Property-based Slicing for Agent Verification

RAFAEL H. BORDINI, *Department of Computer Science, University of Durham, Durham, UK.*
E-mail: R.Bordini@inf.ufrgs.br
Present Address: Institute of Informatics, Federal University of Rio Grande do Sul, Porto Alegre, Brazil.

MICHAEL FISHER and MICHAEL WOOLDRIDGE, *Department of Computer Science, University of Liverpool, Liverpool, UK.*
E-mail: MFisher@liverpool.ac.uk; mjw@liverpool.ac.uk

WILLEM VISSER, *Department of Mathematical Sciences, Computer Science Division, Stellenbosch University, Stellenbosch, South Africa.*
E-mail: wvisser@cs.sun.ac.za

Abstract

Programming languages designed specifically for multi-agent systems represent a new programming paradigm that has gained popularity over recent years, with some multi-agent programming languages being used in increasingly sophisticated applications, often in critical areas. To support this, we have developed a set of tools to allow the use of model-checking techniques in the verification of systems directly implemented in one particular language called AgentSpeak. The success of model checking as a verification technique for large software systems is dependent partly on its use in combination with various state-space reduction techniques, an important example of which is *property-based slicing*. This article introduces an algorithm for property-based slicing of AgentSpeak multi-agent systems. The algorithm uses literal dependence graphs, as developed for slicing logic programs, and generates a program slice whose state space is stuttering-equivalent to that of the original program; the slicing criterion is a property in a logic with LTL operators and (shallow) BDI modalities. In addition to showing correctness and characterizing the complexity of the slicing algorithm, we apply it to an AgentSpeak program based on autonomous planetary exploration rovers, and we discuss how slicing reduces the model-checking state space. The experiment results show a significant reduction in the state space required for model checking that agent, thus indicating that this approach can have an important impact on the future practicality of agent verification.

Keywords: Program verification, multi-agent programming languages, property-based slicing, model checking, multi-agent systems.

1 Introduction

The last decade has seen significant growth in both the volume and maturity of research being carried out in the area of *agent-based systems*. The agent metaphor has been shown to be useful in many practical domains, particularly those involving complex systems comprising flexible, autonomous and distributed components. This, together with the availability of increasingly powerful agent development platforms (often as extensions to Java) has meant that the *industrial* uptake of this technology [4] is growing too.

But why is the agent approach so popular? An agent can be seen as an *autonomous* computational entity—essentially, an agent makes its own decisions about what activities to pursue. We are particularly concerned with *rational agents*, which can be seen as agents that make such decisions in a rational and *explainable* way. Since agents are autonomous, understanding *why* an agent chooses a

particular course of action is vital. Therefore, the key new aspects that agent-oriented programming languages bring is the need to consider, when designing or analysing programs, not just what agents do but *why* they do it. With this paradigm shift, the agent metaphor has been shown to be useful in the development of various applications, including air-traffic control [29], autonomous spacecraft control [35], health care [34] and industrial systems control [25]. Clearly, these are areas for which we often demand *dependability* and *security*.

As agent-based solutions are used in increasingly complex and critical areas, so there is a greater need to analyse rigorously the behaviour of such systems. Not surprisingly, therefore, formal verification techniques tailored specifically for agent-based systems is an area that is also attracting a great deal of attention. Indeed, our work is at the forefront of this new area [5, 6, 14], where we have extended and adapted *model-checking* techniques to the verification of agent-based systems. Examples of other uses of model checking techniques in the area of multi-agent systems include [2, 19, 36, 38, 46, 48, 53]. For a detailed survey of logic-based approaches for programming and verifying multi-agent systems, see [15].

Model checking [11] is a technique whereby a finite description of a system is analysed with respect to a temporal logic formula in order to ascertain whether *all* possible executions of the system satisfy the property described by the formula. Temporal logics are important in a variety of theoretical and practical aspects of Computer Science and Artificial Intelligence [16], but the interest here is in their use for formal specification and verification of hardware and software [1, 32]. In particular, model checking [11, 13, 23, 24, 41], is now very popular and increasingly used outside academia.

In our work, we have developed model-checking techniques for agent-based systems developed using the agent programming language AgentSpeak [7, 8, 39]. As described above, it is vital not only to verify the behaviour the agent system has, but to verify *why* the agents are undertaking certain courses of action. Thus, the temporal basis of model checking must be extended with notions such as agent *belief* and agent *intention*, both of which are characterized via *modal logics*. While the temporal component captures the *dynamic* nature of agent computation, the modal components capture the *informational* ('belief'), *motivational* ('desire') and *deliberative* ('intention') aspects of a rational agent.

Perhaps the key difficulty in applying model-checking techniques to real-world applications is that of the *state-explosion problem*. As a consequence, approaches to reducing the state space required by the checking process are the subject of much ongoing research. *Program slicing* is a widely studied technique for simplifying the analysis of conventional programs [47, 55]. The basic idea behind program slicing is to eliminate elements of a program that are not relevant to the analysis in hand. In our case, since we wish to verify some property, the idea is to use the property as a slicing criterion, eliminating parts of the program that can play no part in affecting whether or not the property is realized. This approach is called *property-based slicing*. Property-based slicing can be understood as a type of automated under-approximation (i.e. whereby fewer behaviours are present in the abstracted system than in the original one), which leads to *precise abstraction* in the sense that the result of model checking the given property using the abstract model is the same as though the original model had been used.

Although slicing techniques have been successfully used in conventional programs to reduce the state space, these standard techniques are either not applicable to agent programs (e.g. they are language dependent) or they are only partially successful when applied to multi-agent programs. What we require are slicing techniques tailored to the *agent-specific* aspects of (multi-)agent programs. This is what we describe in this article: a new slicing algorithm for AgentSpeak, and its application in model checking. It is also worth mentioning that slicing has various other uses in software engineering (such as program comprehension, reuse and testing).

