Model Checking Jason Programs

Steen Vester

Kongens Lyngby 2010
IMM-B.Sc.-2010-10

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

IMM-B.Sc.-2010-10

Summary

Agent-Oriented Programming (AOP) is a relatively new paradigm used for developing multi-agent systems. Within this paradigm agents are often described using the Belief-Desire-Intention model (BDI).

This report focuses on developing a system for verification of programs written in Jason, which is a Java-based interpreter for an extended version of the language AgentSpeak.

AgentSpeak is build around the BDI model and is a high level programming language for developing intelligent agents. This is done with concepts like beliefs, goals and plans.

When describing the properties to be verified we use an extension of the branching-time temporal logic CTL (Computation Tree Logic) which makes it possible to reason about BDI properties over time.

A formal semantics for the language will be used to make a procedure that generates transitions systems representing the programs to be verified. Then a model checking algorithm will be applied to these transition systems to verify that some given program properties holds in all possible states of the programs.

Resumé

Agent-Orienteret Programming (AOP) er et relativt nyt paradigme til udvikling af multi-agent systemer. Indenfor dette paradigme bliver agent ofte beskrevet vha. Belief-Desire-Itention modellen (Tro-Ønske-Hensigt modellen på dansk)

Denne rapport fokuserer på udviklingen af et verifikationssystem for programmer skrevet i Jason, som er en Java-baseret fortolker for en udvidet version af sproget AgentSpeak.

AgentSpeak er bygget op omkring BDI modellen og er et højniveau programmeringssprog til at udvikle intelligente agenter. Dette gøres ved at bruge koncepter som tro, mål og planer.

Når vi beskriver egenskaber der skal verificeres bruger vi Computation Tree Logic (Beregningstræslogik på dansk) som gør det muligt at beskrive BDI egenskaber over tid.

Der vil blive brugt en formel semantik for AgentSpeak til at lave en procedure der genererer transitionssystemer som representerer de programmer der skal verificeres. Derefter vil en model checking algoritme blive anvendt på disse transitionssystemer for at verificere at nogle givne program egenskaber holder i all mulige tilstande af programmerne.

Preface

This report has been developed as a bachelor project of the Software Technology program at DTU Informatics.

My primary prerequisites for the work done in this report has been the courses 02141 Computer Science Modelling, 02152 Concurrent Systems, 02180 Introduction to Artificial Intelligence, 02285 Artificial Intelligence and Multi-Agent Systems, 02156 Formal Logical Systems, 02105 Algorithms and Datastructures I, 02110 Algorithms and Datastructures II and 02249 Computationally Hard Problems.

My knowledge about Jason and Multi-Agent Systems comes from the course 02285. Knowledge about First-Order Logic acquired from 02180 and 02156 as well as Linear Time Temporal Logic from 02156 and 02152 have helped me a lot when trying to understand the Computation Tree Logic used in this report.The courses 02141 and 02152 have been helpful in introducing the concepts of model checking and operational semantics, respectively. Finally, the three algorithm courses 02105, 02110 and 02249 have definitely been useful.

Lyngby, Juni 2010

Steen Vester

Acknowledgements

I want to thank my supervisor Jørgen Villadsen from the Institute of Mathematical Modelling at the Danish Technical University. He has helped by pointing me in the right direction and being available for questions at any time. I also want to thank my dad for reading and commenting on the report.

Contents

Chapter 1

Introduction

The purpose of this report is to create a verification system for agent programs written in Jason, which is an interpreter for the language AgentSpeak. Formal verification of system properties is a good supplement to classical testing methods since it will reach all branches of a given program. The problem is that it is often very computationally expensive to generate all possible system states.

Before doing any verification we will first introduce a formal semantics of the version of AgentSpeak that we use. This will be used to develop an algorithm for creating abstract models of AgentSpeak programs automatically.

The abstract models are transition systems representing AgentSpeak programs and thus, if we verify properties of the abstract models we will automatically verify properties of the corresponding AgentSpeak programs.

The properties that we verify will be specified using Computation Tree Logic, which makes it possible for us to reason about the beliefs, desires and intentions of an agent over time. CTL properties can be verified by using the technique of model checking for which there exists an efficient algorithm. This algorithm will also be presented.

Chapter 2

Description of Jason

Jason is an interpreter for an extension of the agent oriented programming language AgentSpeak. In this chapter we will give an introduction to programming in Jason and create a sample program to be used throughout the report. Most of the information in this chapter comes from [1].

In the field of artificial intelligence an agent is typically viewed as a kind of autonomous machine (physical or non-physical) which inhabits an environment. An agent receives information from its environment through percepts which are assumed to be part of its architecture. This architecture also consists of a number of actuators making the agent able to perform actions on its environment.

Jason is a high level programming language used for programming agents that have the ability to sense things about its environment and act on it. A Jason program should tell an agent how to perceive the information it gets and how to act on its environment to fulfill its goals (even if it has multiple and/or conflicting goals). It should be able to deliberate over time, which means that it might be necessary to change its beliefs about the world if it is confronted with new information. It should also be able to make detailed plans for accomplishing its goals in a changing world.

Jason can be used to program these reactive planning systems in a wide range of different environments and is quite flexible when it comes to handling imperfect

information and partially observable environments.

Agents are programmed with notions like beliefs, goals and plans as the central building blocks. There are also possibilities of communication between agents, though we will not be focusing a lot on this aspect.

We start out with describing the approach to beliefs in Jason agents:

2.1 Beliefs

2.1.1 Basic Beliefs

An agent keeps a belief base consisting of statements about the environment currently believed to be true. A belief is represented by a predicate with a non-negative number of terms as parameters. Examples of beliefs are:

```
soccer_is_fun      age(steen, 23)      favourite_color(red)
```

These three beliefs in the belief base of an agent would indicate that the agent believes that soccer is fun, that the age of steen is 23 and that the favourite color of the agent is red. In the first belief there are no terms, but just the predicate `soccer_is_fun`. The predicate `age` is binary and the predicate `favourite_color` is unary.

The different kinds of terms is illustrated by Figure 2.1 (which is taken from [1]).

A predicate followed by a non-negative number of terms as parameters is called an atomic formula. If it does not contain any variables it is called a ground atomic formula.

Note that variables starts with an uppercase letter and atoms starts with a lowercase letter. More information about the use of variables will be presented later.

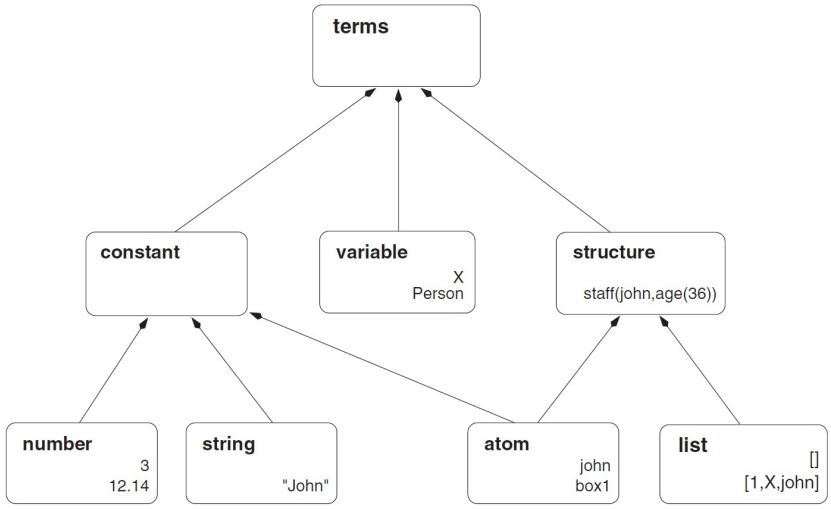

Figure 2.1: Types of terms in Jason

2.1.2 Annotations

Beliefs can also have annotations where one can put information about the source of the belief, the degree of certainty of the belief etc. Annotations are written as a comma separated list in square brackets as follows:

```
taste(mcdonalds, great)[source(tv2)]
```

This means that the agent believes that McDonalds tastes great and it has acquired the knowledge from a TV2 commercial. It is possible for an agent to let its decisions depend on the annotation (for example, if the source was an article from Food and Wine Magazine it might be more likely that McDonalds food actually tastes great).

2.1.3 Strong Negation

Agents in Jason work under the closed world assumption, which means that anything that is not known to be true will assumed to be false. However, there is also the possibility of using the strong negation symbol "$\sim$" to indicate that

something is specifically believed not to be true. For example, an agent might have a specific belief like:

$$\sim \texttt{likes(spinach)}$$

which obviously means that the agent does not like spinach. There is a difference between having this statement in the belief base and not having it. In this example the agent has probably tasted spinach and found out that it does not like it, whereas if it did not have this belief, it might be the case that it likes spinach.

We call an atomic formula with or without a preceeding strong negation symbol a literal, which means that $\sim$ `likes(spinach)` is a literal, but not an atomic formula. All beliefs in the belief base must be literals.

2.1.4 Rules

In addition to the belief base, an agent can be programmed with rules with which it can infer new information from its belief base. Rules are very similar to rules in Prolog. For example, the rule:

```
can_afford_steak_at(S) :- sale_at(S) & has_coupons_for(S).
```

means that the agent can only afford a steak at the shop S if there is a sale at the shop and it also has coupons for that same shop. More precisely, `can_afford_steak_at(S)` is true if there exists an s such that `sale_at(s)` and `has_coupons_for(s)` are beliefs in the agents belief base.

2.2 Goals

In Jason we work with two different kinds of goals: test goals and achievement goals. Test goals are written with a "?" symbol proceeded by a literal and achievement goals are written with a "!" symbol proceeded by a literal.

Test goals are quite similar to goal clauses used in Prolog. By using unification, one can use test goals to get information about the belief base of the agent. For

example, if the belief base contains the belief `age(steen,23)` then the test goal `?age(steen,X)` can be used to retrieve the age of steen by binding the variable `X` to the value `23`.

Whereas Test Goals are primarily used for lookups in the belief base, achievement goals are used to describe goals that the agent should try to accomplish. An example of an achievement goal could be `!go(supermarket)` or `!create_world_peace` which are goals of going to the supermarket and creating world peace respectively. Achievement goals are strongly connected to plans, since plans are the means of the agent to accomplish its goals.

2.3 Plans

A plan in Jason consists of three parts: a triggering event, a context and a plan body. The format of a plan is `te:ct <- h` where `te` is the triggering event, `ct` is the context and `h` is the plan body. We start by describing the triggering event part of the plans:

2.3.1 Triggering Event

A triggering event is used to specify which events a given plan is relevant for. Thus, when a triggering event of a plan matches a given event, then the plan will be a candidate for handling the event.

A triggering event starts with either a "+" or a "-" symbol that is proceeded by either a belief (that must be a literal) or a goal. This gives us the following six types of events: (where l must be a literal)

- $+l$ (Belief Addition)
- $-l$ (Belief Deletion)
- $+?l$ (Test Goal Addition)
- $-?l$ (Test Goal Deletion)
- $+!l$ (Achievement Goal Addition)
- $-!l$ (Achivement Goal Deletion)

The Belief Addition and Deletion events are relevant when there is a change in the agents belief base. This could either be as a consequence of the agents perceptions of the environment or the agent updating its own belief base. For example, if we have a plan p with the triggering event `+hungry(X)` and the agent come to believe `hungry(dog)`, then p will be considered a relevant plan to handle the belief update since the triggering event unifies with the belief.

This is the same way that we select relevant plans to handle goals of the agent. If the triggering event of a plan unifies with the goal of the agent, then the plan will be a relevant candidate for accomplishing the given goal.

2.3.2 Context

The context of a plan is used to find out if relevant plans for a given event are applicable in a given context. A context is usually a conjunction of default literals used to find out if there is a substitution that makes the context follow from the belief base. (there are also possibilities of using disjunction, relational expressions like $X > Y$ and a couple of other constructs, but that is not too important for the focus of this report).

