Imperial College of Science, Technology and Medicine
Department of Computing

Automatic Verification of Teleo-reactive Programs Using a Model Checker

Author:
Kaho Sato

Supervisor:
Alessio Lomuscio

Abstract

A *teleo-reactive program* is a formalism to define an autonomous agent. Though it was coined over two decades ago, the amount of work that has been done in its verification is scarce, and there is no work done in its automatic verification. This report presents a methodology to automatically verify a program written in a teleo-reactive programming language, *TeleoR*, using a model checker *MCMAS*. This involves defining a mapping from TeleoR programs to interpreted systems, and building a compiler which performs this translation.

Acknowledgements

I would like to first thank my supervisor, Prof. Alessio Lomuscio for his technical and mental support throughout the project.
I would also like to thank my second marker, Dr. Krysia Broda for her enthusiasm, and Peter Robinson for having kindly assisted me with understanding TeleoR.
I wish to take this opportunity to thank everybody who supported me throughout these four rough years at Imperial. It is impossible to make a full list of people who encouraged me when I was deprived of confidence. However my special thanks goes to:

- Dr. David Ham, the best personal tutor one could ever wish for,
- Davide Cultrera, a family away from home,
- Yujun Tan, a miracle who manages to put up with me, and
- my family back in Japan, for their unconditional love.

Contents

List of Figures

Chapter 1

Introduction

Autonomous agents are agents that can make decisions on their own, about their action, without interaction from other systems. Examples can be easily found in science fantasy movies amongst which, the most famous would be the charming, C-3P0 from Star Wars. Although our technology is not quite at the stage where such sophisticated robots can be realised, work on autonomous agents continues to be an active area of research.

A Teleo-reactive program is a formalism which specifies autonomous agents using a set of rules, first presented by Nils Nilsson[1]. These rules have a form of implication, where a condition on perceptual inputs and data is at the head, and an action is at the body. For instance, a rule such as *low_battery* $\sim>$ *recharge* specifies that the agent should go and charge itself when its battery is low. To react to the dynamically changing environment, a program continuously reevaluates the triggering conditions and the rule of the highest priority whose condition is satisfied, gets applied.

In general, it is a keen interest of developers to verify that a program achieves what they intended it to achieve. Safety-critical systems are more and more ubiquitous in our life, and they are dangerously more complex. In fact, there have been many unfortunate incidents triggered by a bug in software, some of which costed human lives[2]. However crucial, it is time-consuming for a human to reason about or test programs, and as we all know too well, it is almost impossible to be completely thorough with a complex system.

Teleo-reactive programs greatly benefit from verification for many reasons. For starters, being an autonomous agent, it might be deployed in an unsupervised and remote environment. It is hard to thoroughly reason about teleo-reactive programs that are deployed in a dynamic environment where little can be assumed[1]. Moreover to add to the complexity, agents could be specified so that a multitude of them collaboratively try to attain a goal[3]. Despite this, there has been almost no work done in verification of teleo-reactive programs.

We wanted to approach this problem using *formal methods*, which use symbolic structuring and formal logic. Formal methods are increasingly popular both in industry and in academia as it is guaranteed to be correct and complete, if used appropriately. One of the popular formal methods is model checking. Model checking involves translating a given system into a model and a given property that needs to be proven on the system, into a formula. Once translated, the next step is to check that that formula is true when evaluated on the model.

This project presents a novel mechanism to automatically verify teleo-reactive programs. It is a compiler, *TRTIS*, which takes a teleo-reactive program and outputs a model which can be automatically verified by a model checker. Specifically, we look at programs defined in *TeleoR*, a teleo-reactive programming language developed by Clark and Robinson[3]. For model checking to potentially support multi-agent scenarios in the future, we use MCMAS, a model checker that is specialised in verification of multi-agent systems.

1.1 Objectives

The goal of this project is to develop a mechanism to automatically verify a teleo-reactive program using a model checker. It demonstrates the versatility of model checking as a means to verify a program, especially because teleo-reactive programming is completely different from other programming paradigms. Our objectives to achieve this goal are the following:

1. **Research teleo-reactive programs.**
   The first objective of this project is to explore teleo-reactive programming and its modern implementation, TeleoR. We do this to have a context needed to understand what the properties of interest would be, and consider the potential difficulties in its verification. This was necessary to formulate a more concrete scope of the project.

2. **Formulate the mapping from TeleoR programs to an interpreted system.**
   To use a model checker to verify a system, we need to represent it as a model that the checker can run on. As we wanted to use MCMAS, which requires the system to be expressed as an interpreted system, we had to come up with a mapping from TeleoR program to interpreted system.

3. **Devise a compiler which translates TeleoR programs into ISPL**
   Our aim is to *automatically* verify a teleo-reactive program. MCMAS takes an interpreted system specified in ISPL and automatically verifies a set of given properties. Therefore what we were left to automate, is the process of describing a TeleoR program in ISPL, thus decided to write a compiler.

1.2 Challenges

This project was challenging for various reasons:

- **Almost no previous work in verification of teleo-reactive programs.**
  Though teleo-reactive program has been studied by many[4], there has not been much work in verification of teleo-reactive programs. This meant that there was no resource that we could directly refer to. This made finding a good starting point incredibly difficult when scoping the project.

- **Not much formal source available.**
  In order to express a system as a formal model, it is critical to know how *exactly* the system behaves. Unfortunately, except for a subset, we did not have a formal description of how TeleoR programs are executed until the very end of the project, which limited what could be accomlished.

- **Young yet complex enough language without many references.**
  TeleoR was first presented in 2014[5]. It is a very young language with many features to enable a great expressivity to the developers of robotic agents. Unfortunately, not many examples or documentations are available, which made it hard to understand even the most basic things, such as how to launch an agent or how percepts are communicated.

- **Different temporal semantics.**
  The action of TeleoR agent is durative, meaning that it lasts over a period of time. This implies that the temporal semantics of the agent must be continuous and can not be expressed by an interpreted system, which assumes a discrete state transition. We realised that by only considering which action is chosen at a given time we can reason about the system with a discrete state transition. However we did spend much time to make this shift and convince ourselves that it was the right thing to do.

- **Unspecified environment.**
  In TeleoR program, we know how the agent behaves given what the agent knows about the environment. However we do not know how the environment changes, therefore we do not know how the knowledge of the agent changes. Without such information, we doubted whether it is possible at all to prove an interesting property of the agent. For example, how could we ever say that the agent reaches to a point A, if the environment could possibly be such that it always moves the agent back to where it started? This is overcome by allowing the user to input the assumption of the environment, as described in Chapter 4.

1.3 Contributions

We specified and implemented a compiler which translates TeleoR program into ISPL, which can then be fed to MCMAS. This is the first work done in automated verification of a teleo-reactive programs using a model checker, and we produced the following:

- **Identifying non-trivial problems in applying model checking to TeleoR programs.**
  Sometimes identifying the problems is harder than actually solving them. While researching TeleoR and determining the subset of TeleoR programs to verify, we recognised some non-trivial problems associated with applying model checking to verify a TeleoR program. We hope that by stating them, more work may be done in order to overcome them in the future.

- **Mapping from a subset of TeleoR programs to an interpreted system.**
  We describe one verifiable subset of TeleoR programs as an interpreted system. We also present the transition system which represents a subset of TeleoR program. We then mathematically prove that the interpreted system is equivalent to this transition system.

- **Methodology of automatic verification of a TeleoR program.**
  This project establishes a methodology of automatically verifying a TeleoR program; which is to define TeleoR program as a formal model, on which a model checker can be used, and devise a compiler which can automatically produce the specification of the model.

- **Algorithms to translate TeleoR program into an interpreted system.**
  We came up with an algorithm to translate components of TeleoR program into components of interpreted system. This may be beneficial to those who want to translate TeleoR programs into a formal model in general.

- **TRTIS - TeleoR To ISPL compiler**
  We specified and implemented TRTIS, a compiler which takes a TeleoR program and the assumption on the environment, and returns an ISPL file. As we see in Chapter 6, it can be used not only to verify programs, but also to debug them. Though admittedly the constraint on the input TeleoR program is fairly strong, we show that it does work with some programs, and that this is an effective methodology.

Chapter 2

Background

2.1 TeleoR

What are teleo-reactive programs?

The teleo-reactive program is a formalism for specifying the behaviour of autonomous agents. It was first presented by Nilsson together with TR language for its development in 1994[1]. Prefix “teleo” originates from a word “end” in Ancient Greek, and used in English as “end, purpose, or goal”. Together with the word “reactive”, it describes the core property of teleo-reactive program; it is a goal-directed program that continuously monitors the dynamic environment and reacts to it using its robotic resources.
The change in the environment is not entirely under the agent’s control. It may be caused by other entities such as human, and it could be in favour of the agent’s goal or otherwise. The teleo-reactive agents need to react to such a change quickly, by robustly recovering from the setback and opportunistically taking advantage of the favourable change by skipping some actions.[3]
Teleo-reactive agents may collaborate together to achieve a shared goal. Let us define *agent* to be an entity whose behaviour is specified by a single teleo-reactive program. Agents communicate each other via messaging and influence each other’s action. These multiple agents may belong to one physical robot, to multiple robots or both. Conversely, a single agent may work towards multiple goals, alternating from a task to another. Possible teleo-reactive robots may be:

- A drone, running one agent, that patrols over multiple blocks, one after the another, and reports any abnormality.
- An automated production process involving several independently usable controllable devices such as presses and robot arms. Each device is controlled by one agent, and sensor inputs are independent of the devices. They can collaborate in the means of messaging[6].
- An ambient intelligence consists of multiple nodes, each of them possibly running multiple agents, communicating with each other. This can be used for monitoring and controlling the environments[7].

Note that teleo-reactive program concerns how an agent reason about the environment and decides its next action. It assumes the mechanism of collecting the sensor input and interacting with the environment.

Disambiguation

Note the following differences:

- Teleo-reactive program is a formalisation for specifying an autonomous agent presented by Nilsson[1].
- TeleoR program is a program written in TeleoR, a teleo-reactive programming language presented by Clark and Robinson[3].
- TR program is a program written in TR, a teleo-reactive programming language presented by Nilsson[1].

Overview of TeleoR

TeleoR is a teleo-reactive programming language developed by Clark and Robinson[3]. Note that TeleoR and TR are two different teleo-reactive programming languages. TeleoR, similarly to TR language, specifies the decision-making logic of the agent using a set of TeleoR rules. These rules take the form of:

```
condition -> action
```

For instance, if we want to make the agent pick up a block when it sees one. Such a behaviour can be specified with a TeleoR rule as following:

```
see(block) -> pick_up
```

In TeleoR, the condition of the rules are checked against the deductive system called *BeliefStore*. *BeliefStore* contains facts about the environment, and inference rules to reason about more complex concepts. *BeliefStore* is defined using a logic programming language called QuLog. These two languages used to write a TeleoR program is much interwound with each other, for conciseness, when we discuss “TeleoR” we often mean “TeleoR and Qulog”. In the next subsection we see how components specified in two languages interact.
Clark et al.[6] lists few ways in which TeleoR extends TR language:

- Concurrent execution of primitive actions.
- The optional attaching of a BeliefStore update, a communication by means of messages, or a call to a Qulog 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.

In this project, we specifically look at TeleoR program and not other teleo-reactive programming languages.

Architecture

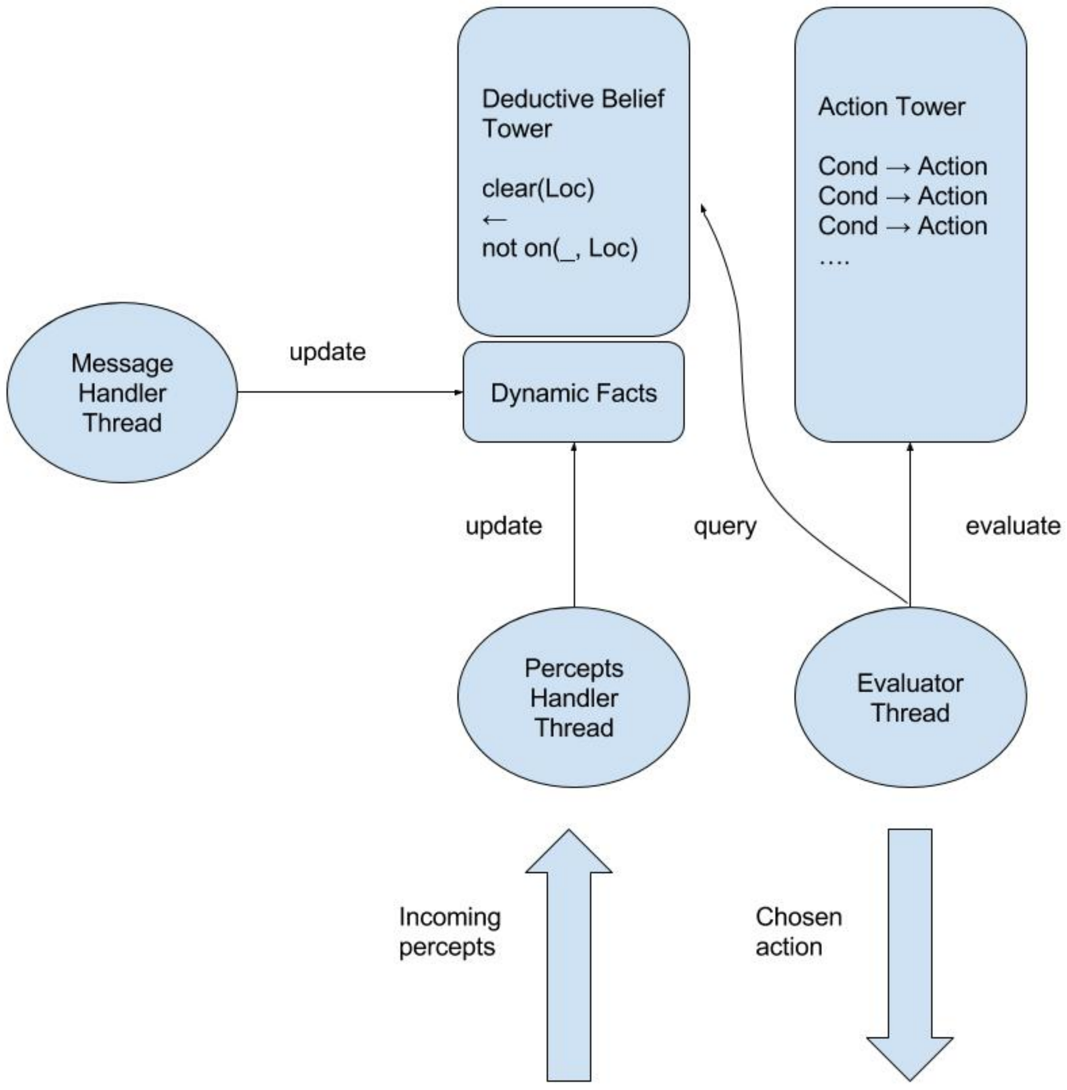

Figure 2.1: Two tower architecture of TeleoR agent.

What is described in a TeleoR program can be categorised into two components:

- BeliefStore
  BeliefStore represents the knowledge of the agent. It contains inference rules and fixed facts defined specified in TeleoR program using Qulog. They can be thought as what the agent knows about the world in general. *Dynamic facts* on the other hand are facts about the environment that the agent is situated in, which are frequently updated. Using this knowledge, the agent is able to reason about the surroundings.

- TeleoR task
  TeleoR task specifies the decision-making logic of the agent. The task is defined with *TeleoR rules*, which are evaluated against the knowledge of the agent, i.e. BeliefStore, and determines which action is chosen.

There are three kinds of threads which all together make up of an agent:

- Percepts handler thread
  This thread takes incoming percepts delivered from the sensors via *Pedro*, inter-agent and inter-process communication server, and atomically updates the dynamic facts in the BeliefStore accordingly.
- Message handler thread
  This thread takes delivered from the sensors via Pedro and atomically updates the dynamic facts in the BeliefStore accordingly.
- Evaluator thread
  This thread computes which action should be taken, by evaluating the TeleoR rules in the TeleoR task and querying the triggering conditions to the BeliefStore. It runs after each atomic update of the BeliefStore, so the agent reacts to a change in the environment and is executing the appropriate action at all times. The precise algorithm of how the thread determines which TeleoR rules are to fire is described later.

Note that there can be multiple threads of each category per agent. For instance, an agent could be executing multiple tasks in parallel, each of them having its own evaluator thread.

2.1.1 Basic Components in TeleoR

We had an overview on how a given TeleoR program is run. In this subsection, we introduce how components in TeleoR program may be defined. First we introduce some terminology often used in logic programming languages, including Qulog. For each component of TeleoR program, we first introduce the syntax, and present its semantics through examples.

Glossary of Terms

We introduce some terminologies used to describe TeleoR.

- Anonymous variable
  Anonymous variables are never bound to a value, and used to mean "any term". For instance, query $foo(\_)$ would succeed as long as foo with any value hold, and *not* $foo(\_)$ would only succeed if foo does not hold with any value.
- Ground term
  Ground terms are terms without any variable. Similarly, a term is *ground* if it does not contain any variable.
- Template term
  Template terms are terms which may contain variables[6].

- Binding
  Binding is a set of mapping from a variable to a value. In this report, we use θ to denote a binding.
- Instantiate
  One may *instantiate* a term using a binding by substituting the variable with a value that the variable is mapped to.

Relations

Relations are used in many things we described so far. For instance, in TeleoR dynamic facts are fully ground relational term. Inference rules are a relation definition for more complex concepts, in terms of other relations.
The syntax to define a relation in EBNF grammar is the follows:

```
relation_def = head "::" cond ("&" cond)* "<=" body ("&" body)*
```

This represents that, if all `cond` holds, `head` holds with values specified in `body` if not ground.
Relation can be seen as a Boolean function, which returns true if it does not fail infinitely and false otherwise (negation as failure) if all arguments are ground.
Example would be:

```
ancestor(Ans, Des) :: ancestor(Ans, X) & ancestor(X, Des) <=
ancestor(Parent, Child) :: parent_of(Parent, Child) <=
```

According to this definition, `ancestor(alice, bob)` holds if `alice` is a parent of `bob`, or there is a descendant of `alice` that is an ancestor of `bob`. In the context of teleo-reactive program, if BeliefStore contains the fact that `alice` is a parent of `bob` or the fact that a descendant of `alice` is an ancestor of `bob`, then the fact `ancestor(alice, bob)` is in BeliefStore. Note that in Qulog variables (`Ans, Des...`) are capitalised and constants (`ancestor, alice...`) are not.
If some of the arguments are not ground (i.e. variables are passed instead), relation works almost as a function returning multiple values; it "returns value" by instantiating the variables with values specified in *`Body`*. For example, a relation *doubleTheLarger* takes two ground integers and instantiate the third non-ground variable to be double the larger given integers can be defined as:

```
doubleTheLarger(X, Y, Double) :: X >= Y <= Double = 2 * X
doubleTheLarger(X, Y, Double) :: X < Y <= Double = 2 * Y
```

With this declaration, `doubleTheLarger(5, 2, X)` will be caught by the first rule as 5 is larger than 2, and `X` is instantiated by 10.

In order for a relation to be instantiated as an dynamic fact in the BeliefStore, it needs to be declared as either percept or belief. This can be done as follows:

```
percept_decl = "percept" name ":"  args_type (, name ":" args)*
belief_decl = "belief" name ":" args_type (, name ":" args)*
args_type = "(" type* ")"
```

The role of `percept` and `belief` are very similar in a sense that they are both facts that composes BeliefStore. In fact, a TeleoR developer, Peter Robinson, told us that they decided to remove distinction between these two, and it is left up to the programmers to treat these dynamic facts.

Functions

The following EBNF grammar defines the syntax of the function definition in Qulog:

```
function_def = function_name "::" predicate ("&" predicate)* "->" expression
```

The return value of function is what `expression` evaluates to, satisfying all `predicate`. For example, a function that calculates the absolute value of a function would be:

```
abs(N) :: N >= 0 -> N
abs(N) :: N <= 0 -> 0 - N
```

Qulog Actions

The syntax to define a new Qulog action is by the following EBNF grammar:

```
action_def = head "::" predicate ("&" predicate)* "~>>" (action_body;)*
```

The semantics of action definitions is the same as for relation definitions. The difference is that at least one `action_body` is an action which has a side effect such as sending a message, starting an agent, forking a new evaluator thread or updating the BeliefStore[8]. For instance, Qulog action can be used to define how the messages from other agents are handled. The following example that Clark et al. presented in[6] shows an example of how one may define `handle_message_`.

```
handle_message_(rem(colour(Th,C)),Nm@Host)::thing(Th)&col(C) ~>>
        remember colour(Th,C);
        remembered(colour(Th,C)) to Nm@Host

handle_message_(unrem(colour(Th,C)),Nm@Host)::thing(Th)&col(C) ~>>
        forget colour(Th,C);
        forgotten(colour(Th,C)) to messages:Nm@Host

handle_message_(_,_) % Ignore all other messages
```

This agent remembers the colour of an object specified in the message of the format `rem(colour(Th, C))`, forgets if the message is in the format `unremcolour(Th, C)` and ignore all the other messages. As this behaviour is not default, an action `handle_message_` should be defined. Note that `remember/forget` are built-in actions for manipulating the BeliefStore, and `remembered/forgotten` are user-defined actions to send an acknowledgement to the sender.

TeleoR Actions

TeleoR actions are to be differentiated from Qulog actions. While Qulog are "relations with side effects", TeleoR actions are what we most likely associate with actions of robots. Note that in this report, "action" or "primitive action" typically refers to a TeleoR action, not a Qulog action.
There are two kinds of actions:

- Durative actions
  Durative actions are actions that continue until stopped and which may be modifiable whilst executing[6]. For instance, "turning to a certain direction at certain speed" would be more appropriate as durative action.

- Discrete actions
  Discrete actions are those which can not be stopped before they naturally terminate after a short time, and all the other actions will be ignored during their execution[6]. For example, "closing the gripper" might be more appropriate as a discrete action, if the gripper do not terminate before it is completely closed.

In the program, the signature of each TeleoR action should be listed under either `durative` or `discrete` action, like so:

```
durative_decl = "durative" name ":" args_type (, name ":" args_type)*
discrete_decl = "discrete" name ":" args_type (, name ":" args_type)*
args_type = "(" type* ")"
```

TeleoR Rules

TeleoR rules take the form of:

```
teleor_rule = condition "->" action
```

The most general form of `condition` is[8]:

```
condition = guard "while" while_cond "min" while_duration
                  "until" until_cond "min" until_duration
```

`guard`, `while_cond` and `until_cond` is a conjunction of predicates which gets queried to BeliefStore. `while_duration` and `until_duration` are an arithmetic term.
Corresponding action takes place if querying `guard` succeeds, or both of the following holds:

- `while_cond` holds or `while_duration` has not expired.
- `until_cond` holds or `until_duration` has not expired.

Normally once `guard` is achieved, the agent tries to take TeleoR rules that are listed higher than the current TeleoR rule. `until` is used to allow an action to over-achieve the guard by preventing earlier TeleoR rules to be fired[5]. Dual of `while` does something opposite by preventing later rules from firing by executing the action even if the `guard` does not hold any more[5].

`action` can be:

- `primitive_action+`
  A single or a set of durative or discrete actions, which are executed concurrently.
- A call to a TeleoR procedure (c.f. Section 2.1.1)
- `(primitive_action "for" duration)+`
  A sequence of primitive actions, whose durations are specified.
- `primitive_action "wait" duration ^ repeats`
  Repeated action with interval specified as `duration` as many as `repeats` times.

These actions may have a Qulog action attached using ++ operator. This action could be, for instance, sending messages, updating the BeliefStore or forking a thread.

Procedures

Procedures are defined as a sequence of TeleoR rules, whose syntax is:

```
procedure =
        name ":" args_type "~>"
        name "(" vars ")" "{"
          teleor_rule+
        "}"
args_type = "(" type* ")"
vars = variable*
```

Rules that are higher up in the procedure take higher precedence. When the evaluator thread determines on which TeleoR rule is to be applied, it first checks the condition of the TeleoR rule at the top of the procedure, and only if it does not hold, the condition of the second TeleoR rule gets checked, and so on.
Procedures can be seen as subtasks of the procedure that calls them. The subtask represented by the procedure is considered to be "done" if the first rule fires. That is, the goal of a procedure P is often the partially instantiated guard of its first rule. Otherwise, letting its parent procedure be P', the goal of P could be the partially instantiated guard of the rule *before* the rule that fired in P' which called P[6].
For instance, a procedure for a robot to clean up the room could be:

```
cleanUpRoom(room)~>
cleanUpRoom(Room) {
        clean(Room) -> ()
    dirty(Room) -> do_cleaning(Room)
}
```

Clearly, the goal of this procedure is to have the given room to be clean, which is the guard of the first rule.
Procedures allow a teleo-reactive program to use recursion, as a call to a procedure can be in the right-hand side of the TeleoR rules. However it is not common and almost always has a small depth bound[6].

2.1.2 Type Definitions

TeleoR is a typed language; procedure, relation, action, and function need to have its type signature declared. In this section of the program, a developer can define their own types, on top of the following built-in types. The syntax for declaring a new type is defined with the following EBNF grammar:

```
type_def = type_head "::=" type_expression
```

A type can take a type parameter to represent generic. For instance, a tuple of a value of the same type can be represented as:

```
tuple(T) ::= t(T, T)
```

A type can be recursively defined too, like so:

```
myNat := zero || succ(myNat)
```

2.1.3 Qulog Type Declarations

Each relation, action, and function have to have their type declared. This was an important update from TR, and it is there to guarantee that all TeleoR guard evaluations are generated correctly[6]. The syntax to do so is:

```
type_decl = name ":" type
```

The type of the function that returns the absolute function of the number given as an argument presented in Section 2.1.1 should be declared as:

```
abs : num -> num
```

In a type of relations and action, specifying which arguments need to be ground upon calling them is done using *moded types*. There are four moded types, which are the following:

- `"!" type`
  When the relation is called the supplied argument must be ground and of `type`[8].
- `"?" type`
  When the relation is called, the supplied argument must either be ground and of `type` or will be ground to a term of `type` by the call[8].
- `type"?"`
  When the relation is called, the supplied argument must either be ground and of `type` or, if ground by the call, will be ground to a term of `type`[8].
- `"??"type`
  When they relation is called, the supplied argument, if ground, must be of `type` and the call will not further instantiate the argument[8].

For instance, relation `doubleTheLarger` presented in Section 2.1.1 should have its type declared as:

```
doubleTheLarger : (!int, !int, ?int) <=
```

This means that the first two integer arguments should be ground and the third integer can be either ground or a variable, in which case it will be instantiate. Therefore the valid call for example would be:

```
doubleTheLarger(1, 2, 4) % succeeds
doubleTheLarger(1, 2, 3) % fails
doubleTheLarger(1, 2, X) % succeeds and X is instantiated with 4
```

where as the following would be invalid:

```
doubleTheLarger(X, 2, 4)
doubleTheLarger(1, X, 3)
```

Moded types are used similarly in the types of actions. For instance, handle_message_ presented in Section 2.1.1 is typed to:

```
handle_message_ : (term?, process_handle) ~>>
```

where `term` is a most general built-in type and `process_handle` is a type for the process from which the message come from[6].

