. There may again be a sequence of new rule firings with procedure call actions until a rule instance with resource actions is fired. These resource actions, and the resource actions of the most recent previous rule firing with resource actions, are then used to generate new control actions as described in Section 7.2.1, which are then executed.

Example continued Suppose our BeliefStore is updated when new percepts arrive to

\{see(bottle, centre), near(bottle, centre), gripper_open\}

The rules for the first call get_next_to(bottle) are re-checked. The first rule with an inferable guard is rule 5. This switch of fired rule will cause the call to approach(bottle, 1.5, 0.1) to be terminated and a new call approach(bottle, 0.5, 0.2) to be invoked.

The partially instantiated rules for this new approach call are

\[
\begin{align*}
\text{see}(\text{bottle}, \text{centre}) & \rightarrow \text{move}(0.5) \\
\text{see}(\text{bottle}, \text{Dir}) & \rightarrow \text{move}(0.5), \text{turn}(\text{Dir}, 0.2)
\end{align*}
\]

For the approach(bottle, 0.5, 0.2) procedure call the first rule will be fired. As its action is move(0.5), and the last determined set of resource
actions was \{move(1.5), turn(left,0.1)\}, the executed control actions are \textit{mod}(move(1.5), move(0.5)), \textit{stop}(turn(left,0.1))

Assume \textit{BeliefStore} is now updated to

\[
\begin{align*}
\{ & \text{see(bottle, right), near(bottle, right), gripper\_open} \}
\end{align*}
\]

The quantified guard $\exists \text{Dir near(bottle, Dir)}$ of the 5th rule of \texttt{get\_next\_to(bottle)} is inferable. Evaluation of $\text{near(bottle, Dir)}$ will return a different value for \textit{Dir}, but this does not affect the rule action which is still the call \textit{approach(bottle, 0.5, 0.2)}.

The second rule of \textit{approach(bottle, 0.5, 0.2)} will be re-fired with returned binding \{\textit{Dir}=right\}. This results in a new tuple of actions \texttt{(move(0.5), turn(right, 0.2))}. The previous \texttt{move(0.5)} action is allowed to continue unchanged but control action \textit{start(turn(right, 0.2))} will be executed.

\textit{BeliefStore} will typically soon become

\[
\begin{align*}
\{ & \text{next\_to(bottle, centre), gripper\_open} \}
\end{align*}
\]

At that point the parent call \textit{get\_bottle} of \texttt{get\_next\_to(bottle)} will fire a different rule. This will switch from having fired its third rule to firing the second of its four rules

\[
\begin{align*}
\textit{holding} \sim (\) & \\
\text{next\_to(bottle, centre) & gripper\_open} \sim \textit{close\_gripper} & \\
\text{gripper\_open} \sim \textit{get\_next\_to(bottle)} & \\
\text{true} \sim \textit{open\_gripper}
\end{align*}
\]

This causes the call \texttt{get\_next\_to(bottle)} and any sub-procedure calls to be terminated, as well as any movement actions. Control actions \texttt{stop(move(0.5)), stop(turn(right, 0.2)), do(close\_gripper)} will be executed.
7.3 Informal Evaluation Algorithm

We can describe the evaluation cycle of a task executing a start TeleoR procedure call TaskCall with the following 8 step informal algorithm.

FrdRules is the set of indexed active procedure calls. Each element of FrdRules is a 4-tuple of the form \((Dp, \text{Call}, R, \theta)\) where \(Dp - 1\) is the number of intermediary procedure calls between Call and TaskCall, \(R\) is the number of the partially instantiated rule of the procedure for Call that was last fired, and \(\theta\) is the set of generated bindings for all the variables of the action of that rule. \(Dp\) is the index of the tuple.

1. \(LActs := \{\}; FrdRules := \{\}; Index := 1; Call := TaskCall.\)
2. \(\text{Index} > \text{MaxDp}: \text{Signal a call-depth-reached failure.}\)
3. Evaluate the guards for the rules for Call, in turn, to find first rule \(K \Rightarrow A\), number \(R\), with an inferable guard, with \(\theta\) being the first returned answer substitution for variables of \(eK\). Add \((\text{Index}, \text{Call}, R, \theta)\) to FrdRules.
   If there is no such rule signal a no-fireable-rule failure.
4. \(A\theta\) is a procedure call.
   \(Call := A\theta; \text{Index} := \text{Index} + 1; \text{Go to step 2.}\)
5. \(A\theta\) is a tuple of primitive actions.
   Compute controls \(CActs\) using actions of \(A\theta\) and Acts (See 7.2.1);
   Execute \(CActs\); Update \(LActs\) to set of actions in \(A\theta\).
6. Wait for a BeliefStore update. After update Index:=1;
   (Then check all the active calls to see if each previously fired rule instance should continue, beginning with initial TaskCall entry which has \(Dp = 1\).)
7. (Optimisation) We can determine that rule \(R\) of Call, where \((\text{Index}, \text{Call}, R, \theta)\) in FrdRules, must continue as the fired rule of Call with firing substitution \(\theta\), without re-evaluating guards.
   If Index = \#FrdRules goto step 6
   else Index := Index + 1; repeat step 7.
8. Otherwise, evaluate the guards for the rules for Call, in turn, to find first rule \( K \rightarrow A \), number \( R' \), with an inferable guard, with \( \theta' \) being the first returned answer substitution for variables of \( eK \).

(a) If there is no such rule, signal a **no-fireable-rule** failure.

(b) If \( R' = R \) and \( \theta' = \theta \) (still have same rule firing for Call):
    
    If \( \text{Index} = \# \text{FrdRules} \) goto step 6
    
    else \( \text{Index} := \text{Index} + 1; \) goto step 7.