A default literal is either a literal or a literal preceeded with the weak negation operator "`not`". For an atomic formula at, `not` at is true if at is not in the belief base. This is opposed to the strong negation as described earlier, where $\sim$ `at` is true only if $\sim$ `at` is in the belief base.

As an example of the use of contexts, suppose we have an agent that should be able to cook dinner. However, it should use different plans depending on whether there are more than 4 people or not. It could look something like this:

```
+!cook(dinner) :  number_of_guests(X) & X > 4 <- body1.
```

```
+!cook(dinner) :  number_of_guests(X) & X <= 4 <- body2.
```

Both of these plans are relevant for the achievement goal `!cook(dinner)`. However, only the first plan will be applicable if the beliefbase contains `number_of_guests(5)` (and no beliefs of the form `number_of_guests(x)` where $x \leq 4$).

2.3.3 Plan Body

We have now reached the plan body, which is used to handle the belief changes and accomplishing goals.

A plan body is a sequence of formulas separated by semicolons. The types of formula allowed can divided into six major groups: actions, achievement goals, test goals, belief changes, expressions and internal actions.

2.3.3.1 Actions

As noted earlier agents have a number of effectors with which it can act on the environment. Calls to one of these effectors are what we call actions. An action must be atomic formula, where all variables have been instantiated when the action is reached.

2.3.3.2 Test Goals

A formula in a plan body can be a test goal. In the case that the test goal unifies with the belief base, the found substitution will be applied to the rest of the plan. If the test goal is not unifiable with the belief base, the test goal fails. This means that the whole plan fails, since all its formulas must evaluate to true for the plan to succeed. Then, the plan is discarded. (There are possibilities for handling plan failures, but we will not explain the details here). As an example, suppose the agent is executing the plan body `?need(X); buy(X).` If the belief base contains the belief `need(food)`, then the test goal unifies with the belief base, and the substitution {`X = food`} will be performed on the plan, and the agent will continue by performing the action `buy(food)`.

2.3.3.3 Achievement Goals

Achievement goals are used to perform sub goals as a part of a plan. Suppose we have a plan to do shopping. However, to do shopping we first have to accomplish the goal of going to the supermarket. A plan for shopping could look like this:

```
+!shop :  true <- !go(supermarket); buy_stuff; !go(home).
```

This plan consists of two achievement goals and the action `buy_stuff`. Before performing the action `buy_stuff` the agent needs to achieve the goal of going to the supermarket. A new plan with the triggeringevent `+!go(supermarket)` or `+!go(X)` could be used to achieve this subgoal. When the subgoal is achieved, the plan continues. If the subgoal cannot be achieved, the achievement goal fails and then the whole plan fails.

2.3.3.4 Belief Changes

It is possible for an agent to change its belief base by using a belief change in the body of its plans. Belief changes starts with a `+` or a `-` followed by a ground atomic formula. A plus is used to denote the addition of a belief to the belief base and a minus is used to denote the deletion of a belief from the belief base. If the belief base is actually changed (it might not be if we try to add a belief that is already in the belief base or if we try to delete a belief that is not in the belief base) an event is created to handle the belief change.

2.3.3.5 Expressions

Prolog-like expressions can be used in the plan body, to make sure certain conditions hold before continuing with a plan. An expression can be used to evaluate numbers, check equality and the likes. It works just like the expressions used in contexts. If an expression does not evaluate to true, the plan fails. Examples of expressions are `X > Y` and `X*Y <= 4`

2.3.3.6 Internal Actions

Internal Actions work much like the actions described earlier from the perspective of an AgentSpeak program. The difference is that the actions defined earlier were external actions used to act on the environment whereas internal actions are used to access the architecture of the agent itself. For example, the agent could have a device attached for running an efficient Shortest-Path Algorithm that is not implemented within the AgentSpeak program. The internal action can be used to access this device. Internal actions can also be used to give the programmer information about the internal state of the agent. Internal actions can also be used for inter-agent communication, but the details of this will not be described in this report. Internal actions starts with a `"."` symbol followed by an atomic formula. We will not be focusing much on internal actions for the

rest of this report since the precise consequences of internal actions are generally unknown to the AgentSpeak programs which we want to verify.

2.4 Sample Program

In this section we present an example of a Jason agent program. A sample program consists of a number of beliefs, a number of initial goals (works like achievement goals) and a set of plans.

Our agent is a cooking agent that should be able to work in a (very) simple version of a restaurant kitchen. It should continue to check for new orders until it receives an order. When it receives an order, it gets the goal of handling that order. If the ordered item is actually on the menu, the agent will cook it and serve it to the customer who ordered it.

Inititally, the agent can only cook steak. If the agent gets the belief that it can cook other things than just steak, then it will add these things to the menu. The Jason code for the agent can be seen in Figure 2.2. In this program we have used plan labels (identified by "@"), which can be thought of as plan names.

```
/* Beliefs */
can_cook(steak).
on_menu(steak).

/* Initial Goals */
!check_for_new_orders.

/* Plans */
@p1
+!check_for_new_orders :  ordered(X,Y)
    <- !handle_order(X,Y) ; -ordered(X,Y) ; !check_for_new_orders.


@p2
+!check_for_new_orders :  true
    <- !check_for_new_orders.

@p3
+!handle_order(X,Y) : on_menu(Y)
    <- cook(Y) ; serve(X,Y).

@p4
+can_cook(X) : true
    <- +on_menu(X).
```

Figure 2.2: Jason Sample Program

Chapter 3

Formalising AgentSpeak

We introduce a formal syntax as well as an operational semantics of AgentSpeak in this chapter, which will be necessary when developing our verification system.

The version used is a simplified version of the one interpreted by Jason for the sake of simplicity. The simplifications are described later.

In addition we present our definitions of the modalities BELIEF, DESIRE and INTEND of an agent, since they (the last two modalities) are not implemented directly in AgentSpeak.

3.1 Syntax

The syntax of AgentSpeak used throughout the rest of the report is a simplified version of the one used in Jason called AgentSpeak(L) which is used a lot in the literature. However, this should not affect the theoretical limitations of our results, but makes the analysis quite a bit easier. We use the syntax described by the grammar in Table 3.1

In this grammar *ag* is an agent program consisting of a set of beliefs *bs* and a set of plans *ps* as described in last section. The metavariable *at* is used for atomic

ag	::=	$bs\ ps$	
at	::=	$P(t_1, ..., t_n)$	$(n \geq 0)$
bs	::=	$b_1 ... b_n$	$(n \geq 0)$
ps	::=	$p_1 ... p_n$	$(n \geq 1)$
p	::=	$te : ct \leftarrow h$	
te	::=	$+at \mid -at \mid +g \mid -g$	
ct	::=	$true \mid at_1 \& ... \& at_n$	$(n \geq 1)$
h	::=	$true \mid f_1; ... ; f_n$	$(n \geq 1)$
l	::=	$at \mid \neg at$	
f	::=	$A(t_1, ..., t_n) \mid g \mid u$	$(n \geq 0)$
g	::=	$!at \mid ?at$	
u	::=	$+b \mid -b$	

Table 3.1: Syntax of AgentSpeak

formulas. As earlier, these consist of a predicate P and a non-negative number of terms t_i as parameters. We only allow the terms to be variables or constants (no numbers, strings or structures). We also do not allow negations symbols. Each belief b is a ground atomic formula.

A plan p consists of a triggering event te, a context ct and a plan body h. Here, the triggering event must be an addition or deletion of a belief or goal. The context must be a conjunction of atomic formulas. The body is a semi-colon separated string of items that can be either actions, subgoals or an addition or deletion of beliefs.

We still use a notation where variables start with uppercase letters and constants, actions and predicates start with lowercase letters. We allow variables as terms in triggering events, contexts and plan bodys. However, when variables are used in plan bodys as achievement goals, actions or belief changes we require them to be instantiated before being executed. This is not the case with triggering events, contexts or test goals where unification will be used.

We do not allow expressions or internal actions as in Jason.

3.2 Semantics

In this section we introduce operational semantics of AgentSpeak to give a more formal description of the workings of the language. The semantics is taken from [11]. For readability we change the syntax a little (inspired by the syntax in [1]), but the meaning of the rules are still the same. The differences between [1]

and [11] will be discussed at the end of this section.

But before we start defining the semantic rules we take a closer look at an agents reasoning cycle to give a better overview:

3.2.1 Reasoning Cycle

We can use Figure 3.1 (which is borrowed from [11]) to illustrate the process of a reasoning cycle of an agent. This reasonining cycle is repeated for as long as the agent program is running.

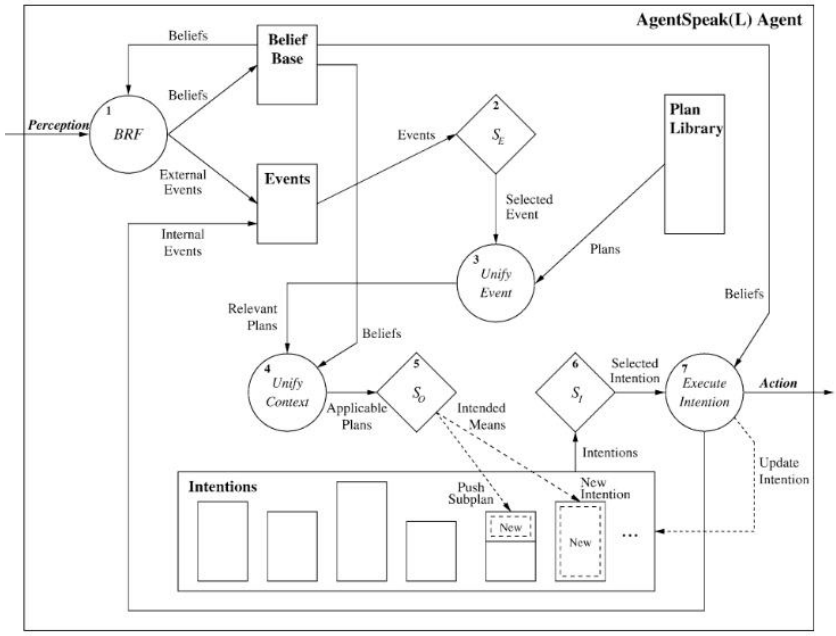

Figure 3.1: An AgentSpeak reasoning cycle

The reasoning cycle starts with the agent receiving percepts from the environment. These percepts are passed through a belief revision function (BRF) which updates the belief base according to the percepts and creates new events if beliefs are changed. The belief revision function, however is not a part of the AgentSpeak language, but is assumed to be done by the architecture of the agent.

The next step in the reasoning cycle is event selection. In each cycle an event is

selected for the agent to deal with (as long as there is an event to deal with). It is the selection function S_ε that decides which event is selected first. In Jason this function can be customised. However in our analysis we assume that this function is non-deterministic. This is done so our program verification will be applicable to as wide a range of programs as possible.

When an event has been selected, the triggering part of the event is attempted unified with the triggering events of the plans in the agents plan library. This is done to create the set of relevant plans for handling the event. Hereafter we take the set of relevant plans and select from this the set of applicable plans. This set consists of the relevant plans which have a context that is a logical consequence of the belief base. These are the final candidates for handling the selected event. Finally, a selection function S_O chooses which one of the applicable plans should be used to handle the event. As the event selection function this is also customisable in Jason and we assume that it is non-deterministic from now on. As shown in Figure 3.1 either a new intention with the selected plan will be added to the set of intentions in case of an external event being handled or the plan will be put on top of a suspended intention in case of an internal event. Suspended intentions will be explained later.

We note that if no relevant or applicable plans is found for a given event, the event is simply discarded.

After selecting plans to handle events, the agent moves on to acting on its current intentions. It starts by using an intention selection function S_I (yes, it is customisable in Jason and yes, we assume it is non-deterministic like the other two selection functions) for choosing an intention to act upon. The agent now executes the first part of the plan on top of this intention. It could be the execution of an action on the environment, a belief addition or deletion, a test goal to be checked or an achievement goal to accomplish. In the case of an achievement goal, a new event is created with the achievement goal as triggering event and with the whole intention on the side. (Events are on the form $\langle te, i \rangle$ where te is a triggering event and i is an intention. More details about this will follow in the next section). The intention is temporarily removed from the set of intentions and is thus suspended until the achievement goal has been accomplished. It is in this case that a plan for handling the achievement goal can be put on top of an existing intention.