2.1.4 Correctness of a Teleo-reactive Program

In order for a teleo-reactive program to be partially correct, each procedure should satisfy the regression property. That is, a rule should be such that its action cause an earlier rule to fire, and the action of the first rule is so that it does not cause its condition to stop holding. In other words, after firing of each rule the program is able to make a meaningful progress to the final goal of the procedure, as the firing of the first rule is the goal of the procedure. This regression property represents the goal-directedness of teleo-reactive programs.
In addition to the regression property, if all the procedures satisfy the completeness property, the program is totally correct. A procedure is complete if there is always a rule that fires. This is true if and only if the disjunction of the guards of all the rules always hold[3].

2.1.5 Operational Semantics of Standard TeleoR

In this subsection, we see the operational semantics of Standard TeleoR, defined by Clark and Robinson[6]. Standard TeleoR assumes that there is only one evaluator thread, and the TeleoR rules are without while/until rules. Note that a bold symbol indicates a term which is partially instantiated.

Notation

The following is a notation that is used in the operational semantics.

- $BS_T \vdash_f C\theta$
  This reads as "the ground instance $C\theta$ is inferable from BS_T, and θ is the first returned binding". In TeleoR, it is possible to to define fact ordering[6]. This is to express the preference on a certain facts over the other, as some change in the BeliefStore may be more important than the other. Therefore the order of the bindings returned are deterministic given BS and a partially instantiated C.

- $\boldsymbol{P}_R$
  The partially ground Rth rule of a procedure P for a call $\boldsymbol{P}$.

- $\boldsymbol{P}_{i,R_i}$
  The partially ground R_ith rule of a procedure P for a call $\boldsymbol{P}_i$.

- $eguard(\boldsymbol{P}_R)$
  $e\boldsymbol{K}$ where $\boldsymbol{P}_R = \boldsymbol{K} \sim> \boldsymbol{A}$ where the variables that does not appear in $\boldsymbol{A}$ are existentially quantified.

- $\equiv$
  This reads as "is defined to be".

- $\prec$
  This reads as "immediately precedes" and is a binary relation defined over two successive $BeliefStore$ states.

- $no\_higher\_fireable\_rule(\boldsymbol{P}, R, T)$
  This reads as "no rule before R in the sequence of rules of call $\boldsymbol{P}$ has an inferable guard at time T. Formally:

$$\begin{aligned} &no\_higher\_fireable\_rule(\boldsymbol{P}, R, T) \equiv \\ &\qquad \forall R'[\ 1 \leq R' < R \rightarrow \neg\exists\theta' BS_T \vdash eguard(\boldsymbol{P}_{R'})\theta'\ ] \end{aligned}$$

- $fire(\boldsymbol{P}, R, \theta, T_F)$
  This reads as "R th rule of a procedure P for a call $\boldsymbol{P}$ fires at T_F".
  Formally,

$$fire(\boldsymbol{P}, R, \theta, T_F) \equiv BS_{T_F} \vdash_f e\boldsymbol{K}\theta \wedge no\_higher\_fireable\_rule(\boldsymbol{P}, R, T)$$

  where $\boldsymbol{P}_R = \boldsymbol{K} \sim> \boldsymbol{A}$.

- $continue(\boldsymbol{P}, R, \theta, T_F, T_C)$
  This reads as "R th rule of a procedure P for a call $\boldsymbol{P}$ has continued to fire from T_F to T_F".
  Formally,

$$continue(\boldsymbol{P}, R, \theta, T_F, T_C) \equiv \forall T[\ T_F \leq T \leq T_C \rightarrow fire(\boldsymbol{P}, R, \theta, T)\ ]$$

  where $\boldsymbol{P}_R = \boldsymbol{K} \sim> \boldsymbol{A}$.

- $refire(\boldsymbol{P}, R, \theta, \psi, T_F, T_{RF}))$
  This reads as "R th rule of a procedure P for a call $\boldsymbol{P}$ that has continued from T_F until just before T_{RF}, and it re-fired at T_{RF} with new binding ψ".
  Formally,

$$\begin{aligned} &refire(\boldsymbol{P}, R, \theta, \psi, T_F, T_{RF})) \equiv \\ &\qquad \forall T[\ T_F \leq T < T_{RF} \rightarrow fire(\boldsymbol{P}, R, \theta, T)\ ] \wedge fire(\boldsymbol{P}, R, \psi, T_{RF}) \wedge \theta \neq \phi \end{aligned}$$

  where $\boldsymbol{P}_R = \boldsymbol{K} \sim> \boldsymbol{A}$.

- $rules(\boldsymbol{P})$
  This denotes a sequence of partially instantiated rules of $\boldsymbol{P}$.

- $primActs(Acts)$
  This reads a "$Acts$ is a set of primitive actions."

- $procCall(Acts)$
  This reads as "$Acts$ is a procedure call.

- $dur(a)$
  This reads as "a is a durative action".

- $dis(a)$
  This reads as "a is a discrete action".

- S_i
  If S is a sequence $s_1, ..., s_n$, S_i is s_i.

- $S_{i:\,j}$
  If S is a sequence $s_1, ..., s_n$, $S_{i:\,j}$ is the sub-sequence $s_i, ..., s_j$.
- $contRI((\boldsymbol{P}, R, \theta, T_F), T_C)$
  Equivalent to $continue(\boldsymbol{P}, R, \theta, T_F, T_C)$.
- $fireRI((\boldsymbol{P}, R, \theta, T_F), T_F)$
  Equivalent to $fire(\boldsymbol{P}, R, \theta, T_F)$.

Evaluation State

The evaluation state at time T for a $TeleoR$ procedure call SP, which began at $ST \leq T$, is a tuple:

$$(T, MaxDp, \boldsymbol{SP}, ST, FrdRules, LActs, Acts, CActs)$$

Informally, each element may be described as following:

- $MaxDp$
  The maximum recursion depth allowed.
- $FrdRules$
  This is a sequence of tuples:

  $$(\boldsymbol{P_1}, R_1, \theta_1, T_1), ..., (\boldsymbol{P_n}, R_n, \theta_n, T_n)$$

  where $(\boldsymbol{P_x}, R_x, \theta_x, T_x)$ represents a rule $\boldsymbol{P}_{i,R_i}$ fired for the first time at T_x with the binding θ_x.
- $LActs$
  $Acts$ from the previous evaluation states.
- $Acts$
  The action that is chosen in this evaluation state.
  $Acts$ can be either:
  - A sequence of primitive actions
  - A procedure call.
  - md_fail
    This indicates an error state due to a procedure calls which exceeds the maximum depth.
  - nfr_fail
    This indicates that the agent is in an error state as it does not have a fireable rule.
- $CActs$
  A set of control action that is computed from $LActs$ and $Acts$.

The evaluation state must satisfy the following conditions, where $\forall i\colon 1 \leq i \leq n$ $\boldsymbol{P_{i,R_i}} = (\boldsymbol{K_i} \sim> \boldsymbol{A_i})$,

$\forall i\colon 1 \leq i \leq n\ \boldsymbol{A_i}\theta_i = \boldsymbol{P_{i+1}}$ and $\boldsymbol{P_1} = \boldsymbol{SP}$.

$$\forall i\colon 1 \leq i \leq n \quad fire(\boldsymbol{P_i}, R_i, \theta_i, T_i)$$

$$\forall i\colon 1 \leq i \leq n \quad procCall(\boldsymbol{P_{i+1}}) \wedge T_i \leq T_{i+1} \leq T$$

$$\begin{array}{ll}
 & [\ Acts = \boldsymbol{A_n}\theta_n \wedge primActs(Acts) \wedge 0 < n \leq MaxDp\ ] \\
\vee & [\ Acts = md\_fail \wedge procCall(\boldsymbol{A_n}\theta_n) \wedge n = MaxDp\ ] \\
\vee & [\ Acts = nfr\_fail \\
\wedge & \exists \boldsymbol{P} \quad [\ \neg\exists(R,\theta) fire(\boldsymbol{P}, R, \theta, T) \\
 & \quad \wedge \quad [\ MaxDp > n > 0 \wedge \boldsymbol{P} = \boldsymbol{A_n}\theta_n \wedge procCall(P)\ ] \\
 & \quad \vee \quad [\ n = 0 \wedge \boldsymbol{P} = \boldsymbol{SP}\ ]\ ]
\end{array}$$

The control actions can be characterised by the following conditions.

$$\begin{array}{l}
CActs = update(LActs, Acts) \wedge \\
\quad \exists ET\ [\ [\ [n > 0 \wedge ET = T_n] \vee [\ n = 0 \wedge ET = T\ ]\ ] \wedge execAt(CActs, ET)\ ]
\end{array}$$

$$[\ Acts = nfr\_fail \vee Acts = md\_fail\ ] \rightarrow update(LActs, Acts) = update(LActs, \{\})$$

$$\begin{array}{l}
[\ Acts \neq nfr\_fail \wedge Acts \neq md\_fail\ ] \rightarrow \\
\quad update(LActs, Acts) = \\
\qquad \{stop(a) | dur(a) \wedge a \in LActs \wedge \neg\exists a'[\ a' \in Acts \wedge mod\_of(a', a)\ ]\} \cup \\
\qquad \{start(a) | dur(a) \wedge a \in Acts \wedge \neg\exists a'[\ a' \in LActs \wedge mod\_of(a', a)\ ]\} \cup \\
\qquad \{mod(a', a) | dur(a) \wedge a \in Acts \wedge a' \in LActs \wedge mod\_of(a', a) \wedge a \neq a'\} \cup \\
\qquad \{do(a) | dis(a) \wedge a \in Acts\}
\end{array}$$

Initial Evaluation State

There are three possible initial evaluation states.
If there is a fireable rule for $\boldsymbol{SP}$ using BS_{ST} and the procedure calls does not exceed $MaxDp$, the initial evaluation state is:

$$(T, MaxDp, \boldsymbol{SP}, ST, (\boldsymbol{SP}, R_1, \theta_1, ST), ..., (\boldsymbol{P}_{ns}, R_{ns}, \theta_{ns}, ST)), \{\}, Acts_{ST}, update(\{\}, Acts_{ST}))$$

If there is a fireable rule for $\boldsymbol{SP}$ and the procedure calls exceeds $MaxDp$, the initial evaluation state is an error state:

$$(T, MaxDp, \boldsymbol{SP}, ST, (\boldsymbol{SP}, R_1, \theta_1, ST), ..., (\boldsymbol{P}_{MaxDp}, R_{MaxDp}, \theta_{MaxDp}, ST)), \{\}, md\_fail, \{\}))$$

If there is no fireable rule for $\boldsymbol{SP}$, the initial evaluation state is an error state:

$$(T, MaxDp, \boldsymbol{SP}, ST, \{\}, \{\}, nfr\_fail, \{\}))$$

State Transition

When the evaluator is notified of a modification to the BeliefStore, it it is not in a non-error state, we have the transition rule:

$$
\begin{aligned}
& BS_T \prec BS_{NT} \rightarrow \\
& (T, MaxDp, SP, ST, FrdRules, LActs, Acts, CActs) \longmapsto \\
& \qquad\qquad (NT, MaxDp, SP, ST, NFrdRules, Acts, NActs, NCActs) \\
& \wedge \\
& n = \#NFrdRules \geq 0 \\
& \wedge \\
& \exists j \colon 0 \leq j \leq n \\
& [\quad NFrdRules_{0:\,j} = FrdRules_{0:\,j} \quad \wedge \\
& \quad (j = n \quad \vee \\
& \quad \neg contRI(FrdRules_{j+1}, NT) \quad \wedge \\
& \quad \forall \colon 0 \leq i \leq j, contRI(FrdRules_i, NT) \quad \wedge \\
& \quad \forall \colon j+1 \leq i \leq n, fireRI(NFrdRules_i, NT) \quad]
\end{aligned}
$$

2.1.6 TeleoR Single Task Evaluation Algorithm

The following diagram shows the algorithm[5] executed by the evaluator thread compute the action to be taken by the agent. This algorithm works out the chain of procedure calls and list of primitive action for the agent to be executed, and continues to update them.

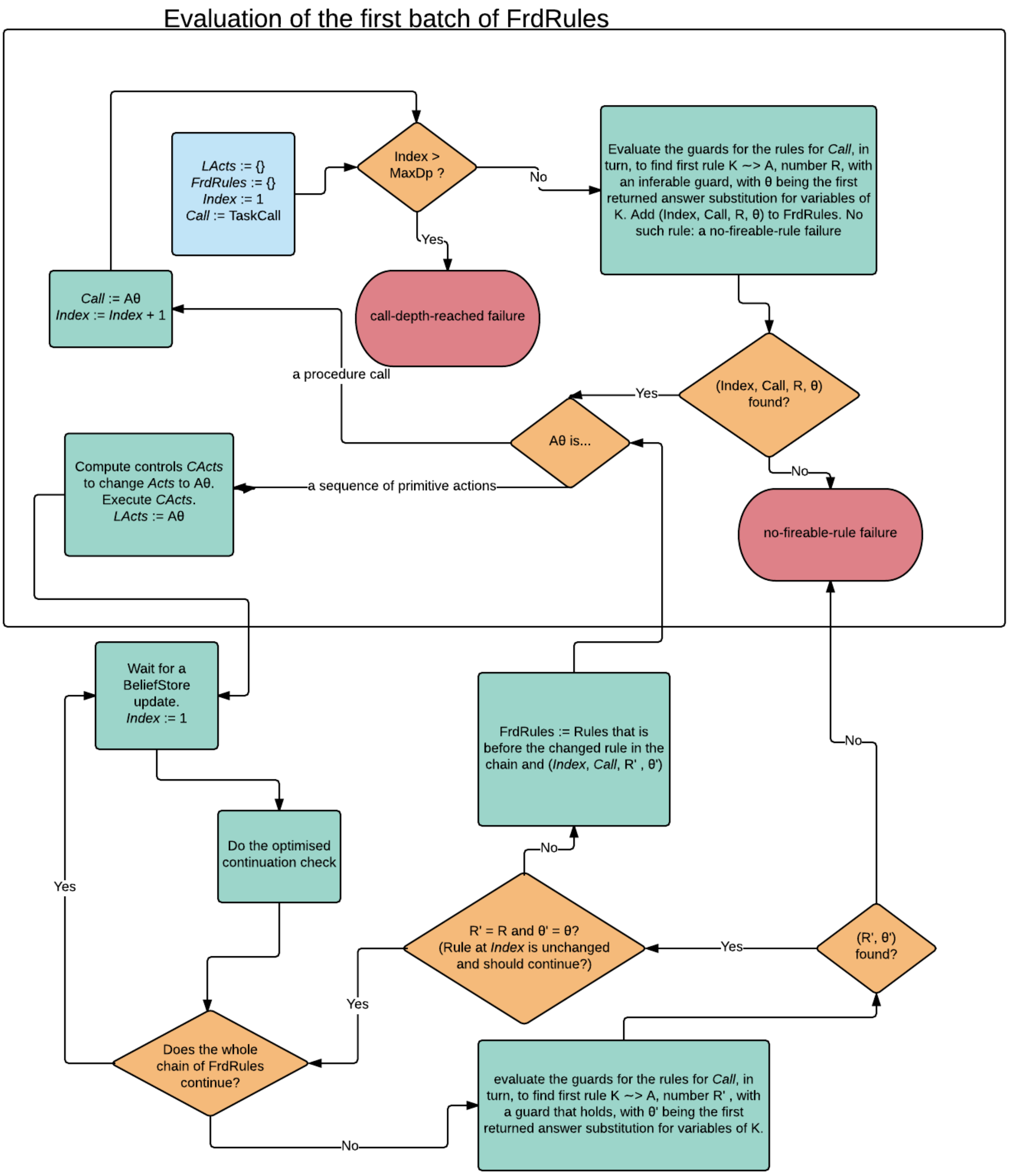
Evaluation of the first batch of FrdRules
LActs := {}
FrdRules := {}
Index := 1
Call := TaskCall
Index > MaxDp ?
No
Yes
Evaluate the guards for the rules for Call, in turn, to find first rule K ~> A, number R, with an inferable guard, with θ being the first returned answer substitution for variables of K. Add (Index, Call, R, θ) to FrdRules. No such rule: a no-fireable-rule failure
call-depth-reached failure
Call := Aθ
Index := Index + 1
a procedure call
(Index, Call, R, θ) found?
Yes
Aθ is...
No
no-fireable-rule failure
Compute controls CActs to change Acts to Aθ. Execute CActs. LActs := Aθ
a sequence of primitive actions
Wait for a BeliefStore update. Index := 1
FrdRules := Rules that is before the changed rule in the chain and (Index, Call, R' , θ')
No
Do the optimised continuation check
Yes
No
R' = R and θ' = θ? (Rule at Index is unchanged and should continue?)
Yes
(R', θ') found?
Yes
Does the whole chain of FrdRules continue?
evaluate the guards for the rules for Call, in turn, to find first rule K ~> A, number R' , with a guard that holds, with θ' being the first returned answer substitution for variables of K.
No

Figure 2.2: TeleoR Single Task Evaluation Algorithm

Let us see how the first batch of *FrdRules* are worked out, given the example following.

```
obj ::= treasure | rubbish
dir ::= left | right | centre

durative : move, turn(dir)

discrete : release, grab

percept :
        % See obj in direction dir in the distance of nat.
        see(obj, nat, dir),
        % The robot grip is closed.
        holding

collect_treasure : () ~>
collect_treasure {
        % The goal of the task is to put the treasure in storage.
        in(storage, treasure) ~> ()

        % If the goal is not achieved, get the treasure.
        true ~> get(treasure)
}

get : (obj) ~>
get (Obj) {
        % The goal of this subtask is to have and see it
        % in the centre with no distance.
        holding & see(Obj, 0, centre) ~> ()

        % If the robot does not have its grip closed,
        % but Obj is just there, grab it.
        not holding & see(Obj, 0, centre)  ~> grab

        % If the grip is open and Obj is not there, reach to it.
        not holding ~> reach(Obj)

        % If the grip is closed and Obj is not there, open the grip.
        true ~> release
}

reach : (obj) ~>
reach (Obj) {
        % The goal of this subtask is to have Obj near and in front.
        see(Obj, 0, centre) ~> ()

        % If the object is in front, shorten the distance.
        see(Obj, _, centre) ~> move

        % If the object is not seen in front, turn to that direction.
        see(Obj, _, Dir) ~> turn(Dir)
        % When the object is not in the sight, wonder around.
        true ~> turn(left) for 5 ; move for 5
}
```

Initialising FrdRules

It starts at state 1, where the fields are initialised. *LActs* is the list of non-procedure action to be executed. *FrdRules* are the chain of rules that get fired as it is the rule of the highest priority in the called procedure whose guard is inferable. *FrdRules* consists of quadruples, each of them representing a chosen rule. The first element is the call depth, the second is the name of the procedure to which the rule belongs, the third is the index of the rule in the procedure and the forth is the substitution applied to the guard. *Index* is the depth of the current procedure calls. *Call* is the procedure which the evaluator is currently evaluating. θ represents the substitution applied to the guard of the rule inspected.

Let *TaskCall* to be `collect_treasure` and the set of the dynamic fact to be {`see(treasure, 5, centre)`}. After initialisation, *LActs* and *FrdRules* are empty lists, *Call* is `collect_treasure` and *Index* is 1.

Inspecting `collect_treasure`, the second rule is chosen, as `in(storage, treasure)` is not part of BeliefStore. As the guard of the second rule does not have a variable to be instantiated, θ is {}. The current index is 1, therefore we add (1, `collect_treasure`, 2, {}) to *FrdRules.*

The action of the second rule of `collect_treasure` is a call of a procedure `get` with treasure as its argument, therefore we increment *Index* and set *Call* to `get` and θ is {`Obj = treasure`}. Going through the rules in `get`, the third rule of `get` is taken, as neither `holding` is not inferable nor `see(treasure, 0, centre)`. (2, get, 3, {`Obj = treasure`}) is added to *FrdRules.* As the action of the second rule is a call to a procedure, we increment *Index.*

We repeat a similar computation. The action of the second rule of `get` is a call of a procedure `reach` with the argument Obj = treasure. Therefore we increment *Index* and set *Call* to `reach`. The second rule of `get` is chosen as we have `see(treasure, 5, centre)` holds. θ is still `Obj = treasure`. We add (3, `reach`, 2, {`Obj = treasure`}) to *FrdRules.* As the action of the first rule is a primitive action, we can say that the whole chain is worked out. *LActs* is instantiated to be {`move`}.

Updating FrdRules

After the BeliefStore is updated, we need to check whether all the rules in *FrdRule* continue to fire. Let us say that the robot got close to the treasure and BeliefStore is updated to{`see(treasure, 0, centre)`}.
Before blindly going through all the rules, one can see whether the change in the BeliefStore *may* falsify the continue condition of any of the rule that is currently fired. This is done by maintaining a list of dependant predicates. Dependant predicates indicate which predicate and whether addition or deletion of the predicate may cause a given rule to stop firing. To log that addition of a predicate may falsify the continue condition, we add `++pred` and `--pred` for deletion to the list. For instance, `FrdRules` contain the second rule of `collect_treasure`:

```
collect_treasure {
        in(storage, treasure) ~> ();
        % The goal of the task is to put the treasure in storage.
        true ~> get(treasure);
        % If the goal is not achieved, get the treasure.
}
```

We record `++in` as the second rule can be falsified by addition of a predicate `in`. This list can be computed while `FrdRules` are computed and updated. Given the current `FrdRules`:
{(1, `collect_treasure`, 2, {}), (2, get, 3, {`Obj = treasure`}), (3, `reach`, 2, {`Obj = treasure`})}
the list of dependent predicates would be:

`[++in, ++holding, ++see, --see]` .

In this case a predicate `see` was removed and added, therefore we have the necessary condition for going through `FrdRules`. For each element, (*Depth*, *Proc*, *Rule*, *Subst*), we check *Proc* and see which rule should be taken and which substitution was applied to its guard. If its index corresponds to Rule and substitution to *Subst*, we carry on with the check. Otherwise the rest of `FrdRules` is discarded, and the rest is calculated in a similar way as `FrdRules` was initialised.

Take (1, `collect_treasure`, 2, {}). Looking at `collect_treasure`, the index of the rule taken is 2 and the substitution is still {}.
Take (2, `get`, 3, {`Obj = treasure`}). With the current BeliefStore, {`see(treasure, 0, centre)`}, the index of the chosen rule is 2. As this does not correspond to 3, the rest of the *FrdRules* is discarded, and (2, `get`, 3, {`Obj = treasure`}) is updated to (2, `get`, 2, {`Obj = treasure`}). As the action of the second rule is `grab`, *LActs* is updated to {`grab`}.

2.2 MCMAS

2.2.1 Model Checking

The goal of system verification is to prove that a system S satisfies a specification P. Model checking is a technique to automatically check whether a model satisfies a logical formula. Model checking can be applied to system verification by:

- Translating S into a model M
- Translating P into a formula ψ
- Applying model checking algorithms to prove $M \vDash \psi$

Given that M includes all the possible computation in S and ψ faithfully represents P, $M \vDash \psi$ if and only if M satisfies P. This means that model checking provides an automatic way of verifying that a system satisfies a specification.

Model checking approaches can characterised by how the model and the specification can be described. In MCMAS, a user defines the system as an *interpreted system*, which is an extended *Kripke Model*. The interpreted system gets translated into a model called *ordered binary decision diagram (OBDD)* and supports a range of temporal and epistemic specification languages. We look at these in the following sections.

2.2.2 Kripke Model

Definition 2.1 (Kripke Model). *A Kripke model is a structure represented by* (S, R, V)*, where:*

- *S is a finite set of states.*
- *$R \subseteq \mathcal{P}(S \times S)$ is a relation that represents the transition.*
- *V is a function $V : AP \to \mathcal{P}(S)$, where AP is a set of propositional atoms.*

2.2.3 Interpreted System

An interpreted system is an extension of Kripke model, used to reason about multi-agent system[9].

Definition 2.2 (Interpreted System)**.** *Let $A = 1, ...n$ be a set of agents, E an environment agent, $Agt = \{E\}$ is the set of all possible agents and AP a set of propositional atoms. Then an interpreted system is a tuple $\langle (L_i, Act_i, P_i, t_i)_{i \in Agt}, I, V \rangle$ where*

- *L_i is a set of all possible private local states, where $L_i \times L_E$ is a set of all possible local states. Global state is defined as a tuple of the local states of all agents and the environment. Therefore all possible global states S is $L_1 \times ... \times L_n \times L_E$.*

- *Act_i is a set of all possible local actions. Global action Act is defined as a tuple of the local action of all the agents and the environment. Therefore all possible global action Act is $Act_1 \times ... \times Act_n \times Act_E$.*

- *$P_i : L_i \times L_E \to \mathcal{P}(Act_i)$ is a protocol function which takes a local state and returns the set of local actions available at that state.*
- *$\tau_i : L_i \times L_E \times Act_1 \times ... \times Act_n \times Act_E \to L_i$ is an evolution function which takes a local state and the local actions of all the agent in Agt, and returns the "next" local state.*
- *$I \in S$ is a set of initial global states.*
- *$V : AP \to \mathcal{P}(S)$ is a valuation function which maps propositional atoms to the set of global states where it holds.*

2.2.4 Ordered Binary Decision Diagram

Ordered Binary Decision Diagram, *OBDD*, is the model an interpreted system is translated into and a model checking algorithm is applied to in MCMAS. It is a representation of propositional formulas indicating the value of the formula given an interpretation. *OBDD* is a special instance of *Binary Decision Diagram*, which is an optimised *Binary Decision Tree* for the purpose. Let us look at these models in order.

Binary Decision Tree

Binary Decision Tree, *BDT*, can be set to represent a propositional formula. Each path in such *BDT* represent one interpretation, and the leaf node indicate the value of the formula. The propositional variables are represented by the non-leaf nodes and two branches represent two values of the variable. Suppose we want to represent $x \lor y$. Then one possible *BDT* is:

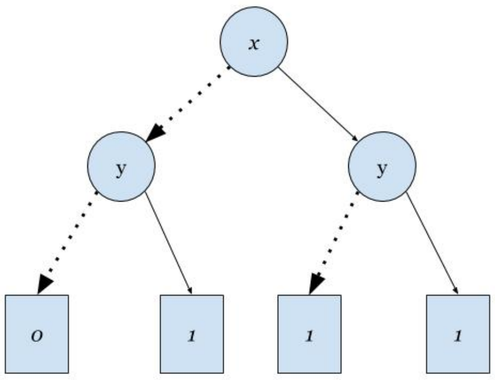

Figure 2.3: BDT for $x \vee y$.

Binary Decision Diagram

Binary Decision Diagram, *BDD* is a more efficient representation of a propositional formula, obtained from *BDT* by removing unnecessary nodes and branches.
Bryant[10] presents the following three rules of doing so without altering the semantics:

1. Remove duplicate terminals.
   Eliminate all but one leaf node with a given label and redirect all arcs into the eliminated leaf nodes to the remaining one.

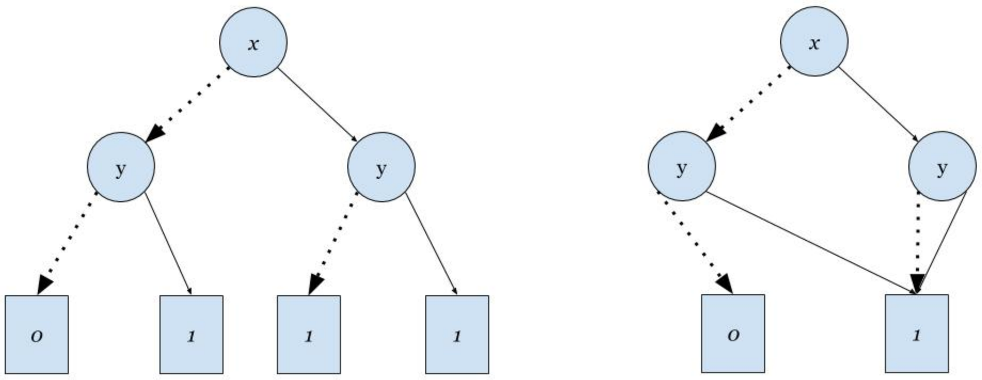

Figure 2.4: BDT for $x \vee y$ simplified by rule (1).