The remainder of this article is organized as follows. In Section 2, we survey the key ideas needed to understand the remainder of the article: agent programming, AgentSpeak syntax and semantics, and slicing techniques. The actual algorithm for property-based slicing of AgentSpeak is described in detail in Section 3, as are correctness and complexity results. In order to examine the practical use of the approach, we introduce a particular case study in Section 4, show an AgentSpeak program for that scenario, apply the slicing algorithm, and discuss the results of model-checking experiments. Finally, in Section 5, we provide concluding remarks and highlight future work.

2 Background

The background section of this article is unusually large. The reason is that the article introduces a *state-space reduction technique* aimed at use with *model checking* for a particular *agent-oriented programming language* (i.e. a language that is specifically formulated for programming *multi-agent systems*), and draws upon existing *slicing techniques* that apply to *logic programming languages*. It is unrealistic to expect that readers will have the required background in all these different areas, so we here try to summarize the background that is required for the remainder of this article, in particular about the relevant agent-oriented programming language, its semantics, our previous work on model-checking techniques for systems programmed in that language and slicing techniques for logic programming languages.

2.1 AgentSpeak

The AgentSpeak(L) programming language was introduced in [39]. It is a natural extension of logic programming for the development of reactive planning systems, and provides an elegant abstract framework for programming BDI agents. In this article, we only give a brief introduction to AgentSpeak(L); see [8, 39] for more details.

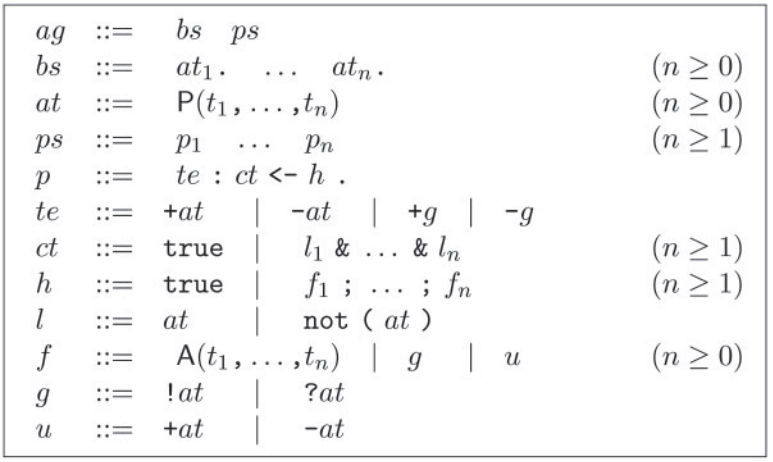
An AgentSpeak(L) agent is created by the specification of a set of base beliefs and a set of plans. A *belief atom* is simply a first-order predicate in the usual notation, and belief atoms or their negations are *belief literals*. An *initial set of beliefs* is just a collection of ground belief atoms.

AgentSpeak(L) distinguishes two types of goals: *achievement goals* and *test goals*. Achievement goals are predicates (as for beliefs) prefixed with the '!' operator, while test goals are prefixed with the '?'operator. Achievement goals state that the agent wants to achieve a state of the world where the associated predicate is true. (In practice, achievement goals initiate the execution of *sub-plans*.) A *test goal* states that the agent wants to test whether the associated predicate is one of its beliefs (i.e. whether it can be unified with a predicate in that agent's base beliefs).

Next, the notion of a *triggering event* is introduced. It is a very important concept in this language, as triggering events define which events may initiate the execution of plans; the idea of *event*, both internal and external, will be made clear below. There are two types of triggering events: those related to the *addition* ('+') and *deletion* ('−') of mental attitudes (beliefs or goals).

Plans refer to the *basic actions* that an agent is able to perform on its environment. Such actions are also defined as first-order predicates, but with special predicate symbols (called *action symbols*) used to distinguish them. The actual syntax of AgentSpeak(L) programs is based on the definition of plans, below. Recall that the designer of an AgentSpeak(L) agent specifies a set of beliefs and a set of plans only.

If e is a triggering event, $b_1, \ldots, b_m$ are belief literals, and $h_1, \ldots, h_n$ are goals or actions, then '$e : b_1 \wedge \cdots \wedge b_m \leftarrow h_1; \ldots; h_n$.' is a *plan*. An AgentSpeak(L) plan has a *head* (the expression to the left



ag	::=	$bs \quad ps$	
bs	::=	$at_1 . \quad \ldots \quad at_n .$	$(n \geq 0)$
at	::=	$\mathsf{P}(t_1, \ldots, t_n)$	$(n \geq 0)$
ps	::=	$p_1 \quad \ldots \quad p_n$	$(n \geq 1)$
p	::=	te : ct <- h .	
te	::=	$+at \mid -at \mid +g \mid -g$	
ct	::=	true $\mid$ l_1 & ... & l_n	$(n \geq 1)$
h	::=	true $\mid$ f_1 ; ... ; f_n	$(n \geq 1)$
l	::=	$at \mid$ not (at)	
f	::=	$\mathsf{A}(t_1, \ldots, t_n) \mid g \mid u$	$(n \geq 0)$
g	::=	$!at \mid ?at$	
u	::=	$+at \mid -at$	

FIGURE 1. The concrete syntax of AgentSpeak.

of the arrow), which is formed from a triggering event (denoting the purpose for that plan), and a conjunction of belief literals representing a *context* (separated from the triggering event by ':'). The conjunction of literals in the context must be satisfied if the plan is to be executed (the context must be a logical consequence of that agent's current beliefs). A plan also has a *body* (the expression to the right of the arrow), which is a sequence of basic actions or (sub)goals that the agent has to achieve (or test) when the plan is triggered.

The grammar in Figure 1 gives the concrete syntax of AgentSpeak (we use AgentSpeak a name for any variant of the original AgentSpeak(L) language). In the grammar, P stands for any predicate symbol, A for any action symbol, while the t_i are first-order terms. As in Prolog, an uppercase initial letter is used to denote variables and lowercase initial letters denote terms and predicate symbols. The grammar in the figure also includes a simple, common extension of the language originally defined by Rao [39], namely the possibility to add or remove beliefs from within a plan body.