(c) If \( R' \neq R \) or \( \theta' \neq \theta \) (new rule firing for Call)
    
    \( \text{FrdRules} := \{ (Dp,N,C,\psi) \mid (Dp,N,C,\psi) \in \text{FrdRules} \land Dp < \text{Index} \} \)
    
    \( \cup \{ (\text{Index}, \text{Call}, R', \theta') \}; \) Go to step 4.

The call tuple with index \( Dp + 1 \) is the offspring of the call tuple with index \( Dp \) and the Call component of the \((Dp + 1)\)th tuple is the fully instantiated procedure call action of the rule fired in the \( Dp \)th tuple.

Steps 1 to 5 of the algorithm will effectively generate a call stack of procedure call descendants of TaskCall using the initial state of the BeliefStore. The last entry in FrdRules with index \( \# \text{FrdRules} \) when step 5 is executed will record the firing of a rule with primitive actions, control actions for which will be executed by step 5.

MaxDp is the maximum number of allowed active calls. Acts is the last tuple of determined actions for TaskCall, initialised to \( () \). Step 1, followed by an iteration of steps 2 to 4, generate the initial set of FrdRules for TaskCall for the initial state of the BeliefStore. This initial iterative generation terminates successfully when a rule is fired with a tuple of primitive actions \( A\theta \). \( LActs = \{ \} \) and the set of actions of \( A\theta \) are then used to generate the initial control actions CActs that are executed in step 5. \( LActs \) is updated to the new set of actions.

The algorithm then suspends at step 6 until the BeliefStore is updated. It then sets Index to 1 and checks each tuple in FrdRules one at a time, starting at the initial entry with \( Dp = 1 \), to see if a different rule, or a different instance of the same rule, should be fired. Step 7, occasionally augmented with step 8(b) when guards of rules need to be re-checked, is repeated until there is a change of rule firing. Step 7 on its own is an optimisation that uses information about which percepts have been changed as discussed in Section 7.8. The 7, 8(b) iteration terminates when either
Index = \#FrdRules, or step 8(c) finds that there should be another rule firing for some call, or for some call there is now no rule that can be fired (an error condition).

Note there is a new rule firing when a different rule is fired, or the same rule is fired with a different set of bindings \( \theta' \) for the variables of the rule’s \( eK \), resulting in different action \( A\theta' \). The Index entry of FrdRules is then replaced and all entries on FrdRules above this entry are discarded. The algorithm then switches into the steps that add new entries into FrdRules when we have a sequence of new rule firings with procedure call actions (iteration of steps 4, 2, 3). This continues until a rule is fired with primitive actions (step 5) and new control actions are executed. It then re-suspends at step 6 until there is another BeliefStore update.

In other words, on each BeliefStore update either there is no change to FrdRules, or some entry is replaced by a new tuple which records the firing of a primitive actions rule, or a new sequence of tuples are added recording the firing of rules with procedure call actions until a last entry is added that fires a rule with primitive actions. Whenever there is a change of FrdRules new control actions are computed and executed unless one of the two possible error conditions occurs.

We have step 7, that explicitly tests whether the previously fired rule of Call must continue. We shall sometimes be able to determine that this is the case without re-evaluating rule guards of its procedure call. This is an optimisation, as will be discussed in Section 7.8. It also allows us to use essentially the same algorithm for the new forms of rules we shall introduce in Chapter 10. These have their own special continuation tests as given in Chapter 11.

7.3.1 Example evaluation revisited

Revisiting our example evaluation of the previous section, we have

\[
\text{Belief Store} = \{\text{see(bottle, left), gripper\_open}\}.
\]

We start the algorithm at step 1 making

\[
LActs = \{}, FrdRules = \{}, Index = 1, TaskCall = \text{get\_bottle}.
\]

Let us assume we have \( MaxDp = 20 \). We skip to step 3 since \( Index < 20 \) and find that rule 3 of the get\_bottle call is the first rule with an inferable guard, giving \( R = 3, \theta' = \{\} \). \( FrdRules := \{(1, \text{get\_bottle}, 3, (())\} \). As the ac-
tion of rule 3 is procedure call $get\_next\_to(bottle)$, $Index$ is incremented to 2, $Call$ is changed to $A\theta$. The algorithm loops back to step 2 and immediately skips to step 3.

Step 3 finds that the 6$^{th}$ rule of the call $get\_next\_to(bottle)$ for the current $BeliefStore$ is the first with an inferable guard with answer bindings {}.

$(2, get\_next\_to(bottle), 6, ())$ is added to $FrdRules$ which is now

$$\{(1, get\_bottle, 3, ()), (2, get\_next\_to(bottle), 3, ())\}.$$  

The fired rule action is procedure call $approach(bottle, 1.5, 0.1)$ so step 4 makes this the value of $Call$ and increments $Index$ to 3. The algorithm again loops back to step 2 and immediately skips to step 3.

At step 3, because we have $see(bottle, left)$ in $BeliefStore$, the second rule of the $approach(bottle, 1.5, 0.1)$ call is the first with an inferable guard so $R = 2, \theta = \{Dir = left\}$. $(3, approach(bottle, 1.5, 0.1), 2, \{Dir = left\})$ is added to $FrdRules$. The instantiated action of the rule is $move(1.5), turn(left, 0.1)$, a tuple of primitive actions.

At step 5, $CActs = \{start(move(1.5)), start(turn(left, 0.1))\}$ is computed from $\{move(1.5), turn(left, 0.1)\}$ and $LActs = \{\}$, and executed. $LActs$ is updated to $\{move(1.5), turn(left, 0.1)\}$. We move to step 6 to wait for a $BeliefStore$ update.