2. Remove duplicate non-leaf nodes.
   If non-leaf node m and n are labelled with the same variable and has the branches of the same type are both going into the same nodes, then eliminate one of the two nodes and redirect all incoming branch to the other node. For instance, BDT of $(x \vee y) \wedge z$, simplified using rule (1), can be simplified as follows using rule (2).

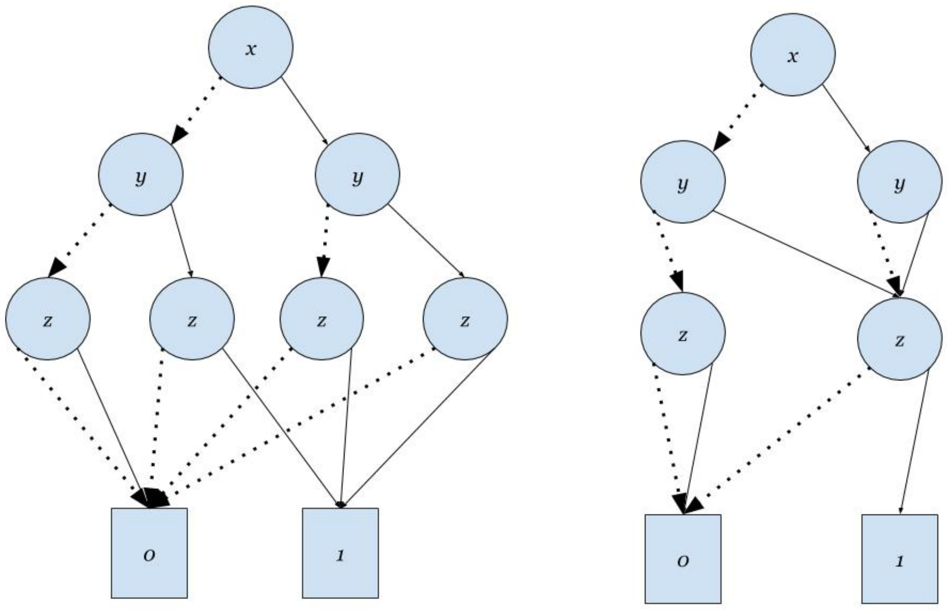

Figure 2.5: BDD of $(x \vee y) \wedge z$ further simplified using rule (2).

3. Remove redundant test.
   If non-leaf node n has both branches going into the same node m, then eliminate n and redirect all incoming arcs of n to m. For instance, BDD of $x \vee y$ simplified with rule (1) can be simplified with this rule as follows:

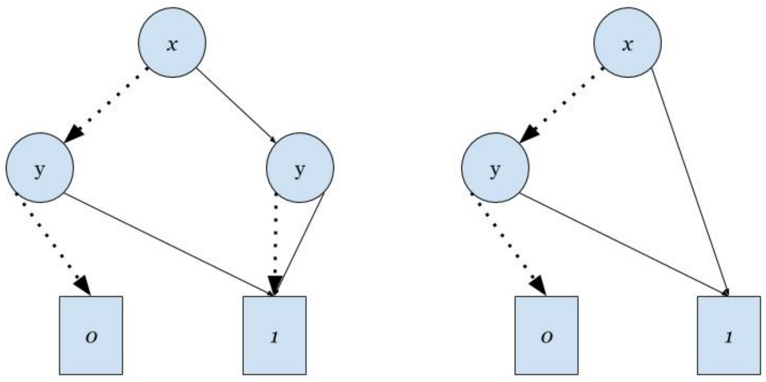

Figure 2.6: BDD of $x \vee y$ further simplified with rule (3).

A BDD is said to be *reduced* if none of the above rules can be applied any more. For instance, the BDD on the right in Figure 2.6 is reduced.

Ordered Binary Decision Diagram

According to the ordering of the variable, there can be multiple BDD. *Ordered Binary Decision Diagram* is a BDD with a fixed order of the variable. Huth and Ryan[11] define OBDD as "Let $[x_1, ..., x_n]$ be an ordered list of variables without duplications and let B be a BDD all of whose variables occur somewhere in the list. We say that B has the ordering $[x_1, ..., x_n]$ if all variable labels of B occur in that list and, for every occurrence of x_i followed by x_j along any path in B, we have $i < j$."
According to this definition, one possible OBDD representing a formula $(x_1 \vee x_2) \wedge x_3$ with the ordering $[x_1, x_2, x_3]$ is:

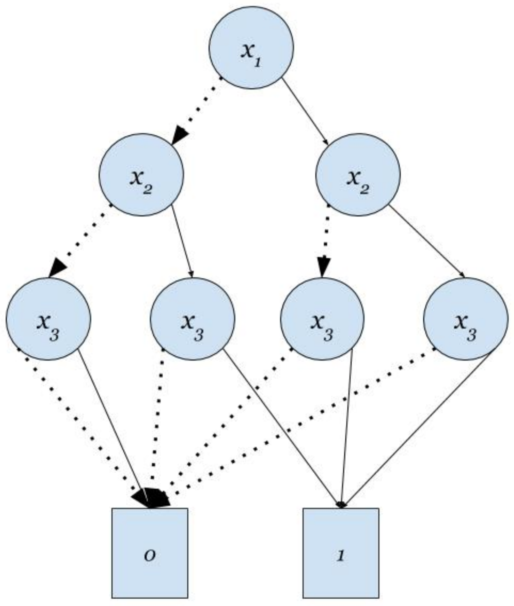

Figure 2.7: An OBDD representing $(x_1 \vee x_2) \wedge x_3$ with the ordering $[x_1, x_2, x_3]$.

For a given Boolean formula and an ordering of the variables, there is one canonical reduced structure of OBDD.

2.2.5 Specification Languages

Many model checkers are built to prove a temporal specification[12]. This is because the application is often verification of a single program or of a piece of hardware. In applications of multi-agent system, temporal specifications might not be enough.
Consider verifying a protocol for two agents to communicate a message on a faulty line. A property of interest might be such as "if the receiver receives the message, and the sender *knows* that he did". To specify such a property, temporal modalities are not a natural choice, if not insufficient. For this reason, MCMAS supports the automatic verification of specifications that use epistemic modalities, as well as correctness, and fairness modalities[13].
This section describes temporal and epistemic operators which can be used to define a formula in MCMAS. Correctness and fairness modalities will be described along with the grammar of MCMAS.

2.2.6 Computational Tree Logic

Computational Tree Logic, *CTL* is a sub-class of *temporal logic*, a modal logic that reasons about time. It allows one to reason about all possible paths initiated from a state.
The syntax is the following:

$$\phi ::= p \mid \neg\phi \mid \phi \wedge \phi \mid EX\phi \mid EG\phi \mid E(\phi U \phi)$$

The semantics on an interpreted system IS and one of its states s is:

- $(IS, s) \vDash \phi$
  $s \in V(s)$
- $(IS, s) \vDash EX\phi$
  There exists a path $s_1, s_2, s_3, ...$ such that $s_1 = s$ and $(IS, s) \vDash \phi$.
- $(IS, s) \vDash EG\phi$
  There exists a path $s_1, s_2, s_3, ...$ such that $s_1 = s$ and $(IS, s_i) \vDash \phi$ for all $i \geq 1$
- $(IS, s) \vDash E(\phi U \psi)$
  There exists a path $s_1, s_2, s_3, ...$ such that $s_1 = s$ and there exists a $i \geq 1$ such that $(IS, s_i) \vDash \psi$ and $(IS, s_j) \vDash \phi$ for all $1 \leq j \leq i$.

For each of them a dual operator can be defined:

- $(IS, s) \vDash AX\phi$
  For all paths $s_1, s_2, s_3, ...$ such that $s_1 = s$, $(IS, s) \vDash \phi$ holds.
- $(IS, s) \vDash AG\phi$
  For all paths $s_1, s_2, s_3, ...$ such that $s_1 = s$, $(IS, s_i) \vDash \phi$ holds for all $i \geq 1$
- $(IS, s) \vDash A(\phi U \psi)$
  For all paths $s_1, s_2, s_3, ...$ such that $s_1 = s$, there exists a $i \geq 1$ such that $(M, s_i) \vDash \psi$ and $(IS, s_j) \vDash \phi$ for all $1 \leq j \leq i$.

Alternating-Time Temporal Logic

CTL allow specifying a property of *closed system* well, providing a way to specify existential and universal satisfaction. A closed system is a system whose behaviour is completely determined by the state of the system[14].
In a *open system* on the other hand, we have the behaviour of environment as an additional factor in the choice that system makes. Alternating-time temporal logic, *ATL* offers a way to write an alternating specification. Alternating specification is a statement about existence of the *strategy*, which decides which action is to be performed in a given situation, of an agent so that the satisfaction of a property is guaranteed no matter how the environment behaves[15].
The syntax is the following:

$$\phi ::= p \mid \neg\phi \mid \phi \wedge \phi \mid \langle\langle\Gamma\rangle\rangle X\phi \mid \langle\langle\Gamma\rangle\rangle G\phi \mid \langle\langle\Gamma\rangle\rangle(\phi U \phi)$$

The informal reading of each operator is:

- $\langle\langle\Gamma\rangle\rangle X\phi$
  The agents in Γ have a joint strategy such that ϕ becomes true at the next state no matter what other agents do.

- $\langle\langle\Gamma\rangle\rangle G\phi$
  The agents in Γ have a joint strategy such that ϕ is forever true no matter what other agents do.
- $\langle\langle\Gamma\rangle\rangle(\phi U \psi)$
  The agents in Γ have a joint strategy such that ψ becomes true at some point and ϕ is true until them, no matter what other agents do.

Though originally the interpretation of ATL was defined on concurrent game structures[15], it can also be interpreted on interpreted systems[16].

Epistemic Operators

In MCMAS, a formula can be prefixed with epistemic operators to express a property on agent's, or a group of agents', knowledge.
Let $s \sim_i s'$ to hold if and only if local state of agent i in s and s' are the same. Then, the semantics of such an operator can be expressed on an interpreted system IS and a state s:

- $(IS, s) \vDash K_i\phi$
  For all s' such that $s \sim_i s'$ we have that $(IS, s') \vDash \phi$.
- $(IS, s) \vDash GK_\Gamma\phi$
  For all s' such that $s(\cup_{i\in\Gamma} \sim_i)s'$ we have that $(IS, s') \vDash \phi$.
- $(IS, s) \vDash DK_\Gamma\phi$
  For all s' such that $s(\cap_{i\in\Gamma} \sim_i)s'$ we have that $(IS, s') \vDash \phi$.
- $(IS, s) \vDash GCK_\Gamma\phi$
  For all s' such that $s(\cup^{+}_{i\in\Gamma} \sim_i)s'$ we have that $(IS, s') \vDash \phi$.

Deontic Operator

The deontic operator O_i can be prefixed to a formula, so it reads "in all the possible correctly functioning alternatives of agent, the following formula holds"[17]. Prohibited local states are marked as *red state*.
Extend IS by including red states RS_i for each agent, and call it DS. Define a binary relation R_i over a tuple of global states, so that $R_i(s, s')$ holds if and only if local state of i in s' is not in RS_i. Then:

- $(DS, s) \vDash \phi$
  Equivalent to $(IS, s) \vDash \phi$
- $(DS, s) \vDash O_i\phi$
  For all s', $R_i(s, s') \rightarrow (DS, s') \vDash \phi$

2.2.7 ISPL Syntax

In MCMAS, ISPL is used to define the interpreted system and the formula. According to the documentation[18], the general structure of ISPL is the following:

```
Semantics = MultiAssignment (MA) | SingleAssignment (SA)
Agent Environment
Obsvars:
...
end Obsvars
Vars:
...
end Vars
RedStates:
...
end RedStates
Actions = {...};
Protocol:
...
end Protocol
Evolution:
...
end Evolution
end Agent
Agent AgentName
Lobsvars = {...};
Vars:
...
end Vars
RedStates:
...
end RedStates
Actions = {...};
Protocol:
...
end Protocol
Evolution:
...
end Evolution
end Agent
Evaluation
...
end Evaluation
InitStates
...
end InitStates
Groups
...
end Groups
Fairness
...
end Fairness
Formulae
...
end Formulae
```

Variables

Variables of an agent make up its local state, and they are declared in `Vars`. It can be Boolean, enumeration and bounded integer[18], which can respectively be declared as:

```
x : boolean; -- x can be either true or false
y : {one, two}; -- y can be either one or two
z : 1 .. 5; -- z can be an integer from 1 to 5
```

Local Observable Variables

Local observable variables of an agent are variables of the environment that the agent can observe. The syntax is:

```
"Lobsvars" "=" "{" var+ "}"";"
```

Red States

Red states of an agent is specified as a Boolean formula over its local variable and local observable variables if the agent is normal. The operator that can be used is `and`, `or`, ! and `->`. The states that satisfy the formula are red state, and a green state otherwise. It can be defined for instance:

```
x = false and Environment.envV1 % where envV1 is a part of Lobsvars
```

Actions

Actions is defined as enumeration, like so:

```
"Actions" "=" {" action+ "}"";"
```

Protocol

A protocol is defined as list of *protocol rules*:

```
condition ":" action_set";"
```

When the condition is satisfied, one action in the corresponding set of actions is non-deterministically chosen. If multiple conditions are true, then all the actions are considered possible by MCMAS.[18] Keyword `Other` can be used to specify the case where none of the condition is hold, like so:

```
Other : {a1, a2};
```

Evolution

An evolution is defined as set of *evolution rules* in the form of:

```
assignment "if" condition ";"
```

There are two ways of interpreting an evolution:

- Multi-assignment
  In one evolution rule, one can assign values to multiple variables. If there are multiple rules whose condition holds, only one rule is non-deterministically chosen.
- Single assignment
  In one evolution rule, the assignment of only one variable can be specified. Each rules are grouped according to the variable it updates, and one rule from each group whose condition holds is non-deterministically applied.

Evaluation

An evaluation defines an atom in terms of the variable of agents, which can be used in the specification formulae. The format is:

```
atom "if" condition";"
```

`atom` is evaluated to be true at a state if `condition` holds, and false otherwise.

Initial States

Initial States are represented as Boolean formulae over variables of the agents that initial sates must satisfy.

Groups

Groups are used in formulae which uses the concept of multiple agents, such as *Gamma* in $E_{Gamma}\phi$. It can be define as a set, like so:

```
g1 = {agent1, Environment};
```

Fairness

A Boolean formula using `and`, `or`, ! and `->` can be defined as fairness property. The specification is checked only on the path the fairness conditions hold infinitely often. It can be used for instance to express a property such as “The faulty line would not drop all the messages forever” in the example of two agents to communicate over a faulty line.

Specification Formulae

Specification Formulae are defined under `Formulae`, using atoms defined in `Evaluattion`, `and`, `or`, !, `->` and operators introduced in the previous section. That is[18]:

```
formula = "(" formula ")"
| formula "and" formula
| formula "or" formula
| "!" formula
| formula "->" formula
| "AG" formula
| "EG" formula
| "AX" formula
| "EX" formula
| "AF" formula
| "EF" formula
| "A" "(" formula "U" formula ")"
| "E" "(" formula "U" formula ")"
| "K" "(" agent_name "," formula ")"
| "GK" "(" group_name "," formula ")"
| "GCK" "(" group_name "," formula ")"
| "DK" "(" group_name "," formula ")"
| "O" "(" agent_name "," formula ")"
| "<" group_name ">" "X" formula
| "<" group_name ">" "F" formula
| "<" group_name ">" "G" formula
| "<" group_name ">" "(" formula "U" formula ")"
| Atom
```

2.3 Related Works

There has not been much work done in verification of teleo-reactive programs. The only work we are aware of is that of Dongol et al.[19]. In this work, they have developed a real-time logic that is based on Duration Calculus to formulate the semantics of teleo-reactive programs. They present a Prolog program which generates the rely condition that the environment needs to satisfy to guarantee the regression property of the program. This is a partially mechanised method, as the user needs to manually prove this condition.

The compilation methodology we use in this project is presented in the work of Boureanu et.al.[20], applied to verify cryptographic protocols specified in a specification language CAPSL. In this work, they define a translation from CAPSL models into interpreted system, and rewrite CAPSL goals as modal specification. Then they present a compiler which implements this translation, so that the compiler takes CAPSL and outputs ISPL.

Chapter 3

MAS models for Single Bound Programs

Our aim is to provide a mechanism to automatically verify teleo-reactive programs using a model checker. We are aware of no similar work done previously and in fact, to our knowledge there is only one work in verification of teleo-reactive program in general, which was discussed in the background.

Specifically, in this project we explore how we may verify a TeleoR program using MCMAS. We do this by devising a compiler that compiles a TeleoR program into an ISPL file. The output ISPL defines an interpreted system which models all possible computation done by the agent running the input TeleoR program. In this chapter, we formally specify this interpreted system which we wish to represent in target ISPL.

We first define Single Bound Programs, a class of TeleoR programs which we wish to deal with in this project. The whole set of TeleoR programs can not be realistically verified for three reasons. Firstly, TeleoR programs are Turing complete, and thus algorithmic verification is undecidable; we define a Turing machine in TeleoR to demonstrate this. Secondly TeleoR is a rather large language, which contains a full blown logic programming language Qulog, and provides mechanisms to support developers implementing rich semantics which is not only difficult to support, but also leads to state explosion. Lastly, the formal operational semantics which covers the whole TeleoR was not available, and the ambiguity is fatal in model checking.

We then present a transition system which represents all the possible computation of the agent. When we say "the computation of the agent, we refer to the computation done by the evaluator thread, presented as the operational semantics in Section 2.1.5. Unfortunately, we can not directly use these operational semantics as the transition system, as it has some unnecessary components, and more importantly, it only specifies what the valid states and transitions are, and not which transitions are actually possible.

This is because the actual transitions are dictated by how the BeliefStore is updated. In this chapter, we assume for two functions, *init* and *next*. They together, represent all possible sequences of the BeliefStore updates that may occur in the environment the agent is placed in. We discuss how these two functions are actually specified in the next chapter.

We then formally specify the interpreted system that the target ISPL should represent, given the transition system defined from a Single Bound Program, *init* and *next*.

Finally, we mathematically show that this mapping between the transition system to the interpreted system is correct; that is, a specification holds for the Single Bound Program if and only if it holds for the interpreted system.

3.1 Turing Completeness of TeleoR

Algorithmic verification of a program written in a Turing-complete language is undecidable, directly following the Halting Problem. TeleoR is, unfortunately, Turing-complete because its subset, Qulog is. Below is a Turing machine emulated in Qulog, based on a Turing machine emulated in Prolog[21].

```
action ::= left | stay | right
state ::= normal_state || accept
% add dummy qf1, as Qulog does not allow enumeration of one value
accept ::= qf | qf1

turing : (![symbol], ?[symbol]) <=
turing(Tape0, Tape) <=
    perform(q0, [], Ls, Tape0, Rs) &
    reverse(Ls, Ls1) &
    Tape = Ls1 <> Rs.

perform : (!state, ![symbol], ?[symbol], ![symbol], ?[symbol]) <=
perform(qf, Ls, Ls, Rs, Rs).
perform(Q0, Ls0, Ls, Rs0, Rs) <=
    get_symbol(Rs0, Sym, RsRest)&
    once(rule(Q0, Sym, Q1, NewSym, Action))&
    make_action(Action, Ls0, Ls1, [NewSym|RsRest], Rs1)&
    perform(Q1, Ls1, Ls, Rs1, Rs).

get_symbol : (![symbol], ?symbol, ?[symbol])  <=
get_symbol([], b, []).
get_symbol([Sym|Rs], Sym, Rs).

make_action : (!action, ![symbol], ?[symbol], ![symbol], ?[symbol]) <=
make_action(left, Ls0, Ls, Rs0, Rs) <= shift_left(Ls0, Ls, Rs0, Rs).
make_action(stay, Ls, Ls, Rs, Rs).
make_action(right, Ls0, [Sym|Ls0], [Sym|Rs], Rs).

shift_left : (![symbol], ?[symbol], ![symbol], ?[symbol]) <=
shift_left([], [], Rs0, [b|Rs0]).
shift_left([L|Ls], Ls, Rs, [L|Rs]).

rule : (!state, !symbol, ?state, ?symbol, ?action) <=
```

A specific Turing Machine can be configured by adding `rule` facts and setting `symbol` and `normal_state` appropriately. For instance, adding the following lines to the above code makes up a Turing Machine that, given a sequence of “a” return a tape with one more “a” appended.

```
symbol ::= a | b
normal_state ::= q0 | q1
```

```
rule(q0, a, q0, a, right).
rule(q0, b, qf, a, stay).
```

As the whole set of Qulog programs is not algorithmically verifiable, it was necessary to define a subset on which model checking could be performed.

3.2 Definition of a Single Bound Program

In this section I list the constraints that a Single Bound Program, a verifiable subset of TeleoR programs supported by the compiler, should satisfy. By doing so we introduce some non-trivial issues in verifying a TeleoR program using a model checker and why they are non-trivial.

- **Single thread and process.**
  The scenarios of multiple TeleoR agents by can be characterised by:
  - A TeleoR program with multiple threads running separate tasks, synchronised on resources.
  - Multiple processes each running a TeleoR program.
  - Both of the above.

  In order to use a model checker on a system, one needs to know its exact semantics. As the formal semantics of how concurrent tasks run was not available until the very end of the project, we were limited to the verification of a program with a single thread. With the formal semantics, one may apply the exact same methodology and extend this project. We also left the verification of multiple processes running a single thread as a future extension, discussed in Section 7.2.
- **No TeleoR rule with while, until.**
  We did not have the formal semantics for while/until until the end of the project, and therefore we had no choice not to avoid them. With the formal semantics, one may apply the exact same methodology we show in this project.
- **Arguments given to TeleoR procedures, functions, relations and actions must be bound.**
  We say that a term is *bound* if there is only a finite possible groundings for it. Otherwise, a term is *unbound*. For example, `int` or `num` are considered to be unbound, whereas the range of integer or enumeration are considered bound.

  ```
  abc ::= a | b | c  % bound
  123 ::= 1..3 % bound
  ```

  Unbound terms are problematic in model checking, because they lead to an infinite set of states or paths in the model that represents the program.
- **No concept of time.**
  A TeleoR program can be defined using the time obtained from the system clock. For example, a TeleoR rule below reads as: “if an agent sees something in a direction for at least 3 units of time, it moves towards it for 5 units of time”.

```
see(Dir, X) min 3 ~> move(Dir) for 5
```

We considered encoding the time by representing a tick of the system clock as a possible state transition which results in an increment of the integer variable that keeps track of the "current time". However this would have meant having to multiply the number of states by the upper limits of the ticks, leading to state explosion.

- **No TeleoR rules with ambiguity.**
  In TR language, a teleo-reactive language developed by Nilsson, it is ambiguous whether a rule should be re-fired if another instance of its guard becomes the first inferable instance while the old instance is still inferable[6]. Consider a TeleoR rule:

  ```
  see(bottle, Col) ~> approach(bottle, Col)
  ```

  Let us say that $see(bottle, green)$ was inferable, and the agent takes $approach(bottle, green)$. Then supposed that, with a sensor input, $see(bottle, blue)$ is added. In TR, it is ambiguous whether $approach(bottle, green)$ should continue, or $approach(bottle, blue)$ should start instead. In TeleoR, there is an implicit ordering among the dynamic facts which are in the BeliefStore. If $see(bottles, blue)$ happened to come before $see(bottle, green)$, the rule re-fires, and the agent engages with $approach(bottle, blue)$. Otherwise, it continues with $approach(bottle, green, 1.5)$. This meant that we need to consider all the orderings of the dynamic facts, at least among those with the same identifiers. However we could easily guess that this would greatly increase the number of states. Say we have 5 relations declared as percept or belief, and each of them have 10 possible instantiations. As the number of permutation of these facts is $10!^5$, there will be $6.2923832e + 32$ states where all possible dynamic facts are in the BeliefStore. Clearly, even the verification of a small program would be infeasible. That being the case, we decided that it is better to avoid programs with such an ambiguity, where there are multiple instances of the guard which lead to different instantiation of the action.

- **No user-defined relation, function or action.**
  MCMAS and Qulog fraction of TeleoR are massively different languages and it is out of the scope of this project to encode all features of a logic programming language into MCMAS.

Additionally, we have the following constraints. They are, admittedly, compared to other constraints, more trivial to support. However they are not essential features, and we wanted to avoid an additional complexity.

- **No comparison of terms, unless they are numbers or equality and inequality.**
  Qulog defines the standard ordering between terms[8]. As it is fairly complex and did not seem to be the case that it is used often, we solely supported arithmetic comparisons and equality and inequality of two terms.

- **No tuple types.**
  In Qulog, a data can be a tuple, and both the whole tuple and each of its elements may be unified to one variable. For example, if `foo` is a relation which takes a tuple of integers, it is possible for one variable to unify to its argument, and it is likewise for a tuple of variables.

```
percept foo : ((int, int))
foo(X)      % unify the whole tuple to X
foo((Y, Z)) % unify each argument to Y and Z
```

We think that tuples are not as essential, as one could substitute them with a sequence of elements.

3.3 Single Bound Programs as Transition system P.

In this section, we present a transition system which represents all the possible computation of the evaluator thread of a Single Bound Program. In this system, the state transition occurs when the BeliefStore is updated, and at each state of the program, the agent computes which action to take by consulting the updated BeliefStore.
We do this by modifying the operational semantics of the Standard TeleoR program presented by Clark and Robinson[6], which we saw in Section 2.1.5. We first refer to functions that we assume, *init* and *next*, which we define the possible sequences of BeliefStore states. Then, we see how we modify the evaluation state. Finally, we formally define the transition system P.

3.3.1 Functions defining BeliefStore Updates

The operational semantics of the Standard TeleoR program defines valid evaluation states of the evaluator thread and how the valid transition may look like. This however is not sufficient to define a transition system that we can verify, as it does not define which transitions actually happens. This is because the transition is determined by how the BeliefStore is updated, thus how the environment that the agent is placed behaves. To fill this gap, we assume that we have *next* which tells us how the BeliefStore may be updated, given the current BeliefStore state. In the next chapter, we have a further discussion on the exact definition of *next*. We also assume a function *init*, which returns all possible BeliefStore states the agent may have at the start.

3.3.2 Modification to Evaluation States

We are defining a state in P by modifying the evaluation state in the operational semantics for the Standard TeleoR program presented by Clark and Robinson[6] , which we saw in Section 2.1.5. The modifications break down into three categories.

- Removing components that are used for executing the TeleoR program, but we would not be interested in having a specification on.
- Removing components or simplify the components that are not needed to represent the Single Bound Program.
  Though the operational semantics itself does not cover the whole set of TeleoR programs, it serves as a base on which an extended operational semantics can be defined. As a result of that and additional assumptions made for Single Bound Programs, there are components that are not necessary, or overly complex, which we can remove or simplify.
- Adding components necessary to define fairness conditions.
  In the next chapter, we discuss how we may check the specification against an environment

which is "cooperative" enough that certain actions eventually succeed. We do this by specifying fairness conditions on the previous BeliefStore state and the action which was just taken. We add these informations to the states.