Besides the belief base and the plan library, an AgentSpeak interpreter also manages a set of *events* and a set of *intentions*, and its functioning requires three *selection functions*. *Intentions* are particular courses of actions to which an agent has committed in order to achieve a particular goal; each intention is a stack of *partially instantiated plans*, i.e. plans where some of the variables have been instantiated. An *event*, which may trigger the execution of a plan, can be *external*, when originating from perception of the agent's environment, or *internal*, when generated from the agent's own execution of a plan (e.g. an achievement goal within a plan body is a goal-addition event which may be a triggering event). The event selection function ($\mathcal{S}_\mathcal{E}$) selects a single event from the set of events; another selection function ($\mathcal{S}_\mathcal{O}$) selects an 'option' (i.e. an applicable plan) from a set of applicable plans; and a third selection function ($\mathcal{S}_\mathcal{I}$) selects one particular intention from the set of intentions. The selection functions are supposed to be agent specific, in the sense that they should make selections based on an agent's characteristics.

2.2 *Operational semantics of AgentSpeak*

The semantics presented in this section is taken from [50], which is a refinement of the semantics that appeared in [8]. We define the semantics of AgentSpeak using operational semantics, a widely

used method for giving semantics to programming languages and studying their properties [37]. The operational semantics is given by a set of rules that define a transition relation between configurations $\langle ag, C, T, s\rangle$ where:

- An agent program ag is, as defined above, a set of beliefs bs and a set of plans ps.
- An agent's circumstance C is a tuple $\langle I, E, A\rangle$ where:
  - I is a set of *intentions* $\{i, i', \ldots\}$. Each intention i is a stack of partially instantiated plans.
  - E is a set of *events* $\{(te, i), (te', i'), \ldots\}$. Each event is a pair (te, i), where te is a triggering event and i is an intention (a stack of plans in case of an internal event, or the empty intention T in case of an external event). When the belief revision function (which is not part of the AgentSpeak interpreter but rather of the agent's overall architecture), updates the belief base, the associated events — i.e. additions and deletions of beliefs — are included in this set. These are called *external* events; internal events are generated by additions or deletions of goals.
  - A is a set of *actions* to be performed in the environment. An action expression included in this set tells other architectural components to actually perform the respective action on the environment, thereby changing it.
- It helps to use a structure which keeps track of temporary information that is required in subsequent stages within a single reasoning cycle. T is the tuple $\langle R, Ap, \iota, \varepsilon, \rho\rangle$ with such temporary information; it has as components:
  - R for the set of *relevant plans* (for the event being handled).
  - Ap for the set of *applicable plans* (the relevant plans whose contexts are true).
  - ι, ε and ρ record a particular intention, event, and applicable plan (respectively) being considered along the execution of one reasoning cycle.
- The current step s within an agent's reasoning cycle is symbolically annotated by $s \in \{\mathsf{ProcMsg}, \mathsf{SelEv}, \mathsf{RelPl}, \mathsf{ApplPl}, \mathsf{SelAppl}, \mathsf{AddIM}, \mathsf{SelInt}, \mathsf{ExecInt}, \mathsf{ClrInt}\}$, which stands for: processing a message from the agent's mail inbox, selecting an event from the set of events, retrieving all relevant plans, checking which of those are applicable, selecting one particular applicable plan (the intended means), adding the new intended means to the set of intentions, selecting an intention, executing the selected intention and clearing an intention or intended means that may have finished in the previous step.
  The graph in Figure 2 shows all possible transitions between the various steps in an agent's reasoning cycle (the labels in the nodes name each step in the cycle).

In the interests of readability, we adopt the following notational conventions in our semantic rules:

- If C is an AgentSpeak agent circumstance, we write C_E to make reference to the component E of C. Similarly for all the other components of a configuration.
- We write $T_\iota = \_$ (the underscore symbol) to indicate that there is no intention presently being considered in that reasoning cycle. Similarly for T_ρ and T_ε.
- We write $i[p]$ to denote the intention that has plan p on top of intention i.

As mentioned above, the AgentSpeak interpreter makes use of three *selection functions* that are defined by the agent programmer. The selection function $\mathcal{S}_\mathcal{E}$ selects an event from the set of events C_E; the selection function S_{Ap} selects an applicable plan given a set of applicable plans; and $\mathcal{S}_\mathcal{I}$ selects an intention from the set of intentions C_I (the chosen intention is then executed). Formally, all the selection functions an agent uses are also part of its configuration (as is the social acceptance function that we mention below). However, as they are defined by the agent programmer at design time and do not (in principle) change at run time, we avoid including them in the configuration, for the sake of readability.

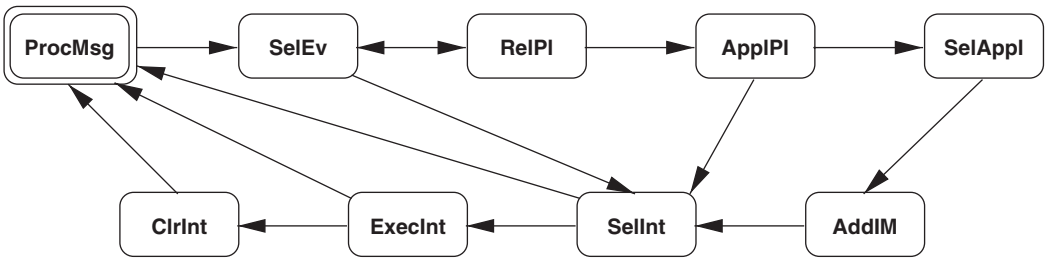


FIGURE 2. Transitions between reasoning cycle steps.

Further, we define some auxiliary syntactic functions to help the presentation of the semantics. If p is a plan of the form $te : ct \leftarrow h$, we define $\mathsf{TrEv}(p) = te$ and $\mathsf{Ctxt}(p) = ct$. That is, these projection functions return the triggering event and the context of the plan, respectively. The TrEv function can also be applied to the head of a plan rather than the whole plan, but is assumed to work similarly in that case. In order to improve readability of the semantic rules, we use two operations on belief bases (i.e. sets of annotated predicates). We write $bs' = bs + b$ to indicate that bs' is as bs except that $bs' \models b$. Similarly, $bs' = bs - b$ will indicate that bs' is as bs except that $bs' \not\models b$.