3.2.2 Circumstance of an agent

We now move to the definition of an agent circumstance, which can describe the state of an agent at any point of program execution.

The circumstance of an agent C is a tuple $\langle I, E, A, R, Ap, \iota, \rho, \varepsilon \rangle$ where each element consists of the following:

- I is a set of intesions. An intention is a stack of partially instantiated plans.
- E is a set of events. Each event $e = (te, i)$ is a pair consisting of a triggering event te and an intention i. The plan on top of intention i is the one that generated the triggering event te in the case of an internal event. In the case of an external event, $te = \top$ where $\top$ is the empty intention.

  It should be noted that it is not the AgentSpeak interpreter that adds the external events to this set, but rather the architecture of the agent (including the belief revision function) that adds external events to this set.
- A is the set of actions that the agent has chosen to execute. When an action is added to this set, it is the agents way of telling its architecture to execute the action.
- R is a set of relevant plans for an event the agent has chosen to handle. It is empty at the beginning of each reasoning cycle.
- Ap is a set of applicaple plans for an event the agent has chosen to handle. It is empty at the beginning of each reasoning cycle.
- ι is a particular intention the agent has chosen for processing. It is empty at the beginning of each reasoning cycle.
- ρ is a particular plan the agent has chosen for handling an event. It is empty at the beginning of each reasoning cycle.
- ε is a particular event the agent has chosen to handle. It is empty at the beginning of each reasoning cycle.

As in [11] we use the following notation:

- C_E is used to denote the E component of a circumstance C. The same notation is used for the other components.
- We write $C_\iota = \_$ when the agent does not consider any intentions in a given reasoning cycle. Likewise for C_ρ and C_ε.
- Intentions are denoted by $i, i', ...$ and $i[p]$ is the notation for an intention i with plan p on top.

Before moving on to presenting transition rules of the AgentSpeak system, we need to introduce some auxiliary functions.

3.2.3 Auxiliary Functions

For a given plan $p = te : ct < -h$ we define the functions $\text{TrEv}(p) = te$ and $\text{Ctxt}(p) = ct$ to make notation simpler in the rest of this chapter.

Relevant Plans
Given an agent with plans ps and a triggering event te we define the set of relevant plans $\text{RelPlans}(ps, te)$ as

$$\text{RelPlans}(ps, te) = \{p\theta \mid p \in ps \wedge \theta = mgu(te, \text{TrEv}(p))\}$$

The set of relevant plans consists of all plans which has a triggering event that can be unified with te. The mgu function is the most general unifying substitution of two expressions as defined in standard first-order logic.

Applicable Plans
Given a set of relevant plans R and an agent with the beliefs bs we define the set of applicable plans $\text{AppPlans}(bs, R)$ as

$$\text{AppPlans}(bs, R) = \{p\theta \mid p \in R \wedge \theta \text{ is s.t. } bs \models Ctxt(p)\theta\}$$

The set of applicable plans consists all of the relevant plans for a given event for which the context is a logical consequence of the belief base.

Test Goals
Given a formula at and a set of beliefs bs, we define the function $\text{Test}(bs, at)$ as follows:

$$\text{Test}(bs, at) = \{\theta \mid bs \models at\theta\}$$

This function is used for test goals where one wants to test formulas against the current belief base of an agent. If this function returns the empty set the formula is not a logical consequence of the belief base, otherwise it is.

3.2.4 Inference Rules

We are now ready to formalize the process of a reasoning cycle.This is done with an operational semantics that uses inference rules to describe legal transitions of an AgentSpeak configuration $\langle ag, C, s \rangle$ where ag is an agent program with plans ps and beliefs bs and C is a circumstance tuple as described earlier.

The element s is the current state of the reasoning cycle. It can have the values SelEv, RelPl, ApplPl, SelAppl, AddIM, SelInt, ExecInt and ClearUp which stand for selecting an event, selecting relevant plans, selecting applicable plans, adding intended means, selecting intentions, executing intentions and clearing up, respectively. This item is the biggest difference in notation from the operational semantics used in [11]. In [1] where this notation is also used, there is an additional possible state called ProcMsg which is used for processing messages when communicating with other agents. For simplicity we do not cover this possibility, but it is a possible extension of the system developed in this report (though that form of communication is not a part of the AgentSpeak language).

The process of a reasoning cycle is illustrated in Figure 3.2.

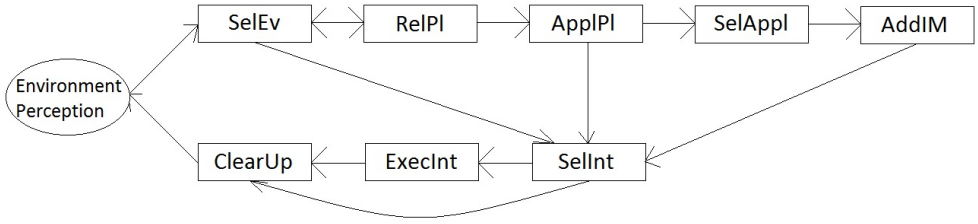

Figure 3.2: Steps in an AgentSpeak reasoning cycle

This figure shows that each reasoning cycle starts with the agent getting perceptions from the environment. The steps in the top row is concerned with handling new events whereas the bottom row is concerned with acting on current intentions. We now move to the inference rules of the system which specifies the legal transitions between the steps of the reasoning cycle and the consequences they have.

We start with the two event selection rules:

SelEv$_1$:

$$\frac{S_\varepsilon(C_E) = \langle te, i\rangle}{\langle ag, C, \text{SelEv}\rangle \rightarrow \langle ag, C', \text{RelPl}\rangle}$$

where

$$C'_E = C_E \setminus \{\langle te, i\rangle\}$$

$$C'_\varepsilon = \langle te, i\rangle$$

SelEv$_2$:

$$\frac{C_E = \{\}}{\langle ag, C, \text{SelEv}\rangle \rightarrow \langle ag, C, \text{SelInt}\rangle}$$

The first rule is used when there is at least one event in the set C_E to handle. Here, the event is removed from the set of events and added to the C_ε component. The reasoning cycle continues with selection of relevant plans for the event. In the case that there are no events to handle, the reasoning cycle skips to the intention selection.

Now, we move on to the rules for finding relevant plans for a selected event:

RelPl$_1$:

$$\frac{C_\varepsilon = \langle te, i\rangle \qquad \text{RelPlans}(ag_{ps}, te) \neq \{\}}{\langle ag, C, \text{RelPl}\rangle \rightarrow \langle ag, C', \text{ApplPl}\rangle}$$

where

$$C'_R = \text{RelPlans}(ag_{ps}, te)$$

RelPl$_2$:

$$\frac{C_\varepsilon = \langle te, i\rangle \qquad \text{RelPlans}(ag_{ps}, te) = \{\}}{\langle ag, C, \text{RelPl}\rangle \rightarrow \langle ag, C', \text{SelEv}\rangle}$$

The first rule is used when there is at least one relevant plan for the selected event. Here, the set of relevant plans for the event is added to the C_R component. If there are no relevant plans, the event is just abandoned (it was already

removed from the set of events in the former step) and a new event can be chosen.

Next, we consider rules for finding applicable plans:

ApplPl$_1$:

$$\frac{\text{AppPlans}(ag_{bs}, C_R) \neq \{\}}{\langle ag, C, \text{ApplPl}\rangle \rightarrow \langle ag, C', \text{SelAppl}\rangle}$$

where

$$C'_{Ap} = \text{AppPlans}(ag_{ps}, C_R)$$

ApplPl$_2$:

$$\frac{\text{AppPlans}(ag_{bs}, C_R) = \{\}}{\langle ag, C, \text{ApplPl}\rangle \rightarrow \langle ag, C, \text{SelInt}\rangle}$$

If there are applicable plans for the selected event, we save these in the C_{Ap} component and move on to selecting one of the applicable plans to commit to. If there are no applicable plans, we move on to selecting an intention to act upon.

We now consider the rule for selecting an applicable plan:

SelAppl :

$$\frac{S_O(C_{Ap}) = p}{\langle ag, C, \text{SelAppl}\rangle \rightarrow \langle ag, C', \text{AddIM}\rangle}$$

where

$$C'_\rho = p$$

This rule uses the non-deterministic selection function S_O for selecting one of the applicable plans to commit to.

The next step is the addition of the selected plan to the set of intentions. We have one rule for handling external events and one for handling internal events. It is easy to tell the difference between these two types of events, since the external events have empty intentions and internal events does not. We use the symbol $\top$ to denote the empty intention:

ExtEv :

$$\frac{C_\varepsilon = \langle te, \top \rangle \qquad C_\rho = p}{\langle ag, C, \text{AddIM} \rangle \rightarrow \langle ag, C', \text{SelInt} \rangle}$$

where

$$C'_I = C_I \cup \{[p]\}$$

IntEv :

$$\frac{C_\varepsilon = \langle te, i \rangle \qquad C_\rho = p}{\langle ag, C, \text{AddIM} \rangle \rightarrow \langle ag, C', \text{SelInt} \rangle}$$

where

$$C'_I = C_I \cup \{i[p]\}$$

As explained earlier, a new intention is created in the case of an external event. In the case of an internal event, the selected plan will be put on top of the suspended intention that follows with the event.

We now move to the part of the reasoning cycle that has to do with acting on current intentions. We start by selecting an intention to act upon (if there are any):

SelInt$_1$:

$$\frac{C_I \neq \{\} \qquad S_I(C_I) = i}{\langle ag, C, \text{SelInt} \rangle \rightarrow \langle ag, C', \text{ExecInt} \rangle}$$

where

$$C'_\iota = i$$

SelInt$_2$:

$$\frac{C_I = \{\}}{\langle ag, C, \text{SelInt}\rangle \rightarrow \langle ag, C', \text{ClearUp}\rangle}$$

If there are no intentions, we go back to handling new events. If there are intentions, the non-deterministic selection function S_I chooses the intention to act upon.

We now move on to describing rules for executing intentions. There are five types of formulas that we need to be able to handle. There are actions, achievement goals, test goals, belief additions and belief deletions. Each of these types of formulas have their own rule. Test goals have two rules depending on whether the test goal is a logical consequence of the agents beliefs or not.

First we present the action rule:

Action :

$$\frac{C_\iota = i[head \leftarrow a; h]}{\langle ag, C, \text{ExecInt}\rangle \rightarrow \langle ag, C', \text{ClearUp}\rangle}$$

where

$$C'_A = C_A \cup \{a\}$$

$$C'_I = (C_I \setminus \{C_\iota\}) \cup \{i[head \leftarrow h]\}$$

Thus, when the selected intention has a plan on top where the first formula of the plan body is an action, the action is added to the set of actions to be performed by the agent and the action is removed from the plan body.

AchvGl :

$$\frac{C_\iota = i[head \leftarrow !at; h]}{\langle ag, C, \text{ExecInt}\rangle \rightarrow \langle ag, C', \text{ClearUp}\rangle}$$

where

$$C'_E = C_E \cup \{\langle +!at, C_\iota \rangle\}$$

$$C'_I = C_I \setminus \{C_\iota\}$$

When the formula is an achievement goal, a new event is created with the addition of the achievement goal as triggering event and with the intention on the side. The intention is temporarily removed from the set of intentions the agent has chosen to act upon. This is what we call a suspended intention, since the intention is suspended until the achievement goal has been accomplished.