For the first category, we remove control actions. For the second category, we remove time stamps, and simplify BeliefStore states. For the third category, we add the previous BeliefStore state, and the action that was just taken.

Control Actions

Control actions define the operations done to the action of the agent that is currently undertaken. For example if the agent is executing a durative action *move*, and the agent chooses another action due to a BeliefStore update. Then one of the control actions will be to stop executing *move*. Since our concern is on which action is chosen at each evaluation, we do not have to keep track of control actions.

Time stamps

In the operational semantics of standard TeleoR programs, the list of fired rules $FrdRules$ is a list of tuples:

$$(P_1, R_1, \theta_1 T_1), (P_2, R_2, \theta_2, T_2)...(P_n, R_n, \theta_n T_n)$$

T_i denotes the time when the R_i rule of the TeleoR procedure P_i has fired.
We want to remove this as Single Bound Programs do not use this. In TeleoR programs, T_i is used for two purposes. One is to support a durative action that lasts for a certain amount of time. For instance, the following rule reads as "If an agent sees something in its left, it moves towards left for 5 units of time".

```
see(left, _) ~> move(left) for 5
```

Since Single Bound Programs are assumed not to have a rule which uses the concept of time, the time stamp is not necessary for specifying the actions.
The other is to see which rule has been continuing to fire over a period of time. This is to evaluate rules that contain *while*, *until* or *min*. For instance, the following rule reads as "If the agent sees something on the left, it turns for at least 5 units of time, until it sees it in the centre".

```
see(left, X) until see(centre, X) min 5~> turn(left)
```

This rule fires from the moment `see(left, _)` is inferable, until `see(centre, _)` is true even if `see(left, _)` is not inferable any more, at least for 5 units of time. Consequently, the state transition semantics needs to remember from which time stamp this rule has been firing.

All in all, for a Single Bound Program, neither the evaluation of the condition nor the assignment of the action requires the time stamps. Thus we can drop the time stamps from the operational semantics.

BeliefStore

Recall that in TeleoR the terms in BeliefStore are ordered. The BeliefStore state BS is used to define $fire$ has $\vdash_f$ operator defined, where

$$BS \vdash_f C\theta$$

holds if and only if the ground instance $C\theta$ is inferable from BS, and θ the *first* inferable binding. That is, $C\theta$ is the "first" grounding among all the groundings of C which is in BS. This matters as, for a rule $C \sim> A$, different bindings of C might instantiate A differently.
Recall that we assumed that none of the rules in Single Bound Programs are ambiguous. That is, there are no rules such that there are multiple instances of the guard which lead to different instantiation of the action. In other words, for any rule $C \sim> A$, all the bindings for C should bind the variables in A to the same values.
This means that the BeliefStore for a Single Bound Program does not need the notion of ordering among the terms. Besides, as Single Bound Program do not have user-defined relations, the BeliefStore consists only of dynamic facts.
Therefore a BeliefStore state for a Single Bound Program only needs to know which dynamic fact currently holds. Let us define a signature for these simplified BeliefStore states:

Definition 3.1 (BS_C)**.** *BS_C is a signature which denotes BeliefStore states, which consist only of dynamic facts, which are not ordered.*

Definition 3.2 ($\vdash$)**.**

$$\forall BS \in BS_C \; BS \vdash C\theta$$

holds if and only if the ground instance $C\theta$ is inferable from BS.

Previous BeliefStore States and Action Just Taken

As we mentioned before, later we want to specify fairness conditions to rule out unreasonable behaviour of the environment. To do so, we need the previous BeliefStore state and the action which was just taken as a part of evaluation state. We justify this in the next chapter.

3.3.3 Transition System

Now we define a transition system $P = (S_P, I_P, R_P)$, where S_P is a set of states, I_P is a set of initial states, and R_P is a relation which defines the transition, for a Single Bound Program whose starting task is $\boldsymbol{SP}$ and the maximum call depth is $MaxDp \in \mathbb{Z}$.
First, let us define a auxiliary function:

Definition 3.3. *For a sequence s, $length(s)$ is a function takes a sequence and returns its length.*

Let us define a set of possible states S_P as follows:

Definition 3.4 (A set of possible states S_P)**.** *A state in S_P, a possible set of states of a transition system which represents a Single Bound Program whose starting task $\boldsymbol{SP}$ and the maximum call depth $MaxDp \in \mathbb{Z}$ is formalised as a tuple:*

$$(BS, LBS, MaxDp, SP, FrdRules, Acts, LActs)$$

where

- *$BS \in BS_C$ is the current BeliefStore state*
- *$LBS \in BS_C$ is the previous BeliefStore state*
- *$FrdRules$ is the sequence of n, $0 \leq n \leq MaxDp$ tuples:*

  $$(\boldsymbol{P_1}, R_1, \theta_1, BS), ..., (\boldsymbol{P_n}, R_n, \theta_n, BS)$$

  where $\forall i : 1 \leq i \leq n$ $\boldsymbol{P_i}$ is a partially instantiated procedure call, and $\forall i : 1 \leq i \leq n$ $\boldsymbol{R_i}$ is an integer.
- *$Acts$ is a set of primitive actions which is chosen according to BS, or an action to indicate an error state. (i.e. nfr_fail or md_fail)*
- *$LActs$ is a set of primitive actions which was chosen in the previous state*

For the ease of notation, let us declare some "getter" functions:

Definition 3.5.

$$\begin{aligned}
&BS((BS, LBS, MaxDp, SP, FrdRules, Acts, LActs)) = BS \\
&LBS((BS, LBS, MaxDp, SP, FrdRules, Acts, LActs)) = LBS \\
&Acts((BS, LBS, MaxDp, SP, FrdRules, Acts, LActs)) = Acts \\
&LActs((BS, LBS, MaxDp, SP, FrdRules, Acts, LActs)) = LActs
\end{aligned}$$

Definition 3.6 (fire)**.** *$fire$ is a relation on a partially instantiated procedure call, integer, binding and BS_C, defined as:*

$$fire(\boldsymbol{P}, R, \theta, BS) \equiv \exists \theta' BS \vdash \boldsymbol{P}_R \theta' \wedge \theta \subseteq \theta' \wedge no\_higher\_fireable\_rule(\boldsymbol{P}, R, BS)$$

where

$$no\_higher\_fireable\_rule(\boldsymbol{P}, R, BS) \equiv \forall R' [\ 1 \leq R' < R \rightarrow \nexists \theta BS \vdash \boldsymbol{P}_{R'} \theta\ ]$$

On S_P, define an unary relation *valid*:

Definition 3.7 (*valid*)**.** *For a state $s = (BS, LBS, MaxDp, SP, FrdRules, Acts, LActs) \in S_P$, define a unary relation $valid(s)$ which holds if and only if the state satisfies the following conditions, where $\forall i\colon 1 \leq i \leq n$ $\boldsymbol{P_{i,R_i}} = (\boldsymbol{K_i} \sim> \boldsymbol{A_i})$, $\forall i\colon 1 \leq i \leq n$ $\boldsymbol{A_i}\theta_i = \boldsymbol{P_{i+1}}$ and $\boldsymbol{P_1} = \boldsymbol{SP}$.*

$$\forall i\colon 1 \leq i \leq n \quad fire(\boldsymbol{P_i}, R_i, \theta_i, BS)$$

$$\forall i\colon 1 \leq i \leq n \quad procCall(\boldsymbol{P_{i+1}})$$

$$\begin{aligned}
&[\ Acts = \boldsymbol{A_n}\theta_n \wedge primActs(Acts) \wedge 0 < n \leq MaxDp\ ] \\
\vee\ &[\ Acts = md\_fail \wedge procCall(A_n) \wedge n = MaxDp\ ] \\
\vee\ &[\ Acts = nfr\_fail \\
\wedge\ &\exists P \quad [\ \neg\exists(R, \theta) fire(P, R, \theta, BS) \\
&\quad \wedge \quad [\ MaxDp > n > 0 \wedge P = \boldsymbol{A_n}\theta_n \wedge procCall(P)\ ] \\
&\quad \vee \quad [\ n = 0 \wedge P = \boldsymbol{SP}\ ]\ ]
\end{aligned}$$

The third condition is a disjunction of three terms, and each term characterises one kind of valid evaluation state.
The first term represents the non-erroneous state. In this kind of state, a set of primitive actions is successfully chosen, because the sequence of fireable rules given the BeliefStore state BS, is such that the length does not exceed $MaxDp$, and the action of the last fired rule is a set of primitive action.
The second term represents an erroneous state due to a "call stack over flow". In this state, the sequence of fireable rules given the BeliefStore state BS has length $MaxDp$ and the action of the last fired rule is a procedure call.
The third term represents another erroneous state where there is a call to a procedure such that none of the guard of its rules holds for BS.

We define R_P as follows:

Definition 3.8 (State transition R_P)**.** *A valid transition is defined with a relation R_P defined on two states $s, s' \in S_P$:*

$$\begin{aligned} R_P(s,s') \iff & LBS(s') = BS(s) \wedge LActs(s') = Acts(s) \\ & \wedge\ Acts(s) \neq md\_fail \wedge Acts(s) \neq nfr\_fail \\ & \wedge\ valid(s) \wedge valid(s') \\ & \wedge\ BS' \in next(BS) \end{aligned}$$

Note that the erroneous states do not transition to any other state.

The initial states I_P is defined as:

Definition 3.9 (Initial States I_P)**.**

$$\begin{aligned} I_P = \{(BS_I, \{\}, & MaxDp, \boldsymbol{SP}, FrdRules_I, Acts_I, \{\}) \\ & | valid((BS_I, \{\}, MaxDp, \boldsymbol{SP}, FrdRules_I, Acts_I, \{\})), BS_I \in init\} \end{aligned}$$

Now we defined the transition system P. For later purpose, we want to define a function that takes a BeliefStore state, task call and the maximum call depth, and returns the action the agent chooses. To do this, first let us show that it is possible to define such a function; that is, we want to show that the action chosen and the rules fired by is dependent only on the current BeliefStore state, the task call and the maximum call depth.

Remark. *For an arbitrary state $s \in S_P$, if we know its BeliefStore state $BS = BS(s)$, there is only one possible value for $Acts(s)$ and a sequence rules that fires for $BS(s)$, for a Single Bound Program with the maximum call depth $MaxDp$ and the starting procedure call $\boldsymbol{SP}$.*

Proof. (sketch)
Let $s = (BS, LBS, MaxDp, \boldsymbol{SP}, FrdRules, Acts, LActs)$,
$s' = (BS, LBS', MaxDp, \boldsymbol{SP}, FrdRules', Acts', LActs')$.
Assume $valid(s)$ and $valid(s')$.
Let $n = length(FrdRules)$ and $n' = length(FrdRules')$.
Let $FrdRules = (\boldsymbol{P_1}, R_1, \theta_1, BS)...(\boldsymbol{P_n}, R_n, \theta_n, BS)$ and $FrdRules' = (\boldsymbol{P'_1}, R'_1, \theta'_1, BS)...(\boldsymbol{P'_n}, R'_n, \theta'_n, BS)$.

Assume $FrdRules \neq FrdRules'$.
Then $\exists i : 1 \leq i \leq min(n,n')\ R_i \neq R'_i \wedge P_i = P'_i$ or $\exists i : 1 \leq i \leq min(n,n')\ \theta_i \neq \theta'_i \wedge P_i = P'_i \wedge R_i = R'_i$.
Assume $\exists i : 1 \leq i \leq min(n,n')\ R_i \neq R'_i$. Assume $R_i < R'_i$. Then $fire(\boldsymbol{P_i}, R_i, \theta_i)$ contradicts with $no\_fireable\_rule(\boldsymbol{P_i}, R'_i, BS)$. By symmetry, we have that $R'_i < R_i$, does not hold either. Therefore

$R_i = R'_i$ and $\nexists i : 1 \leq i \leq min(n,n')\ R_i \neq R'_i$.
Assume $\exists i : 1 \leq i \leq min(n,n')\ \theta_i \neq \theta'_i \wedge P_i = P'_i \wedge R_i = R'_i$. Then there must be a rule where there is a multiple bindings for the right hand side, given the bindings for the left hand side, which contradicts with the assumption about Single Bound Programs. Therefore $\nexists i : 1 \leq i \leq min(n,n')\ \theta_i \neq \theta'_i \wedge P_i = P'_i \wedge R_i = R'_i$

Therefore we have $FrdRules = FrdRules'$. Then as $valid(s)$ and $valid(s')$, it follows that $Acts = Acts'$. □

As this remark holds, we can define the following function.

Definition 3.10 (action). *action is a function which takes $BS \in BS_C$, a partially ground procedure call $\boldsymbol{SP}$, a maximum call depth $MaxDp$, and returns $Acts$ such that*

$$\exists LBS, FrdRules, LActs\ valid((BS, LBS, MaxDp, SP, FrdRules, Acts, LActs))$$

3.4 Single Bound Program P as Interpreted System IS_P

We define the interpreted system, IS_P for a transition system P defined with *next* and *init*, representing the Single Bound Program whose starting task is $\boldsymbol{SP}$ and the maximum call depth is $MaxDp$.

3.4.1 Local States of the Environment Agent

The local states of the environment agent L_E is a tuple $(BS, LBS, LActs)$, where $BS, LBS \in BS_C$ and $LActs$ is a set of primitive actions.
For Single Bound Program, we could have also put these information in the private local state of the agent. However we thought it was better to put it in the local state of the environment agent, when we considered extending the implementation to support the system with multiple agents.
The interpreted system which can be defined in ISPL is more powerful than the definition of the interpreted system we use in this report, as one may define a fraction of local state of the environment which can be seen by each agent, using local observable variables. Therefore putting more information in the environment agent allows a higher flexibility, especially when the system with multiple agents are considered. For example, we thought it might make sense that some information is coherent between among multiple agents, such as the location of the object in the field. This can be achieved by the agents observing the corresponding fraction of the local state of the environment agent.
We put $LActs$ in the local state of the environment agent as well, as the action of an agent is visible to all other agents, therefore it makes sense that the *previous* action is visible to other agents too.

3.4.2 Local States of the Agent

The private local state of the agent L_{ag} is empty as there is nothing left in a state in P which we could map to it. Therefore we have the local state of the agent to be $((BS, LBS, LActs), ())$ where $(BS, LBS, LActs)$ is the local state of the environment agent.

3.4.3 Actions of the Environment Agent

In P, there is nothing we could map to action of the environment agent. As a place-holder, let $Act_E = \{update\}$.

3.4.4 Actions of the Agent

The possible set actions of the agent Act_{ag} correspond to the possible values for the return value of the protocol function of the agent, which is a set of ground primitive actions.

3.4.5 Local Protocol Function of the Environment Agent

The local protocol function of the environment agent P_E takes a local state of the environment and returns "update", its only possible action. Formally, $P_E : L_E \rightarrow \mathcal{P}(Act_E)$ is defined as:

$$P_E(s) = \{update\}$$

3.4.6 Local Protocol Function of the Agent

In P, there is an outgoing transition from an evaluation state if and only if it is not an erroneous state. To reflect this, the local protocol function P_{ag} of the agent is such that it takes a local state of the agent $((BS, LBS, LActs), ())$ and returns an empty set if BS characterises an erroneous state, and returns the chosen sequence of primitive action otherwise. Formally, $P_{ag} : L_{ag} \times L_E \rightarrow \mathcal{P}(Act_{ag})$ is defined as:

$$P_{ag}(((BS, LBS, LActs), ())) = \begin{cases} \{\} & \text{if } action(BS, \boldsymbol{SP}, MaxDp) \\ & \qquad \in \{md\_fail, nfr\_fail\} \\ \{action(BS, \boldsymbol{SP}, MaxDp)\} & \text{otherwise} \end{cases}$$

3.4.7 Local Evolution Function of the Environment Agent

The local evolution function of the environment agent $\tau_E : L_E \times (Act_E \times Act_{ag}) \rightarrow L_E$ takes the local state of the environment agent $(BS, LBS, LActs)$ and the global action $(a, update)$, and returns a set of new local states (BS', BS, a) where $BS' \in next(BS)$.

$$\begin{aligned} &\tau_E((BS, LBS, LActs), (a, update)) = \\ &\qquad \{(BS', BS, a) | BS' \in next(BS)\} \end{aligned}$$

3.4.8 Local Evolution Function of the Agent

The local evolution function of the agent $\tau_{ag} : (L_E \times L_{ag}) \times (Act_E \times Act_{ag}) \rightarrow L_{ag}$ takes a local state of the agent, $((BS, LBS, LActs), ())$ and returns an empty private local state of the agent, $()$.

$$\tau_{ag}(s, a) = \{()\}$$

3.4.9 Initial States

The set of initial global states I consists of $((BS, \{\}, \{\}), ())$, where $BS \in init$.

Again, for the ease of notation later on, let us define some "getter" function for the global states of this interpreted system.

Definition 3.11.

$$BS(((BS, LBS, LActs), ())) = BS$$
$$LBS(((BS, LBS, LActs), ())) = LBS$$
$$LActs(((BS, LBS, LActs), ())) = LActs$$

3.5 Correctness of the Mapping

In this section, we argue that the mapping from a Single Bound Program P to the interpreted system IS_P is correct. By correct, we mean that the validity of the modal specification evaluated at the initial states of P is preserved in the initial states of IS_P and vice versa.
We have not defined the class of specification we want to prove for P. We do this by defining the valuation function on P and IS_P for the propositional atoms that may appear in the specification. The valuation function for a transition system is a function which takes a propositional atom and returns the set of states in the system where the given atom holds.
We show that a modal specification evaluated at the initial states of P holds if and only if it holds at the initial states of IS_P by showing that there is a *bisimulation* between P and IS_P. By finding a bisimulation, we can claim that these two transition systems have exactly the same moves where the given valuation function is concerned, therefore the validity of the modal specification is preserved.

3.5.1 Valuation Function

Propositional Atoms AP

The natural candidate for specifications which the TeleoR developer may be interested in would be:

- The agent never goes to an error state; that is there is no evaluation state reachable from the initial state which chose nfr_fail or md_fail.
- The top-goal of the TeleoR program is achieved; that is, the condition of the top-goal evaluates to true when evaluated against the BeliefStore state.

Therefore, we want to be able to have propositional atoms about the actions chosen at each evaluation state, and which dynamic facts are in the BeliefStore.
We also want to have propositional atoms for the actions that were chosen in the previous state, and which dynamic facts held in the previous state, to define fairness condition. Again, this is described in Chapter 4. To define this set of propositional atoms, AP, let us define some terms:

Definition 3.12 (*groundToAtom*)**.** *groundToAtom is a bijective function which takes a ground term, and returns a propositional atom.*

Definition 3.13 (*groundToPreviousAtom*)**.** *groundToPreviousAtom is a bijective function which takes a ground term, and returns a propositional atom. We assume that the value it returns is never returned by groundToAtom for any input.*

Definition 3.14 (*allGroundings*)**.** *allGroundings is a function which takes the identifier of a relation or a TeleoR action and returns all possible groundings.*

Definition 3.15 (*facts*)**.** *facts is a set of the identifier of all the relations that are declared as a percept or belief of the agent.*

Definition 3.16 (*actions*)**.** *actions is a set of the identifier of all the actions that are declared as a discrete or durative action of the agent.*

Then AP contains the atoms below, and nothing else.

- $groundToAtom(ground\_fact) \cup groundToPreviousAtom(ground\_fact)$
  For all $ground\_fact$ in $allGroundings(fact)$ for all $fact \in facts$.
- $groundToAtom(ground\_action) \cup groundToPreviousAtom(ground\_action)$.
  For all $ground\_action$ in $allGroundings(action)$ for all $action \in actions$.
- $groundToAtom(md\_fail) \cup groundToPreviousAtom(md\_fail)$
- $groundToAtom(nfr\_fail) \cup groundToPreviousAtom(nfr\_fail)$

Valuation Function of P

Let S_P be the set of states in P. Then valuation function of P, $V_P \in AP \to \mathcal{P}(S_P)$ is defined as follows:

$$V_{IS_P}(p) = \begin{cases} \{s|s \in S_P\ BS(s) \vdash F\} & \text{if } \exists f \in facts, \exists F \in allGroundings(f), p = groundToAtom(F) \\ \{s|s \in S_P\ LBS(s) \vdash F\} & \text{if } \exists f \in facts, \exists F \in allGroundings(f), \\ & \quad p = groundToPreviousAtom(F) \\ \{s|s \in S_P\ A \in Acts(s)\} & \text{if } \exists a \in actions \cup \{md\_fail, nfr\_fail\}, \\ & \quad \exists A \in allGroundings(a), p = groundToAtom(A) \\ \{s|s \in S_P\ A \in LActs(s)\} & \text{if } \exists a \in actions \cup \{md\_fail, nfr\_fail\}, \\ & \quad \exists A \in allGroundings(a), p = groundPreviousToAtom(A) \end{cases}$$

Valuation Function of IS_P

Let S_{IS_P} be the set of global states in IS_P. Then valuation function of IS_P, $V_{IS_P} \in AP \to \mathcal{P}(S_{IS_P})$ is defined as follows:

$$V_P(p) = \begin{cases} \{s|s \in S_{IS_P}\ BS(s) \vdash F\} & \text{if } \exists f \in facts, \exists F \in allGroundings(f), p = groundToAtom(F) \\ \{s|s \in S_{IS_P}\ LBS(s) \vdash F\} & \text{if } \exists f \in facts, \exists F \in allGroundings(f), \\ & \quad p = groundToPreviousAtom(F) \\ \{s|s \in S_{IS_P}\ A \in action(s)\} & \text{if } \exists a \in actions \cup \{md\_fail, nfr\_fail\}, \\ & \quad \exists A \in allGroundings(a), p = groundToAtom(A) \\ \{s|s \in S_{IS_P}\ A \in LActs(s)\} & \text{if } \exists a \in actions \cup \{md\_fail, nfr\_fail\}, \\ & \quad \exists A \in allGroundings(a), p = groundPreviousToAtom(A) \end{cases}$$

3.5.2 Bisimulation

Bisimulation on Kripke models can be defined as follows[22]:

Definition 3.17 (Bisimulation). *Let $\mathcal{M} = (S, R, V)$ and $\mathcal{M}' = (S', R', V')$ be Kripke models, where $V \in A \to \mathcal{P}(S)$ and $V' \in A \to \mathcal{P}(S')$ where A is a set of propositional atoms.*
Let $t \in S, t' \in S'$. Then, bisimulation between (S, t) and (S', t') as a binary relation $B \subseteq S \times S'$ such that:

- *$B(t, t')$.*

and for all $u \in S$ and $u' \in S'$ such that $B(u, u')$:

- *For all $p \in A$, $u \in V(p) \iff s' \in V'(p)$.*
- ***(forth)** If $v \in S$ and $R(u, v)$, then there is $v' \in S'$ with $R'(u', v')$ and $B(v, v')$*
- ***(back)** If $v' \in S'$ and $R'(u', v')$, then there is $v \in W$ with $R(u, v)$ and $B(v, v')$.*

A term "bisimilar" is defined as follows[22]:

Definition 3.18 (Bisimilar). *$(\mathcal{M}, t)$ and $(\mathcal{M}', t')$ are bisimilar if there exists a bisimulation between $\mathcal{M}, t$ and $\mathcal{M}', t'$.*

We have the following theorem[22]:

Theorem 3.1 (Bisimulation invariance of modal formulae). *Let $(\mathcal{M}, t)$ and $(\mathcal{M}', t')$ be bisimilar and let F be any modal formula. Then $\mathcal{M}, t \vdash A \iff \mathcal{M}', t' \vdash A$*

3.5.3 Proof

We want to show that an arbitrary modal formula F is true at the initial states of a Single Bound Program P if and only if it is true at the initial states IS_P. We do this by first defining a Kripke model for P and IS_P, and finding a bisimulation which relates all the initial states of these Kripke models to each other.

Let $\mathcal{M}_P = (S_P, R_P, V_P)$ where:

- S_P is the set of all possible evaluation states of P
- R_P is the transition relation of P
- V_P is as previously defined in Section 3.5.1

Let $\mathcal{M}_{IS_P} = (S_{IS_P}, R_{IS_P}, V_{IS_P})$ where:

- S_{IS_P} is the set of all possible global states in IS_P.
- $R_{IS_P} \subseteq S_{IS_P} \times S_{IS_P}$ and defined as

$$R_{IS_P}(s, s') \iff \exists a \; s' \in \tau(s, (a, update))$$

  where τ is the global evolution function of IS_P.

- V_{IS_P} is as previously defined in Section 3.5.1

Now we show the following remark:

Remark. *The following relation B is a bisimulation:*

$$B(s_P, s_{IS_P}) \iff valid(s_P) \wedge BS(s_p) = BS(s_{IS_P}) \wedge LBS(s_p) = LBS(s_{IS_P}) \wedge LActs(s_p) = LActs(s_{IS_P})$$

Proof. We prove this by showing:

1. For an arbitrary $s_P \in S_P$ and $s_{IS_P} \in S_{IS_P}$ where $valid(s_P)$, if $B(s_P, s_{IS_P})$, then for all $p \in AP$, $s_P \in V_P(p) \iff s_{IS_P} \in V_{IS_P}$.
2. All initial states of P and IS_P can be related with B.
3. Assuming that $B(s_P, s_{IS_P})$, the following properties hold:
   - For all $p \in AP$, $s_P \in V_P(p) \iff s_{IS_P} \in V_{IS_P}(p)$.
   - (**forth**) If $s'_P \in S_P$ and $R_P(s_P, s'_P)$, then there is $s'_{IS_P} \in S'_{IS_P}$ with $R_{IS_P}(s_{IS_P}, s'_{IS_P})$ and $B(s'_P, s'_{IS_P})$
   - (**back**) If $s'_{IS_P} \in S'_{IS_P}$ and $R_{IS_P}(s_{IS_P}, s'_{IS_P})$, then there is $s'_P \in S_P$ with $R_P(s_P, s'_P)$ and $B(s'_P, s'_{IS_P})$.

Remark 1
Take an arbitrary atom $p \in AP$ We consider the four cases.
Assume there is F such that $\exists f \in facts\ F \in allGroundings(f)\ p = groundToAtom(F)$. Then $V_P(p) = \{s | s \in S_P\ BS(s) \vdash F\}$ and $V_{IS_P}(p) = \{s | s \in S_P\ BS(s) \vdash F\}$. As $BS(s_P) = BS(s_{IS_P})$, $s_P \in V_P(p) \iff s_{IS_P} \in V_{IS_P}$.

Assume there is F such that $\exists f \in facts\ F \in allGroundings(f)\ p = groundToPreviousAtom(F)$. Then $V_P(p) = \{s | s \in S_P\ LBS(s) \vdash F\}$ and $V_{IS_P}(p) = \{s | s \in S_P\ LBS(s) \vdash F\}$. As $LBS(s_P) = LBS(s_{IS_P})$, $s_P \in V_P(p) \iff s_{IS_P} \in V_{IS_P}$.

Assume there is A such that $\exists a \in actions\ A \in allGroudings(a)\ p = groundToAtom(A)$. Then $V_P(p) = \{s | s \in S_P\ A \in Acts(s)\}$ and $V_{IS_P}(p) = \{s | s \in S_P\ A \in action(s)\}$. By definition of *action* and we have $valid(s_P)$, $action(s_{IS_P}) = Acts(s_P)$ if $BS(s_{IS_P}) = BS(s_P)$.