A plan is considered *relevant* with respect to a triggering event if it has been written to deal with that event. In practice, this is checked by trying to unify the triggering event part of the plan with the triggering event within the event that has been selected for treatment in that reasoning cycle. Below, we use θ to denote substitutions obtained by this unification exactly as in logic programming languages such as Prolog; note that apart from some extra prefixing notation (which must match for two AgentSpeak terms to unify), atomic formulæ in AgentSpeak are very much like Prolog structures (i.e. compound terms). Further, note that in the semantics of AgentSpeak used here, we are interested in obtaining all possible unifying substitutions rather than a most general unifier, unlike unification in Prolog. We use the notation $t\theta$ to denote the term obtained by applying substitution θ to term t; substitution (and composition of substitutions) is also as usual in logic programming [30]. We define the following auxiliary functions to facilitate the presentation of semantic rules.

DEFINITION 2.1
Given the plans ps of an agent and a triggering event te, the set $\mathsf{RelPlans}(ps, te)$ of relevant plans is given as follows:

$$\mathsf{RelPlans}(ps, te) = \{(p, \theta) \mid p \in ps \text{ and } \theta \text{ is s.t. } te = \mathsf{TrEv}(p)\theta\}.$$

A plan is *applicable* if it is relevant and its context is a logical consequence of the agent's beliefs. An auxiliary function for applicable plans is defined as follows.

DEFINITION 2.2
Given a set of relevant plans R and the beliefs bs of an agent, the set of applicable plans $\mathsf{AppPlans}(bs, R)$ is defined as follows:

$$\mathsf{AppPlans}(bs, R) = \{(p, \theta' \circ \theta) \mid (p, \theta) \in R \text{ and } \theta' \text{ is s.t. } bs \models \mathsf{Ctxt}(p)\theta\theta'\}.$$

Finally, we need an auxiliary function to help in the semantic rule that is used when the agent is executing a test goal. The evaluation of a test goal $?at$ requires testing if the formula at is a logical

consequence of the agent's beliefs. The auxiliary function returns a set of most general unifiers all of which make the formula *at* a logical consequence of a set of formulæ *bs*, as follows.

DEFINITION 2.3
Given a set of formulæ *bs* and a formula *at*, the set of substitutions $\mathsf{Test}(bs, at)$ produced by testing *at* against *bs* is defined as follows:

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

Next, we present the rules that define the operational semantics of the reasoning cycle of AgentSpeak.

2.3 Semantic rules

In the general case, an agent's initial configuration is $\langle ag, C, T, \mathsf{ProcMsg}\rangle$, where *ag* is as given by the agent program, and all components of C and T are empty. The initial step of the reasoning cycle is ProcMsg, which deals with agent communication. However, communication is one of the extension of the original AgentSpeak(L) language which are omitted in this article for the sake of clarity. So here we consider that the reasoning cycle starts with an event selection (SelEv) being made, which is the reasoning cycle as originally defined for the language, as embodied in the semantics presented below. We should also emphasize that the '*where*' part of the semantic rules formalize all components of the transition system configuration that change as a consequence of applying that semantic rule; all other components remain unaltered but, to avoid cluttering the rules, this is not formally stated.

2.3.1 Event selection

The rule below assumes the existence of a selection function $\mathcal{S}_\mathcal{E}$ that selects events from a set of events E. The selected event is removed from E and it is assigned to the ε component of the temporary information. Rule **SelEv$_2$** skips to the intention execution part of the cycle, in case there is no event to handle.

$$\frac{\mathcal{S}_\mathcal{E}(C_E) = (te, i)}{\langle ag, C, T, \mathsf{SelEv}\rangle \longrightarrow \langle ag, C', T', \mathsf{RelPl}\rangle} \quad \textbf{(SelEv$_1$)}$$

$$\textit{where: } \begin{array}{l} C'_E = C_E \setminus \{(te, i)\} \\ T'_\varepsilon = (te, i) \end{array}$$

$$\frac{C_E = \{\}}{\langle ag, C, T, \mathsf{SelEv}\rangle \longrightarrow \langle ag, C, T, \mathsf{SelInt}\rangle} \quad \textbf{(SelEv$_2$)}$$

2.3.2 Relevant plans

Rule **Rel$_1$** assigns the set of relevant plans to component T_R. Rule **Rel$_2$** deals with the situation where there are no relevant plans for an event; in that case, the event is simply discarded. In fact, an intention associated with that event might be completely discarded too; if there are no relevant plans to handle an event generated by that intention, it cannot be further executed. (In practice, this leads to activation of the plan failure mechanism, which we do not discuss here for clarity of presentation.)

$$\frac{T_\varepsilon = (te, i) \qquad \mathsf{RelPlans}(ag_{ps}, te) \neq \{\}}{\langle ag, C, T, \mathsf{RelPl}\rangle \longrightarrow \langle ag, C, T', \mathsf{ApplPl}\rangle} \quad \textbf{(Rel}_\mathbf{1}\textbf{)}$$

$$\textit{where: } T'_R = \mathsf{RelPlans}(ag_{ps}, te)$$

$$\frac{T_\varepsilon = (te, i) \qquad \mathsf{RelPlans}(ag_{ps}, te) = \{\}}{\langle ag, C, T, \mathsf{RelPl}\rangle \longrightarrow \langle ag, C', T, \mathsf{SelEv}\rangle} \quad \textbf{(Rel}_\mathbf{2}\textbf{)}$$

$$\textit{where: } C'_E = C_E \setminus \{(te, i)\}$$

2.3.3 Applicable plans

The rule **Appl$_1$** assigns the set of applicable plans to the T_{Ap} component; rule **Appl$_2$** applies when there are no applicable plans for an event, in which case the event is simply discarded. Again, in practice, this normally leads to the plan failure mechanism being used, rather than simply discarding the event (and the whole intention with it).