TestGl$_1$:

$$\frac{C_\iota = i[head \leftarrow ?at; h] \qquad \text{Test}(ag_{bs}, at) \neq \{\}}{\langle ag, C, \text{ExecInt} \rangle \rightarrow \langle ag, C', \text{ClearUp} \rangle}$$

where

$$\theta \in \text{Test}(ag_{bs}, at)$$

$$C'_I = (C_I \setminus \{C_\iota\}) \cup \{i[(head \leftarrow h)\theta]\}$$

TestGl$_2$:

$$\frac{C_\iota = i[head \leftarrow ?at; h] \qquad \text{Test}(ag_{bs}, at) = \{\}}{\langle ag, C, \text{ExecInt} \rangle \rightarrow \langle ag, C', \text{ClearUp} \rangle}$$

where

$$C'_I = C_I \setminus \{C_\iota\}$$

When the formula to be handled is a test goal, there are two cases. One, if there is a substitution that makes the test goal follow from the beliefs and one if there is no such substitution.

If there is a substitution, it is performed on the rest of the plan that is on top of the selected intention. If there is no such substitution the intention is

discarded. This is one of the differences between AgentSpeak(L) and the version of AgentSpeak interpreted by Jason. In Jason an internal event will be created to handle the situation whereas, in this simpler version, the intention is simply thrown away.

The next type of formula to consider is belief additions and belief deletions:

AddBel :

$$\frac{C_\iota = i[head \leftarrow +b; h]}{\langle ag, C, \text{ExecInt}\rangle \rightarrow \langle ag', C', \text{ClearUp}\rangle}$$

where

$$ag'_{bs} \models b$$

$$C'_E = \begin{cases} C_E \cup \{\langle +b, \top\rangle\} & \text{if } ag_{bs} \not\models b \\ C_E & \text{if } ag_{bs} \models b \end{cases}$$

$$C'_I = (C_I \setminus \{C_\iota\}) \cup \{i[head \leftarrow h]\}$$

DelBel :

$$\frac{C_\iota = i[head \leftarrow -b; h]}{\langle ag, C, \text{ExecInt}\rangle \rightarrow \langle ag', C', \text{ClearUp}\rangle}$$

where

$$ag'_{bs} \not\models b$$

$$C'_E = \begin{cases} C_E \cup \{\langle -b, \top\rangle\} & \text{if } ag_{bs} \models b \\ C_E & \text{if } ag_{bs} \not\models b \end{cases}$$

$$C'_I = (C_I \setminus \{C_\iota\}) \cup \{i[head \leftarrow h]\}$$

These two formulas are used to update the belief base of the agent internally (which means that the beliefs are not acquired from percepts, but follows from

plan executions). When the belief base is actually changed, an event is created with an empty intention to handle the change of the agents beliefs. The formula ($+b$ or $-b$) is then removed from the plan being executed.

We now move to the last rule, which is used for "cleaning up" before the next reasoning cycle:

ClearUp :

$$\frac{}{\langle ag, C, \text{ClearUp}\rangle \rightarrow \langle ag, C', \text{EnvPerc}\rangle}$$

where

$$C'_I = \begin{cases} C_I \setminus \{C_\iota\} & \text{if } C_\iota = [head \leftarrow] \\ (C_I \setminus \{C_\iota\}) \cup \\ \{i'[head \leftarrow h]\} & \text{if } C_\iota = i'[head \leftarrow !at; h][head \leftarrow] \\ C_I & \text{otherwise} \end{cases}$$

$$C'_R = C'_{Ap} = \{\}$$

$$C'_\iota = C'_p = C'_\varepsilon = \_$$

If the selected intention is empty, it is removed from the set of intentions. If the body of the plan on top of the intention is empty, the plan is removed from the top of this intention (this should only happen when we have accomplished an achievement goal). In all other cases, no change of the intention is needed.

In addition to handling empty plan bodys and empty intentions, this final rule clears the temporary elements R, Ap, ι, p and ε of the circumstance making the state of the system ready for the next reasoning cycle.

This ends the section of inference rules in AgentSpeak. It seems appropriate to sum up the subtle differences between the rules of this language and the language interpreted by Jason:

- AgentSpeak do not support communication between agents (this would happen in the start of the reasoning cycle along with other perceptions of the environment).

- In Jason, an event is discarded if there exist relevant plans but no applicable plan for the event. This is not the case in AgentSpeak. Here, the event can be chosen again in another reasoning cycle with the possibility that the context of one of the relevant plans will then be true according to the belief base. A reason for this choice in Jason might be that if developers implement a static event selection function, the same event could be chosen in each reasoning cycle even though there will not be any applicable plans for it. Thus it will be "blocking" other events.
- AgentSpeak do not handle failure of test goals in the same way that Jason does. In AgentSpeak the intention is thrown away, whereas in Jason an event is created to handle the failure of the plan. Not dealing with plan failure certainly simplifies AgentSpeak in a number of situations which will make our further analysis easier.

The first and last point of this list are possible extensions of the work done in this report, but both will likely complicate things quite a bit. The second point seems like a matter of taste, but since we adopted the rules of AgentSpeak in the other cases we do it in this case also since the difference does not seem too important.

3.2.5 BDI Modalities

To be able to verify Belief-Desire-Intention properties of AgentSpeak systems we need to define these formally since they are not all implemented clearly in AgentSpeak. We define each of the three modalities BEL, DESIRE and INTEND in the same way as is done in [11]. In that paper some relations between the three modalities which hold generally in AgentSpeak programs are proved. It seems like an advantage to work with the same definitions as they do, both since they seem natural and since their results could possibly be combined with the results of the system developed in this report.

We define the three modalities with respect to an agent $ag = \langle bs, ps \rangle$ and a circumstance C. We start with the belief modality, which is the easiest since it is directly implemented in AgentSpeak:

$$\text{BEL}_{\langle ag, C \rangle}(\varphi) \equiv bs \models \varphi$$

Thus an agent believes a formula φ if and only if the belief base entails φ which seems very straightforward.

Before defining the intention modality, we need to introduce an auxiliary function *agoals* that takes an intention as input and returns all achivement goals in the triggering events of the plans of the intention. The function is defined inductively on the number of plans of an intention (where $\top$ is the empty intention):

$$\begin{array}{l} agoals(\top) = \{\} \\ agoals(i[p]) = \begin{cases} \{at\} \cup agoals(i) & \text{if } p = +!at : ct \leftarrow h \\ agoals(i) & \text{otherwise} \end{cases} \end{array}$$

Thus, all plans which do not have achievement goals as triggering events will be ignored, and all achievement goals working as triggering events will be added to the set $agoals(i)$.

We are now ready to define the INTEND modality:

$$\text{INTEND}_{\langle ag,C\rangle}(\varphi) \equiv \varphi \in \bigcup_{i \in C_I} agoals(i) \vee \varphi \in \bigcup_{\langle te,i\rangle \in C_E} agoals(i)$$

This means that an agent intends φ if and only if φ is an achievement goal in the triggering event of a plan in one of the intentions of C_I or in the intention part of a suspended intention.

The reason that we are only interested in achievement goals is that they represent the goals and subgoals that the agent is trying to accomplish. Other triggering events like test goals and belief changes do not say anything about the states of the world that the agent is trying to bring about. The definition seems quite natural since an agent intends precisely the achievement goals that the agent has committed to accomplish in earlier reasoning cycles (but has not yet accomplished).

The third modality, DESIRE, is defined as follows:

$$\text{DESIRE}_{\langle ag,C\rangle}(\varphi) \equiv \text{INTEND}_{\langle ag,C\rangle}(\varphi) \vee \langle +!\varphi, i\rangle \in C_E$$

Thus, an agent desires a formula φ if and only if it intends the formula or there is an event which has $+!\varphi$ as triggering event. In this way, the agent desires all the formulas it intends, but in addition it desires accomplishment goals which it has not yet committed to pursue. Further discussion of these definitions of

the BELIEF, DESIRE and INTEND modalities for AgentSpeak programs can be found in [1] and [11].

3.3 Reasoning Cycle Example

In this section we will walk through an example of a reasoning cycle of an AgentSpeak program. We the sample Jason program from Figure 2.2 which is also a valid AgentSpeak program.

We haven't discussed initial goals in the context of AgentSpeak. All initial goals will be added to the set of events with empty intentions to the initial configuration of the system. This means that the initial configuration of our AgentSpeak program is

$$\mathit{Config}_0 = \langle ag_0, C_0, \mathrm{SelEv} \rangle$$

where

$$ag_{0,bs} = \{\texttt{can\_cook(steak)}, \texttt{on\_menu(steak)}\}$$

$$C_0 = \langle \{\}, E_0, \{\}, \{\}, \{\}, \_, \_, \_ \rangle$$

$$E_0 = \{\langle \texttt{+!check\_for\_new\_orders}, \top \rangle\}$$

Thus, initially most of the circumstance is empty, except for the single initial event. For readability we refer to the triggering event `+!check_for_new_orders` by t_0 from now on.

The only applicable rule is now $\mathbf{SelEv}_1$ since we are in the state SelEv and the set of events is not empty. This leads us to the next configuration:

$$\mathit{Config}_1 = \langle ag_0, C_1, \mathrm{RelPl} \rangle$$

where

$$C_1 = \langle \{\}, \{\}, \{\}, \{\}, \{\}, \_, \_, \langle t_1, \top, \rangle \rangle$$

The only applicable rule is now $\mathbf{RelPl}_1$ since the state is RelPl and there are two relevant plans, namely the plans labeled by `p1` and `p2`.

The next configuration then becomes

$$\mathit{Config}_2 = \langle ag_0, C_2, \text{ApplPl} \rangle$$

where

$$C_2 = \langle \{\}, \{\}, \{\}, \{\texttt{p1}, \texttt{p2}\}, \{\}, \_, \_, \langle t_1, \top, \rangle \rangle$$

The only rule we can use now is $\mathbf{AppPlans}_1$ since the state is ApplPl and `p2` is applicable. (

Thus, the next configuration is

$$\mathit{Config}_3 = \langle ag_0, C_3, \text{SelAppl} \rangle$$

where

$$C_3 = \langle \{\}, \{\}, \{\}, \{\texttt{p1}, \texttt{p2}\}, \{p2\}, \_, \_, \langle t_1, \top, \rangle \rangle$$

Now, the only applicable rule is SelAppl which will choose the only applicable plan we have, giving us:

$$\mathit{Config}_4 = \langle ag_0, C_4, \text{AddIM} \rangle$$

where

$$C_4 = \langle \{\}, \{\}, \{\}, \{\texttt{p1}, \texttt{p2}\}, \{p2\}, \_, \texttt{p2}, \langle t_1, \top, \rangle \rangle$$

Now, the selected plan will be added to the set of intentions according to rule **ExtEv**:

$$Config_5 = \langle ag_0, C_5, \text{SelInt} \rangle$$

where

$$C_5 = \langle \{[\texttt{p2}]\}, \{\}, \{\}, \{\texttt{p1}, \texttt{p2}\}, \{p2\}, \_, \texttt{p2}, \langle t_1, \top, \rangle \rangle$$

Now rule **SelInt**$_1$ will be applied by selecting our only intention to act on:

$$Config_6 = \langle ag_0, C_6, \text{ExecInt} \rangle$$

where

$$C_6 = \langle \{[\texttt{p2}]\}, \{\}, \{\}, \{\texttt{p1}, \texttt{p2}\}, \{p2\}, [\texttt{p2}], \texttt{p2}, \langle t_1, \top, \rangle \rangle$$

Since our selected intention has a plan on top (it consists of only one plan) that has an achievement goal as its first formula, the next configuration is obtained by using rule **AchvGl** giving us:

$$Config_7 = \langle ag_0, C_7, \text{ClearUp} \rangle$$

where

$$C_7 = \langle \{\}, \{\langle t_0, [\texttt{p2}] \rangle\}, \{\}, \{\texttt{p1}, \texttt{p2}\}, \{p2\}, [\texttt{p2}], \texttt{p2}, \langle t_1, \top, \rangle \rangle$$

Thus, the set of intentions is now empty and we have a new event with the old intention on the side.

Finally, we use the **ClearUp** rule to clear all the temporary information, giving us:

$$Config_8 = \langle ag_0, C_8, \text{EnvPerc} \rangle$$

where

$$C_8 = \langle \{\}, \{\langle t_0, [\texttt{p2}]\rangle\}, \{\}, \{\}, \{\}, \_, \_, \_\rangle$$

This ends the reasoning cycle.