On the next $BeliefStore$ update, say to

$$BeliefStore = \{see(bottle, centre), gripper\_open\}$$

the algorithm checks each rule firing in $FrdRules$, which is

$$\{(1, get\_bottle, 3, {}), (2, get\_next\_to(bottle), 3, {}), (3, approach(bottle, 1.5, 0.1), 2, \{Dir = left\})\}$$

It starts with the first firing in $FrdRules$ as $Index$ is reset to 1. It checks if the 3$^{\text{rd}}$ fired rule of the call $get\_bottle$ should continue as its fired rule.

This re-test will discover that this is the case. This will either be because step 8(b) confirms this, or because of optimisations we shall discuss in Section 7.8 allowing us to avoid re-evaluating rule guards for the call. $Index$ is increased to 2 and the algorithm re-applies step 7.

The same will happen with the checking of the rule firing for the second call $get\_next\_to(bottle)$ followed by another repetition of step 7 with $Index = 3$. This time, since we have $see(bottle, center)$ in $BeliefStore$, the 2$^{\text{nd}}$ rule of the $approach(bottle, 1.5, 0.1)$ call will not continue as its fired rule, and step
8 will determine that its 1st rule is the first one with an inferable guard. We have, $R' = 1 \neq R$, so $(3, \textit{approach}(\textit{bottle}, 1.5, 0.1), 2, \{\text{Dir} = \textit{left}\})$ is replaced by $(3, \textit{approach}(\textit{bottle}, 1.5, 0.1), 1, \{\text{Dir} = \textit{center}\})$ and we jump back to step 4, immediately skipping to step 5 as we do not have a procedure call action. We have the resource action $\textit{move}(1.5)$ of the new rule firing.

$CActs = \{\textit{stop}(\textit{turn}(\textit{left}, 0.1))\}$ is computed from the action $\textit{move}(1.5)$ of rule 1 and $LActs = \{\textit{move}(1.5), \textit{turn}(\textit{left}, 0.1)\}$. $CActs$ is executed and $LActs$ becomes $\{\textit{move}(1.5)\}$. We move to step 6.

The informal algorithm leaves unspecified the precise logical conditions that determine whether a rule may \textit{continue} as a call’s fired rule. The formal semantics of the next section makes this precise. The formal semantics will also distinguish between the two cases dealt with by step 8(c). When $R' \neq R$, this a new rule \textit{firing}. When $R' = R$ but $\theta \neq \theta'$, this a \textit{re-firing}. The cases dealt with by steps 7 and 8(b) are a \textit{continuation} of the fired rule.

The formal semantics will also abstract from the details of the updating of $\textit{FrdRules}$. It will logically define the relationship between the tuples in $\textit{FrdRules}$ before and after a $\textit{BeliefStore}$ update. Every such update of $\textit{BeliefStore}$ will be deemed to determine a new state of evaluation even if $\textit{FrdRules}$ remains unchanged after the update - a situation handled by the jumps back to step 6 from steps 7 and 8(b) of the algorithm. This happens when the last entry of $\textit{FrdRules}$ has been checked and no change of a fired rule has been found for any entry.

7.4 Re-firing When Previous Guard Instance is Still Inferable

We allow a rule to be re-fired if its guard query to the $\textit{BeliefStore}$ re-turns a different first solution $\psi$. This may happen even when the instance $eK_i\theta$ of the previous firing is still inferable.

As an example of how this can happen consider the following $\textit{BeliefStore}$ rules where bottles have different colours and the colour and the used vision percepts are defined in terms of $\textit{see\_patch}$ (see Section 3.6). The bottle colour, and that of the drop, is also an argument to $\textit{approach}$.
This is part of a bottle collecting program where there is a premium on collecting blue bottles. It is a little artificial but will illustrate the point.

Let us suppose that after a percepts update \( \text{see} \text{patch}(\text{green}, 80, \text{left}) \) is the only \( \text{see} \text{patch} \) fact in the BeliefStore. The TeleoR rule will be fired with green as the colour of bottle to be approached. Suppose that while the green bottle is being approached a blue bottle is put down within sight of the robot and \( \text{see} \text{patch}(\text{blue}, 55, \text{right}) \) is added to the BeliefStore on a subsequent percepts update. The patch size is 55 as the blue bottle is further away. \( \text{see} \text{patch}(\text{green}, 90, \text{left}) \) also replaces \( \text{see} \text{patch}(\text{green}, 80, \text{left}) \) as the green bottle, a little closer, is still in sight as well.

The TeleoR rule will be re-fired switching the robot from approaching the green bottle to approaching the more distant blue bottle that has just appeared, but \( \text{see} \text{patch}(\text{green}, \_ , \_ ) \) will still be inferable. This is an example of where we have used BeliefStore fact/rule ordering to implement a choice preference that can cause a change of behaviour when new percepts arrive, just as TeleoR rule ordering reflects action preference that can and often does affect behaviour when new percepts arrive.

In [91] TeleoR rule re-firing on BeliefStore updates was not mentioned and so the issue of whether or not a rule should be re-fired when the previously fired guard instance was still inferable was left ambiguous. In the earlier papers [89],[88], which did not have BeliefStore rules, the informal semantics for TeleoR procedures was based on the concept of analogue inputs and outputs. Re-firing a rule when the previous guard instance still held was not possible. There could only be one analogue input at any time.

We made the choice to allow re-firing as the default, even when the previous guard instance was still inferable, so that BeliefStore rules could be used to express behaviour preferences, as in this example. However, there are situations where one would like to stick with the previously determined action when it is still an option, especially when this is a procedure call that may have acquired resources in a multi-tasking multi-resource using agent. We will return to the question in Section 17.2 when discussing future
7.5 Fire, Continue, Re-fire Semantics of TeleoR

To formalise the operational semantics the first thing we need to do is to more precisely define the conditions for when a rule of a TeleoR procedure call is the fired rule of the procedure call \( P \), preferably without explicitly referring to rule ordering. We should also be more precise about the condition which allows the rule to continue being the fired rule. Finally, we need to specify when it may be re-fired.