Assume there is A such that $\exists a \in actions\ A \in allGroudings(a)\ p = groundToPreviousAtom(A)$. Then $V_P(p) = \{s | s \in S_P\ A \in LActs(s)\}$ and $V_{IS_P}(p) = \{s | s \in S_P\ A \in LActs(s)\}$. As $LBS(s_P) = LBS(s_{IS_P})$, $s_P \in V_P(p) \iff s_{IS_P} \in V_{IS_P}$.

Remark 2
The initial states of P is defined as:

$$\begin{aligned} I_P = \{s_P | s_P \in S_P, BS(s_P) \in init \\ valid(s_P), LBS(s_P) = \{\}, LActs(s_P) = \{\}\} \end{aligned}$$

The initial states of IS_P is defined as:

$$\begin{aligned} I = \{s_{IS_P} | s_{IS_P} \in S_{IS_P}, BS(s_{IS_P}) \in init \\ LBS(s_{IS_P}) = \{\}, LActs(s_{IS_P}) = \{\}\} \end{aligned}$$

Clearly, we can draw a relation B between all states in I_P and I.

Remark 3 Take arbitrary $s_P \in S_P$ and $s_{IS_P} \in S_{IS_P}$ and assume $R(s_P, s_{IS_P})$.
From remark 1, if $R(s_P, s_{IS_P})$, we have that for all $p \in AP$ $s_P \in V_P(p) \iff V_{IS_P}(p)$.
By definition of *action* and we have $valid(s_P)$, $action(s_{IS_P}) = Acts(s_P)$ if $BS(s_{IS_P}) = BS(s_P)$. Let $a = Acts(s_P) = action(s_{IS_P})$.
If $a \in md\_fail, nfr\_fail$, neither s_P or s_{IS_P} has an outgoing transition. Therefore forth and back property trivially holds.
Otherwise, s_P transits to:

$$\begin{aligned} N_P = \{s'_P | s'_P \in S_P, BS(s'_P) \in next(BS(s_P)), \\ valid(s'_P), LBS(s'_P) = BS(s_P), LActs(s'_P) = a\} \end{aligned}$$

s_{IS_P} transits to:

$$\begin{aligned} N_{IS_P} = \{s'_{IS_P} | s'_{IS_P} \in S'_{IS_P}, BS(s'_{IS_P}) \in next(BS(s_{IS_P})), \\ LBS(s'_{IS_P}) = BS(s_{IS_P}), LActs(s'_{IS_P}) = a\} \end{aligned}$$

Clearly, we can draw a relation B between all states in N_P and N_{IS_P}. Therefore we have forth and back property. $\square$

3.6 Summary

In this chapter we discussed what Single Bound Programs are and how the behaviour of the evaluator thread can be defined as a transition system. We left out the definition of functions which defines the sequence of the BeliefStore update. In the next chapter we discuss how they are defined. Then we define an interpreted system which is equivalent to this transition system. We formally show their equivalence by stating a bisimulation.

Chapter 4

Specifying a Reasonable Environment

In the previous chapter, we specified the interpreted system which represents all the possible computation of the agent running a Single Bound Program, given *next* and *init*. In this chapter, we formalise these two functions, thus the set of all possible sequences of BeliefStore updates in the environment.

Preferably, we would like to verify whether the agent successfully attained its top-goal in a reasonable environment. In other words, we want to exclude unlikely sequences of BeliefStore updates. For instance, if we had an agent that collects a bottle, we do not want to bother with an environment where there is no bottle at all.

The question is how we could know what a "reasonable" environment is. The best option would be to work it out from the information in the TeleoR program. Unfortunately, such information is scarce, as we will see in this chapter.

Instead of attempting to squeeze out the little information it may contain, we decided to allow the users to feed an extra file which we call environment configuration. The environment configuration contains statements that hold in the environment the agent may be situated in. Naturally, the empty environment configuration is equivalent to making no assumption at all on the BeliefStore updates, thus the specification is checked against a non-deterministic environment. We discuss what would be a useful class of statements.

We might have alternatively had an interpreted system which assumes nothing of this environment, and specify the assumption as a part of the specification. In this chapter we show that this is not possible, hence it should be expressed as a part of the definition of the interpreted systems.

Finally we discuss a precise semantics of the environment configuration on the interpreted system for Single Bound Program.

4.1 What Could We Know About The Environment?

We would like to verify that a specification holds for the TeleoR program, given that the agent is in a reasonable environment. The difficulty is to work out what we can assume for this reasonable environment. If we could deduce this from the TeleoR program itself, naturally that would be optimal. In this section, we see that in some set-ups, TeleoR programs can give us a limited amount of information on how the BeliefStore might be updated.

4.1.1 Percept Handlers

The agent is started by the built-in Qulog action `start_agent(Name, Handle, Convention)`, where[8]:

- `Name`
  The name of the agent.
- `Handle`
  The message address of the robot interface or simulation with which the agent will interact.
- `Convention`
  The keyword that specifies how the dynamic facts are updated.

The agents are notified of the changes in the sensor inputs with the messages from `Handle`. These messages may be in three different forms, depending on the value of `Convention`.

- `all`
  The message is a list of all the dynamic facts that hold.
- `updates`
  The message contains which of the dynamic facts that are in the BeliefStore now, should be removed and added.
- `user`
  The format of the message and how they are handled is specified in the built-in action `handle_percepts_` in the TeleoR program.

If `Convention` is `all` or `updates`, all the information on how the BeliefStore is updated, is in the simulation file, which may be written in Python, Java, C and C++.[6]. However if `Convention` is `user`, then we may able to gain some information about how the BeliefStore is updated. The following snippet is from bottle.qlg that comes with Qulog 0.4[8].

```
percept_message_type ::= set(percept) | unset(percept)

handle_percepts_(Ps) :: ground(Ps) ~>>
   forall P (P in Ps & type(P, percept_message_type)
         ~>> handle_percept(P))
handle_percept(set(gripper_open))
         ~>> forget holding; remember gripper_open
handle_percept(unset(gripper_open)) ~>> forget gripper_open
```

From this declaration, we know that if `gripper_open` holds, `holding` would never hold. Though this might be useful information, the applicability is limited as we may only take programs with "user" configuration.

4.1.2 Attached Qulog Action

In TeleoR, it is possible to specify which Qulog action may be executed after a primitive action. This Qulog action may add or remove the dynamic facts in the BeliefStore. Again, the following snippet is from bottle.qlg in Qulog 0.4 distribution[8].

```
..
true ~> open_gripper ++ update_and_communicate_count(OthrAg)
..
update_and_communicate_count: pedro_handle
update_and_communicate_count(OthrAg) ~>>
                          collected +:= 1;
                          count($collected) to OthrAg
```

From this, we may deduce that, if the above rule is selected, `collected` is incremented. However as can be seen, attached Qulog actions are used to record some knowledge-based information and possibly communicate it to the others. Therefore if the program is heavily based on sensor inputs, we do not learn much about the dynamic facts.

4.2 Environment Configuration

We saw that the information of how the BeliefStore is updated is scarce in most of the TeleoR programs. Rather than attempting to utilise what little clues they might contain, we decided to take an additional file which we refer to as *environment configuration* which contains statements that hold in the environment.
In this section, we specify the environment configuration rules, the statements which compose the environment configuration. We discuss what kind of rules would be useful to define the environment. What we want to achieve with the environment configuration is to check the specification only on the path with reasonable BeliefStore updates. Intuitively, it should be possible to have a specification such as "if the global state changes in this manner, then the agent satisfies this specification". This way we can simply have an interpreted system which contains all possible BeliefStore updates, rather than defining an interpreted system which reflects the environment configuration rules. Unfortunately we can not have the statements that we want to express about the environment as a part of the specification, due to the limitation in the specification language provided by MCMAS. We show that later in this section.
Then we formally define the environment configuration rules, and how we can define *next* and *init* from these rules.

4.2.1 Wish List

We would like the environment configuration rules to be expressive enough for a user to express their assumption fully. The following are what we thought the user may want to specify:

- A definite effect of the action
  We may assume that some actions would always succeed, and therefore know about how some dynamic facts are modified in the BeliefStore. For instance, we may think that the gripper of an agent is sophisticated enough that opening it would always result to adding `open_gripper` and removing `close_gripper`.

- An eventual effect of the action
  We may assume that if a durative action continues for a long enough time, or if a discrete action is taken for enough times consecutively, then it should succeed at one point. For instance, we may assume that, if the agent pushes the door for a while, it should at some point open it.

- A possible effect of the action
  We may assume that taking some action may result in a change in some dynamic fact. For instance, we may assume that if the agent turns around, some new object might come in its sight, or some objects might go out of its sight.
- How the environment is at the start
  We may assume that the environment is in a certain state at the start. For instance, an agent that builds a tower using blocks may assume that all of the blocks are on the table, rather than on each other.
- Which dynamic fact may not change without the action of the agent
  We may assume that, without the agent doing anything, some factor in the environment will not change. For instance, we may assume that without the agent moving, the distance from an object will not change.

Preferably, we would like the environment configuration rules to be expressive enough for such statements.

4.2.2 Limitation of Specification Languages

Naturally one may think that it must be possible to express a specification which explicitly states the assumption, thus it is not necessary to change the ISPL. Unfortunately, the specification language for MCMAS, CTL, is not expressive enough to do this.
Take the assumption on the definite effect as an example. Say we want to assume that "if p holds and the agent takes $act1$, then in the next state q will always be true", and we want to verify that, under this assumption "the agent will always be idle at one point". Let $idle$ hold if the agent is idle, $taken\_act1$ be true at the state if the local action of the agent that led to its state is $act1$, and $prev\_p$ be true if p holds in the previous state.
It seems sensible to express this as:

$$AG(((prev\_p \wedge taken\_act1) \rightarrow q) \rightarrow AF(idle)) \tag{4.1}$$

However, this is not quite what we want. Consider the following transition system with two paths, where the initial state is marked with red:

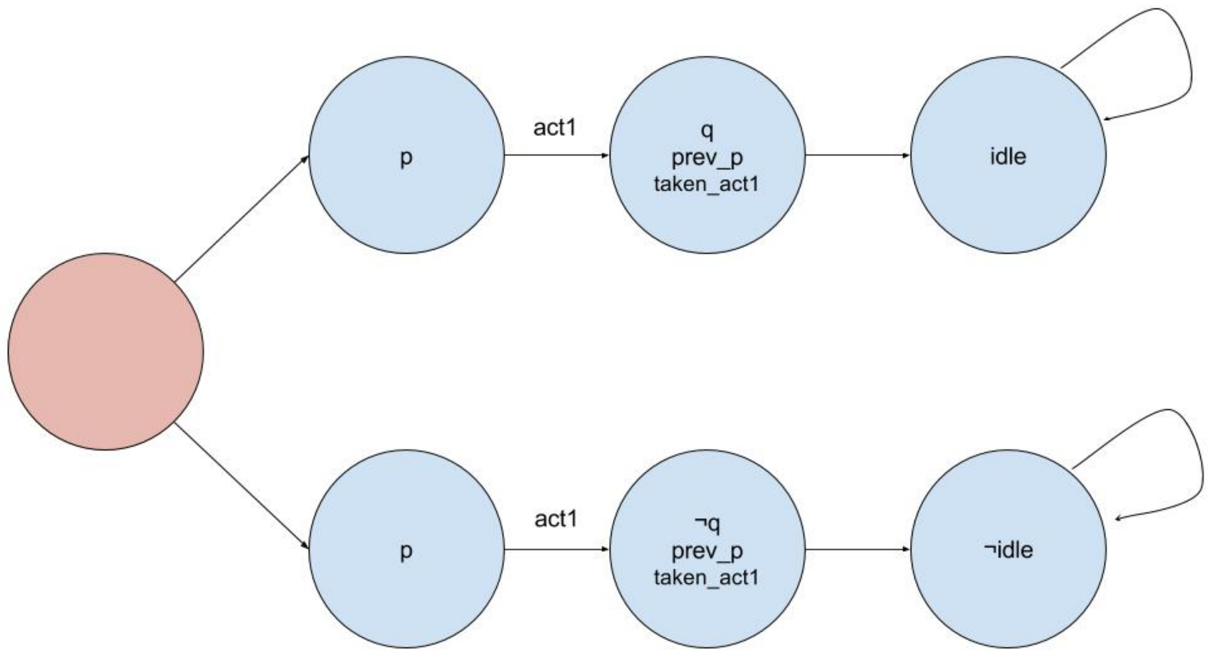

Figure 4.1: Example transition system.

What we would like to express is that if the environment conforms to the assumption, then we will at some point have *idle*. In other words, we would like to express that “for all paths from the initial state that follows the assumption of the environment, there is always a state where *idle* would hold”. This statement should hold in the transition system above, as only the path at the top satisfies the assumption and we only need to have $AF(idle)$ for that path.
However (4.1) does not hold. At the initial state $(prev\_p \wedge taken\_act1) \rightarrow q$ holds, yet $AF(idle)$ does not as the path on the bottom does not have a state which has *idle*.
As specifying the assumption as a part of the specification will not do, we are left to modify the ISPL such that all paths from the initial state satisfy the assumptions.

4.2.3 Formal Definition of Environment Configuration

In this subsection, we present the formal definition of the environment configuration we implemented. First we define $fact\ modifiers$ which we use to specify changes in the BeliefStore state that we wish to see in the rules.

Definition 4.1 (Fact modifier)**.** *A fact modifier is a tuple, which specifies the modification of a BeliefStore state. It can be either of the following:*

- *(remember, F) where F is a template term of a relation declared as percept or belief. When this modifier is applied to the BeliefStore, the grounding of F is added to the BeliefStore.*

- *(forget, F) where F is a template term of a relation declared as percept or belief. When this modifier is applied to the BeliefStore, the grounding of F is added to the BeliefStore.*

We now define the environment configuration, which allows us to specify the possible sequences of the BeliefStore updates.

Definition 4.2 (Environment configuration)**.** *Environment configuration is a set of tuples, which can be one of the following:*

- $(init, C)$ *where C is a BeliefStore clause. This rule specifies the conditions that should hold for the first BeliefStore states that the agent may experience.*
- $(dontFlip, f)$ *where* $f \in facts$*. This rules specifies that dynamic fact with identifier* f *does not change, unless specified otherwise by other rules.*
- $(definitely, C, A, Ms)$ *where C is a BeliefStore clause, A is a primitive action and* Ms *is a sequence of fact modifiers. This rule specifies that if C holds at the current state and the action A is taken, then the next BeliefStore will have a change specified in* Ms*.*
- $(maychange, C, A, F)$ *where C is a BeliefStore clause, A is primitive action and F is a template term of a relation declared as percept or belief. This rule specifies that if C holds at the current state and the action A is taken, then the next BeliefStore may have facts obtained from grounding F added or removed.*
- $(eventually, C, A, Ms)$ *where C is a condition that may be used as a triggering condition of the TeleoR rule, A is a primitive action and* Ms *is a sequence of fact modifiers. This rule specifies that if the agent continues to take action A from the state where C holds, then eventually the BeliefStore will have a change specified in* Ms*.*

We restrict that there can be only one rule with definitely per action signature. Consider an environment configuration with the following rules.

$$(definitely, move(X), (remember, see(bottle)))$$
$$(definitely, move(X), (forget, see(bottle)))$$

When the agent takes an action $move(1)$*, it is ambiguous whether the first rule should be applied first, therefore* $see(bottle)$ *would not be in the next BeliefStore, or the second rule should be applied first and therefore* $see(bottle)$ *will be in the next BeliefStore, along with other facts.*

4.2.4 Semantics of Environment Configuration on IS_P

In this subsection, we formally specify the semantics of rules defined in the previous subsection on IS_P, by defining *init* and *next* functions that we used to define IS_P.
First let us define some auxiliary terms:

Definition 4.3 (subsequence)**.** *If seq is a sequence , $seq_{i:}$ is a subsequence of seq from index i, including the element at index i.*

Definition 4.4 (bindings)**.** *bindings is a function which takes a template term and returns a set of bindings such that it does not instantiate input term to false. For instance:*

$$bindings(foo(X)\&X = 1)$$

if foo may take 1 or 2, it returns a set of one element, which is a binding which binds X to 1. A binding which binds X to 2 is not included, as the whole term would evaluate to false.

Definition 4.5 ($getModifiers$)**.** *$getModifiers$ is a function which takes a BeliefStore state BS, an environment configuration EC and a ground action a and returns a tuple of a list of modifiers which should be applied to the BeliefStore and set of bindings for the modifiers. If there is a "definitely" rule which applies for BS and a, $getModifiers$ returns the following:*

$$\begin{aligned} &getModifiers(BS, EC, a) = (Ms, \theta s)\\ &\qquad\qquad where\ (definitely, C, A, Ms) \in EC, \theta s = \{\theta | BS \vdash C\theta' \wedge a = A\theta\}\end{aligned}$$

Note that we assumed that there is at most one "definitely" rule per identifier of a primitive action. Otherwise,

$$getModifiers(BS, EC, a) = (\{\}, \{\})$$

Definition 4.6 (changing)**.** *changing is a function which takes a BeliefStore state BS, an environment configuration EC and returns all the ground facts which may flip due to "maychange" rule in EC which applies for BS.*

$$changing(BS, EC) = \{F\theta | (maychange, C, A, F) \in EC, \theta \in \{\theta | BS \vdash C\theta' \wedge A\theta \in action(BS)\}$$

Then *init* can be defined as:

Definition 4.7 (init)**.** *init is a function which returns all possible BeliefStore states that the agent may have at the start.*

$$init = \{BS \mid BS \in BS_C\ \forall(init, C) \in EC\ \forall\theta \in bindings(C)\ BS \vdash C\theta\}$$

next can be defined as:

Definition 4.8 (next)**.** *$next(BS)$ is a function which takes the current BeliefStore state and returns all the BeliefStore state $BS' \in BS_C$ that the agent may see next. Specifically, the BeliefStore states returned are such that for all possible dynamic fact F with identifier f:*

$$\begin{aligned} &[\exists(dontFlip, f) \in EC \wedge F \notin changing(BS, EC)\ \wedge\ F \notin modified(BS, EC) \rightarrow\\ &\qquad\qquad BS' \vdash F \iff BS \vdash F]\\ &\quad\wedge\\ &[\exists(dontFlip, f) \in EC \wedge F \notin changing(BS, EC) \wedge F \in modified(BS, EC) \rightarrow\\ &\qquad\qquad BS' \vdash F \iff modify(BS, BS, EC, action(BS)) \vdash F]\end{aligned}$$

where $modifed(BS, EC)$ is a function that returns a set of dynamic facts that are modified in $modify(BS, BS, EC, action(BS))$, which is defined as follows:

$$modify(BS, BS', EC, Acts) = \begin{cases} BS' & \text{if } length(Acts) = 0 \\ modify'(BS, BS'', EC, Acts_{2:}) & otherwise \\ \quad where & \\ \quad BS'' = modify(BS', (Ms, \theta s)) & \\ \quad (Ms, \theta s) & \\ \qquad = getModifiers(BS, EC, Acts_1) & \end{cases}$$

$$modify'(BS, (Ms, \theta s)) = \begin{cases} BS & if\ |\theta s| = 0 \\ & \quad \vee (Ms, \theta s) = (\{\}, \{\}) \\ modify'(BS', (Ms, \theta s')) & otherwise \\ \quad where & \\ \quad BS' = modify''(BS, Ms, \theta) & \\ \quad \theta \in \theta s & \\ \quad \theta s' = \theta s \backslash \{\theta\} & \end{cases}$$

$$modify''(BS, Ms, \theta) = \begin{cases} BS & if\ length(Ms) = 0 \\ modify''(BS', Ms_{2:}, \theta) & otherwise \\ \quad where & \\ \quad BS' = modify'''(BS, Ms_1, \theta) & \end{cases}$$

$$modify'''(BS, (remember, F), \theta) = BS \cup \{F\theta\}$$
$$modify'''(BS, (forget, F), \theta) = BS \backslash \{F\theta\}$$

Note that the set operation $\cup$ and $\backslash$ is used on a BeliefStore state to represent "adding" and "removing" a set of dynamic facts.
Note also that with an empty environment configuration, *next* returns a set of all possible BeliefStore states.
As can be seen, to express the semantics of the statements we had in our "wish list", we only needed the current BeliefStore state. If one may wish to extend the environment configuration rules, it might happen that the current BeliefStore is not enough. In this case the interpreted system IS_P should be extended so the local state contains whatever information the new configuration rules depend on, and feed it to *next* as a parameter.

Fairness

With $eventually(C, A, Ms)$ rules, we want to express that if the BeliefStore has C and the agent takes A, then eventually the next BeliefStore state would have changed as specified in Ms. We can not express this by modifying the model, and as we discussed previously it can not be expressed as a part of the specification. Instead, we use fairness conditions. By declaring a formula as a fairness condition, the specification is checked only on the paths where the formula holds infinitely often.
We first consider the formula as the following, where θ ground C and A:

$$(LBS \vdash C\theta) \wedge (A\theta \in LActs) \wedge (BS = modify'(BS, (Ms, \{\theta\})))$$

However this is not what we want. Consider that the specification, "the agent at some point achieves its top goal". Imagine we have a path which consists of one state transiting to itself, and at this state the top goal of the agent holds. Naturally we do not want to exclude such paths, but if the above formula does not hold, which is probably the case as C is most likely representing an intermediate BeliefStore state getting towards the top goal, it does get excluded from checking.

Then we considered the following formula, where θ ground C and A:

$$(LBS \vdash C\theta) \wedge (A\theta \in LActs) \rightarrow (BS = modify'(BS, (Ms, \{\theta\})))$$

This seems more appropriate, as this would only exclude paths where C holds and the agent continues to take A, without seeing the desired change in BS.
Then the set of fairness condition which we wish to produce from an environment configuration EC is the following:

$$\begin{aligned}&\{(LBS \vdash C\theta \wedge A\theta \in LActs) \rightarrow (BS = modify'(BS, (Ms, \{\theta\}))\\&\quad \mid (eventually, C, A, Ms) \in EC, \theta \in bindings((C, A))\}\end{aligned}$$

4.3 Summary

In this chapter, we introduced environment configuration, which defines the possible sequence of BeliefStore updates.
We came up with a rough idea of what kind of rules would be useful in environment configurations. Then we formalised these rules, and defined *next* and *init* function to fill the missing piece in the previous chapter. As the rules to express the eventual effect of an action can not be expressed in terms of the structure of the interpreted system, we defined fairness conditions instead so only the paths where the agent eventually succeeds in the action are checked against the specification.
We also discussed why taking an environment configuration from the user and specifying the interpreted system accordingly was a necessity, and how alternative methods, such as extracting the information from TeleoR or specifying the assumption on the BeliefStore updates in the specification formulae, are not sufficient.

Chapter 5

Implementation

In this section, we see the overview of the implementation and the algorithm used to implement the compiler.

5.1 Java

I chose Java to implement the compiler. It was expected that the code-base of the compiler would be fairly large. Therefore it was as safer to choose a statically typed language, so that simple mistakes can be caught at compile time.

C++, Scala and Haskell were considered. Although in most cases Java is slower than C++, Java still has a fairly good absolute performance and provides automated garbage collection, which meant that the memory management is not a concern.

Java has been popular for a long time, and as a result a large array of tools and libraries are available. Using a great IDE, IntelliJ[23], and not having to implement your own parser or a library to manipulate propositional logic expression sped up the development massively. This factor led me to ultimately forgo Haskell.

In Scala it is possible to use libraries in Java, and an IDE is available. However in the end I decided against Scala, as I have no experience in it and the learning curve for Scala is notoriously steep. It is a multi-paradigm language combining object-oriented-programming, functional programming and language-orientated-programming[24], and consequently there are many ways of achieving the same thing, though only a few among them are best practices. For example, there are eight ways of creating and populating a list in Scala[25].

5.2 ANTLR

A parser generator ANTLR[26] is used to implement the front-end of the compiler.

Initially, I had hoped to reuse the front-end of the compiler of Qulog. There were two reasons that I concluded that it was better to generate a new parser on my own. First, it was written in QuProlog, which I am not familiar with. Second, Qulog compiler compiles Qulog into QuProlog[27] which is another logic language and its intermediate states are not appropriate for compiling Qulog to MCMAS, which is a very different language from Qulog.

There are various parser generators whose output language is Java, including Beaver[28], JavaCC[29]. I chose ANTLR as it had by far the best documentation including a book[30] available and had IDE support.

Thankfully EBNF grammar for Qulog was availlable[31] and after adapting the syntax to the ANTLR grammar, the only substantial modification necessary was to resolve the mutual left recursion among some rules.

The following simplified example demonstrates the problem I encountered. ANTLR grammar does not allow a grammar below, because ANTLR generates an LL parser and `foo` and `bar` refers to each other at the start of the rule and so does `foo` and `par`.

```
foo : bar | par;
bar : foo '+' foo | INT;
par : foo '-' foo | INT;
INT : (-)?[1-9][0-9]+;
```

This can be partially resolved by in-lining `bar` and `par` in `foo`.

```
foo : foo '+' foo |  foo '-' foo | INT;
INT : (-)?[1-9][0-9]+;
```

ANTLR still will not accept this grammar, it is not clear whether

```
1 + 2 - 3
```

should be parserd as

```
(1 + 2) - 3
```

or

```
1 + (2 - 3)
```

In other words, the associativity is ambiguous for the operators + and -.
In ANTLR it is possible to specify the associativity as following:

```
foo :
    <assoc=left> foo '+' foo |
    <assoc=left> foo '-' foo |
    INT;
INT : (-)?[1-9][0-9]+;
```

The above grammar is accepted by ANTLR and exactly corresponds to the first grammar.
The intermediate representation that ANTLR parser provides is Context, which is essentially a multi-way tree that contains the information of a code fragment. It also provides a base Visitor class with a visit method for each class of Context which one can extend to implement their own Visitor.

5.3 Other Input Files

5.3.1 Starting Task Information

In TeleoR programs, how the agent is started is specified in a Qulog action. As we assume that there is no action in Single Bound Program, we decided to accept another file, which lists the argument to the starting task.
For instance, if the starting task should run with the argument "box", then this file should be such that it contains just a word "box".

5.3.2 Environment Configuration

We formally discussed environment configuration in the previous section. The syntax of the environment configuration in ANTLR grammar is the following:

```
config ::= rule*
rule ::= 'DONTFLIP' atom (',' atom)*';' |
         'INIT' atom  ':' trRuleLHS (',' trRuleLHS)* ';'|
          condition 'DEFINITELY' bsModifier, (';' bsModifier)+';' |
          condition 'EVENTUALLY' bsModifier, (';' bsModifier)+';' |
          condition 'MAYCHANGE' lhsRelationalTerm';' |

condition ::= ('TRUE'| trRuleLHS) '+' actionWithAnonVar
bsModifier ::= 'remember' lhsRelationalTerm+ | 'forget' lhsRelationalTerm+
```

where

- `trRuleLHS` is a BeliefStore clause
- `actionWithAnonVar` has is a primitive action, whose arguments may be anonymous variables
- `lhsRelationalTerm` is a relational term without `not`

For instance, for program with the declaration in Figure 5.1, Figure 5.2 is a syntactically valid environment configuration.

```
dir::= left | centre | right
thing::= box | ball
percept
    holding : (thing),
    next_to:(thing,dir),
    see:(thing,dir)
durative
    move:(),
    turn:(dir)
```

Figure 5.1: Example declaration.

```
DONTFLIP holding, see;
INIT holding : not holding(_);
next_to(X, centre) + grab(X) DEFINITELY remember holding(X);
TRUE + turn(X) EVENTUALLY remember see(box, centre);
see(X, centre) + move(_) MAYCHANGE next_to(X, centre);
```

Figure 5.2: Example environment configuration.