3.4 Belief-Desire-Intend Example

We continue with an example of using the BEL, DESIRE and INTEND modalities introduced earlier by looking at what the agent believes, desires and intends in the different configurations we have just been through.

In this reasoning cycle, the belief base did not change, which means that we have

$$\text{BEL}_{\langle ag,C\rangle}(\texttt{can\_cook(steak)}) = \text{BEL}_{\langle ag,C\rangle}(\texttt{on\_menu(steak)}) = true$$

and

$$\text{BEL}_{\langle ag,C\rangle}(\varphi) = false$$

for all other atomic formula φ in all states of our system.

In the beginning of our reasoning cycle we have

$$\text{DESIRE}_{\langle ag,C\rangle}(\texttt{check\_for\_new\_orders}) = true$$

since E contains the event $\langle \texttt{+!check\_for\_new\_orders}, \top\rangle$.

We have that

$$\text{INTEND}_{\langle ag,C\rangle}(\texttt{check\_for\_new\_orders}) = true$$

in all states after we have added an intention to the set I. Even when we remove our intention from the set I and create the new event $\langle \texttt{+!check\_for\_new\_orders}, [\texttt{p2}]\rangle$ this holds, since the achievement goals is a triggering event in the single plan of the intention [p2].

The agent does not intend or desire anything else during the reasoning cycle we just described.

3.5 Jason vs AgentSpeak

We will round this chapter off with a discussion of the differences between the language interpreted by Jason and AgentSpeak which is used in this report. Some major Jason features which we do not support in our version of AgentSpeak are:

- AgentSpeak do not support rules. This restriction is purely for simplicity and is an obvious extension possibility.
- We do not support communication between agents. It is a possible extension of our system, but would probably require quite a bit of work. In [1] they actually introduce a formal semantics for the use of messages for inter-agent communication which could be used for this purpose. This would also give the possibility of verifying systems of multiple agents, whereas this report focuses on the verification of a single agent.
- Jason gives more possibilities of dealing with plan failure than we do in AgentSpeak. As we stated with the inference rule $\mathbf{TestGl}_2$, intentions are simply removed if a test goal cannot be unified with the belief base. This should not be too difficult to change, but since we have followed the AgentSpeak rules from [11] which are also used in a number of other papers on the subject we have chosen to go with their choice in this case too.
- Jason allows variables to be bound to predicates where we only allow variables to be bound to constants. We do not allow this use of higher-order variables
- Jason support all the terms presented in Figure 2.1 whereas we only use constants and variables for simplicity. Jason also allows expressions as defined earlier. This is not a part of AgentSpeak either.

AgentSpeak is clearly simpler than the language interpreted by Jason. However, most of the theory used in this report could probably be extended to deal with these extensions.

Chapter 4

Transition System Generation

In this section we will present procedures for generating abstract models of AgentSpeak programs. Model checking techniques will later be applied to these abstract models to verify properties of the corresponding AgentSpeak programs.

4.1 Transition Systems

To apply model checking techniques to verify properties of an AgentSpeak program we need to create a finite transition system representing our program.

More formally, our model checking algorithm (which will be presented later) will need as input a transition system on the form $T = \langle S, R, L \rangle$ where S is the set of possible states of the system, $R \subseteq S \times S$ is a transition relation specifying the legal transitions between states and $L : S \times AP \rightarrow \{true, false\}$ is a labeling function that specifies whether a given formula $f \in AP$ is true or false in a given state. AP is the set of formulas on the form BEL(p), DESIRE(p) or INTEND(p) where p is a ground atomic formula. In addition R is required to be total, which means that every state will have at least one successor state.

Our job in this chapter is to systematically take an AgentSpeak program and create a transition system representing the program properly.

To do this we will be using the operational semantics of AgentSpeak. We let each state in our transition system represent a configuration $\langle ag, C, s\rangle$ of the program which is defined in the same way as in the last chapter. The inference rules of the operational semantics will be used to generate successor states and to build the transition relation R of the system. The details of this procedure will be presented in a later section.

For each state $s \in S$ that represents a configuration $\langle ag, C, s\rangle$ it will be easy to create a labeling function L using the definition of the BEL, DESIRE and INTEND modalities from last chapter such that the abstract model tells us which beliefs, desires and intentions the agent has in the different states of the system.

4.2 Finite Systems

A problem with generating transition systems from our programs is that an AgentSpeak program can easily have an infinite number of states. This is not acceptable if we are to perform model checking on the corresponding transition systems. For example, the size of the belief base of an agent is not bounded.

In our system we have chosen an approach where we specify a number of bounds before doing verification. This includes:

- B_{max} the maximum number of belief atoms allowed in the belief base at a time
- I_{max} the maximum number of intentions allowed at a time.
- P_{max} the maximum number of plans in an intention.
- E_{max} the maximum number of events.

These bounds make sure that there will only be a finite number of different states in our transition system. This can be realised by looking at the process of a reasoning cycle. After doing environment perception all the temporary elements of an agents circumstance (Re, Ap, ι, ρ and ε) will all be cleared. Also, the set of actions A will be empty since we assume that any actions added to this set during a reasoning cycle will be removed when doing environment perception, since the action will be sent to the architecture of the agent to be processed. This leaves us with the set of intentions I, the set of events E and the belief base.

The different possible entries in the belief base can be assumed to be finite for now. How we deal with this is explained later. This means that the possible number of different events we can receive from the environment is also finite, since we can only receive belief changes from the environment. Thus, the number of different possible events receivable from the environment is finite.

We have a bound on the maximum number of plans in an intention. Since each of these plans must be a subplan of the finite number of plans of our agent and the variables of these plans can only be instantiated with the finite number of possible beliefs, there is a finite number of different intentions. This also means that there is a finite number of different events, since events either comes externally from the environment or internally with a triggering event and an intention on the side. Since there is a maximum number of intentions and a maximum number of events at a time, there is only a finite number of possibilities for the sets E and I.

Thus, we have shown that if there is a finite number of possible beliefs, then after the environment perception state, we will have a finite number of different configurations for an agent.

If there is a finite number of different configurations after the environment perception state, then the number of different configurations in total will also be finite since there clealy is only a finite number ways that all the temporary components R, Ap, ι, ρ and ε can be instantiated given our assumptions.

The maximum number of belief atoms allowed in the belief base at a time B_{max} is strictly speaking not necessary for the belief base to have a finite number of different configurations since there can not be two equal belief atoms in the belief base at a time whereas there can easily be two equal events in the set E. However, it will be very computationally expensive to generate systems where B_{max} is large since it will make the state space very large without necessarily making the verification of the system better.

There are a number of important things to think about when specifying the bounds. It is important to set them high enough that the transition system actually gives a good representation of the program. However, if they are set too high, it will take a very long time to verify system properties since the state space will increase very quickly.

For example, when verifying properties of a program like the sample program in chapter 2 it will not be necessary to have a belief base size of 30, since it will not change the dynamics of the system. However, setting it to 1 will be a problem, since it will not allow us to verify properties in all relevant states of the system.

4.3 Environment

The environment type used in our program is non-deterministic. This means that the properties we can verify are properties that should hold no matter what could possibly happen to the agent. The challenge is that this implies an infinite number of possible perceptions from the environment. For example our cooking agent from chapter 2 could possibly receive percepts like `elvis_is_alive` and `cars_are_cool` even though one would expect the agent to receive percepts like `can_cook(chicken)` or `ordered(mr_johnson, steak)`.

To deal with this problem we first assume that if the agent receives a belief change from the environment with a predicate that does not occur in his initial belief base or in any of his plans then the belief change, it will simply be discarded since it will not have any effect on how the agent will act or deliberate. In fact, we also discard the belief change if the predicate occurs somewhere in the agent program, but only with a different arity. For example the belief addition `ordered(steak)` will be discarded since it is only used as a binary predicate in our agent program and thus the agent will not act upon the event or use the information.

By using this abstraction there will only be a finite number of predicates in the belief changes we receive from the environment. However, the terms can still take an infinite number of different values. We work our way around this by using generic constants which are constants that could stand for anything. Or, rather it could stand for any constant that does not already occur in the agent program. This is because constants that occur in a plan, the belief base or the circumstance should possibly be handled differently from other constants.

For example, consider the simple example program in Figure 4.1 consisting of one initial beliefs and two plans.

```
/* Beliefs */
like(red).

/* Plans */
+house_is(X) : like(X) <- act_happy.
+house_is(X) : true <- act_sad.
```

Figure 4.1: AgentSpeak example

In our program our agent can only perceive one thing from the environment each reasoning cycle. This means that in the sample program in Figure 4.1

the first time we perceive the environment the different possible events we can receive are:

- $E = \{\}$
- $E = \{\langle$`+like(_G0)`$, \top\rangle\}$
- $E = \{\langle$`+house_is(red)`$, \top\rangle\}$
- $E = \{\langle$`+house_is(_G0)`$, \top\rangle\}$
- $E = \{\langle$`-like(red)`$, \top\rangle\}$

The generic constants have the form `_Gi`. Note that `like(red)` is not added, since it is already in the belief base which means that it is possible to receive a belief deletion from the environment.

Thus, we take each predicate that occurs in a configuration and use them as belief additions with all the possible combinations of constants as terms. Also, for each belief in the belief base a belief deletion can be perceived. More formally the set of possible events that can be perceived in a reasoning cycle is defined by:

$$E_{pos} = \left(\bigcup_{b \in ag_{bs}} \langle -b, \top \rangle \right) \cup \left(\bigcup_{p \in P} \bigcup_{k_1,...,k_{a(p)} \in K_{a(p)}} \{\langle +p(k_1, ..., k_{a(p_i)}), \top \rangle\} \right)$$

where P is the set of predicates occuring in the configuration of the agent and K_i is the set of containing all constants occuring in the configuration of an agent and i new generic constants. We use $a(p)$ to denote the arity of the predicate p.

For example if we have an agent where the only predicate occuring in its configuration is `like(steen, dogs)` which is a belief and there are no other constants in the configuration of the agent, then

$$
\begin{aligned}
E_{pos} = \{ \quad & \langle \texttt{-like(steen, dogs)}, && \langle \texttt{+like(steen, steen)}, \top \rangle, \\
& \langle \texttt{+like(steen, \_C0)}, && \langle \texttt{+like(steen, \_C1)}, \top \rangle, \\
& \langle \texttt{+like(dogs, steen)}, && \langle \texttt{+like(dogs, dogs)}, \top \rangle, \\
& \langle \texttt{+like(dogs, \_C0)}, && \langle \texttt{+like(dogs, \_C1)}, \top \rangle, \\
& \langle \texttt{+like(\_C0, steen)}, && \langle \texttt{+like(\_C0, dogs)}, \top \rangle, \\
& \langle \texttt{+like(\_C0, \_C0)}, && \langle \texttt{+like(\_C0, \_C1)}, \top \rangle, \\
& \langle \texttt{+like(\_C1, steen)}, && \langle \texttt{+like(\_C1, dogs)}, \top \rangle, \\
& \langle \texttt{+like(\_C1, \_C0)}, && \langle \texttt{+like(\_C1, \_C1)}, \top \rangle \}
\end{aligned}
$$

This use of generic constants gives us the possibility of dealing with all possible environment percepts while we still keep a finite number of states in our transition systems.

4.4 System Generation

We are now ready to discuss the algorithm used to generate our transition systems. The algorithm can be seen in Figure 1.

The algorithm takes an AgentSpeak configuration as input (which can easily be obtained from an AgentSpeak program) and adds it to the set of unprocessed states U. The set of states S of the transition system is initially empty and the so is the transition relation T.

The algorithm now picks the unprocessed states one at a time. If there is already a processed state $t \in S$ with the same configuration, the new state s will not be added to the set of states. Instead, the transition relation will be altered so that all predecessors of s will be predecessors of t instead. This is done so we never have two states in S with the same configuration, which would lead to an infinite number of generated states.