We assume a function \( BS \) such that \( BS_T \) is the BeliefStore state at time \( T \). The state changes when new percepts arrive, which can happen as frequently as every few hundredth’s of a second, and not necessarily at fixed intervals. We will define the following:

\[
\text{fire}(P,R,\theta,T_F) - K \rightarrow A, \text{ the } R^{th} \text{ rule of called procedure } P, \text{ is the fireable rule of } P \text{ because rule } R \text{ is the first rule with an inferable guard from BeliefStore state } BS_T, \text{ and } eK\theta \text{ is the first inferable instance of its existentially quantified and partially instantiated guard } eK \text{ using } BS_T.
\]

\[
\text{continue}(P,R,\theta,T_F,T_C) - K \rightarrow A, \text{ the } R^{th} \text{ rule of called procedure } P, \text{ which was its fireable rule at time } T_F, \text{ and has been its fireable rule for every } T \text{ in } T_F < T \leq T_C, \text{ with the same } \theta \text{ bindings.}
\]

\[
\text{refire}(P,R,\theta,\psi,T_F,T_{RF}) - K \rightarrow A, \text{ the } R^{th} \text{ rule of called procedure } P, \text{ was its fireable rule at time } T_F \text{ and has been its fireable rule for every } T \text{ in } T_F < T < T_{RF} \text{ with the same } \theta \text{ bindings. It is still its fireable rule at time } T_{RF} \text{ but } eK\psi, \psi \neq \theta \text{ is now the first inferable instance from } BS_{T_RF}.
\]

These definitions will apply to any standard TeleoR rule, whether its action is a TeleoR procedure call, or one or more resource actions that can be discrete, durative, or a mixture of the two.
7.5.1 The formal definitions of fire, continue and refire

Some notation

\( BS_T \vdash_f C\theta \) means the ground instance \( C\theta \) is inferable from \( BS_T \) with \( \theta \) the \textit{first} returned set of bindings for its variables using the QuLog query evaluator of Section 4.

\( P_R \) is the partially instantiated \( R^{th} \) rule of a procedure \( P \) for a call \( P \).

\( P_{i,R_i} \) is the partially instantiated \( R^{th} \) rule of a procedure call \( P_i \).

\( eguard(P_R) \) is the guard \( eK \) of a rule \( P_R=K\rightarrow A \) existentially quantified with respect to its variables not in \( A \).

\( \theta \) and \( \psi \) each give variable free values for all the free variables of \( eK \).

\( \equiv \) may be read as \textit{is defined as}.

\( \prec \), read as \textit{immediately precedes}, is the relation between successive different BeliefStore states. It is defined by

\[
BS_{T'} \prec BS_T \equiv T' < T \land BS_T \neq BS_{T'} \land \neg \exists T'' (T' < T'' < T \land BS_{T'} \neq BS_{T''})
\]

Finally we define an auxiliary relation \textit{no higher fireable rule}(\( P, R, T \)) which means that no rule before rule \( R \) in the sequence of rules of call \( P \) has an inferable guard at time \( T \).

\[
\text{no higher fireable rule}(P, R, T) \equiv \forall R' ( 1 \leq R' < R \Rightarrow \neg \exists \theta' BS_T \vdash eguard(P_{R'})\theta' )
\]

We can now give the formal definitions of fire, continue, refire.
\[ P_R = K \rightarrow A \]

\[
\text{fire}(P, R, \theta, T_F) \equiv BS_{T_F} \downarrow eK\theta \land \text{no\_higher\_fireable\_rule}(P, R, T_F)
\]

\[
\text{continue}(P, R, \theta, T_F, T_C) \equiv \forall T(T_F \leq T \leq T_C \Rightarrow \text{fire}(P, R, \theta, T))
\]

\[
\text{refire}(P, R, \theta, \psi, T_F, T_{RF}) \equiv \forall T(T_F \leq T < T_{RF} \Rightarrow \text{fire}(P, R, \theta, T)) \land \text{fire}(P, R, \psi, T) \land \theta \neq \psi
\]

Note that for any procedure call \( P \) there will be at most one fireable rule, and there may be none. Also note that \textit{continue} and \textit{refire} are defined over an interval which starts with a time at which the rule was the fireable rule of its procedure call \( P \) and is such that the rule continued to be the call’s fireable rule up to the interval end. In the case of \textit{continue} it was also the fireable rule at the end point \( T_C \) with the same substitution \( \theta \) for its action variables, whereas in the case of \textit{refire} there was a different firing substitution at the end point \( T_{RF} \). This is what makes it a \textit{refire}.

Because both \( \theta \) and \( \psi \) are only substitutions for the variables of \( eK \), which are the variables of the partially instantiated rule’s action \( A \), we have a re-fire only when \( \psi \) determines a different action \( A\psi \) to be executed. When \( A \) contains durative actions the currently executing durative actions of \( A\theta \) will be modified to have new argument values given by \( \psi \).

### 7.6 State Transition Semantics

We can now use these definitions to precisely define the state of a \textit{TeleoR} evaluation that begins with some call \texttt{SP} to a \textit{TeleoR} procedure. As is usual when giving this type of semantic definition, we shall define the state as some tuple of values which includes everything we need to have represented and stored in some implementation of the evaluation, or which the implementer needs to know about.

In the next section, 7.7, we give an example three step evaluation of an initial call to \texttt{collect\_bottle} using the mathematical representation of the evaluation state. The fired rules used are those of the program of Chapter 3. You might find it useful to alternate between reading this quite mathematical section and the example evaluation.
7.6.1 Informal description of a TeleoR task evaluation state

The evaluation state at time $T$ for some TeleoR procedure call $SP$ started at $ST \leq T$, is an 8-tuple

$$(T, MaxDp, SP, ST, FrdRules, LActs, Acts, CActs)$$

with the following informal description that we shall then formalise.