The formal representation of this environment configuration would be:

$$\begin{aligned}\{\ &(dontFlip, holding), (dontFlip, see),\\ &(init, \boldsymbol{not}\ holding(\_)),\\ &(definitely, next\_to(X, centre), grab(X), ((remember, holding(X)))),\\ &(eventually, \boldsymbol{true}, turn(X), ((remember, see(box, centre)))),\\ &(maychange, see(X, centre), move(\_), next\_to(X, centre))\ \}\end{aligned}$$

The visitor classes for parsing the Single Bound Program could be reused to parse the environment configuration file, which saved a lot of effort.

5.4 Collecting the Type Information

To compile an ISPL, we often need to compute possible instantiations of terms in the TeleoR program. We do this based on the type of the term. In this section, we see the algorithms used to extract type informations from the TeleoR program.
The type informations can be found in two parts of the programs; one is the type definition, and the other is the type declaration.
In the type definition, the developer may define its own types. The types may be defined in terms of other types. We will see how we resolve this aliasing.
The type declaration specifies the type of the terms. We also see how we check that all terms are bound, as they should be to be in a Single Bound Program.

5.4.1 Type Definition

In TeleoR, it is possible to define a new type as previously mentioned in Section 2.1.2. Percepts and beliefs are to be encoded to IS local variables for the IS environment agent and TeleoR actions are encoded to IS actions by practically grounding them, which we see later. Therefore it is important to understand which value is possible, and it is in the type where such information lies.

Types defined through an enumeration of atoms or a range of integers are straightforward. However, type aliasing introduces some complications. Type aliasing refers to defining a type in terms of other types. The following is an example of type aliasing.

```
ab ::= a | b
cd ::= c | d
abcd ::= ab || cd
other_name_int ::= int
int_num ::= num || other_name_int
```

Figure 5.3: Example type declaration.

After recording all the type definitions, the compiler checks for circular dependencies as the Qulog compiler does not do so. Circular dependencies are detected by constructing a dependency graph of the types, and finding a loop in it. Each vertex represents a type, and a vertex V has an edge to another vertex V' if V' aliases V. In the above example, a graph would look as follows.

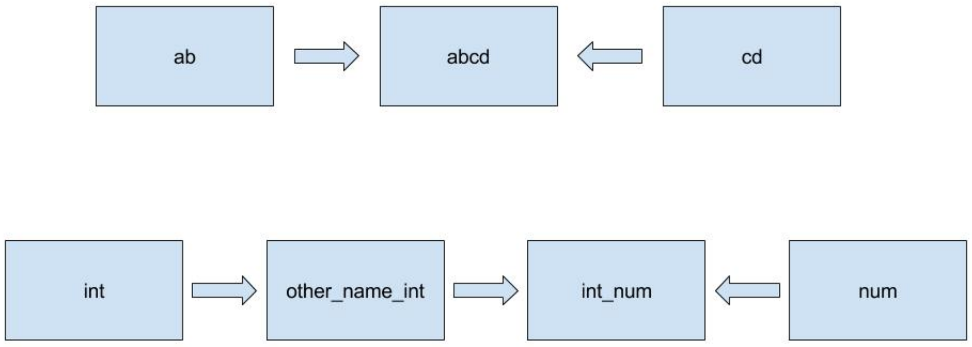

Figure 5.4: Dependency graph of types in Figure 5.3.

If circular dependency is detected, the compilation is aborted.

Later on, we need to compute a possible grounding for relations and actions and in order to do so, we use their type information. As each type is most likely used multiple times, rather than traversing the dependency graph each time, it was better to resolve the aliasing all at once. Resolution of the aliasing is resolved in a topological order, worked out from the dependency graph. That is, a type is resolved strictly after those which it depends on. The complexity of ordering the types in a topological order is O(V+E) where V is the number of vertices and E is the number of the edges.

Resolution of each type is done as follows.

- If a type does not alias anything, no resolution is needed. (as in `ab`, `cd`)
- If a type aliases one type, resolve it as that type. (as in `other_name_int`)
- If a type aliases types that enumerate values, then create a new enumerating type referring to all the values that these aliased types refer to. (as in `abcd`)

- If a type aliases types which alias types that are not an enumeration, flatten the aliasing by creating a new aliasing type referring to all the values that these aliased types alias. (as in `int_num`)

Note that as it is done in a topological order, all the aliased types have their own aliasing resolved already. As a result of this resolution, the above example would resolve to the following types.

- `other_name_int` : An `int` type
- `int_num` : An aliasing type of `int` and `num`
- `abcd` : An enumeration type of `a`, `b`, `c` and `d`
- `ab`: An enumeration type of `a` and `b`
- `cd` : An enumeration type of `c` and `d`

As can be seen, after the resolution, there is no type which aliases a user-defined type.

5.4.2 Type Declaration

In type declaration, the type of each term is specified. As a part of the assumption on a Single Bound Program, all the terms need to have limited possible ground terms that can be instantiated from them. Specifically, a relation, a TeleoR procedure, and an action are bound if all of their arguments are enumeration or integer range type.

5.5 TeleoR Procedures as Protocol Rules

The definition of the protocol function of the agent in IS_P is such that, given the local state with the current BeliefStore state BS, it returns $action(BS)$ if $action(BS)$ is a set of primitive actions, and returns an empty set otherwise. We want to find a way to produce a set of ISPL protocol rules which is equivalent to this protocol function.

The most straight-forward way to do so is to produce protocol rules where each condition only holds for a single BeliefStore state BS, and the action is $\{action(BS)\}$ if $action(BS)$ is a set of primitive actions, and otherwise $\{\}$. However this way we produce as many protocol rules as possible BeliefStore states, therefore it is not efficient.

Recall that there is only one possible sequence of fired rules for each BeliefStore state. Recall also that if given a sequence of fired rules, we know which action is taken by the agent. Then we can define one protocol rule per possible sequence of fired rules. This way we produce as many protocol rules as possible sequences of fired rules, which is fewer than the possible BeliefStore states.

Let us be more concrete. In order for the agent with the BeliefStore state BS to fire a sequence of fired rules, $(\boldsymbol{P_1}, R_1, \theta_1, BS)...(\boldsymbol{P_n}, R_n, \theta_n, BS)$, BS satisfies:

$$\forall i\colon 1 \leq i \leq n \quad fire(\boldsymbol{P_i}, R_i, \theta_i, BS) \tag{5.1}$$

We want to define a function $toMCMASProtocol$ which returns a set of protocol rules where, for all possible sequences of fired rules, there is a rule whose condition is satisfied by the local state of the agent $((BS', LBS, LActs), ())$ if and only if (5.1) holds with $BS = BS'$, and action is as can be

worked out from the sequence. Then the protocol function consists of the return value of the function, excluding the elements whose action is not a set of primitive actions.
First let us define notations we use to refer to BeliefStore clauses and protocol rules. Then we consider how we may translate $fire$ into a condition of protocol rules, and finally we can define $toMCMASProtocol$ using this translation.

5.5.1 Notation

In this subsection, we introduce notations for BeliefStore clauses, MCMAS condition and ISPL protocol rules which we use to present the algorithm to translate BeliefStore clauses into MCMAS condition, and TeleoR procedures into protocol rules.

BeliefStore Clauses

We use the following notation for the BeliefStore clauses which may appear in the Single Bound Program, based on the TeleoR grammar[8] and the restriction of Single Bound Program. Note that the terms in bold are keywords.

Definition 5.1 (BeliefStore clauses)**.**

$$\begin{aligned}
&clause ::= literal \mid literal \ \& \ clause\\
&literal ::= dynamic(args) \mid \boldsymbol{true_{SB}} \mid \boldsymbol{not}\ literal \mid comparison\\
&args ::= arg(, arg)*\\
&arg ::= val \mid anonVar \mid normalVar
\end{aligned}$$

where

- $dynamic \in facts$ *is a identifier of a relation declared as percept or belief*
- *comparison is a comparison term, such as* $X < 2$ *or* $X = left$*, which when grounded resolve to either* $\boldsymbol{true_{SB}}$ *or* $\boldsymbol{not}\ \boldsymbol{true_{SB}}$*.*
- *val is a concrete value that is not variable*
- $anonVar$ *is an anonymous variable*
- $normalVar$ *is a non-anonymous variable*

MCMAS Conditions

We use the following notation for the MCMAS conditions which may appear in ISPL, based on the grammar of ISPL[18]. Note that the terms in bold are keywords.

Definition 5.2 (MCMAS condition)**.**

$$\begin{aligned}
mcmasCondition \ ::= \ & mcmasVar = value \mid\\
& (mcmasCondition) \mid\\
& \mathbf{!}(mcmasCondition) \mid\\
& mcmasCondition \ \boldsymbol{and} \ mcmasCondition \mid\\
& mcmasCondition \ \boldsymbol{or} \ mcmasCondition \mid\\
& \boldsymbol{true}
\end{aligned}$$

where $mcmasVar$ refers to a variable in the local state, and value is a possible value of a variable, such as ***true***.

ISPL Protocol Rule

We use the following notation for protocol rules, which defines the local protocol function in ISPL.

Definition 5.3 (ISPL protocol rule)**.**

$$(mcmasCondition\colon action)$$

where *action* is a local action of an agent, and *mcmasCondition* is a MCMAS condition term.

5.5.2 Condition to Fire a TeleoR Rule as ISPL

Recall the definition of $fire$:

$$fire(\boldsymbol{P}, R, \theta, BS) \equiv \exists\theta' BS \vdash \boldsymbol{P}_R\theta' \wedge \theta \subseteq \theta' \wedge no\_higher\_fireable\_rule(\boldsymbol{P}, R, BS)$$

Let us consider how we may express $\exists\theta' BS \vdash C\theta'$. We can do this by

1. Producing all bindings θs which ground named variables C
2. For all $\theta \in \theta s$, compute an equivalent MCMAS condition for $BS \vdash C\theta$
3. Express that at least one of the conditions produced in (2) holds.

We achieve (1) with a function *getAllBindings*. We can not simply express $BS \vdash C\theta$ as $toVar(C\theta) = true$, where $toVar$ takes a dynamic fact and returns a IS local variable which is true if the BeliefStore state the local state represents contains the input fact. This is because the term $C\theta$ may contain an anonymous variable, therefore may possibly unify with multiple facts in the BeliefStore. For instance with declaration:

```
ab ::= a | b
percept foo : (ab)
```

Then $BS \vdash foo(\_)$ holds if either $foo(a)$ or $foo(b)$ is present in BS. Therefore to implement (2), we define a function *groundFully*, which takes a literal L and a binding θ, and compute all possible facts which may unify with $L\theta$. Then we can express $BS \vdash L\theta$ as “at least one variable representing a fact in $groundFully(L, \theta)$ is true” Finally we present $toMCMASCond$ which use *groundFully* and *getAllBindings* to translate $\exists\theta' BS \vdash C\theta'$.
With means to express $\exists\theta BS \vdash C\theta$, we can also express $no\_higher\_fireable\_rule$, which is defined in terms of $\nexists\theta BS \vdash C\theta$; we simply negate the expression for $\exists\theta BS \vdash C\theta$.

Auxiliary Functions

Definition 5.4 (*relation*)**.** *relation is a function which takes a BeliefStore literal and returns true if it is in the form of dynamic*(*args*)*.*

Definition 5.5 (id, $args$)**.** *id and $args$ is a function which takes a literal L where $relation(L)$ is true, and returns f, As respectively where $L = f(As)$.*

Definition 5.6 ($normalVar$, val)**.** *$normalVar$ and val is a function which takes an argument for a relational BeliefStore literal, and returns true if it is a normal variable, and a concrete value, respectively.*

Definition 5.7 ($possibleValues$)**.** *$possibleValues$ is a function which takes an identifier $f \in facts \cup actions$ i, and returns all the possible values for the ith argument of f according to its type. For instance, with a declaration shown in Figure 5.1, $possibleValues(holding, 1)$ would return $\{box, ball\}$. Note that the returned set is always finite as Single Bound Program would only have action or relation which are "bound".*

Definition 5.8 ($toVar$)**.** *$toVar$ is a bijective function which takes a ground relational term or action, and returns a MCMAS variable.*

Definition 5.9 ($toVal$)**.** *$toVal$ is a function which takes a set of primitive actions, and returns the value in ISPL which represents it.*

Definition 5.10 ($\bigwedge$ for *mcmasCondition*)**.** *Let condition $\in$ mcmasCondition has a free variable i..j, var_conds be a set of conditions that describes i...j and S be a set of all instantiated condition with variables that satisfy var_conds.*
Then:

$$\bigwedge_{var\_conds} condition$$

*is **true** if S is empty, and otherwise it is a mcmasCondition that is constructed from connecting all mcmasCondition in S with **and**.*

Definition 5.11 ($\bigvee$ for *mcmasCondition*)**.** *Let condition $\in$ mcmasCondition has a free variable i..j, var_conds be a set of conditions that describes i...j and S be a set of all instantiated condition with variables that satisfy var_conds.*
Then:

$$\bigvee_{var\_conds} condition$$

*is **false** if S is empty, and otherwise it is a mcmasCondition that is constructed from connecting all mcmasCondition in S with **or**.*

Definition 5.12 ($groundNonRelatioal$)**.** *$groundNonRelational$ is a function that takes a non relational BeliefStore literal, such as $\boldsymbol{true_{SB}}$ or comparison term $X > 1$ and a binding, and returns either $\boldsymbol{true_{SB}}$ or **not** $\boldsymbol{true_{SB}}$ appropriately.*

getAllBindings

Definition 5.13 (*getAllBindings*)**.** *getAllBindings takes a BeliefStore clause and returns all the bindings that grounds the non-anonymous variables.*

$$getAllBindings(L \ \& \ C, \theta) = \bigcup_{\theta' \in \theta s} getAllBindings(C, \theta'), \ \ where\ \theta s = getAllBindings(L, \theta)$$

$$getAllBindings(L, \theta) = \begin{cases} \{\} & if\ \neg relation(L) \\ & \quad \wedge\ groundNonRelational(L, \theta) \\ & \qquad = \boldsymbol{not\ true_{SB}} \\ \{\theta\} & if\ \neg relation(L) \\ & \quad \wedge\ groundNonRelational(L, \theta) \\ & \qquad = \boldsymbol{true_{SB}} \\ getAllBindings'(id(C), arg(C), 1, \theta) & otherwise \end{cases}$$

$$getAllBindings'(f, args, i, \theta) = \begin{cases} \{\theta\} & if\ i > length(args) \\ getAllBindings(f, args, i+1, \theta) & if\ \neg normalVar(args_i) \\ & \quad \vee\ \exists v \{args_i \rightarrow v\} \subseteq \theta \\ \bigcup_{\theta' \in \theta s} getAllBindings'(f, args, i+1, \theta') & \\ \quad where & \\ \theta s = \{\theta \cup \{args_i \rightarrow v\} & \\ \qquad\qquad \mid v \in possibleValues(f, i)\} & if\ normalVar(args_i) \\ & \quad \wedge\ \nexists v \{args_i \rightarrow v\} \subseteq \theta \\ getAllBindings(f, args, i+1, \theta) & otherwise \end{cases}$$

groundFully

Definition 5.14 (*groundFully*)**.** *groundFully is a function that takes a BeliefStore literal L and a binding θ and returns a set of ground BeliefStore literals that may unify with $C\theta$.*

$$groundFully(L,\theta) = \begin{cases} \{L' \mid id(L) = id(L'), \\ \qquad arg(L') = groundArgs(id(L), arg(L), \theta, 1)\} & \textit{if } relation(L) \\ \\ \{groundNonRelational(L,\theta)\} & \textit{otherwise} \end{cases}$$

$$groundArgs(f, args, \theta, i) = \begin{cases} () & \textit{if } i > length(args) \\ \\ \{s \mid s_1 = groundArg(args_i, \theta) \\ \qquad s_{2:} \in groundArgs(f, args, \theta, i+1)\} & \textit{if } val(groundArg(args_i, \theta)) \\ \\ \{s \mid v \in possibleValues(f, i), \\ \qquad s_1 = v, \\ \qquad \theta' = \theta \cup \{args_i \rightarrow v\}, \\ \qquad s_{2:} \in groundArgs(f, args, \theta', i+1)\} & \textit{if } normalVar(groundArg(args_i, \theta)) \\ \\ \{s \mid s_1 \in possibleValues(f, i), \\ \qquad s_{2:} \in groundArgs(f, args, \theta, i+1)\} & \textit{otherwise} \end{cases}$$

$$groundArg(arg, \theta) = \begin{cases} v \textit{ where} \{arg \rightarrow v\} \subseteq \theta & \textit{if } normalVar(arg) \wedge \exists v \{arg \rightarrow v\} \subseteq \theta \\ arg & \textit{otherwise} \end{cases}$$

toMCMASCond

Definition 5.15 (*toMCMASCond*)**.** *toMCMASCond is a function that takes a BeliefStore clause, and returns an equivalent MCMAS condition term.*
It is defined as follows:

$$toMCMASCond(L \;\&\; C, \theta) = toMCMASCond(L, \theta) \textbf{ and } toMCMASCond(C, \theta)$$

$$toMCMASCond(\textbf{not } L, \theta) = \textbf{!(} \bigvee_{L' \in GL} toVar(L') = \textbf{ true)}$$
$$\textit{where } GL = groundFully(L, \theta)$$

$$toMCMASCond(L, \theta) = \textbf{(} \bigvee_{L' \in GL} toVar(L') = \textbf{ true)}, \textit{ where } GL = groundFully(L, \theta)$$

All in all, $fire(\boldsymbol{P}, R, \theta, BS)$ is equivalent to an MCMAS condition of the following:

$$[\bigvee_{\theta \in AB} toMCMASCond(\boldsymbol{P_R}, \theta)] \textbf{ and } [\bigwedge_{1 \le i < R} \bigwedge_{\theta' \in AB'} \textbf{!}toMCMASCond(\boldsymbol{P_i}, \theta')]$$

where $AB = getAllBindings(P_R, \{\})$ and $AB' = getAllBindings(P_i, \{\})$.

5.5.3 Protocol Generation Algorithm

Now we know how to generate an MCMAS condition equivalent to $fire$. The action of a TeleoR agent is determined by the sequence of TeleoR rules are fired. In other words, its BeliefStore state at the time is such that it satisfies the condition of firing this particular sequence of TeleoR rules, which is, the conjugate of $fire$ for all the rules in the sequence. To recreate this, *toMCMASProtocol* generates a set of protocol rules such that it includes a protocol rule per a possible sequence of fired rules, such that its condition is equivalent to the conjugate of $fire$ for the rules in the sequence.
Recall that we also have two erroneous states, where the agent of IS_P do not take any action. With *toMCMASProtocol*, we compute protocol rules for the sequence of fired rules for those states as well, and then later remove them from the protocol rules. We use the removed rules to define the valuation function, as we need to have propositional atoms which hold at the erroneous states.

Definition 5.16 (*toMCMASProtocol*)**.** *The following Python-like pseudo code defines toMCMASProtocol, which returns a set of protocol rules, where curateForRHS takes an action and a set of bindings, and return a binding which binds the variables in the action in the same way as all the input bindings.*

```
toMCMASProtocol($\boldsymbol{P}$, firedRuleCount)
    if firedRuleCount > MaxDp:
        return {($\mathbf{true}$: $md\_fail$)}

    toReturn = {}
    prevRulesDontApply = $\mathbf{true}$;
    R = 1

    while R <= getRuleCount(P):
        $\boldsymbol{K}$ ~> $\boldsymbol{A}$ = $\boldsymbol{P}_R$
        bindings = getAllBindings($\boldsymbol{K}$, {})
        protocol = translateRHS($\boldsymbol{A}$, curateForRHS($\boldsymbol{A}$, bindings), firedRuleCount)
        guard = $\mathbf{true}$
        for $\theta$ in bindings:
            guard = guard $\mathbf{or}$ toMCMASCond($\boldsymbol{K}$, $\theta$)
        for rule in protocol
            add (guard $\mathbf{and}$ prevRulesDontApply $\mathbf{and}$ rule.cond : rule.action) to toReturn
        for $\theta$ in bindings:
            prevRulesDontApply = prevRulesDontApply $\mathbf{and}$ !(toMCMASCond($\boldsymbol{K}$,$\theta$))
        R += 1

    add (prevRuleDontApply : $nfr\_fail$) to toReturn
    return toReturn


translateRHS($\boldsymbol{A}$, $\theta$, firedRuleCount) :
    if $primActs$($\boldsymbol{A}\theta$):
        return {(true : toVal($\boldsymbol{A}\theta$))}
    if $procCall$($\boldsymbol{A}\theta$):
        return toMCMASProtocol($\boldsymbol{A}\theta$, firedRuleCount+1)
```

Note that there is always a binding returned by *curateForRHS* as we have no ambiguous rule.
Note also that to produce a set of protocol rules for the agent whose starting task is $\boldsymbol{SP}$, one needs to call $toMCMASProtocol(\boldsymbol{SP}, 1)$.

5.6 Translating Environment Configuration into ISPL

In this section, we see the algorithm to translate environment configuration rules into ISPL. We previously saw that environment configuration rules are made up of the following four rules:

1. $(definitely, C, A, Ms)$
2. $(maychange, C, A, F)$
3. $(dontFlip, f)$
4. $(init, C)$
5. $(eventually, C, A, Ms)$

The evolution function of the environment agent in IS_P is defined in terms of *next*, which is specified by the first three rules. We see how *next* can be expressed as ISPL. Then we see how (4) can be translated into the initial state condition in ISPL. Finally, we see how (5) can be translated into the fairness condition in ISPL.

5.6.1 Evolution Rules

Assignment Semantics

As mentioned in Section 2.2.7 in the background, there are two ways to define the evolution in ISPL; one is single assignment and the other is multi-assignment. We chose single assignment semantics, as it is more natural and concise to express the change of each local variable as with an evolution item. For instance, consider a set of n boolean variables which we want them to be changed non-deterministically. In the multi-assignment semantics, we need to have 2^n rules, each specifying possible updates to all n variables. For instance, if we have three boolean variables, we would have 8 rules:

```
v1 = true and v2 = true and v3 = true if Environment.Action = update;
v1 = false and v2 = true and v3 = true if Environment.Action = update;
v1 = true and v2 = false and v3 = true if Environment.Action = update;
...
```

With single assignment semantics, we only need to have $2n$ rules, where there are two rules per variable, one setting the variable to true and the other to false. For instance, if we have three boolean variables, we would have 6 rules:

```
v1 = true if Environment.Action = update;
v1 = false if Environment.Action = update;
v2 = true if Environment.Action = update;
...
```

Note that in ISPL, if there are multiple rules on the same variable whose condition evaluates to true at a given state, one rule is arbitrarily chosen.
Let us say we have one variable v which changes its value to true under a certain condition C among this n variables, and the others still change their value non-deterministically. With multi-assignment semantics, the rule will be extremely hard to read, as there will be 2^{n-1} rules with C as the condition, all assigning true to v, and each of them updating $n-1$ variables differently.

Generating Evolution Rules

With the assignment semantics fixed, we can show the algorithm to generate the evolution rules from the environment configuration.
Let us use the following notation to represent an evolution rule:

Definition 5.17 (Evolution Rules)**.**

$$(mcmasCondition, mcmasVar, value)$$

where mcmasCondition is defined in Definition 5.2, mcmasVar is a local variable in ISPL, and value is a value that is assigned to mcmasVar if mcmasCondition holds.

We saw that the environment configuration rules specify the BeliefStore update using a condition on the BeliefStore state, and a condition on the TeleoR action. Therefore we want to define a function similar to *toMCMASCond*, but for the TeleoR actions rather than BeliefStore clauses.
Before doing so, consider that we might have a rule when the chosen agent take multiple actions. These actions are executed in parallel[6]. We assume that these actions change different sets of facts, as they must be using separate robotic resources for the actions to be executed concurrently. This means that an environment configuration rule which specifies the BeliefStore update when an action is taken should come into effect whenever a TeleoR action is a set of primitive actions which includes it.
To define *toMCMASCond* for actions, we assume the following function:

Definition 5.18 (*getMultipleActions*)**.** *getMultipleActions is a function which takes a fully ground action, and returns the set of values which represent the set of TeleoR actions which includes the given action. For instance, given* $move(1)$*, if there is a protocol rules which lead to* $\{move(1), turn(1)\}$ *and a rule with* $\{move(1)\}$*, then it returns* $\{toVal(\{move(1), turn(1)\}), toVal(\{move(1)\})\}$*.*

In TRTIS, this is implemented by using a map from a single action to the set of action sets which is computed while computing the protocol rules. Now define *toEvolution*, which translate the environment configuration rules to evolution rules.
Now we can define *toMCMASCond* for the TeleoR action. Let us extend *getAllBindings* in Definition 5.13 so that an action is treated the same as relational BeliefStore literals, and do similarly for *groundFully* defined in Definition 5.14. Then we define *toMCMASCond* for the action of the agent, and of the environment.

Definition 5.19 ($toMCMASCond_A$, $toMCMASCond_E$)**.**

$$toMCMASCond_A(A, \theta) = \Big(\bigvee_{A' \in GA} \bigvee_{A'' \in MA(A')} \boldsymbol{Agent\_0.Action} \; = \; A''\Big),$$
$$\text{where } GA = groundFully(A, \theta), MA(A') = getMultipleActions(A')$$
$$toMCMASCond_E(update) = \boldsymbol{(Environment.Action} \; = \; update\boldsymbol{)}$$

where A is a single action.

Regarding how we can translate the modifiers into assignments in the evolution rule of ISPL, consider the following rule:

$$(definitely, \boldsymbol{true_{SB}}, turn(1), ((forget, see(\_)), (remember, see(box))))$$

where *see* can take either *box* or *ball*. We want this syntax to have semantics "when ever the agent $turn(1)$, then it forgets everything that it sees, then remember $see(box)$". In other words, the rule is equivalent to:

$$(definitely, \boldsymbol{true_{SB}}, turn(1), ((forget, see(ball)), (forget, see(box))(remember, see(box))))$$

We can not express this straight-forwardly in ISPL as $see(box)$ appears multiple times, therefore it would be ambiguous whether in the next state $see(box)$ would hold or not.
Instead, we need compute an equivalent set of ground modifiers to this list of ground modifiers, so no fact appears multiple times. For instance, the above rule is equivalent of having:

$$(definitely, \boldsymbol{true_{SB}}, turn(1), ((forget, see(ball)), (remember, see(box))))$$

Let us define a function to do so:

Definition 5.20 (*toSet*)**.** *toSet is a function which takes a binding and a list of modifiers, and returns a set of modifiers which is equivalent to the given list fully grounded by the given binding. toSet does so by first applying groundFully to the elements in the list to create a list of set of facts, then iterate it backwards to take the latest occurrence of each fact.*

Now we can define *toEvolution* as a function which returns the union of evolution rules computed from *definitely*, *maychange* and *dontflip* rules.