$$\frac{\mathsf{AppPlans}(ag_{bs}, T_R) \neq \{\}}{\langle ag, C, T, \mathsf{ApplPl}\rangle \longrightarrow \langle ag, C, T', \mathsf{SelAppl}\rangle} \quad \textbf{(Appl}_\mathbf{1}\textbf{)}$$

$$\textit{where: } T'_{Ap} = \mathsf{AppPlans}(ag_{bs}, T_R)$$

$$\frac{\mathsf{AppPlans}(ag_{bs}, T_R) = \{\} \qquad T_\varepsilon = (te, i)}{\langle ag, C, T, \mathsf{ApplPl}\rangle \longrightarrow \langle ag, C', T, \mathsf{SelEv}\rangle} \quad \textbf{(Appl}_\mathbf{2}\textbf{)}$$

$$\textit{where: } C'_E = C_E \setminus \{(te, i)\}$$

2.3.4 Selection of an applicable plan

This rule assumes the existence of a selection function S_{Ap} that selects a plan from a set of applicable plans T_{Ap}. The selected plan is then assigned to the T_ρ component of the configuration.

$$\frac{S_{Ap}(T_{Ap}) = (p, \theta)}{\langle ag, C, T, \mathsf{SelAppl}\rangle \longrightarrow \langle ag, C, T', \mathsf{AddIM}\rangle} \quad \textbf{(SelAppl)}$$

$$\textit{where: } T'_\rho = (p, \theta)$$

2.3.5 Adding an intended means to the set of intentions

Events can be classified as external or internal (depending on whether they were generated from the agent's perception of its environment, or whether they were generated by the previous execution of other plans, respectively). Rule **ExtEv** says that if the event ε is external (which is indicated by T in the intention associated to ε) a new intention is created and the only intended means in it is the plan p assigned to the ρ component. If the event is internal, rule **IntEv** says that the plan in ρ should be put on top of the intention associated with the event.

$$\frac{T_\varepsilon = (te, \mathsf{T}) \qquad T_\rho = (p, \theta)}{\langle ag, C, T, \mathsf{AddIM}\rangle \longrightarrow \langle ag, C', T, \mathsf{SelInt}\rangle} \quad \textbf{(ExtEv)}$$

$$\textit{where: } C'_I = C_I \cup \{\ [p\theta]\ \}$$

$$\frac{T_\varepsilon = (te, i) \quad T_\rho = (p, \theta)}{\langle ag, C, T, \mathsf{AddIM} \rangle \longrightarrow \langle ag, C', T, \mathsf{SelInt} \rangle} \qquad \textbf{(IntEv)}$$

$$\textit{where: } C'_I = C_I \cup \{\, i[(p\theta)] \,\}$$

Note that, in rule **IntEv**, the whole intention i that generated the internal event needs to be inserted back into C_I, with p as its top. This issue is related to suspended intentions, see rule **AchvGl**.

2.3.6 Intention selection

Rule $\mathbf{SelInt_1}$ assumes the existence of a function $\mathcal{S}_\mathcal{I}$ that selects an intention (i.e. a stack of plans) for processing next, while rule $\mathbf{SelInt_2}$ takes care of the situation where the set of intentions is empty (in which case, the reasoning cycle is simply restarted).

$$\frac{C_I \neq \{\} \qquad \mathcal{S}_\mathcal{I}(C_I) = i}{\langle ag, C, T, \mathsf{SelInt} \rangle \longrightarrow \langle ag, C, T', \mathsf{ExecInt} \rangle} \qquad \textbf{(SelInt}_\mathbf{1}\textbf{)}$$

$$\textit{where: } T'_\iota = i$$

$$\frac{C_I = \{\}}{\langle ag, C, T, \mathsf{SelInt} \rangle \longrightarrow \langle ag, C, T, \mathsf{ProcMsg} \rangle} \qquad \textbf{(SelInt}_\mathbf{2}\textbf{)}$$

2.3.7 Executing an intended means

These rules express the effects of executing the body of a plan; each rule deals with one type of formula that can appear in a plan body. The plan to be executed is always the one on top of the intention that has been selected in the previous step; the specific formula to be executed is the one at the beginning of the body of that plan.

2.3.8 Actions

The action a in the body of the plan is added to the set of actions A. The action is removed from the body of the plan and the intention is updated to reflect this removal.

$$\frac{T_\iota = i[head \leftarrow a\,;\, h]}{\langle ag, C, T, \mathsf{ExecInt} \rangle \longrightarrow \langle ag, C', T, \mathsf{ClrInt} \rangle} \qquad \textbf{(Action)}$$

$$\begin{aligned} \textit{where: } & C'_A = C_A \cup \{a\} \\ & C'_I = (C_I \setminus \{T_\iota\}) \cup \{i[head \leftarrow h]\} \end{aligned}$$

2.3.9 Achievement goals

The rule below registers a new internal event in the set of events E. This event can then be eventually selected (see rule **SelEv**). When the formula being executed is a goal, the formula is not removed from the body of the plan, as in the other cases. This only happens when the plan used for achieving that goal finishes successfully; see rule $\mathbf{ClrInt_2}$. The reasons for this are related to further instantiation of the plan as well as handling plan failure.

$$\frac{T_\iota = i[head \leftarrow !at\,;\,h]}{\langle ag, C, T, \mathsf{ExecInt}\rangle \longrightarrow \langle ag, C', T, \mathsf{ProcMsg}\rangle} \qquad \textbf{(AchvGl)}$$

$$\begin{array}{rl} \textit{where:} & C'_E = C_E \cup \{(+!at, T_\iota)\} \\ & C'_I = C_I \setminus \{T_\iota\} \end{array}$$

Note how the intention that generated the internal event is removed from the set of intentions C_I; this captures the idea of *suspended intentions*. When the event with the achievement-goal addition is treated and a plan for it is selected (rule **IntEv**), the intention can be resumed by executing the plan for achieving that goal. If we have, in a plan body, '$!g;f$' (where f is any formula that can appear in plan bodies), this means that, before f can be executed, the state of affairs represented by goal g needs to be achieved (through the execution of some relevant, applicable plan). This newly added goal is treated as any other event, which means it will go into the set of events until it is eventually selected in a later reasoning cycle. Meanwhile, that plan (with formula f to be executed next) can no longer be executed, hence the whole intention (recall that an intention is a stack of plans) is suspended by being placed, within an event, in the set of events and removed from the set of intentions. When a plan for achieving g has been selected, it is pushed on top of the suspended intention, which is then resumed (i.e. moved back to the set of intentions), according to rule **IntEv**. The execution of that intention proceeds with the plan at the top (in this case, for achieving g), and only when that plan is finished will f be executed (as it will be at the top of the intention again). See [8] for a more detailed discussion of suspended intentions.