- $MaxDp$ is the call depth limit.
- $FrdRules$ is sequence of $n, 0 \leq n \leq MaxDp$, of four-tuples $(P_1, R_1, \theta_1, T_1), \ldots, (P_i, R_i, \theta_i, T_i), \ldots, (P_n, R_n, \theta_n, T_n)$

  where each $P_i$ is a TeleoR procedure call, $P_i R_i = (K_i \rightarrow A_i)$ is one of the partially instantiated rules of $rules(P_i)$, that was its fireable rule at time $T_i$ with substitution $\theta_i$ for the variables of $A_i$, and continued to be the fireable rule of the call $P_i$ from $T_i$ up to time $T$.

- For each $i < n$ we always have the fully instantiated action $A_i \theta_i$ as the call $P_{i+1}$ of the next 4-tuple in the sequence. $P_1 = SP$ if $n > 0$.

- $LActs$ is the value of $Acts$ of the previous evaluation state. $LActs = \{\}$ if this is the start state.

- $Acts = A_n \theta_n$ when this is a tuple of resource actions, including $()$.

- $Acts = nfr_{-}fail$ when $A_n \theta_n$ is a procedure call or $n = 0$. If $n > 0$ there is no fireable for the procedure call $A_n \theta_n$ of the $FrdRules$ sequence at time $T$. If $n = 0$ there are no tuples in $FrdRules$, there is therefore no fireable rule for the task procedure call $SP$ at time $T$. (This is slightly different from the informal algorithm of Section 7.3 when we recorded a call with no fireable rule by having a tuple in its $FrdRules$ set with a fired rule number 0.)

- $Acts = md_{-}fail$ when $n = MaxDp$ and $A_n$ is a procedure call.

- $CActs$ is a set of control actions computed from the $LActs$ and $Acts$ sets sent at time $T_n$ to the robotic resource. $CActs = \{\}$ if $n = 0$ or $A_n \theta_n$ is a procedure call. $T_n$ is usually $T$, but will not be if no new rule was fired in any active procedure call at time $T$. 

114
• If Acts = nfr_fail or Acts = md_fail the state is a terminal error state. There is no next state. CActs is the set of control actions that would be computed from LActs and Acts = {}. CActs stops all durative actions on LActs.

Typically the tuples of a tail sub-sequence FrdRules will have the time \( T_n \) for their rule firing. If \( T_n = T \) this sub-sequence records the rules that were fired when the BeliefStore was updated to state BS\( T \). If \( T_n < T \) the last BeliefStore update did not result in a change of FrdRules and Acts. \( T_n \) then records the most recent time when a BeliefStore update did result in new rule firings.

We do not need to store the indexes in tuple in FrdRules as this is a sequence not a set as it was in the informal algorithm. Instead we store the time at which the rule became the fireable rule of the procedure call of the tuple, or the time that it was re-fired.

### 7.6.2 Formal specification of an evaluation state

**More notation**

If \( S \) is a sequence \( s_1, \ldots, s_n \), \( S_i \) is \( s_i \) and \( S_{i:j} \) is the sub-sequence \( s_i, \ldots, s_j \). \( \in \) can be used to access elements of a sequence or tuple.

For a TeleoR procedure headed \( P \), rules(\( P \)) is the sequence of partially instantiated rules \( K \rightarrow A \) of a call \( P \).

**Definition**

The evaluation state at time \( T \) for some TeleoR procedure call \( SP \) started at \( ST \leq T \), is an 8-tuple

\[
(T, MaxDp, SP, ST, FrdRules, LActs, Acts, CActs)
\]

where FrdRules is the sequence of \( n, 0 \leq n \leq MaxDp \), 4-tuples,

\[
(P_1, R_1, \theta_1, T_1), \ldots, (P_i, R_i, \theta_i, T_i), \ldots, (P_n, R_n, \theta_n, T_n)
\]

and for \( 1 \leq i < n, P_{i,R_i} = (K_i \rightarrow A_i) \) and \( P_1 = SP \) if \( n > 0 \).

It satisfies the six conditions 7.1 to 7.6 given below.
Conditions characterising a valid evaluation state

\[ \forall i : 1 \leq i \leq n \ fire(P_i, R_i, \theta_i, T_i) \quad (7.1) \]

\[ \forall i : 1 \leq i < n \ (A_i \theta_i = P_{i+1} \land T_i \leq T_{i+1} \leq T) \quad (7.2) \]

\[ \begin{align*}
A_i &= A_{i-1} \Box resActs(A_i) \land 0 < n \leq MaxDp \\
\lor & \quad \forall \ Acts = md\_fail \land procCall(A_n) \land n = MaxDp \\
\lor & \quad \ Acts = nfr\_fail \land \\
\exists P((n > 0 \land P = A_n \theta_n \land procCall(P) \\
\quad \lor n = 0 \land P = SP) \\
\quad \land \neg \exists(R, \theta)fire(P, R, \theta, T))
\end{align*} \quad (7.3) \]

Conditions constraining \( CA_i \)

\[ \begin{align*}
CA_i &= update(LActs, Acts) \land \\
\exists ET ((n > 0 \land ET = T_n \lor n = 0 \land ET = T) \\
\quad \land execAt(CA_i, ET)) \\
\quad \land \neg \exists(R, \theta)fire(P, R, \theta, T)))
\end{align*} \quad (7.4) \]

\[ \begin{align*}
update(LActs, Acts) &= update(LActs, \{\}) \Leftarrow \\
& \quad Acts = nfr\_fail \lor Acts = md\_fail
\end{align*} \quad (7.5) \]

\[ \begin{align*}
update(LActs, Acts) &= \\
\{ & \quad \text{stop}(a) \mid \text{dur}(a) \land a \in LActs \land \\
& \quad \neg \exists a'(a' \in Acts \land \text{mod}_of(a', a)) \} \cup \\
\{ & \quad \text{start}(a) \mid \text{dur}(a) \land a \in Acts \land \\
& \quad \neg \exists a'(a' \in LActs \land \text{mod}_of(a', a)) \} \cup \\
\{ & \quad \text{mod}(a', a) \mid \text{dur}(a) \land a \in Acts \land a' \in LActs \land \\
& \quad \text{mod}_of(a', a) \land a \neq a' \} \cup \\
\{ & \quad \text{do}(a) \mid \text{dis}(a) \land a \in Acts \} \Leftarrow \\
& \quad Acts \neq nfr\_fail \land Acts \neq md\_fail
\end{align*} \quad (7.6) \]