If there is no processed state in S with the same configuration we will first check if the new state is legal. A state is considered legal if its configuration satisfies the four bounds specified in the section about finite systems. If the state is legal it will be added to S and for all possible successors of its configuration a new state is added to the set of unprocessed states. The possible successors of a state are determined by the operational semantics of an AgentSpeak configuration. In the way the semantic rules are defined, there will always be a finite number of successors. If the configuration state is EnvPerc there will also be a finite number of successors obtained in the way described in the last section about non-deterministic environments.

Algorithm 1 TransSystemGeneration(Configuration:$conf$)

```
s0 = new State(conf);

List U = {s0}; //The set of unprocessed states

S = {}; // The set of states in the transition system
T = {}; // The transition relation of the transition system

while U ≠ {} do
  State s = removeFirst(U);
  if ∃t ∈ S : conf(t) == conf(s) then
    for all p : (p, s) ∈ T do
      T = T \ {(p, s)};
      if (p, t) ∉ T then
        T = T ∪ (p, t);
      end if
    end for
  else
    if isLegal(s) then
      for all c ∈ succ(conf(s)) do
        child = new State(c);
        T = T ∪ {(s, child)};
        U = U ∪ {child};
      end for
      S = S ∪ {s};
    else
      recursivelyRemove(s);
    end if
  end if
end while
```

Algorithm 2 recursivelyRemove(State:s)

```
for all p : (p, s) ∈ T do
  T = T \ {(p, s)}
  if ∄t : (p, t) ∈ T then
    recursivelyRemove(p);
  end if
end for
```

If the state is not legal it will not be added to S. In addition, if any of its predecessors does not have anymore successors they will be removed too. And by recursion, if they have any predecessors without successors they will be removed too and so on. This is done because we demand that the transition relation T is total. And when a state has no legal successors the only way to deal with it is to remove it, since any program passing through that state would also pass through illegal states.

After this algorithm terminates (which it will since there are only a finite number of legal states), we will have a transition system where S is the set of states and T is the transition relation. The labeling function L will be obtained by running through all states in S and labeling them with BEL(φ) for each belief φ in the belief base of the configuration of the state. Each state will also be labeled with INTEND(φ) for all achievement goals φ occuring as a triggering event in an intention $i \in I$ or as a triggering event in the intention part of an event $e \in E$. Finally, each state will be labeled with DESIRE(φ) if it is labeled with INTEND(φ) or if φ occurs as a triggering event in one of the events in E.

Chapter 5

Computation Tree Logic

In this section we introduce a logic called computation tree logic (CTL). This choice is inspired by [6] and [12]. However, we will not use the BDI modality extensions of CTL that are used in these papers, but stick to the version of the logic defined in [4]. The differences and the motivation for this will be described later.

CTL can be used to describe the possible computation paths of transition systems like the ones in the last chapter. In addition an efficient model checking algorithm for the logic is known.

5.1 Logical operators

When describing computation paths we use two types of path quantifiers:

- **A** (For all paths) - Requires a property to hold for all computation paths
- **E** (For some path) - Requires a property to hold For some computation path

In addition we have three basic unary temporal operators:

- **X** (Next Time) - Requires a property to hold in the next state of the path
- **F** (Eventually) - Requires a property to hold in some state of the path
- **G** (Globally) - Requires a property to hold in every state of the path

Also, we have two basic dual temporal operators:

- **U** (Until) - Holds if the second property holds in some state of the path and the first property holds in every preceeding state.
- **R** (Release) - Holds if the first property holds in all states until and including the first state where the second property holds.

In addition to path quantifiers and temporal operators we also use the usual logical operators:

- $\neg$ (Negation)
- $\wedge$ (Conjunction)
- $\vee$ (Disjunction)

Finally, we have three modal operators BEL, DESIRE and INTEND used for describing beliefs, desires and intentions of agents.

5.2 Syntax

After explaining the meaning of our operators we move on to describing rules for using these in our CTL formulas. We have two types of formulas: state formulas and path formulas. State formulas can be true or false in a single state of the system whereas path formulas can be true or false along a path of the system.

Let p be a ground atomic formula as defined earlier and let AP be the set of formulas that have the form BEL(p), DESIRE(p) or INTEND(p). Then state formulas are defined as follows:

1. If $p \in AP$, then p is a state formula
2. If f and g are state formulas, then $\neg f$, $f \vee g$ and $f \wedge g$ are state formulas.
3. If f is a path formula, then $\mathbf{E}\ f$ and $\mathbf{A}\ f$ are state formulas.

   Path formulas are defined as follows:
4. If f and g are state formulas, then $\mathbf{X}\ f$, $\mathbf{F}\ f$, $\mathbf{G}\ f$, $f\ \mathbf{U}\ g$ and $f\ \mathbf{R}\ g$ are path formulas.

In standard CTL the set AP is just the set of atomic proposition symbols. Our choice does not change this, but just reflects the types of formulas we wish to describe with CTL. From this point on, the formulas in AP can just be thought of as simple atomic propositions (with not so simple names).

In [6] and [12] they let AP be the set of atomic proposition symbols and use a fourth axiom stating that if ϕ is a state formula, then BEL(ϕ), DESIRE(ϕ) and INTEND(ϕ) are also state formulas. This means that their logic is more expressive since it includes all the same formulas as ours, but also includes formulas like BEL($\mathbf{AG}(p)$) for some proposition symbol p.

Our choice is simpler and it works well with our knowledge of states of AgentSpeak programs. With our definition of BEL, DESIRE and INTEND earlier, we will not have a circumstance where e.g. $\text{BEL}_{\langle ag,C\rangle}(\phi) = f \wedge g$ or $\text{DESIRE}_{\langle ag,C\rangle}(\phi) = \mathbf{A}f$ for ground atomic formulas f and g. This would require an extension of our BEL, DESIRE and INTEND modalities to include rules like $\text{BEL}(f) \wedge \text{BEL}(g) \Rightarrow \text{BEL}(f \wedge g)$. However, this is a possible extension of the possibilities of reasoning we could do about our AgentSpeak programs.

5.3 Semantics

The semantics of CTL is defined with respect to a transition system $T = \langle S, R, L\rangle$ as defined in the last chapter where S is the set of states, $R \subseteq S \times S$ is the transition relation and L is labeling function for atomic proposition symbols in each state s. In other words L is a function $L : S \times AP \to \{true, false\}$ where AP is defined as in the last section.

Remember that we require R to be total, which means that all states have at least one successor state (which is a natural requirement since we are programming reactive systems that should theoretically be able to run for infinitely long time).

We define a path in a transition system T as an infinite sequence of states, $\pi = s_0, s_1, ...$ such that for every $i \geq 0$ we have $(s_i, s_{i+1}) \in R$. The suffix π^i of π is defined as the path $s_i, s_{i+1}, ...$

Now, let T be a transition system, s be a state in T and f be a state formula. Then the notation $T, s \models f$ means that f holds in the state s of the transition system T. Likewise if g is a path formula then $T, \pi \models g$ means that g holds along the path π in the transition system T.

Assume f_1 and f_2 are state formulas, g_1 and g_2 are path formulas, T is a transition system, s is a state and π is path in T. Then the relation of logical consequence ($\models$) is defined in the following way:

1. $T, s \models p \iff L(s, p) = true$
2. $T, s \models \neg f_1 \iff T, s \not\models f_1$
3. $T, s \models f_1 \vee f_2 \iff T, s \models f_1$ or $T, s \models f_2$
4. $T, s \models f_1 \wedge f_2 \iff T, s \models f_1$ and $T, s \models f_2$
5. $T, s \models \mathbf{E}\ g_1 \iff$ There exists a path π from s such that $T, \pi \models g_1$
6. $T, s \models \mathbf{A}\ g_1 \iff$ For every path π starting from s it holds that $T, \pi \models g_1$
7. $T, \pi \models f_1 \iff s$ is the first state of π and $T, s \models f_1$
8. $T, \pi \models \neg g_1 \iff T, \pi \not\models g_1$
9. $T, \pi \models g_1 \vee g_2 \iff T, \pi \models g_1$ or $T, \pi \models g_2$
10. $T, \pi \models g_1 \wedge g_2 \iff T, \pi \models g_1$ and $T, \pi \models g_2$
11. $T, \pi \models \mathbf{X}\ g_1 \iff T, \pi^1 \models g_1$
12. $T, \pi \models \mathbf{F}\ g_1 \iff$ There exists a $k \geq 0$ such that $T, \pi^k \models g_1$
13. $T, \pi \models \mathbf{G}\ g_1 \iff$ For all $i \geq 0 : T, \pi^i \models g_1$
14. $T, \pi \models g_1\ \mathbf{U}\ g_2 \iff$ There exists a $k \geq 0$ such that $T, \pi^k \models g_2$ and for all $0 \leq j < k : T, \pi^j \models g_1$
15. $T, \pi \models g_1\ \mathbf{R}\ g_2 \iff$ For all $j \geq 0$ if for every $i < j : T, \pi^i \not\models g_1$ then $M, \pi^j \models g_2$

When doing model checking we only model check state formulas. Given the definition of state formulas above, the temporal operators **X**, **F**, **G**, **U** and **R** must always be preceeded with one of the path quantifiers **A** and **E**. This leaves us with the ten combinations:

$$\mathbf{AX}, \mathbf{EX}, \mathbf{AF}, \mathbf{EF}, \mathbf{AG}, \mathbf{EG}, \mathbf{AU}, \mathbf{EU}, \mathbf{AR}, \mathbf{ER}$$

However, these ten operators can all be expressed in terms of **EX**, **EG** and **EU**, which means that when developing model checking algorithms for this logic, we only need to concentrate on these three possibilities since all the others can be transformed to these with apropriate preprocessing. We use the following equivalences for preprocessing the CTL formulas:

- $\mathbf{AX}f = \neg\mathbf{EX}(\neg f)$
- $\mathbf{EF}f = \mathbf{E}(True\ \mathbf{U}\ f)$
- $\mathbf{AG}f = \neg\mathbf{EF}(\neg f)$
- $\mathbf{AF}f = \neg\mathbf{EG}(\neg f)$
- $\mathbf{A}(f\ \mathbf{U}\ g) = \neg\mathbf{E}[\neg g\ \mathbf{U}\ (\neg f \wedge \neg g)] \wedge \neg\mathbf{EG}\neg g$
- $\mathbf{A}(f\ \mathbf{R}\ g) = \neg\mathbf{E}(\neg f\ \mathbf{U}\ \neg g)$
- $\mathbf{E}(f\ \mathbf{R}\ g) = \neg\mathbf{A}(\neg f\ \mathbf{U}\ \neg g)$

5.4 Example

On Figure 5.1 we illustrate some transition systems $T = \langle S, R, L\rangle$, where the states in S are represented by nodes in the graph and an edge from nodes representing states s_i and s_j represent the pair $(s_i, s_j) \in R$. We want to illustrate CTL formulas that are true in state s_0 of the systems. Suppose formula f is true in the black nodes and false in the white nodes, then the two graphs illustrate the formulas $T, s_0 \models \mathbf{AG}f$ and $T, s_0 \models \mathbf{EG}f$ respectively.

On Figure 5.2 we illustrate the formulas $T, s_0 \models \mathbf{EX}f$ and $T, s_0 \models \mathbf{AF}f$ respectively.