2.3.10 Test goals

These rules are used when a test goal formula $?at$ should be executed. Rule $\textbf{TestGl}_\mathbf{1}$ is used when there is a set of substitutions that can make at a logical consequence of the agent's beliefs. If the test goal succeeds, *one* such substitution[1] is applied to the whole intended means, and the reasoning cycle can be continued. If this is not the case, it may be that the test goal is used as a triggering event of a plan, which is used by programmers to formulate more sophisticated queries. Rule $\textbf{TestGl}_\mathbf{2}$ is used in such cases: it generates an internal event, which may trigger the execution of a plan, as with achievement goals. If, in order to carry out a plan, an agent is required to obtain information (at the time of actual execution of the plan) which is not directly available in its belief base, a plan for a test goal can be written which, for example, sends messages to other agents, or processes available data, so that the particular test goal can be concluded (producing an appropriate instantiation of logical variables). If an internal event is generated for the test goal being executed, the process is very similar to achievement goals, where the intention is suspended until a plan is selected to achieve the goal, as explained above.

$$\frac{T_\iota = i[head \leftarrow ?at\,;\,h] \qquad \mathsf{Test}(ag_{bs}, at) \neq \{\}}{\langle ag, C, T, \mathsf{ExecInt}\rangle \longrightarrow \langle ag, C', T, \mathsf{ClrInt}\rangle} \qquad \textbf{(TestGl}_\mathbf{1}\textbf{)}$$

$$\begin{array}{rl} \textit{where:} & C'_I = (C_I \setminus \{T_\iota\}) \cup \{i[(head \leftarrow h)\theta]\} \\ & \theta \in \mathsf{Test}(ag_{bs}, at) \end{array}$$

[1] In practical implementations of AgentSpeak such as ***Jason*** [7], the first successful substitution is used, which depends on the order of the beliefs in the belief base. In the semantics, we purposely leave this vague, as how a particular substitution is chosen is not important from the point of view of the overall behaviour of the language interpreter, thus left as an implementation choice.

$$\frac{T_\iota = i[head \leftarrow ?at\,;h] \qquad \mathsf{Test}(ag_{bs}, at) = \{\}}{\langle ag, C, T, \mathsf{ExecInt}\rangle \longrightarrow \langle ag, C', T, \mathsf{ClrInt}\rangle} \quad \textbf{(TestGl}_\mathbf{2}\textbf{)}$$

$$\begin{array}{rl} \textit{where:} & C'_E = C_E \cup \{(+?at, T_\iota)\} \\ & C'_I = C_I \setminus \{T_\iota\} \end{array}$$

2.3.11 Updating beliefs

Rule **AddBel** simply adds a new event to the set of events E. The formula $+b$ is removed from the body of the plan and the set of intentions is updated properly. Rule **DelBel** works similarly. In both rules, the set of beliefs of the agent should be modified in a way that either the predicate b (with annotation `self`) is included in the new set of beliefs (rule **AddBel**) or it is removed from there (rule **DelBel**). Note that a request to delete beliefs can have variables (at), whilst only ground atoms (b) can be added to the belief base.

$$\frac{T_\iota = i[head \leftarrow +b\,;h]}{\langle ag, C, T, \mathsf{ExecInt}\rangle \longrightarrow \langle ag', C', T, \mathsf{ClrInt}\rangle} \quad \textbf{(AddBel)}$$

$$\begin{array}{rl} \textit{where:} & ag'_{bs} = ag_{bs} + b \\ & C'_E = C_E \cup \{(+b, \mathsf{T})\} \\ & C'_I = (C_I \setminus \{T_\iota\}) \cup \{i[head \leftarrow h]\} \end{array}$$

$$\frac{T_\iota = i[head \leftarrow -b\,;h]}{\langle ag, C, T, \mathsf{ExecInt}\rangle \longrightarrow \langle ag', C', T, \mathsf{ClrInt}\rangle} \quad \textbf{(DelBel)}$$

$$\begin{array}{rl} \textit{where:} & ag'_{bs} = ag_{bs} - b \\ & C'_E = C_E \cup \{(-b, \mathsf{T})\} \\ & C'_I = (C_I \setminus \{T_\iota\}) \cup \{i[head \leftarrow h]\} \end{array}$$

2.3.12 Clearing intentions

Finally, the following rules remove empty intended means or intentions from the set of intentions. Rule **ClrInt$_1$** simply removes a whole intention when nothing remains to be executed in that intention. Rule **ClrInt$_2$** clears the remainder of the plan with an empty body currently at the top of a (non-empty) intention. In this case, it is necessary to further instantiate the plan below the finished plan on top of that intention, and remove the goal that was left at the beginning of the body of the plan below (see rules **AchvGl** and **TestGl**). Note that, in this case, further 'clearing' might be necessary, hence the next step is still ClrInt. Rule **ClrInt$_3$** takes care of the situation where no (further) clearing is required, so a new reasoning cycle can start (step ProcMsg).

$$\frac{j = [head \leftarrow \mathsf{T}], \text{ for some } j \in C_I}{\langle ag, C, T, \mathsf{ClrInt}\rangle \longrightarrow \langle ag, C', T, \mathsf{ProcMsg}\rangle} \quad \textbf{(ClrInt}_\mathbf{1}\textbf{)}$$

$$\textit{where:} \quad C'_I = C_I \setminus \{j\}$$

$$\frac{j = i[head \leftarrow \mathsf{T}], \text{ for some } j \in C_I}{\langle ag, C, T, \mathsf{ClrInt}\rangle \longrightarrow \langle ag, C', T, \mathsf{ClrInt}\rangle} \quad \textbf{(ClrInt}_\mathbf{2}\textbf{)}$$

$$\begin{array}{l} \textit{where:} \quad C'_I = (C_I \setminus \{j\}) \cup \{k[(head' \leftarrow h)\theta]\} \\ \textit{if } \; i = k[head' \leftarrow g\,;h] \textit{ and } \; \theta \textit{ is s.t. } g\theta = \mathsf{TrEv}(head) \end{array}$$

$$\frac{j \neq [head \leftarrow \mathsf{T}] \wedge j \neq i[head \leftarrow \mathsf{T}], \text{ for } any\ j \in C_I}{\langle ag, C, T, \mathsf{ClrInt} \rangle \longrightarrow \langle ag, C, T, \mathsf{ProcMsg} \rangle} \quad (\mathbf{ClrInt_3})$$