The first condition 7.1 tells us something about the history of each tuple in \( FrdRules \) when \( n > 0 \). It tells us that each \( R_i \) identifies one of the
partially instantiated rules of the TeleoR procedure call $P_i$ of the 4-tuple; and it tells us that that rule was the fireable rule of the call at time $T_i$ with it guard evaluation returning bindings $\theta_i$ for the local variables of the action of that rule.

The second condition 7.2 is the 4-tuple adjacency condition. It tells us how the FrdRules list is generated from SP. All but the last 4-tuple of the list must have a procedure call action for its fired rule instance and this is always the procedure call of the next tuple in the list. This next tuple has a rule firing no earlier than that of the preceding 4-tuple, and no later than $T$.

The third condition 7.3 gives the value of Acts relative to the FrdRules. It is the set of fully instantiated actions of $A_n\theta_n$ if this is a tuple of resource actions, tested by resActs. We have Acts = $md\_fail$ if $A_n\theta_n$, the action of the last 4-tuple, is a procedure call action, tested by procCall, and the maximum call depth $MaxDp$ has been reached. We have Acts = $nfr\_fail$ if $A_n\theta_n$ is a procedure call action and there is no fireable rule for this procedure call, or if FrdRuules is empty because at time $T$ there was no fireable rule for the task start call SP.

The remaining three conditions constrain the value of CActs relative to the values of LActs and Acts. Condition 18.7 tells us that the value of CActs is a set of control actions returned by the function update. The execAt(CActs, ET) condition means that the control actions are executed at the time ET. If there are fired rules, $n > 0$, this is time $T_n$ when the last 4-tuple was added to the FrdRules sequence. If $n = 0$, it is the time $T$ of the current state, as this will be the time that no rule could be fired for the task start call SP.

The last two conditions 7.5, 7.6 define the update function. The first gives its value for special failure indicating values of Acts. Together with condition 7.6 they tell us that when there is an error condition all durative actions are terminated.

Condition 7.6 deals with the general case. $dur(a)$ means $a$ is a durative action, $dis(a)$ that $a$ is a discrete action. The condition $mod\_of(a', a)$ means that the two actions have the same name, such as move, but as $a' \neq a$ they must have different arguments. The fact that there is a $do(a)$ control on CActs whenever a discrete action $a$ is on Acts, means that a discrete action is re-done each time it appears in Acts, even if it was also in LActs. It is not repeated if there is no change to FrdRules on a BeliefStore update.

The conditions leave unspecified the value of LActs. For this we need to be told what happens when one evaluation state transitions into the next evaluation state, and we need to know its value in the initial state.

117
Initial Evaluation State

If there is a fireable rule for \( SP \) using \( BS_{ST} \), the initial evaluation state is the 8-tuple

\[
(\text{ST}, \text{MaxDp}, SP, \text{ST}, ((SP, R_1, \theta_1, \text{ST}), \ldots, (P_{n_S}, R_{n_S}, \theta_{n_S}, \text{ST})), \{\}, \text{Acts}_{ST}, \text{update}(\{\}, \text{Acts}_{ST}))
\]

and \( n_S \geq 1 \). It is a failure state if the action of \( P_{R_{n_S}} \) is a procedure call or if \( n_S = \text{MaxDp} \). Note that the firing time for every rule is the start time \( \text{ST} \).

If there is no fireable rule for the task procedure call \( SP \) at \( \text{ST} \) the initial state is the failure state

\[
(\text{ST}, \text{MaxDp}, SP, \text{ST}, (), \{\}, nfr\_fail, \{\})
\]

If the initial state is not a failure state, the next state will materialise at time \( T_{\text{ST}+D} \) when the \( \text{BeliefStore} \) is changed from state \( BS_{ST} \) to \( BS_{ST+D} \) at time \( ST + D \). This will be generated from the initial evaluation state using the state transition rule below. Every evaluation state is generated from the initial state by a sequence of applications of the state transition rule.

7.6.3 State transition

We will make use of two auxiliary definitions enabling us to directly test tuples in \( \text{FrdRules} \) for continuation and firing.

\[
\text{contRI}( (P, R, \theta, T_F), T_C) \equiv \text{continue}(P, R, \theta, T_F, T_C)
\]

\[
\text{fireRI}( (P, R, \theta, T_F), T_F) \equiv \text{fire}(P, R, \theta, T_F)
\]

These will allow us to test if an entry of \( \text{FrdRules} \) records a rule instance firing that will continue as the call’s fired rule.