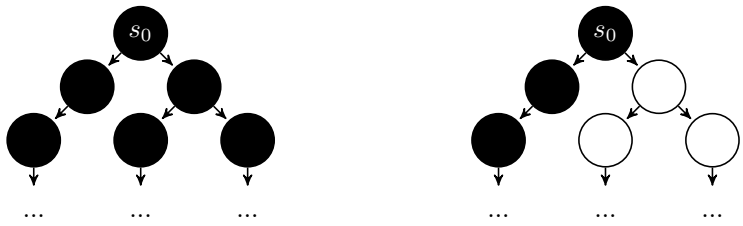

Figure 5.1: AG illustrated to the left, EG to the right

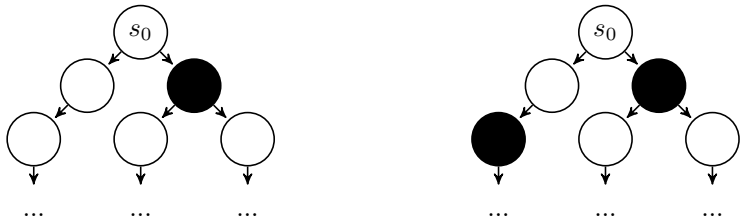

Figure 5.2: EX illustrated to the left, AF to the right

Chapter 6

Model Checking

Model checking is the process of taking a transition system $T = \langle S, R, L \rangle$ and a CTL state formula p and find all states $s \in S : T, s \models p$.

In our case, where the transition system is an abstract representation of an AgentSpeak program and p is a specification of a system property, model checking will help us determine if the specification formula p is satisfied in all possible program states. Thus, it can verify that the system works properly with respect to specifications expressed in CTL.

6.1 Subroutines

Before presenting the total model checking algorithm we start with presenting a number subroutines used by the algorithm.

6.1.1 Formula Preprocessing

The first algorithm is used to transform a CTL formula into an equivalent CTL formula that only uses the operators $\neg$, $\vee$, **EX**, **EU** and **EG**. We use the

equivalences presented in the last chapter to do this. The algorithm works recursively as follows:

Algorithm 3 PreProcess(Formula:f)

switch(f)
case($\neg f_1$): **return** $\neg$(PreProcess(f))
case($f_1 \vee f_2$): **return** PreProcess(f_1) $\vee$ PreProcess(f_2))
case($f_1 \wedge f_2$): **return** $\neg(\neg$ PreProcess(f_1) $\vee \neg$PreProcess(f_2))
case(**EX**f_1): **return** **EX**(PreProcess(f_1))
case(**EG**f_1): **return** **EG**(PreProcess(f_1))
case(**E**(f_1 **U** f_2)): **return** **E**(PreProcess(f_1) **U** PreProcess(f_2))
case(**AX**f_1): **return** $\neg$**EX**($\neg$PreProcess(f_1))
case(**EF**f_1): **return** **E**($true$ **U** PreProcess(f_1))
case(**AG**f_1): **return** $\neg$**E**($true$ **U** $\neg$PreProcess(f_1))
case(**AF**f_1): **return** $\neg$**EG**($\neg$PreProcess(f_1))
case(**A**(f_1 **U** f_2)):
  return $\neg$(**E**[$\neg$(PreProcess(f_1) **U** ($\neg$(PreProcess(f_1) $\vee$ PreProcess(f_2))] $\vee$
EG $\neg$PreProcess(f_2))
case(**A**(f_1 **R** f_2)): **return** $\neg$**E**($\neg PreProcess(f_1)$ **U** $\neg PreProcess(f_2)$)
case(**E**(f_1 **R** f_2)): **return** $\neg$**A**($\neg PreProcess(f_1)$ **U** $\neg PreProcess(f_2)$)
default: **return** f

The running time of this algorithm is $O(|f|)$ since there are made at most $|f|$ recursive calls of PreProcess (because the inputs formulas to the algorithm get shorter for each recursion) where $|f|$ is the number of symbols in f. This argument also ensures that the algorithm terminates. By induction on the structure of CTL formulas, it is easy to see that the resulting formula will only use the operators $\neg$, $\vee$, **EX**, **EU** and **EG**.

6.1.2 Proposition Symbol Labeling

This algorithm takes a transition system $T = \langle S, R, L\rangle$ and a CTL formula f as inputs and labels each state s with all atomic proposition symbols p present in f such that $L(s, p) = true$.

We use the function $label(s)$ to denote the set of labels of s.

This algorithm takes $O(|S| \cdot |f|)$ time where $|S|$ is the number of states and $|f|$ is the number of symbols in the formula f.

Algorithm 4 PropLabeling(TransSystem:$\langle S, R, L\rangle$, Formula:f)

```
for s ∈ S do
  label(s) ← {};
  for p ∈ PropSymbols(f) do
    if L(s, p) = true then
      label(s) ← label(s) ∪ {p};
    end if
  end for
end for
```

6.1.3 Negation Labeling

This algorithm takes as input a transition system $T = \langle S, R, L\rangle$ and a CTL formulas f. We require that:

$$\forall s \in S : (T, s \models f \iff f \in label(s))$$

In other words, we require for all states $s \in S$ that s is labeled with f if and only if f is true in state s.

The algorithm is used to label all states where $\neg f$ holds, given that all states are labeled correctly with respect to f. Thus, it works by labeling all states that are not labeled with f:

Algorithm 5 NegLabeling(TransSystem:$\langle S, R, L\rangle$, Formula:f)

```
for s ∈ S do
  if f ∉ label(s) then
    label(s) ← label(s) ∪ {¬f};
  end if
end for
```

This algorithm runs in $O(|S|)$ time, given that we can make a lookup of whether $f \in label(s)$ in constant time. This is possible if the labelled sets are implemented as hash tables.

6.1.4 Disjunction Labeling

This algorithm takes as input a transition system $T = \langle S, R, L\rangle$ and two CTL formulas f_1 and f_2. We require that all states where f_1 and f_2 are true are

labeled with the formulas.

The algorithm is used to label all states where $f_1 \vee f_2$ holds, given that all states are labeled correctly with respect to f_1 and f_2. Thus, it works by labeling all states that are labelled with either f_1 or f_2:

Algorithm 6 DisjLabeling(TransSystem:$\langle S, R, L\rangle$, Formula:f_1, Formula:f_2)

```
for s ∈ S do
  if f1 ∈ label(s) ∨ f2 ∈ label(s) then
    label(s) ← label(s) ∪ {f1 ∨ f2};
  end if
end for
```

This algorithm also runs in $O(|S|)$ time.

6.1.5 EX Labeling

This algorithm takes as input a transition system $T = \langle S, R, L\rangle$ and a CTL formula f. We require that all states where f are true are labeled with the formula.

The algorithm is used to label all states where $\mathbf{EX}f$ holds, given that all states are labelled correctly with respect to f. Thus, it labels all states that has a successor where f holds according to the definition of $\mathbf{EX}f$:

Algorithm 7 EXLabeling(TransSystem:$\langle S, R, L\rangle$, Formula:f)

```
for s ∈ S do
  for all t : (s, t) ∈ R do
    if f ∈ label(t) ∧ EXf ∉ label(s) then
      label(s) ← label(s) ∪ {EXf};
    end if
  end for
end for
```

This algorithm runs in $O(|S| + |R|)$ since the inner loop will not be executed more than $|R|$ times. This is because a given pair $(s, t) \in R$ only can make the condition of the inner loop true in one the executions of the outer loop.

6.1.6 EU Labeling

This algorithm takes as input a transition system $T = \langle S, R, L \rangle$ and two CTL formulas f_1 and f_2. We require that all states where f_1 and f_2 are true are labeled with the formulas.

The algorithm is used to label all states where $\mathbf{E}(f_1 \mathbf{U} f_2)$ holds, given that all states are labeled correctly with respect to f_1 and f_2. Thus, it needs to label all states from which there exists a path such that f_1 holds until f_2 holds.

We can start by labeling all states where f_2 holds because here we have $\mathbf{E}(f_1 \mathbf{U} f_2)$ according to the definition of $\mathbf{E}$ and $\mathbf{U}$. From these states we work our way backwards and label all predecessors where f_1 holds. Then we work further back to predecessors of these states where f_1 holds and so on, until there are no states left:

Algorithm 8 EULabeling(TransSystem:$\langle S, R, L \rangle$, Formula:f_1, Formula:f_2)

```
A ← {s | f2 ∈ label(s)};
for all s ∈ A do
  label(s) ← label(s) ∪ {E(f1 U f2)};
end for
while A ≠ ∅ do
  choose s ∈ A;
  A ← A \ {s};
  for all t : (t, s) ∈ R do
    if E(f1 U f2) ∉ label(t) ∧ f1 ∈ label(t) then
      label(t) ← label(t) ∪ {E(f1 U f2)};
      A ← A ∪ {t};
    end if
  end for
end while
```

This algorithm runs in $O(|S| + |R|)$. The reason is that the while loop runs at most $|S|$ times since a state cannot be added to the set A more than once. For the same reason as with Algorithm 7 the inner for loop can at most run $|R|$ time in total and thus the total asymptotic running time is $O(|S| + |R|)$.

6.1.7 EG Labeling

This algorithm takes as input a transition system $T = \langle S, R, L \rangle$ and a CTL formula f. We require that all states where f are true are labeled with the

formula.

The algorithm is used to label all states where $\mathbf{EG}f$ holds, given that all states are labelled correctly with respect to f.

Before presenting the algorithm we need to introduce the notion of strongly connected components of directed graphs and a lemma that is used to give an understanding of the algorithm.

Definition (Strongly Connected Component)

A Strongly Connected Component C of a directed graph G is a maximal subgraph such that every node in C can be reached from every other node in C along a directed path where all nodes lies entirely within C.

We abbreviate Strongly Connected Components with SCC from now on.

According to the definition, no two SCCs can contain the same node. Further, all nodes are contained in exactly one SCC. SCCs with more than one node or one node with a self loop a called nontrivial SCCs.

We are now ready to present a lemma about SCCs that will help producing our algorithm: (A formal proof can be found in [4])

Lemma 1
Let f be a formula in CTL. Let $T = \langle S, R, L\rangle$ be a transition system and let $T' = \langle S', R', L'\rangle$ be a restricted version of T obtained by letting $S' = \{s \in S : T, s \models f\}$, $R' = R\,|_{S'\times S'}$ and $L' = L\,|'_S$.
It now holds that $T, s \models \mathbf{EG}f_1$ if and only if

1. $s \in S'$
2. There exists a path in T' that leads from s to some node t in a nontrivial SCC of T'.

The lemma does not seem unreasonable given the definition of the CTL formula $\mathbf{EG}\ f$. This formula is true in a state s of transition system T if there exists an infinite path starting in s such that f holds in all states of the path. This implies that there must be an SCC in the reduced system T', because if there was no SCC, there would not be an infinite path where f holds in all states.

The formula is clearly true in all states of SCCs of T', and further it will be true in states from which there is path to one of the states of an SCC of T' where f

holds in all states along the path.

The principle of algorithm is now to find all SCCs of T' and label all states in these SCCs with $\mathbf{EG}f$. Then it moves to predecessors of these states and add the formula to their labels if f also holds there. Then it moves to their predecessors and so on. Finally, it will have labeled all states in T where $\mathbf{EG}f$ holds.

Algorithm 9 EGLabeling(TransSystem:$\langle S, R, L\rangle$, Formula:f)

```
S' ← {s | f ∈ label(s)};
SCC ← {C | C is a nontrivial SCC of S'};
A ← ⋃_{C∈SCC} {s | s ∈ C};
for all s ∈ A do
  label(s) ← label(s) ∪ {EGf};
end for
while A ≠ ∅ do
  choose s ∈ A;
  A ← A \ {s};
  for all t : t ∈ S' ∧ (t, s) ∈ R do
    if EGf ∉ label(t) then
      label(t) ← label(t) ∪ {EGf};
      A ← A ∪ {t};
    end if
  end for
end while
```

We have not yet presented an algorithm for finding SCCs of T'. In our implementation we use Tarjans algorithm, which can be seen in the appendix. Tarjans algorithm runs in time $O(|S'| + |R'|)$ time. This algorithm runs in $O(|S| + |R|)$ for the same reasons that Algorithm 8 does.

6.2 The Model Checking Algorithm

We are now ready to present the model checking algorithm which takes as input a transition system $T = \langle S, R, L\rangle$, a starting state $s \in S$ and a CTL formula f.

It produces the output $true$ if $T, s \models f$ and produces the output $false$ otherwise. The pseudocode can be seen below.