2.4 *Model checking AgentSpeak*

Recall that our main goal in this research is to facilitate model checking of AgentSpeak systems. However, model checking as a paradigm is predominantly applied to *finite state* systems. A first key step in our research was thus to restrict AgentSpeak to facilitate its translation to a finite state model, particularly in PROMELA [24]. We call the cut-down version of the language AgentSpeak(F). This restricted language was described in [6] and is briefly reviewed here so that we can later give an example of our approach to programming and verifying multi-agent systems. The idea is to translate multi-agent systems defined in this language into the input language of existing model checkers, so that we can take advantage of the extensive range of features that those model checkers provide.

Further, we would like to be able to verify that systems implemented in AgentSpeak satisfy (or do not satisfy) properties expressed in a BDI logic [40]. Such logics formalize all the main concepts of the BDI architecture used in reactive planning systems such as those generated by AgentSpeak agents. This section also presents a simplified form of BDI logic which we are able to convert into Linear Temporal Logic (LTL) formulæ, so that we can use existing LTL model checkers for verifying our multi-agent systems.

The purpose of this section is to introduce the features and limitations of the languages used in our approach to code the system and to produce specifications the system should satisfy. Their use will be made clear in Section 4, where we present a case study.

It is important to note that the agent model-checking approach we have developed [6], implements the original AgentSpeak program in an appropriate input language for a model checker and then applies the model checker to a (transformed) property. In this way we can utilize the wealth of work on model-checking tools without having to implement a new model checker. The two approaches we have investigated so far are:

(1) to implement an AgentSpeak(F) interpreter in PROMELA, the input language for SPIN [24, 45], and then apply SPIN to BDI properties translated into LTL; and
(2) to implement an AgentSpeak(F) interpreter in Java, and then apply JPF [26, 51], an *on-the-fly* model checker that works *directly* on Java bytecode, to BDI properties translated into appropriate Java structures.

2.4.1 AgentSpeak(F)

The main difference between AgentSpeak(F) and AgentSpeak(L) (see Section 2.1) is that first-order terms are disallowed. That is, terms t_i in Figure 1 are assumed to be either constants or variables. The full set of features currently *disallowed* in AgentSpeak(F) are as follows:

(1) uninstantiated variables in triggering events;
(2) uninstantiated variables in negated literals in a plan's context (as originally defined by Rao [39]);
(3) the same predicate symbol with different arities (this is specifically for when a PROMELA model is required);
(4) arbitrary (nested) first-order terms.

The first restriction means that an achievement goal cannot be called with an uninstantiated variable; this is the usual means for a goal to return values to be used in the plan where it was called. However,

this restriction can be overcome by storing such values in the belief base, and using test goals to retrieve them. Hence, syntactic mechanisms for dealing with this restriction can be implemented (i.e. this problem can be solved by pre processing). With respect to the second restriction, we point out that this was not allowed in Rao's original definition of AgentSpeak(L), so the second restriction is not an unreasonable one.

Further, the translation to PROMELA in particular also requires a series of parameters to be determined by the user. These include things such as the maximum expected number of intentions to be held by the agent at one time, expected maximum number of beliefs, and so forth; see [6] for details. This is necessary as all PROMELA data structures must have a static size.

Some of the additions to the basic AgentSpeak language are as follows. There are some special action symbols which are denoted by an initial '.' character, and they are referred to as *internal actions*. Some such actions are pre defined and can be used for things such as printing console messages and arithmetic expressions. The action '.send' is used for inter-agent communication, and is interpreted as follows. If an AgentSpeak(F) agent l_1 executes .send(l_2, ilf, at), a message will be inserted in the mailbox of agent l_2, having l_1 as sender, illocutionary force *ilf*, and propositional content *at* (an atomic AgentSpeak(F) formula). At this stage, only three illocutionary forces can be used: tell, untell and achieve (unless others are defined by the user). They have the same informal semantics as in the well-known KQML agent communication language [33]. In particular, achieve corresponds to including *at* as a goal addition in the receiving agent's set of events; tell and untell change the belief base and the appropriate events are generated. These communicative acts only change an agent's internal data structures after user-defined trust functions are checked. There is one specific trust function for belief changes, and another for achievement goals. The latter defines a power relation (as other agents have power over an agent's goals), whereas the belief trust function simply defines the trustworthiness of information sources.

Finally, we remark that the multi-agent system is specified by the user as a collection of AgentSpeak(F) source files, one for each agent in the system. The user can change various predefined functions which are part of the interpretation of AgentSpeak agents. Also, the user has to provide the environment where the agents will be situated; this must be done in the model language of the model checker itself, rather than AgentSpeak(F).

2.4.2 Property specification language

In the context of verifying multi-agent systems implemented in AgentSpeak, the most appropriate way of specifying the properties that the system satisfy (or do not satisfy) is by expressing those properties using a temporal logic combined with modalities for referring to agent's mental attitudes, such as BDI logics [40, 54]. In this section, we review how simple BDI logical properties can be mapped onto LTL formulæ and associated predicates over the AgentSpeak data structures in the system.

In [8], a way of interpreting the informational, motivational and deliberative modalities of BDI logics for AgentSpeak agents was given; this is based on the operational semantics of AgentSpeak given earlier in Section 2.2. We adopt the same framework for interpreting the BDI modalities in terms of data structures within the model of an AgentSpeak(F) agent given in the model checker input language. In this way, we can translate (temporal) BDI properties into LTL formulæ. The particular logical language that is used for specifying such properties is given later in this section.