For a non-terminal evaluation state we have the transition rule
\( BS_T \prec BS_{NT} \Rightarrow \)
\[(T, MaxDp, SP, ST, FrdRules, LActs, Acts, CActs) \mapsto \]
\[(NT, MaxDp, SP, ST, NFrdRules, Acts, NActs, NCActs) \]
\(\wedge \)
\( n = \#NFrdRules \geq 0 \)
\(\wedge \)
\( \exists j : 0 \leq j \leq n \)
\([NFrdRules_{0:j} = FrdRules_{0:j} \wedge \]
\((j = n \lor \neg contRI(FrdRules_{j+1}, NT) \wedge \]
\(\forall i : 0 \leq i \leq j, contRI(FrdRules_i, NT) \wedge \]
\(\forall i : j + 1 \leq i \leq n, fireRI(NFrdRules_i, NT) \]
\]
\( \Rightarrow \) can be read as transitions to. It may be that the \( j + 1 \)th tuple has the same rule number \( R_{i+1} \) and is therefore also a re\( f \)ire. If \( j = 0 \) and \( n > 0 \) then we have a new rule firing for the task start call \( SP \). If \( j = 0 \) and \( n = 0 \) then for \( BS_{NT} \) there is no rule that can be fired for \( SP \).

The transition happens when the percept or message handling threads add and/or remove facts from the BeliefStore \( BS_T \) to generate the next BeliefStore state \( BS_{NT} \) as part of some atomic transaction at time \( NT \).

The new state records the update time \( NT \), and that the \( LActs \) component of the new state is the \( Acts \) value of the preceding state. The values of \( NActs \) and \( NCActs \), are determined by the conditions given above that formally characterise a valid evaluation state. The transition rule specifies the relationship between \( FrdRules \) and \( NFrdRules \).

It tells us that there will be some maximal index \( j \) in the list of \( FrdRules \) such that

\[ NFrdRules_{0:j} = FrdRules_{0:j} \]

where the continue condition holds at time \( NT \) for all the fired rules in this common initial sub-sequence which will be empty if \( j = 0 \). However, after that \( j \)th point, if there are any 4-tuples, \( NFrdRules \) contains tuples with newly fired rules, fired at transition time \( NT \) using the new \( BS_{NT} \) for guard inference, where the first newly fired rule is not a continuing rule of its procedure call \( P_{j+1} \). This is what tells us that \( j \) is the maximal index up to which the continue condition for the new \( BS_{NT} \) holds for the previous \( FrdRules \) entries.
That each of the newly fired rules from the $j^{th}$ entry on will satisfy the adjacency condition $A_i \theta_i = P_{i+1}$ follows from the conditions characterising a valid evaluation state.

The declarative specification of the relationship between the new and old evaluation states does not tell us how to generate the new state from the old. That is the role of the algorithm of Section 7.3. That the algorithm always starts checking the FrdRules tuples from the one with index 1, effectively applying the continue condition to each one in turn, guarantees that it will find the maximal $j$ up to which the current fired rule remains the fired rule of its procedure call.

Note there is no connection in the state transition semantics between the new control actions $NCActs$ and the next update of the BeliefStore that will trigger the next transition. Indeed, the next BeliefStore update may not be the result of the control actions. Actions by other agents or nature may be the cause of the next state transition.

This makes the operational semantics of a TeleoR task very different from the operational semantics of a conventional programming language with actions that directly update the data state. This data state comprises the values of all the variables and the state of any current objects. The next action to be executed in such a language is also determined by the control flow of the program. It is not determined by decision rules that can inspect the entire data state.

Having no direct connection between an action, the next state, and the next action, makes TeleoR a very unusual programming language, as we mentioned in the Preface. We can think of a resource action, even the () action, as being highly non-deterministic in its effect. We should write our TeleoR programs to cope with every contingency.

### 7.7 An Example Three Step Abstract Evaluation

The example we will use is a start procedure that calls $collect\_bottle$ at time $st$. Let us suppose $BS_{st}$ comprises the percept facts

$$\{see(bottle, left), gripper\_open\}$$

and the maximum call depth is 100.
\[ BS_{st} = \{ \text{see}(\text{bottle}, \text{left}), \text{gripper\_open} \} \]

Start state at \( st \):

\[
(st, 100, \text{collect\_bottle}, st, \ % \text{time, call limit, start call, start time}
\]
\[
( (\text{collect\_bottle}, 3, \}, st), \ % \text{FrdRules sequence}
\]
\[
(\text{get\_bottle}, 3, \}, st),
\]
\[
(\text{get\_next\_to\(\text{bottle}\)}, 6, \}, st),
\]
\[
(\text{approach}\(\text{bottle}, 1.5, 0.1\), 2, \{\text{Dir} = \text{left}\}, st)
\]
\)
\[
\}, \ {\text{move}(1.5), \text{turn}(\text{left}, 0.1)}, \ % \text{Last \& new acts \& controls}
\]
\[
\{\text{start}(\text{move}(1.5)), \text{start}(\text{turn}(\text{left}, 0.1))\}
\]

is the initial evaluation state. Now suppose that two seconds later \( BS_{st} \)

is updated to \( \{\text{see}(\text{bottle}, \text{centre}), \text{near}(\text{bottle}, \text{centre}), \text{gripper\_open}\} \). The

next state of evaluation is

\[
BS_{st+2} = \{\text{see}(\text{bottle}, \text{centre}), \text{near}(\text{bottle}, \text{centre}), \text{gripper\_open}\}
\]

State at \( st+2 \):

\[
(st + 2, 100, \text{collect\_bottle}, st,
\]
\[
( (\text{collect\_bottle}, 3, \}, st),
\]
\[
(\text{get\_bottle}, 6, \}, st),
\]
\[
(\text{get\_next\_to\(\text{bottle}\)}, 5, \}, st + 2),
\]
\[
(\text{approach}\(\text{bottle}, 0.5, 0.2\), 1, \}, st + 2)
\]
\)
\[
\}, \ {\text{move}(1.5), \text{turn}(\text{left}, 0.1)}, \ {\text{move}(0.5)},
\]
\[
\{\text{mod}(\text{move}(1.5), \text{move}(0.5)), \text{stop}(\text{turn}(\text{left}, 0.1))\}
\]

For the 1\text{st} and 2\text{nd} \text{FrdRules} tuples the \text{continue} condition for the last fired

rule holds, so the split point is \( j = 2 \). From there on the list is regrown with

newly fired rules at time \( st + 2 \).