We use a procedure called FormulasInOrder(f), which returns a stack of all subformulas of f in order such that the innermost, shortest formula is on top

Algorithm 10 ModelChecking(TransSystem:$\langle S, R, L\rangle$, State:s, Formula:f)

```
f ← PreProcess(f);
stack ← FormulasInOrder(f)
while stack ≠ ∅ do
  g ← stack.pop()
  switch(g)
  case ¬g1 : negLabeling(⟨S, R, L⟩, g1); break;
  case g1 ∨ g2 : disjLabeling(⟨S, R, L⟩, g1, g2); break;
  case EXg1 : EXLabeling(⟨S, R, L⟩, g1); break;
  case E(g1 U g2) : EULabeling(⟨S, R, L⟩, g1, g2); break;
  case EGg1 : EGLabeling(⟨S, R, L⟩, g1); break;
  default : PropLabeling(⟨S, R, L⟩, g1);
end while
return f ∈ label(s)
```

and the outermost, longest is at the bottom (thus, f is actually at the bottom). As an example, suppose $f = \mathbf{EX}(\neg p \vee q)$, then we would have:

$$\text{FormulasInOrder}(f) = \begin{array}{l} p \\ q \\ \neg p \\ \neg p \vee q \\ \mathbf{EX}(\neg p \vee q) \end{array}$$

The algorithm works by removing the top of the stack and label the states of the transition system where the formula holds by using the proper subroutine. It then proceeds by taking the next subformula, doing the same and so on. When the subformulas are taken is this order, the preconditions for all our subroutines will be fulfilled, which means that when all formulas of the stack have been processed, then the states that entails the formula f will be labeled with it.

Since FormulasInOrder(f) should nok take more than $O(|f|)$ time if implemented properly, the total running time of the model checking algorithm is $O(|f| \cdot (|S| + |R|)$ since we pass through the while loop $O(|f|)$ times and the subroutines all take $O(|S| + |R|)$ time.

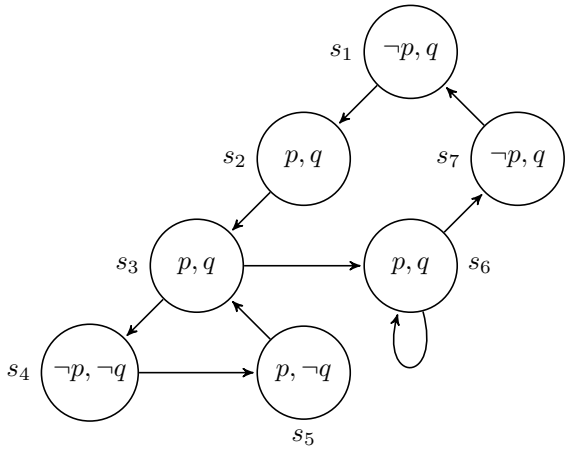

Figure 6.1: Illustration of Transition System

6.3 Example

In this section we will give an example run of the model checking algorithm presented in the last chapter. Since the transition systems generated from AgentSpeak(L) have a large amount of states even for very simple programs, a simple transition system that is not generated from such a program has been chosen to illustrate the process of the algorithm instead.

The transition system T we use as input has the states

$$S = \{s_1, s_2, ..., s_7\},$$

the transition relation of T is defined by

$$R = \{(s_1, s_2), (s_2, s_3), (s_3, s_4), (s_3, s_6), (s_4, s_5), (s_5, s_3), (s_6, s_6), (s_6, s_7), (s_7, s_1)\},$$

and the labeling function of T is defined in Figure 6.2. In this example we use the proposition symbols p and q even though we technically aren't allowed to name the propositions like this, given our definitions earlier. This namecalling does not change the validity of the algorithm in any way, but makes notation slightly simpler in this example. Therefore this choice. A graph is shown in

Figure 6.1 illustrating T, where the states in S are represented by nodes, the transition relation R is represented by edges and the nodes are labeled with the literals that are true in the corresponding state according to L.

$L : AP \times S \rightarrow \{true, false\}$		
	p	q
s_1	false	true
s_2	true	true
s_3	true	true
s_4	false	false
s_5	true	false
s_6	true	true
s_7	false	true

Figure 6.2: Definition of labeling function

We will use the algorithm to find out which of the states in S that satisfies the CTL formula $f = \mathbf{EG}(\mathbf{EX}(p) \wedge q)$.

First, the formula is preprocessed which gives us $f = \mathbf{EG}(\neg[\neg\mathbf{EX}(p) \vee \neg q])$

Secondly, the subformulas of f are added to a stack, such that the elements are sorted with the innermost, shortest subformula on top and outermost subformula at the bottom. Thus, we have

$$stack = \{p, q, \mathbf{EX}(p), \neg q, \neg\mathbf{EX}(p), \neg\mathbf{EX}(p)\vee\neg q, \neg[\neg\mathbf{EX}(p)\vee\neg q], \mathbf{EG}(\neg[\neg\mathbf{EX}(p)\vee\neg q])\}$$

This stack tells us the order in which the subformulas should be processed, starting with the formula p. For simplicity we will denote the set of states in S that are labeled by a formula g by $states(g)$.

The algorithm now proceeds by labeling all states where p holds according to L. Thus, we have

$$states(p) = \{s_2, s_3, s_5, s_6\}$$

The same is done with q which gives us

$$states(q) = \{s_1, s_2, s_3, s_6, s_7\}$$

Next step is to call EXLabeling(T, p) that labels all states that has a successor where p holds with $\mathbf{EX}(p)$. This gives us

$$states(\mathbf{EX}(p)) = \{s_1, s_2, s_3, s_4, s_5, s_6\}$$

We now call NegLabeling(T, q) to label all states where q does not hold with $\neg q$

$$states(\neg q) = \{s_4, s_5\}$$

The same algorithm is used to label states where $\neg\mathbf{EX}(p)$ giving

$$states(\neg\mathbf{EX}(p)) = \{s_7\}$$

Next, we call DisjLabeling$(T, \neg\mathbf{EX}(p) \vee \neg q)$ labeling all states where either $\neg\mathbf{EX}(p)$ or $\neg q$ holds, giving

$$states(\neg\mathbf{EX}(p) \vee \neg q) = \{s_4, s_5, s_7\}$$

Now, using negation we label the other states to get:

$$states(\neg[\neg\mathbf{EX}(p) \vee \neg q]) = \{s_1, s_2, s_3, s_6\}$$

Finally, we use EGLabeling$(T, \neg[\neg\mathbf{EX}(p) \vee \neg q])$ to find the states where $f = \mathbf{EG}(\neg[\neg\mathbf{EX}(p) \vee \neg q])$ holds.

This algorithm starts by finding nontrivial SCCs in the transition system restricted to states where $\neg[\neg\mathbf{EX}(p) \vee \neg q]$ holds. Since this system only consists of the states s_1, s_2, s_3 and s_6 the only non-trivial SCC is the one consisting of s_6 with a self-loop. This means that s_6 immediately gets labeled with f and also that s_1, s_2 and s_3 are labeled with f since there is a path from each of these states to s_6 in the restricted system.

Thus, we have found out that the states of the system T that satisfies f are s_1, s_2, s_3 and s_6 by using model checking.

Chapter 7

Implementation

As a part of the project a prototype for the system described in this report has been implemented.

It can be divided into three major parts that can all be run seperately. These three parts are:

- An AgentSpeak interpreter that can read AgentSpeak programs to generate a configuration and follow the semantic rules described in this report to generate successors of any given configuration.
- A CTL model checker that can perform the model checking algorithm described in the last chapter.
- A State Space Generation program that uses the algorithm from chapter 4 to generate a state space representing an AgentSpeak program. It uses the AgentSpeak interpreter to do this and is interfaced to work with the CTL model checker.

All program components are implemented in Java. Since it is a prototype the program has not been tested a lot. However the tests that have been performed shows that for typical AgentSpeak programs and CTL formulas, the State Space

Generation process is by far the most time consuming part of the program. Even very simple programs creates quite large state spaces.

There is definitely room for improvement of the implementation which will make us able to verify larger programs. It quickly becomes a problem if we increase maximum belief base sizes, maximum number of plans in an intention etc. For now, we have not been able to verify programs with more than 3 possible belief atoms in the belief base at a time.

However, on a positive note, the verification process seems to work correctly which was the primary goal of implementing the system.

Chapter 8

Conclusion

In this report we have presented a system for verifying AgentSpeak programs using model checking. It has been a challenge to create finite abstract models of AgentSpeak programs since there can be an infinite number of possible states in an AgentSpeak program. This follows from our choice to make a verification system for verifying AgentSpeak agents inhabiting non-deterministic environments. However, most of the theory used in this report could probably also be used for creating verfication systems of agents inhabiting deterministic or stochastic environments.

The branching time logic CTL for describing BDI-properties of agents over time has been presented and used in combination with a model checking algorithm to create a system that is able to verify properties of AgentSpeak programs.

Our implementation is still not effective enough to verify large programs which is caused by the great number of possible states of the AgentSpeak programs. There is still a lot of work to be done on this front and probably also possibilities of reducing the state space by using different kinds of abstractions.

However, given enough time and space our programs have been proven able to verify AgentSpeak programs which was the purpose of this report.

Steen Vester

Appendix A

Tarjans Algorithm

Below follows Tarjans algorithm for finding and printing all strongly connected components of a graph.

Algorithm 11 getSCCs (Graph:$G = (V, E)$)

```
index = 0
S = empty
for all v ∈ V do
  if v.index is undefined then
    tarjan(v);
  end if
end for
```

Algorithm 12 Tarjan(Graph:$G = (V, E)$)

```
v.index = index;
v.lowlink = index;
index++;
S.push(v);
for all (v, v') ∈ E do
  if v'.index is undefined then
    tarjan(v');
    v.lowlink = min(v.lowlink, v'.lowlink)
  else
    v.lowlink = min(v.lowlink, v'.index)
  end if
end for
if v.lowlink == v.index then
  print "SCC:"
  repeat
    v' = S.pop;
    print v'
  until v' == v
end if
```

Bibliography

[1] R. H. Bordini, J. F. Hubner and M. Wooldridge 2007, *Programming Multi-Agent Systems in AgentSpeak using Jason*, Wiley

[2] H. R. Nielson and F. Nielson 2007, *Semantics with applications - An Appetizer*, Springer

[3] M. Ben-Ari 2000, *Mathematical Logic in Computer Science*, Springer

[4] W. M. Clarke Jr., O. Grumberg and D. A. Peled 1999, *Model Checking*, The MIT Press

[5] M. Wooldridge 2009, *An introduction to MultiAgent Systems*, Wiley

[6] A. S. Rao and M. P. Georgeff 1998 *Decision Procedures for BDI Logics*, in Journal of Logic and Computation vol. 8 nr. 3

[7] A. S. Rao 1996, *AgentSpeak(L): BDI Agents speak out in a logical computable language*, in Agents Breaking Away: 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World, MAAMAW 1996

[8] R. H. Bordini, M. Fisher, W. Visser and M. Wooldridge 2006, *Verifying Multi-Agent Programs by Model Checking* in Autonomous Agents and Multi-Agent Systems vol. 12, nr. 2 march 2006

[9] R. H. Bordini, M. Fisher, W. Visser and M. Wooldridge 2004, *Verifiable Multi-Agent Programs* in Programming Multi-Agent Systems page 72-89, Springer

[10] R. H. Bordini, M. Fisher, W. Visser and M. Wooldridge 2004, *Model Checking Rational Agents* in IEEE Intelligent Systems, vol. 19, issue 5, September 2004

[11] R. H. Bordini and A. F. Moreira 2004, *Proving BDI properties of agent-oriented programming languages - The asymmetry thesis principles in AgentSpeak(L)* in Annals of Mathematics and Artificial Intelligence 42: 197-226

[12] A. S. Rao and M. P. Georgeff 1993, *A Model-Theoretic Approach to the Verification of Agent-Oriented Systems* in Proceedings of the thirteenth International Joint Conference on Artificial Intelligence, Chamberey, France, 1993