Definition 5.21 (*toEvolution*)**.** *toEvolution is a function which takes the environment configuration rules, and returns the set of evolution rules.*

$$\begin{aligned} toEvolution(EC) = & \{e \mid (definitely, C, A, MS) \in EC, e \in toEvolution'((definitely, C, A, MS))\} \\ & \cup \{e \mid (maychange, C, A, F) \in EC, e \in toEvolution'((maychange, C, A, F))\} \\ & \cup \{(mc, var, val) \\ & \qquad \mid f \in facts, \nexists(dontFlip, f) \in EC, G \in getAllGroundings(f), var = toVar(G), \\ & \qquad\; mc = toMCMASCond_E(update), val \in \{\boldsymbol{true}, \boldsymbol{false}\}\} \end{aligned}$$

where

$$\begin{aligned} & toEvolution'((definitely, C, A, MS)) = \\ & \qquad \{(mc, var, val) \mid \theta \in getAllBindings'(C, A), M \in toSet(MS, \theta), \\ & \qquad\qquad mc = toMCMASCond(C, \theta) \ \boldsymbol{and} \ toMCMASCond_A(A, \theta), \\ & \qquad\qquad (var, val) = toAssignment(M)\} \\ & toEvolution'((maychange, C, A, F)) = \\ & \qquad \{(mc, var, val) \mid \theta \in getAllBindings'(C, A), \\ & \qquad\qquad mc = toMCMASCond(C, \theta) \ \boldsymbol{and} \ toMCMASCond_A(A, \theta), \\ & \qquad\qquad G \in groundFully(F, \theta), var = toVar(G), val \in \{true, false\}\} \\ & getAllBindings'(C, A) = \\ & \qquad \{\theta \mid \theta' \in getAllBindings(C, \{\}), \theta \in getAllGrounding(A, \theta')\} \end{aligned}$$

$$\begin{aligned} & toAssignment((remember, M)) = (toVar(M), \boldsymbol{true}) \\ & toAssignment((forget, M)) = (toVar(M), \boldsymbol{false}) \end{aligned}$$

5.6.2 Initial States Condition

The following function shows how the initial states condition is computed from the environment configuration.

Definition 5.22 (*toInitStatesCondition*)**.** *toInitStatesCondition is a function which takes an environment configuration, and returns a MCMAS condition which specifies the initial states.*

$$toInitStatesCondition(EC) = \bigwedge_{G \in GC} toMCMASCond(G)$$
$$\text{where } GC = \{G \mid (init, C) \in EC, \theta \in getAllBindings(C, \{\}), G \in groundFully(C, \theta)\}$$

5.6.3 Fairness Conditions

The following is the function to produce the fairness condition from the environment configuration.

$$toPrevCond(L\&C, \theta) = \textbf{(} \bigvee_{L' \in GL(L,\theta)} groundPreviousToAtom(L')\textbf{)} \ \textbf{\textit{and}} \ toPrevCond(C, \theta)$$

$$toPrevCond(L, \theta) = \textbf{(} \bigvee_{L' \in GC(L,\theta)} groundPreviousToAtom(L')\textbf{)}$$

$$toFairness(EC) =$$
$$\{\textbf{!(}toPrevCond(C, \theta) \ \textbf{\textit{and}} \bigvee_{A' \in GA(A,\theta)} groundPreviousToAtom(A')\textbf{)} \ \textbf{\textit{or}} \ \textbf{(} \bigwedge_{M \in toSet(MS,\theta)} toAtom(M)\textbf{)}$$
$$\mid eventually(C, A, MS) \in EC, \theta \in getAllBindings'(C, A)\}$$

where

$$GL(L, \theta) = groundFully(L, \theta)$$
$$GA(A, \theta) = groundFully(A, \theta)$$
$$getAllBindings'(C, A) = \{\theta \mid \theta' \in getAllBindings(C, \{\}), \theta \in getAllGrounding(A, \theta')\}$$
$$toAtom((remember, F)) = groundToAtom(F)$$
$$toAtom((forget, F)) = \textbf{!}groundToAtom(F)$$

Note that we utilise the equivalence $\neg(A \wedge B) \vee C \equiv (A \wedge B) \rightarrow C$.

5.7 TeleoR Program as ISPL

We discussed algorithms to translate TeleoR procedures and environment configuration into components in ISPL, in a rather formal way. This section aims to present a full view on how an ISPL file that represents IS_P should look like. We do this by looking at a toy example, and stating how each component of ISPL is expected to be. This should provide a more concrete view of how TeleoR procedures and environment configurations are translated. In addition, it should give a precise idea on how the rest of IS_P is represented in ISPL, as it is not as complex as to require a formal definition. Note that $toVal$ is assumed to be implemented so that it returns a string constructed from joining the identifier and the arguments with "__", and $toVar$ is similar except that it is prefixed by the name of the agent.

5.7.1 Example TeleoR Program

Let us define a toy TeleoR program which we use as an example in this section, the Spinning Agent, which when facing right turns left, and when facing left turns right.

```
dir ::= left | right
percept
        facing:(dir)
durative
        turn:(dir)

task_start spin : ()
spin() {
        facing(right) ~> turn(left)
        facing(_) ~> turn(right)
}
```

Figure 5.5: Spinning Agent.

We use the following environment configuration as an example. It is rather contrived, but it should illustrate how environment configurations should be translated.

```
INIT facing: facing(right);
facing(left) + turn(right) EVENTUALLY remember facing(right);
facing(right) + turn(_) DEFINITELY remember facing(left);
facing(X) + turn(X) MAYCHANGE facing(X);
DONTFLIP facing;
```

Figure 5.6: Environment Configuration for Spinning Agent.

5.7.2 Action

Environment

The only action that is possible of the environment agent in IS_P is *update*.
Therefore the environment agent in the ISPL should have the following line, regardless of the input TeleoR program or environment configuration:

```
Actions = {update};
```

Agent

The set of actions of the agent in IS_P is a set of possible return values of its protocol function. Concretely, the set of actions of the agent in the ISPL should be set up so that it lists all the used grounding of the actions that appear on the right hand side of the TeleoR rules. For instance, for the Spinning Agent, we should have the following line in the agent:

```
Actions = {turn__left, turn__right};
```

If a TeleoR program includes a rule with empty action `()`, then `Actions` contains `idle__`.

5.7.3 Local States

Environment

Recall that the local states of the environment agent in IS_P is represented as a tuple $(BS, LBS, LActs)$, where BS is the current BeliefStore state, LBS is the previous BeliefStore state, and $LActs$ is the previous action.
With the environment configuration, the local variables of the environment should consists of the following:

- a boolean variable to represent whether a dynamic fact is present in the current BeliefStore state.
- a boolean variable to represent whether a dynamic fact was present in the previous BeliefStore state.
- a variable which maintain the action that was just taken.
- a dummy boolean variable which we use to express true and false value, as `(dummy = true) or (dummy = false)` and `(dummy = true) and (dummy = false)` respectively, as ISPL does not have a built-in true false values.

For instance, given the Spinning Agent program shown in Figure 5.5, the local state of the environment agent should be:

```
Vars:
agent__0__facing__left : boolean;
agent__0__facing__right : boolean;
prev__agent__0__facing__left : boolean;
prev__agent__0__facing__right : boolean;
agent__0__taken__action : {none, turn__right, turn__left };
dummy : boolean
end Vars
```

`agent__0__taken__action`, in addition to a possible value for `Agent__0.Action`, contains `none`, a value to which `agent__0__taken__action` is set at the initial states.

Without the environment configuration, we do not need to keep track of the previous BeliefStore state or the action which was just taken. Therefore the local state simplifies to:

```
Vars:
agent__0__facing__left : boolean;
agent__0__facing__right : boolean;
dummy : boolean
```

```
end Vars
```

Agent

Recall that the private local state of the agent is empty in IS_P, and the agent observes the local state of the environment. In MCMAS, local variables in the environment can be "seen" by the agent by declaring them as local observable variables. As the agent does not actually use the previous BeliefStore state or the action that was just taken in the evolution or the protocol, it suffices that it observes only the local variables which are used to represent the current BeliefStore state. Therefore the local state of the agent in the ISPL for the Spinning Agent would be:

```
Lobsvars = {agent__0__facing__left, agent__0__facing__right};
Vars:
dummy : boolean;
end Vars
```

5.7.4 Protocol

5.7.5 Environment

Recall that the local protocol function of the environment always returns "update", its only action. This can be expressed as follows:

```
Other : {update};
```

Agent

The set of protocol rules is computed using $toMCMASProtocol(\boldsymbol{SP}, 1)$, which is defined in Definition 5.16. As mentioned previously, the rules with md_fail or nfr_fail are not added to the declaration of the protocol of the agent.
For example, the concrete implementation of $toMCMASProtocol(spin, 1)$ should produce the following rules:

```
(Environment.agent__0__facing__right) : {turn__left}
!(Environment.agent__0__facing__right = true)
        and (Environment.agent__0__facing__right = true
        or Environment.agent__0__facing__left = true) : {turn__right}
!(Environment.agent__0__facing__right = true)
        and !(Environment.agent__0__facing__right = true
        or Environment.agent__0__facing__left = true) : {nfr_fail}
```

Figure 5.7: Expected output of the concrete implementation of $toMCMASProtocol(spin, 1)$.

Then the protocol function of the agent is defined in ISPL as:

```
Protocol:
(Environment.agent__0__facing__right) : {turn__left}
!(Environment.agent__0__facing__right = true)
        and (Environment.agent__0__facing__right = true
        or Environment.agent__0__facing__left = true) : {turn__right}
end Protocol
```

Note that the condition of the second rule simplifies to `Environment.agent__0__facing__left = true and Environment.agent__0__facing__right = false`.

5.7.6 Evolution

Environment

The evolution of the environment should consist of rules which updates the current BeliefStore states, the previous BeliefStore states and the action which was just taken.
In IS_P, the current BeliefStore state is updated using the *next* function. In the previous section we defined the function *toEvolution* which takes an environment configuration and returns the rules which are all together equivalent to *next*. The following is the set of rules which should be returned by the concrete implementation of *toEvolution* for the Spinning Agent:

```

-- facing(right) + turn(_) DEFINITELY remember facing(left)
agent__0__facing__left = true
if (Agent__0.Action = turn__right or Agent__0.Action = turn__left)
and agent__0__facing__right = true;

-- facing(left) + turn(left) MAYCHANGE facing(left)
agent__0__facing__left = true
if Agent__0.Action = turn__left and agent__0__facing__left = true;
agent__0__facing__left = false
if Agent__0.Action = turn__left and agent__0__facing__left = true;

-- facing(right) + turn(right) MAYCHANGE facing(right)
agent__0__facing__right = true
if Agent__0.Action = turn__right and agent__0__facing__right = true;
agent__0__facing__right = false
if Agent__0.Action = turn__right and agent__0__facing__right = true;
```

The evolution also includes following rules to update the variables representing the previous BeliefStore state.

```
prev__agent__0__facing__left = true if agent__0__facing__left = true;
prev__agent__0__facing__left = false if agent__0__facing__left = false;
prev__agent__0__facing__right = true if agent__0__facing__right = true;
prev__agent__0__facing__right = false
if agent__0__facing__right = false;
```

Finally, the evolution includes the rules to update the variable for the action just taken.

```
agent__0__taken__action = turn__left if Agent__0.Action = turn__left;
agent__0__taken__action = turn__right if Agent__0.Action = turn__right;
```

Note that if we had not had *init* rule for $facing$, the evolution would have also included the rules presented below in Figure 5.8.

With empty environment configuration, the local variables of the environment agent are only those to represent the current BeliefStore state. Reflecting that the user assumes a completely non-deterministic environment, the evolution of the environment in the agent would be:

```
agent__0__facing__left = true if Action = update;
agent__0__facing__left = false if Action = update;
agent__0__facing__right = true if Action = update;
agent__0__facing__right = false if Action = update;
```

Figure 5.8: Expected evolution of Agent 0 in ISPL for the Spinning Agent without the environment configuration.

Agent

The agent does not have anything in its private local state, therefore no evolution rule is necessary. As ISPL does now allow an empty evolution, we should have dummy evolution rules which is the following:

```
Protocol:
        dummy = true if dummy = true;
        dummy = false if dummy = false;
end Protocol
```

Clearly, this does not affect the verification.

5.7.7 Evaluation

In evaluation, one can declare the propositional atoms which can be used in Fairness and the specification. For IS_P recall we defined a valuation function V_{IS_P}. We want to define the atoms which are the domain of V_{IS_P}. For atoms, for facts in the current BeliefStore, the previous BeliefStore and the action which was just taken, it is quite straight-forward. For instance, the atom for $facing(left)$ in current and previous BeliefStore, and the atom which is true if and only if the action just taken was $turn(left)$ should be defined as follows:

```
agent__0__facing__left if Environment.agent__0__facing__left = true;
prev__agent__0__facing__left
if Environment.prev__agent__0__facing__left = true;
agent__0__taken__turn__left
if Environment.agent__0__taken__action = turn__left;
```

The atoms for action that is going to be taken at the state are defined using the protocol rules produced by *toMCMASProtocol*, including those for the erroneous states. The atoms for the action a is defined as the conjugate of the conditions of all the protocol rules in the return value of *toMCMASProtocol* whose action is a. If there is no protocol rule for a, then the atom is defined so it never holds.
Recall we saw the expected output of the concrete implementation of *toMCMASProtocol* in Figure 5.7. The evaluation for the current action should be:

```
agent__0__turn__left if (Environment.agent__0__facing__right = true);
agent__0__turn__right
if !(Environment.agent__0__facing__right = true)
        and (Environment.agent__0__facing__right = true
        or Environment.agent__0__facing__left = true)
agent__0__nfr_fail
if!(Environment.agent__0__facing__right = true)
        and !(Environment.agent__0__facing__right = true
        or Environment.agent__0__facing__left = true)
agent__0__md_fail
if Environment.dummy = false and Environment.dummy = true;
```

Note that this correctly represents that the agent may go to the error state if the agent is not facing either left or right.

If there is no environment configuration, then only the atoms of the current BeliefStore state and the action which is about to taken is defined.

5.7.8 Initial States

The condition which is imposed on initial states specifies that the previous BeliefStore is empty, there was no previous action taken, and that the starting BeliefStore state is in *init* defined from the environment configuration. Not to double up the number of states unnecessarily, it should also fix all the dummy variables to true.
The concrete implementation of *toInitStatesCondition* should output:

```
-- INIT facing : facing(right)
agent__0__facing__right = true
```

Therefore, with the environment configuration, the ISPL for the Spinning agent is expected to have the following:

```
InitStates
Agent__0.dummy = true
and Environment.dummy = true
and Environment.prev__agent__0__facing__left = false
and Environment.prev__agent__0__facing__right = false
and Environment.agent__0__taken__action = none
and Environment.agent__0__facing__right = true;
end InitStates
```

Without the environment configuration, only the conditions on the dummy variable should be present.

5.7.9 Fairness

The fairness condition is generated by *toFairness*. The expected fairness condition for the Spinning Agent is:

```
Fairness
-- facing(left) + turn(right) EVENTUALLY remember facing(right)
!(prev__agent__0__facing__left and agent__0__taken__turn__right )
or agent__0__facing__right;
end Fairness
```

Chapter 6

Evaluation

In this chapter, we evaluate TRTIS by looking at some Single Bound Programs. We also aim to give the reader a more concrete idea on how one may use TRTIS and MCMAS.

We first look at a small TeleoR program, Object Grabbing Agent, which turns around and grabs a specified object. With this example, we want to show two things. One is that TRTIS behaves as described in the previous chapters. We do this by looking at the actual output of TRTIS, with and without the environment configuration. The other is that, for this small example, the environment configuration rules are expressive enough. We do this by describing what would be a reasonable assumption that one may make for such an agent, and express it with the environment configuration rules.

Then we look at two larger TeleoR programs, Bottle Collecting Agent and Tower Building Agent, which are simplified versions of "classic" TeleoR programs. Through these examples, we discuss how the environment configuration rules that we have may be extended to express a more realistic assumption of the environment.

With Tower Building Agent, we see how one can use TRTIS and MCMAS to debug TeleoR programs. The advantage of using a model checker is that the model of the program is tested against every possible execution. With TeleoR program, the debugging is normally done using an environment simulator or Python robot shell, where the update of the BeliefStore is emulated. These techniques in contrast, are incomplete, as it is not possible to check all possible situations.

Finally, we look at the performance of both TRTIS and MCMAS for these three examples, and briefly discuss the user experience of TRTIS.

6.1 Object Grabbing Agent

We start the case studies by looking at Object Grabbing Agent, which turns around and grabs a specified object.
First we see how the TeleoR program is defined for this agent. Then we see the output of TRTIS without the environment as is described in the previous chapters, and run it against some specifications. We then construct environment configuration rules, discussing what assumption could have been made for the environment. Finally we compile an ISPL with the environment configuration rules, and discuss how those rules are reflected as expected. We also show that the specification that could not be verified before may be verified.

6.1.1 TeleoR program

Below is the TeleoR program which defines Object Grabbing Agent.

```
dir::= left | right | centre
obj ::= box | ball

percept
            see : (obj, dir),
            holding : (obj)

durative
            turn : (dir),
            stop : ()

discrete
            grab : (obj),
            release : ()

get_object(X) {
      holding(X) ~> ()
      not holding(X) & see(X, centre) ~> grab(X)
      not holding(_) ~> face(X)
      true ~> release
}

face : (obj) ~>
face(X) {
      see(X, centre) ~> ()
      true ~> turn(left)
}

task_start get_object : (obj)
```

As the starting task is `get_object`, the top-goal of this agent is `holding(X)`, where `X` should be specified in a file with the option `-c`. If it does not hold but sees `X` in the centre, then the agent takes a discrete action `grab`. If it is not the case that the agent is holding anything, then a procedure call `face` is called, where the agent turns to the left until it sees the object it should grab. Otherwise, the agent is not holding or facing the box and it is holding something else. Then the agent executes `release`, which should leave the object.

6.1.2 Compiled ISPL with Non-deterministic Environment

We compile the program by running `trtis input.qlg -c start_task_call.txt -o output.mas -d 100`, where start_task_call.txt contains a line `box` and nothing else.
The below is the resulting ISPL, where some more "obvious" bits are omitted for the clarity. We observe how it corresponds to the original TeleoR program, and then we see which specification may be and may not be verified on this program.

```
Semantics = SA;
Agent Environment

Vars:
agent__0__see__ball__right : boolean;
agent__0__see__ball__centre : boolean;
agent__0__see__ball__left : boolean;
agent__0__see__box__right : boolean;
agent__0__see__box__left : boolean;
agent__0__see__box__centre : boolean;
agent__0__holding__ball : boolean;
agent__0__holding__box : boolean;
dummy : boolean;
end Vars

Actions = {update};

Protocol:
Other : {update};
end Protocol

Evolution:
(agent__0__see__box__centre = false) if (Action = update);
(agent__0__see__box__centre = true) if (Action = update);
....
end Evolution
end Agent


Agent Agent__0

Lobsvars = {agent__0__see__box__left,
            ....
            agent__0__holding__ball};

Vars:
dummy : boolean;
end Vars

Actions = {turn__left, idle__, grab__box, release};

Protocol:
(Environment.agent__0__holding__box = true) : {idle__};

(!(Environment.agent__0__holding__box = true)
and !(Environment.agent__0__see__box__centre = true)
and (Environment.agent__0__holding__ball = true)) : {release};

(!(Environment.agent__0__holding__box = true)
and (Environment.agent__0__see__box__centre = true)) : {grab__box};
```

```
(!(Environment.agent__0__holding__ball = true)
and !(Environment.agent__0__holding__box = true)
and !(Environment.agent__0__see__box__centre = true)): {turn__left};
end Protocol

Evolution:
dummy = true if dummy = true;
dummy = false if dummy = false;
end Evolution

end Agent

Evaluation
agent__0__see__ball__right
    if Environment.agent__0__see__ball__right = true;
agent__0__idle__
        if (Environment.agent__0__holding__box = true);
agent__0__turn__centre
        if (Environment.dummy = false and Environment.dummy = true);
....
end Evaluation
InitStates
((Agent__0.dummy = true) and (Environment.dummy = true));
end InitStates
```

Environment Agent

The environment has a local variable for each instantiation of percepts and beliefs. The only action that it has is `update`, and the protocol is such that it always selects its only action `update`. The evolution of the environment agent is such that it has two rules for each local variable. They have the same condition `Action = update` and one rule sets the local variable to `true`, and another to `false`. This makes each local variable set to either `true` or `false`.

Agent 0

The `Agent__0` agent on the other hand does not have a meaningful local variable and observes the local variable of the environment, so it may use it in the protocol.
`Agent__0` agent has four actions, representing the only four actions taken by Object Grabbing Agent that runs `get_object(box)`, which are `turn(left), (), grab(box)` and `release()`.
The protocol of the `Agent__0` clearly corresponds to that of the source TeleoR program, and the expected output described in the previous section; If the agent is holding box, then it goes idle. If it is not holding the box yet it is already facing the box, grab the box. If it is not holding the ball or the box, and it does not see the box right in front, then turn left. If the agent is not holding or facing the box and it is holding something else, then releases whatever it is holding.

Evaluation

Expectedly, the atoms are defined for the local variables of the environment and the action of the agent. The atom for the action of the agent is defined with the condition of the protocol rules which triggers it if there is any, and otherwise defined with false condition.

All in all ISPL is as expected, and correctly characterises Object Grabbing Agent in a completely non-deterministic environment. We look at the following three specifications:

- `AG(!agent__0__nfr_fail)`
  In no reachable state `agent__0__nfr_fail` holds. That is, it is not possible for Object Grabbing Agent to be in the error state because it does not have a fireable rule.
- `AG(!agent__0__md_fail)`
  In no reachable state `agent__0__md_fail` holds. That is, it is not possible for Object Grabbing Agent to be in the error state because of call stack overflow.
- `AF(agent__0__holding__box)`
  From the initial state, whatever happens, `agent__0__holding__box` holds at one point. That is, the Object Grabbing Agent should achieve its top goal no matter what happens.

Running MCMAS

Running `mcmas -c 3 box_grabbing.mas` gives the following output:

```
Checking formulae...
Verifying properties...
  Formula number 1: (AG (! agent___0___nfr_fail)), is TRUE in the model
  Formula number 2: (AG (! agent___0___md_fail)), is TRUE in the model
  Formula number 3: (AF agent___0___holding___box), is FALSE in the model
  The following is a counterexample for the formula:
   < 0 0 >
  States description:
------------- State: 0 -----------------
Agent Environment
  agent___0___holding___ball = true
  agent___0___holding___box = false
  agent___0___see___ball___centre = true
  agent___0___see___ball___left = false
  agent___0___see___ball___right = false
  agent___0___see___box___centre = false
  agent___0___see___box___left = false
  agent___0___see___box___right = true
  dummy = true
Agent Agent___0
  dummy = true
----------------------------------------
```

Expectedly, the first two properties hold. The third property however does not hold, as the environment is such that it is not affected by the action of an agent.

6.1.3 Environment Configuration

Let us come up with a sensible assumption we may make about the environment, in which we expect the agent to actually be able to grab the box. We see that these assumptions can be expressed with the environment configuration rules.
We say that the environment is static, in a sense that it will not change without the agent doing something. This can be expressed as `DONTFLIP holding, see;`
It is sensible to assume that there is actually a box for the robot to grab, and the turning speed of the agent is slow enough that it can face it, as otherwise it is impossible for the agent to achieve its top goal. In other words, if the agent keeps on turning to the left, eventually it should see the box in the centre. This can be expressed as `TRUE + turn(left) EVENTUALLY remember see(box, centre);`.
Let us assume that the gripper is functional enough that releasing the object should always succeed and it drops everything it was holding, and grabbing the object right in front is always successful. This can be expressed as a rule `TRUE + release() DEFINITELY forget holding(_);` and `see(X, centre) + grab(X) DEFINITELY remember holding(X);`
Summarising, we have the following four rules:

```
DONTFLIP holding, see;
TRUE + turn(left) EVENTUALLY remember see(box, centre);
TRUE + release() DEFINITELY forget holding(_);
see(X, centre) + grab(X) DEFINITELY remember holding(X);
```

6.1.4 Compiled ISPL with Configuration

Now let us compile an ISPL with the environment configuration rules, and confirm that the compiler indeed outputs an ISPL reflecting the environment configuration. Then we run MCMAS on the ISPL to see whether there is a flaw in the program.
Running `trtis input.qlg -c start_task_call.txt -o output.mas -d 100 -e config.txt` where config.txt contains four rules described in the previous subsection outputs:

```
Semantics = SA;
Agent Environment
Vars:
agent__0__see__ball__right : boolean;
....
prev__agent__0__see__ball__right : boolean;
....
agent__0__taken__action
    : {turn__left, idle__, grab__box, release, none};
end Vars

Actions = {update};

Protocol:
Other : {update};
end Protocol

Evolution:
(prev__agent__0__holding__box = false)
```

```
    if (agent__0__holding__box = false);
....
(agent__0__taken__action = turn__left)
    if Agent__0.Action = turn__left;
....
(agent__0__holding__ball = false) if (Agent__0.Action = release);
(agent__0__holding__box = false) if (Agent__0.Action = release);
(agent__0__holding__box = true)
    if ((Agent__0.Action = grab__box)
    and (agent__0__see__box__centre = true));
end Evolution

end Agent


Agent Agent__0
-- Same as the ISPL compiled without the environment configuration rules
end Agent

Evaluation
prev__agent__0__see__box__centre
    if Environment.prev__agent__0__see__box__centre = true;
....
end Evaluation
InitStates
(((Agent__0.dummy = true)
    and (Environment.dummy = true))
    and (Environment.prev__agent__0__see__box__centre = false)
    ...
    and Environment.agent__0__taken__action = none);
end InitStates
Fairness
(!agent__0__taken__turn__left or agent__0__see__box__centre);
end Fairness
```

We highlight the difference between this ISPL and the ISPL compiled without the environment configuration rules.

Environment Agent

The local state of the environment agent is expectedly augmented with local variables for the previous BeliefStore state and the action which was just taken.
The evolution of the environment agent expectedly has new rules to support the semantics of the new local variables. That is, the local variables with an identifier `prev__ID` is updated with the value of the local variable with the identifier `ID`, and `agent__0__taken__action` is updated with the value of `Agent__0.Action`.
In addition, the evolution of the environment agent is augmented with three evolution rules that are derived from two `DEFINITELY` rules in the environment configuration. Specifically:

- `TRUE + release() DEFINITELY forget holding(_);`

This rule adds:

```
(agent__0__holding__ball = false) if (Agent__0.Action = release);
(agent__0__holding__box = false) if (Agent__0.Action = release);
```

- `see(X, centre) + grab(X) DEFINITELY remember holding(X)`
  This rule adds:

  ```
  (agent__0__holding__box = true)
      if ((Agent__0.Action = grab__box)
      and (agent__0__see__box__centre = true));
  ```

As `holding` and `see` are declared in the `DONTFLIP` rule, the evolution of the environment does not have rules that was seen in the ISPL compiled without the environment configuration.

Agent 0

This agent has no change from the one in the ISPL without the environment configuration.

Evaluation

The evaluation is correctly augmented with the definition of the atoms used to directly refer to the new local variables.

Initial State

Terms are added to initialise all `prev__` variables to false, and `agent__0__taken__action` to `none`.

Fairness

There is one fairness condition which must hold infinitely often, added due to one `EVENTUALLY` rule. Specifically:

- `TRUE + turn(left) EVENTUALLY remember see(box, centre);`
  This rule adds:

  ```
  (!agent__0__taken__turn__left or agent__0__see__box__centre);
  ```

  which excludes the paths where the agent keeps on turning left without seeing the box in the centre.

All in all, the produced ISPL is as specified in the previous chapter.