Finally, let us suppose that 3 seconds later there is another percepts up-

date of \text{BeliefStore} to \( \{\text{see}(\text{bottle}, \text{right}), \text{close\_to}(\text{bottle}, \text{right}), \text{gripper\_open}\} \).
\[ BS_{st+5} = \{ \text{see}(\text{bottle}, \text{right}), \text{close\_to}(\text{bottle}, \text{right}), \text{gripper\_open} \} \]

State at \( st+5 \):

\[
(st + 5, 100, \text{collect\_bottle}, st, \\
( \text{collect\_bottle}, 3, \{ \}, st), \\
( \text{get\_bottle}, 6, \{ \}, st), \\
( \text{get\_next\_to\( (\text{bottle}) \)}, 4, \{ \}, st + 5), \\
( \text{approach}(\text{bottle}, 0.2, 0.2), 2, \{ \text{Dir} = \text{right} \}, st + 5) \\
), \\
\{ \text{move}(0.5) \}, \{ \text{move}(0.2), \text{turn}(\text{right}, 0.2) \}, \\
\{ \text{mod}(\text{move}(0.5), \text{move}(0.2)), \text{start}(\text{turn}(\text{right}, 0.2)) \}
\]

Again the split point is \( j = 2 \). Up to that entry in the \( FrdRules \) list the \( continue \) condition for each fired rule holds. After that point new rules are fired at time \( st + 5 \).

### 7.8 Necessary Conditions for Re-evaluation of Rule Guards

The agent’s \( BeliefStore \) comprises a fixed set of rules and facts and changing facts for the sensor percepts and other dynamic beliefs. These other beliefs are beliefs that are added as a result of messages that have been received by the agent’s message handler, or beliefs added to the \( BeliefStore \) as a result of firing a \( TeleoR \) rule.

Our \( TeleoR \) rule evaluator knows which \( BeliefStore \) predicates may have been queried in order to find the first fireable rule of a procedure call. It also knows whether the adding or removal of a fact for each of these dynamic predicates, \( may \) result in the guard of the last fired rule having a different or no inferable instance, \( or \) in the guard for one of the rules above the last fired rule having an inferable instance. Only if one of these specific types of updates happens in the \( BeliefStore \) need the guards of the rules for the procedure call to be re-evaluated. If there has been no such change the same rule with the same action result would be the fired rule - the \( continue \) condition for the last fired rule will still be true.

As an example, suppose that in the procedure
get_bottle{
    holding ~> ()
    next_to(bottle,centre) & gripper_open ~> close_gripper
    gripper_open ~> get_next_to(bottle)
    true ~> open_gripper
}

the third rule has been fired. The adding of one of holding, gripper_open
or a next_to fact may make one of the earlier guards inferable, and the
removal of gripper_open will make the guard of the fired rule fail.

A dynamic predicate is prefixed with ++ if the addition of a fact or
rule for the predicate might falsify the continue condition, and with -- if
it is the removal that will have this effect. The same predicate may have
both a ++ and a -- prefix. These ++ -- annotated predicates are the local
dependent predicates for the procedure call. The union of these for each
active procedure call gives us the dependent predicates of the current state
of a TeleoR evaluation.

So the local dependent predicates for the call to get_bottle that has
fired its third rule is the list

[++holding, ++gripper_open, ++next_to, --gripper_open]

The local dependent predicates of the call to

collect_bottle{
    next_to(drop,_)&next_to(bottle,_)&gripper_open ~> ()
    holding ~> deliver_bottle
    true ~> get_bottle
}

which will have fired its last rule, will be

[+holding, ++gripper_open, ++next_to]

The dependent predicates of the current state of the evaluation will there-
fore be

123
[++holding, ++gripper_open, ++next_to, --gripper_open]

together with the local dependent predicates of the call `get_next_to(bottle)`
and its call to `approach`.

When the `BeliefStore` gets updated, the evaluator only re-checks rules
of any procedure call if one or more of these dependent predicates has been
updated, in its prefix specified way, since the last time the rule firings were
checked. This is because, without such an update, every procedure call
would re-fire the same rule, with the same action result - the `continue` test
for the last fired rule of `every` procedure call will still be true.

When there is such an update, rules are re-checked. But the re-rechecking
of rules is restricted to those procedures calls which have had one of their
local dependent predicates updated in its ++ -- specified way. More than
that, it will only start retrying the partly instantiated guards of the `TeleoR`
rules of these procedure calls from the first guard that queries one of the
local dependent predicates that has been updated in such a way that the
previous evaluation of the guard may have changed. This is similar in intent
to, but much simpler than, the Rete algorithm [52] for efficient re-checking
of rules of a production rule system in which rules are typically fired to
update some problem state with conditions that are tests on that problem
state maintained inside the problem solving agent.

To make this concrete, suppose that for our example call to `get_bottle`
which has fired the third rule the `BeliefStore` has been updated by the adding
of a `next_to` fact, i.e. the recorded changes are just `++next_to`. This in-
cludes one of the changes of the list of local dependencies

[+next_to]

for the procedure call, so its rule guards will be retried. However they need
only be retried from rule 2 as there is no overlap between the list of changes
`[+next_to]` and the changes `[+holding]` that would make the guard of
rule 1 inferable.

For some of the forms of `TeleoR` rule that we shall discuss in Chapter 10,
the check that there has not been a particular type of dynamic belief update
to one or more locally dependent predicates will be replaced or augmented
by other necessary conditions that must be satisfied before we re-test the
guards of certain rules of a procedure call. These are always checks that tell
us that the continuation condition of its last fired rule `must` still hold, hence
that we need not, or `should not`, re-evaluate certain guards of the procedure
call’s `TeleoR` rules.