Running MCMAS

Running MCMAS, we verify that all three formulae holds for the ISPL with environment configuration, thus we may conclude that the Object Grabbing Agent will hold the box without going to the error state, under the assumption that we gave in the environment configuration file.

6.2 Bottle Collecting Agent

Now we look at the Bottle Collecting Agent, which is a simplified example from a literature[6]. With this example, we see an environment configuration that we may write, and how we might want to extend the rules so the assumption we make is not too strong.

6.2.1 TeleoR Program

```
dir::= left | centre | right
thing::= bottle | drop
distance ::= 1..10
angle ::= 1..5
percept
    holding : (thing),
    gripper_open : (),
    next_to:(thing,dir),
    see:(thing,dir)
discrete
    open_gripper : (),
    close_gripper : ()
durative
    move:(distance),
    turn:(dir,angle)

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

get_bottle : () ~>
get_bottle() {
    next_to(bottle, centre) & holding(bottle) ~> ()
    next_to(bottle,centre) & gripper_open ~> close_gripper
    gripper_open ~> get_next_to(bottle)
    true ~> open_gripper
}

deliver_bottle : ()~>
deliver_bottle(){
    next_to(drop,_) & gripper_open ~> ()
    next_to(drop,_) ~> open_gripper
    true ~> get_next_to(drop)
}

get_next_to:(thing)~>
get_next_to(Th){
    Th = bottle & next_to(bottle,centre) ~> ()
    Th = drop & next_to(drop,_) ~> ()
    Th = bottle & next_to(bottle,left) ~> turn(left,2)
    Th = bottle & next_to(bottle,right) ~> turn(right,2)
    see(Th,_) ~> approach(Th,5,1)
    true ~> turn(left,5)
}
approach:(thing,distance,angle)~>
approach(Th,Fs,Ts){
    see(Th,centre) ~> move(Fs)
    see(Th,left) ~> move(Fs),turn(left,Ts)
    see(Th,right) ~> move(Fs),turn(right,Ts)
}
task_start collect_bottle : ()
```

The top goal of this agent is to put a bottle in the drop. This is expressed as being next to both drop and the bottle. If the agent is holding the bottle, then it calls a procedure, `deliver_bottle`, otherwise it calls `get_bottle`.
`deliver_bottle` succeeds if the agent is next to the drop, and it has its gripper open. If it is next to the drop and it does not have its gripper open, it opens the gripper. Otherwise, it calls a procedure `get_next_to`.
`get_bottle` succeeds if the agent is in front of and holding the bottle. If it is in front of the bottle and not holding the bottle, it attempts to grab it. If it is not in front of the bottle, it calls a procedure `get_next_to`.
In `get_next_to`, if the argument is the bottle, then the procedure succeeds if the agent faces it. If it is the drop, then the procedure succeeds if it is next to it, in whatever direction. In order to face the bottle, the agent would turn to the direction in a small step, so it does not lose the object from its sight. If the agent sees the object and is not next to it, then it calls a procedure `approach`, which tries to get next to the object by moving and turning around. Otherwise, it tries to look for the object by turning around at a higher speed.

6.2.2 Environment Configuration

The following is an example environment configuration one may write for this program.

```
INIT : not holding(bottle)
% The environment is static and it does not change unless the agent
% does something.
DONTFLIP holding, gripper_open, see, next_to;

% Turning slowly, it should at one point sees the object
% right in the front.
see(Th, X) + turn(X, 1)
    EVENTUALLY remember see(Th, centre); forget see(Th, X);


% If the agent is next to something, turning at the medium speed
% should allow the agent to face it at one point.
next_to(Th, X) + turn(X, 2) EVENTUALLY remember next_to(Th, centre);
```

```
% By turning at high speed, the agent might find something
% or lose something from its sight.
TRUE + turn(_, 5) MAYCHANGE see(_, _);


% The bottle and drop should be in the field
% where the agent may at some point find them by turning
% at high speed.
TRUE + turn(X, 5) EVENTUALLY remember see(bottle, X);
TRUE + turn(X, 5) EVENTUALLY remember see(drop, X);


% Unless the agent is not holding the object, the agent
% might move away / or get close to the object.
not holding(X) + move(_) MAYCHANGE next_to(X ,_);


% The gripper is functional enough that, if the agent
% is right in front of the bottle and it closes the gripper,
% it will definitely hold the bottle and recognise that
% the gripper is not open any more.
next_to(bottle, centre) + close_gripper
    DEFINITELY remember holding(bottle); forget gripper_open;


% The gripper is functional enough that, opening the gripper
% would always result to dropping whatever it was holding,
% and the agent will recognise that the gripper is open.
TRUE + open_gripper
        DEFINITELY forget holding(_); remember gripper_open;


% If we see the object in front and move straight,
% the agent should get right in front of it at some point
see(Th, centre) + move(5) EVENTUALLY remember next_to(Th, centre);
```

They are all fairly reasonable, except perhaps for the last rule. It is plausible that the agent sees the object in front, but with some angle. If the agent moves straight, then it will probably get next to the object, but either on its left or right. This can not be expressed with the environment configuration we have. We would need to extend the syntax of the rules and the compiler to accept something along the line of:

```
see(Th, centre) + move(5)
        EVENTUALLY remember (next_to(Th, centre)
        or next_to(Th, left)
        or next_to(Th, right));
```

Similarly, it is plausible that by moving straight, the relative position of the object it sees changes. For instance, if the agent sees the object in front and moves, then it might see the object to its left or right. It may be useful to have a rule:

```
see(Th, centre) + move(5)
```

```
        MAYREPLACE see(Th, centre) WITH see(Th, left) or see(Th, right);
```

The model specified in the ISPL compiled with the above environment configuration satisfies the following specifications:

- `AG(!agent__0__nfr_fail)`
  In no reachable state `agent__0__nfr_fail` holds. That is, it is not possible for Bottle Collecting Agent to be in the error state because it does not have a fireable rule.
- `AG(!agent__0__md_fail)`
  In no reachable state `agent__0__md_fail` holds. That is, it is not possible for Bottle Collecting Agent to be in the error state because of call stack overflow.
- `AF((agent__0__next_to__bottle__centre or agent__0__next_to__bottle__left or agent__0__next_to__bottle__right) and (agent__0__next_to__drop__centre or agent__0__next_to__drop__left or agent__0__next_to__drop__right) and agent__0__gripper_open);`
  From the initial state, whatever happens, Bottle Collecting Agent will be next to both the bottle and the drop in some direction, and has its gripper open. That is, the Object Grabbing Agent should achieve its top goal no matter what happens.

6.3 Tower Building Agent

6.3.1 TR Program

```
tab ::= table | floor
block ::= a | b | c
loc ::= block | tab
belief
    holding:(block),
    on:(block, loc),
    over : (loc)

discrete
    grab:(block),
    move_over:(loc),
    release:()

task_start tower : ()
tower() {
    on(a,b) & on(b, c) & on(c, table) ~> ()
    on(b,c) & on(c, table)
        & not on(_, a) & not on(_, b) & holding(a) ~> place(a, b)
    on(b,c) & on(c, table)
        & not on(_, a) & not on(_, b) & not holding(_) ~> grab(a)
    on(c, table) & not on(_, c)
        & not on(_, b) & holding(b) ~> place(b, c)
    on(c, table) & not on(_, c)
        & not on(_, b)  & not holding(_)~> grab(b)
    true ~> putAllOnTable()
}


putAllOnTable : () ~>
putAllOnTable() {
    on(a, table) & on(b, table) & on(c, table) ~> ()
    holding(a) ~> place(a, table)
    holding(b) ~> place(b, table)
    holding(c) ~> place(c, table)
    on(a, X) & not on(_, a) & X \= table ~> grab(a)
    on(b, X) & not on(_, b) & X \= table ~> grab(b)
    on(c, X) & not on(_, c) & X \= table ~> grab(c)
}

place : (block, loc) ~>
place(Block, Loc) {
    not holding(Block) & on(Block, Loc)  ~> ()
    holding(Block) & over(Loc) ~> release()
    holding(Block) ~> move_over(Loc)
}
```

The top-goal of this agent is to build a tower with blocks on the table, where block 'a' is at the top, then followed by block 'b' in the middle, and block 'c' is at the bottom.
In order to do so, the agent accumulates the block from 'c'. If the blocks are arranged in a way that there is no sub tower, then the agent puts all the blocks on the table first, and then starts building the tower.

6.3.2 Environment Configuration

One possible environment configuration is presented below.

```
% The environment is static
DONTFLIP holding, on, over;

% The initial configuration of the blocks
%  b
%  a
%  c
% ----

INIT on : on(b, a), on(a, c), on(c, table)
          , not on(_, floor)
          , not on(_, b)
          , not on(b, c), not on(b, table), not on(b, b)
          , not on(a, b), not on(a, a), not on(a, table)
          , not on(c, c), not on(c, a), not on(c, b);

% At start, the agent is not holding anything
INIT holding: not holding(_);
```

```
% If there is nothing on X, and the agent is not holding
% anything, grabbing X will succeed
% and the agent holds X, and X is not on anything.
not on(_, X) & not holding(_) + grab(X)
    DEFINITELY remember holding(X); forget on(X, _);

% If there is nothing on Loc, moving over eventually
% allow the agent to be directly above Loc
not on(_, Loc) + move_over(Loc)
    EVENTUALLY forget over(_); remember over(Loc);

% If there is nothing on the block, then move_over
% may make the agent be above it, or
% may move the agent away from the block
not on(_, a) & not holding(a) + move_over(_)
    MAYCHANGE over(a);
not on(_, b) & not holding(b) + move_over(_)
    MAYCHANGE over(b);
not on(_, c) & not holding(c) + move_over(_)
    MAYCHANGE over(c);

% By moving over, the agent may move away
% or move over the table.
TRUE + move_over(_) MAYCHANGE over(table);

% The gripper of the agent is functional enough that,
% if the agent holds the block and directly above
% the location, releasing the block put it on
% the location, and the agent does not hold the block
% any more.
holding(Block) & over(Loc) + release
    DEFINITELY forget holding(Block); remember on(Block, Loc);
```

In environment configuration rules, we can to a good extent talk about the property of each action. However, we cannot do so about the dynamic facts. For instance, it would have been useful to be able to express "A block can not be on itself", or "Two blocks can not be on the same block". As we can not convey this, we can not test on all initial states, where blocks are arranged in a reasonable manner. It requires manually changing the `INIT` rule for `on`.

6.3.3 Debugging the TeleoR program

We see how we can utilise TRTIS and MCMAS to debug TeleoR programs. Conventionally, one may test TeleoR programs by running against the environment simulator, or interacting with the agent through Python robot shell, from which one can update the BeliefStore of the agent[6]. These techniques are time-consuming and not thorough, as it is hard, if not impossible, to consider all possible sequences of updates in the BeliefStore.
Using TRTIS and MCMAS on the other hand, given that we have the environment configuration rules

which correctly characterise the environment, is automatic and it is guaranteed to be complete. In this subsection, we see how one might debug the TeleoR program.

We compile ISPL with the configuration rules presented in the previous subsection, and run MCMAS against the following specifications:

- `AG(!agent__0__nfr_fail)`
  In no reachable state `agent__0__nfr_fail` holds. That is, it is not possible for Tower Building Agent to be in the error state because it does not have a fireable rule.
- `AG(!agent__0__md_fail)`
  In no reachable state `agent__0__md_fail` holds. That is, it is not possible for Tower Building Agent to be in the error state because of call stack overflow.
- `AF(agent__0__on__a__b and agent__0__on__b__c and agent__0__on__c__table)`
  From the initial state, whatever happens, the Tower Building Agent builds the tower with 'a' on the top, then 'b', and finally 'c' at the bottom. That is, the Tower Building Agent should achieve its top goal no matter what happens.

Running `mcmas -c 3 tower_building.mas` outputs:

```
Verifying properties...
  Formula number 1: (AG (! agent__0__nfr_fail)), is TRUE in the model
  Formula number 2: (AG (! agent__0__md_fail)), is TRUE in the model
  Formula number 3: (AF ((agent__0__on__a__b && agent__0__on__b__c)
  && agent__0__on__c__table)), is FALSE in the model
  The following is a counterexample for the formula:
   < 0 1 2 3 2 >
  States description:
------------- State: 0 -----------------
Agent Environment
  agent__0__on__a__c = true
  agent__0__on__b__a = true
  agent__0__on__c__table = true
  agent__0__over__a = true
  agent__0__over__floor = true
  agent__0__over__table = true
  agent__0__taken__action = none
 ...
----------------------------------------
------------- State: 1 -----------------
Agent Environment
  agent__0__holding__b = true
  agent__0__on__a__c = true
  agent__0__on__c__table = true
  agent__0__over__a = true
  agent__0__over__floor = true
  agent__0__over__table = true
  agent__0__taken__action = grab__b
 ...
----------------------------------------
------------- State: 2 -----------------
```

```
Agent Environment
  agent__0__on__a__c = true
  agent__0__on__b__a = true
  agent__0__on__b__floor = true
  agent__0__on__b__table = true
  agent__0__on__c__table = true
  agent__0__over__a = true
  agent__0__over__floor = true
  agent__0__over__table = true
  agent__0__taken__action = release
...
----------------------------------------
------------- State: 3 -----------------
Agent Environment
  agent__0__holding__b = true
  agent__0__on__a__c = true
  agent__0__on__c__table = true
  agent__0__over__a = true
  agent__0__over__floor = true
  agent__0__over__table = true
  agent__0__taken__action = grab__b
...
----------------------------------------
```

Note that it is modified so that local variable start with `prev__` is omitted, and only the local variables of the environment agent whose value is true is shown.
We can see what the problem is in the state 2 and the state 3, which alternate to each other forever. At the state 3, we notice that the agent executes `release` to put the block ‘b’ on the table. As the agent is in between multiple objects including the table when doing so, in the state 2, the block is on floor, table, and block ‘a’.
Though it is arguable that this particular path is possible in the actual environment, one would notice that with this program it is possible that the block the agent puts might be placed on two blocks, which is undesirable. To mitigate this problem, we change the procedure `place` to the following.

```
place : (block, loc) ~>
place(Block, Loc) {
    not holding(Block) & on(Block, Loc)  ~> ()
    holding(Block) & over(Other) & Other \= Loc ~> move_over(Loc)
    holding(Block) & over(Loc) ~> release()
    holding(Block) ~> move_over(Loc)
    true ~> grab(Block)
}
```

With this modification, the agent will continue to move if it is in between multiple objects, until it is directly above one object.
By running MCMAS against the new ISPL, we see that the model indeed satisfies all the specifications.

6.4 Performance

In this section we present the performance of TRTIS and MCMAS for the agents presented so far. We also present a performance measure on what we call Bottle Collecting Agent 2, which has the same TeleoR program as Bottle Collecting Agent, but have an environment configuration with one line changed, such that is *dontFlip* rule only has *holding* and *gripper_open*.

We compile these agents with and without the environment configuration, and verify three specification formulae we discussed in the respective case studies. We measure the performance on 64-bit Ubuntu 14.04 Linux machine with a 3.40GHz Intel Core i7-3770 processor and 15.9GiB RAM. The execution time presented is the average of 1000 iterations.

TeleoR program	TRTIS compilation time(seconds)
Object Grabbing	0.4552
Bottle Collecting	2.2896
Tower Building	3.6585

Table 6.1: Mean TRTIS compilation time without environment configuration.

TeleoR program	TRTIS compilation time(seconds)
Object Grabbing	0.5090
Bottle Collecting	2.0126
Bottle Collecting 2	1.8955
Tower Building	3.7674

Table 6.2: Mean TRTIS compilation time with environment configuration.

TeleoR program	reachable states	memory(MB)	execution time(seconds)
Object Grabbing	256	≈ 9.0	0.0041
Bottle Collecting	32768	≈ 9.1	0.0057
Tower Building	8.38861e+06	≈ 9.8	0.0277

Table 6.3: MCMAS benchmark without environment configuration.

TeleoR program	reachable states	memory(MB)	execution time(seconds)
Object Grabbing	512	≈ 9.2	0.0169
Bottle Collecting	79872	≈ 11.8	0.2372
Bottle Collecting 2	1.0068e+08	≈ 11.3	0.1401
Tower Building	432	≈ 10.8	0.5055

Table 6.4: MCMAS benchmark with environment configuration.

With or without the environment configuration, the compilation time of TRTIS is within a reasonable range. Surprising enough, the compilation time do not change much with environment configuration. This is probably because the execution time is mainly spent by $toMCMASProtocol$ the most heavy weight function in TRTIS, and the time spent on additional task of generating additional components such as fairness condition is almost negligible.

During the development, we realised the execution time of $toMCMASProtocol$, in Definition 5.16, gets influenced by how often the boolean expressions of protocol rules are simplified. We use a

jbool_expression library[32] to represent MCMAS condition terms. Recall in *toMCMASProtocol* the expression gets accumulated as we go deeper in the call sequence, and as we go to the later rules in the procedure. If we reduce the expression as we add terms to it, the compilation time gets unacceptably long, sometimes going over a minute for the TeleoR programs we presented in this chapter. Therefore we reduce the expression only once, at the end, and never do so while we are accumulating the expression.
However we realised that for some programs, without an intermediate simplification, the size of the expression gets too big and goes over java heap size. Clearly, we need to find a good middle ground, or consider an alternative approach.

MCMAS executes very quick for all the agents. The execution time does not seem to grow in proportion to the number of states. Bottle Collecting Agent 2 has by far the largest reachable states, but its execution time is not the longest. It does more seem to be the case that the execution time is correlated to the memory usage, however Tower Building Agent has a significantly longer execution time than other agents has smaller memory use and smaller number of reachable states.

In general, the number of reachable state is higher with environment configuration. This may seem counter-intuitive, but this is because without the environment configuration, there is only one state per possible BeliefStore state, whereas there are multiple state with the same BeliefStore state, with possible different previously taken action or the previous BeliefStore states.

6.5 User Experience

When the user input an illegal program, depending on the errors, the feedback from the compiler may or may not be helpful. The compiler can inform the user if for instance:

- There is an ambiguous rule.
- There is a circular dependency among the user-defined types.
- The starting task is not specified.

However, as ANTLR grammar we use for TRTIS is based on EBNF grammar of TeleoR[31], it does not reject a TeleoR program that is not a Single Bound Program. This should be improved by cleaning up the ANTLR grammar so it only accepts Single Bound Program.

6.6 Summary

In this chapter, we saw three Single Bound Programs. With Object Grabbing Agent, we saw the actual behaviour of TRTIS, which was explained in the previous chapters. We also saw that for this simple program, the environment configuration rule we have is good enough.

With Bottle Collecting Agent and Tower Building Agent, we discussed how we need to make a strong assumption on the environment, as the environment configuration rule is not expressive enough. Extending the environment configuration rules such that one may describe the environment fully would be an appropriate extension to this project.

Using Tower Building Agent as an example, we saw how one can debug a TeleoR program with TRTIS and MCMAS. Compared to conventional testing using the environment simulator or the Python agent shell, TRTIS and MCMAS have the advantage of being completely automatic and thorough, given an appropriate environment configuration.

Finally, we saw the performance of TRTIS and MCMAS on these three programs. The performance was good for the agents presented in this chapter. However we discussed that dealing with boolean expressions becomes the bottleneck with TeleoR procedures with more rules and higher call depth, therefore we need to have a better strategy around it.

Chapter 7

Conclusion

7.1 Summary of Work

Our goal was to present a mechanism of automating the verification of teleo-reactive program. This project successfully presented a promising methodology to do so; it involves formalising the semantics of TeleoR program as a model that can be verified by a model checker, and devising a compiler that outputs a corresponding model to the input TeleoR program.

We also presented the formalisation of the semantics of Single Bound Program as an interpreted system. This formalisation is mathematically proved to be correct, therefore the future work to support a larger class of programs can be confidently based on this.

We presented environment configuration rules which can be used to avoid verifying a program against unreasonably hostile environments. We evaluated their expressivity by looking at some actual TeleoR program, and how they may be extended.

We specified and implemented the compiler which translates TeleoR into ISPL. Though the performance and the class of programs it can verify is limited, the behaviour is as expected. The algorithms presented can be reused and extended for future work. We also saw that the compiler can be useful to debug Single Bound Program.

All in all, we believe that this project can be a good foundation for this field which is yet to be cultivated further.

7.2 Future Work

We hope that this work could be a solid basis on which future works may be based. The following is a list of possible extensions of this project:

- **Support multiple robotic agents.**
  As mentioned previously, in teleo-reactive programming, multiple agents may be specified to collaboratively achieve a common goal. We considered supporting the scenarios of multiple robotic agents, each of them running as a separate process, by simply composing the agents produced by TRTIS. However we did not have enough time to think thoroughly on how we could faithfully represent their collaboration. There are two ways these agents may interact. One is by messaging with each other. This requires extending the compiler so it may support

user defined action. The other is an “implicit interaction via dynamic fact about each other. For instance in the example program in Qulog distribution[8], agents makes an action when it *sees* the other agent. We were uncertain how we could represent a reasonable environment with such an “implicit interaction. One possible way of achieving this is to allow the user to specify certain percepts to be shared. For instance, it might make sense to have all the agents know which agent is next to a certain object. This way we may express that, if an agent sees the object, then it sees the agent next to it as well.

- **Support a bigger class of TeleoR.**
  We mentioned that TeleoR, or more precisely Qulog, is Turing-complete. Therefore it is impossible to have an automatic verification on the whole class of TeleoR. However there are various features that one may support, which we could not do so as there was no formal semantics available to us until the very end of the project. We believe that otherwise it would have been interesting to support:

  – Multi-tasking with multiple evaluator threads, which synchronise with each other on the shared resources

  – until / while rules

  – User-defined relation, function and action, avoiding Turing-complete fraction

- **Extend environment configuration rules.**
  The environment configuration would be optimal if one could express the exact assumption of the environment. Therefore the richer the environment configuration rules are, the better it is. We discussed some ways they can be extended, such as

  – MAYREPLACE
  Rules which specify that taking an action may replace some facts with other facts.

  – Rules that describe properties of the dynamic facts
  In a way, environment configuration rules we have give some semantics to the actions. Then it might be beneficial to have rules which describes a properties of the dynamic facts. For instance, it would be convenient to be able to express that some facts are mutually exclusive such as $on(a, b)$ and $on(b, a)$, which say that both block a and b are on top of each other.

Bibliography

[1] N. J. Nilsson, "Teleo-reactive programs for agent control," *JAIR*, vol. 1, pp. 139–158, 1994.

[2] N. G. Leveson and C. S. Turner, "An investigation of the therac-25 accidents," *Computer*, vol. 26, no. 7, pp. 18–41, 1993.

[3] K. L. Clark and P. J. Robinson, "Robotic agent programming in TeleoR," in *Proceedings of International Conference on Robotics and Automation*, pp. 5040–5047, IEEE, 2015.

[4] "Related publications." `http://teleoreactiveprograms.net/?page_id=10`. Accessed: 04-06-2016.

[5] K. L. Clark and P. J. Robinson, "Multi-tasking robotic agent programming in teleor,"

[6] K. L. Clark and P. J. Robinson, "Programming Robotic Agents : A Teleo-Reactive Multi-Tasking Approach." `http://teleoreactiveprograms.net/wp-content/uploads/2014/08/BookSample26-11-14.pdf`. Accessed: 03-01-2016.

[7] K. Broda, K. Clark, R. Miller, and A. Russo, "SAGE: a logical agent-based environment monitoring and control system," in *Proceedings of European Conference of Ambient Intelligence*, pp. 112–117, Springer, 2009.

[8] P. J. Robinson, "The Qulog/TeleoR 0.4 Reference Manual." `http://staff.itee.uq.edu.au/pjr/HomePages/QulogFiles/manual/index.html`. Accessed: 03-01-2016.

[9] A. Lomuscio and F. Raimondi, "Model checking knowledge, strategies, and games in multi-agent systems," in *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pp. 161–168, ACM, 2006.

[10] R. E. Bryant, "Symbolic boolean manipulation with ordered binary-decision diagrams," *ACM Computing Surveys (CSUR)*, vol. 24, no. 3, pp. 293–318, 1992.

[11] M. Huth and M. Ryan, *Logic in Computer Science: Modelling and reasoning about systems.* Cambridge University Press, 2004.

[12] E. Clarke, "The birth of model checking," *25 Years of Model Checking*, pp. 1–26, 2008.

[13] A. Lomuscio and F. Raimondi, "MCMAS: A model checker for multi-agent systems," in *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems*, vol. 3920, pp. 450–454, Springer, 2006.

[14] D. Bliosi, "An introduction to Alternating-time Temporal Logic." `http://www.dis.uniroma1.it/~bloisi/didattica/apprendimento0809/atl.pdf`. Accessed: 26-01-2016.

[15] R. Alur, T. A. Henzinger, and O. Kupferman, "Alternating-time temporal logic," *Journal of the ACM (JACM)*, vol. 49, no. 5, pp. 672–713, 2002.

[16] A. Lomuscio, H. Qu, and F. Raimondi, "MCMAS: A model checker for the verification of multi-agent systems," in *Proceedings of Computer Aided Verification*, pp. 682–688, Springer, 2009.

[17] A. Lomuscio and M. Sergot, "On multi-agent systems specification via deontic logic," in *Intelligent Agents VIII*, pp. 86–99, Springer, 2001.

[18] "MCMAS v1.2.2: User Manual." `http://vas.doc.ic.ac.uk/downloads/manual.pdf`. Accessed: 27-01-2016.

[19] B. Dongol, I. J. Hayes, and P. J. Robinson, "Reasoning about goal-directed real-time teleo-reactive programs," *Formal Aspects of Computing*, vol. 26, no. 3, pp. 563–589, 2014.

[20] I. Boureanu, M. Cohen, and A. Lomuscio, "Automatic verification of temporal-epistemic properties of cryptographic protocols," *Journal of Applied Non-Classical Logics*, vol. 19, no. 4, pp. 463–487, 2009.

[21] "Prolog." `https://en.wikipedia.org/wiki/Prolog`. Accessed: 01-05-2016.

[22] I. Hodkinson, "C499: Modal and temporal logic." University Lecture, Department of Computing, Imperial College London, 2016.

[23] "IntelliJ IDEA." `https://www.jetbrains.com/idea/`. Accessed: 01-05-2016.

[24] "A Tour of Scala." `http://docs.scala-lang.org/tutorials/tour/tour-of-scala.html`. Accessed: 04-05-2016.

[25] "Different ways to create and populate Lists in Scala." `http://alvinalexander.com/scala/different-ways-create-populate-list-scala-cookbook-range-nil-cons`. Accessed: 04-05-2016.

[26] "ANTLR." `http://www.antlr.org/`. Accessed: 01-05-2016.

[27] P. J. Robinson, "Qu-Prolog." `http://staff.itee.uq.edu.au/pjr/HomePages/QuPrologHome.html`. Accessed: 01-05-2016.

[28] "Beaver." `http://beaver.sourceforge.net`. Accessed: 08-05-2016.

[29] "JavaCC." `https://javacc.java.net`. Accessed: 08-05-2016.

[30] T. Parr, *The definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2013.

[31] P. J. Robinson, "EBNF grammar for Qulog." `http://staff.itee.uq.edu.au/pjr/HomePages/QulogFiles/manual/EBNF-Grammar-for-Qulog.html`. Accessed: 01-05-2016.

[32] "jbool_expression library." `https://github.com/bpodgursky/jbool_expressions`. Accessed: 08-06-2016.