As the interpretation is based on the operational semantics of AgentSpeak, it may help to recall some of the notation used in it. The configurations of the transition system giving such operational semantics are defined as a tuple $\langle ag, C, T, s \rangle$, where an agent program *ag* is defined as a set of beliefs *bs*

and a set of plans ps, and C is the agent's present circumstance defined as a tuple $\langle I, E, A\rangle$ containing a set of intentions S, a set of events E and a set of actions A, all as defined in Section 2.1 (the others components are not relevant here).

We give here only the main definitions in [8]; the argumentation on the proposed interpretation is omitted. In particular, that paper provides further details on the interpretation of intentions and desires, as the *belief* modality is clearly defined in AgentSpeak.

DEFINITION 2.4 (Beliefs)
We say that an AgentSpeak agent ag, regardless of its circumstance C, believes a formula φ if, and only if, it is included in the agent's belief base; that is, for an agent $ag = \langle bs, ps\rangle$:

$$\mathrm{Bel}_{\langle ag,C\rangle}(\varphi) \quad \equiv \quad \varphi \in bs.$$

Note that a closed world is assumed, so $\mathrm{Bel}_{\langle ag,C\rangle}(\varphi)$ is true if φ is included in the agent's belief base, and $\mathrm{Bel}_{\langle ag,C\rangle}(\neg\varphi)$ is true otherwise, where φ is an atom (i.e. *at* in Section 2.4.1).

Before giving the formal definition for the *intention* modality, we first define an auxiliary function $agls : \mathcal{I} \to \mathcal{P}(\Phi)$, where $\mathcal{I}$ is the domain of all individual intentions and Φ is the domain of all atomic formulæ (as mentioned above). Recall that an intention is a stack of partially instantiated plans, so the definition of $\mathcal{I}$ is as follows. The empty intention (or true intention) is denoted by T, and $\mathsf{T} \in \mathcal{I}$. If p is a plan and $i \in \mathcal{I}$, then also $i[p] \in \mathcal{I}$. The notation $i[p]$ is used to denote the intention that has plan p on top of another intention i, and C_E denotes the E component of C (and similarly for the other components). The *agls* function below takes an intention and returns all achievement goals in the triggering event part of the plans in it:

$$\begin{aligned} agls(\mathsf{T}) &= \{\} \\ agls(i[p]) &= \begin{cases} \{at\} \cup agls(i) & \text{if } p = \texttt{+!}at\ \texttt{:}\ ct\ \texttt{<-}\ h\texttt{.} \\ agls(i) & \text{otherwise.} \end{cases} \end{aligned}$$

DEFINITION 2.5 (Intentions)
We say an AgentSpeak agent ag intends φ in circumstance C if, and only if, it has φ as an achievement goal that currently appears in its set of intentions C_I, or φ is an achievement goal that appears in the (suspended) intentions associated with events in C_E. For an agent ag and circumstance C, we have:

$$\mathrm{Int}_{\langle ag,C\rangle}(\varphi) \quad \equiv \quad \varphi \in \bigcup_{i \in C_I} agls(i) \ \vee \ \varphi \in \bigcup_{(te,i) \in C_E} agls(i).$$

Note that we are only interested in triggering events that have the form of additions of achievement goals; we ignore all other types of triggering events. The atomic formulæ *at* within those triggering events are the formulæ that represent (symbolically) properties of the states of the world that the agent is trying to achieve (i.e. the intended states). However, taking such formulæ from the agent's set of intentions does not suffice for defining intentions, as there may also be *suspended* intentions. Suspended intentions are precisely those that appear in the set of events.

We are now in a position to define the interpretation of the *desire* modality in AgentSpeak agents.

DEFINITION 2.6 (Desires)
We say an AgentSpeak agent ag desires φ in circumstance C if, and only if, φ is an achievement goal in C's set of events C_E (associated with any intention i), or φ is a current intention of the agent; more

formally:

$$\mathrm{Des}_{\langle ag,C\rangle}(\varphi) \quad \equiv \quad (+!\varphi, i) \in C_E \ \vee \ \mathrm{Int}_{\langle ag,C\rangle}(\varphi).$$

Although this is not discussed in the original literature on AgentSpeak, it was argued in [8] that the *desire* modality in an AgentSpeak agent is best represented by additions of achievement goals presently in the set of events, as well as its present intentions.

The definitions above tell us precisely how the BDI modalities that are used in claims about the system can be mapped onto the AgentSpeak(F) structures implemented either as a PROMELA or Java model. We next review the logical language that is used to specify properties of the BDI multi-agent systems written in AgentSpeak(F).

The logical language we use here is a simplified version of $\mathcal{LORA}$ [54], which is based on modal logics of intentionality [12, 40], dynamic logic [20] and CTL* [1]. In the restricted version of the logic used here, we limit the underlying temporal logics to LTL rather than CTL*, given that LTL formulæ (excluding the 'next' operator $\bigcirc$) can be automatically processed by our target model checkers. Other restrictions, aimed at making the logic directly translatable into LTL formulæ, are described below.

DEFINITION 2.7 (Property specification language)
Let l be any agent label, x be a variable ranging over agent labels, and *at* and a be, respectively, atomic and action formulæ defined in AgentSpeak(F) (see Section 2.4.1). Then the set of well-formed formulæ (*wff*) of this logical language is defined inductively as follows:

- (Bel l *at*), (Des l *at*), (Int l *at*), (Does l a), and *at* are *wff*;
- if φ and ψ are *wff*, so are $(\neg\varphi)$, $(\varphi \wedge \psi)$, $(\varphi \vee \psi)$, $(\varphi \Rightarrow \psi)$, $(\varphi \Leftrightarrow \psi)$, always $(\Box\varphi)$, eventually $(\Diamond\varphi)$, until $(\varphi \mathcal{U} \psi)$, and 'release', the dual of until $(\varphi \mathcal{R} \psi)$;
- nothing else is a *wff*.

In the syntax above, the agent labels denoted by l, over which the variable x ranges, are those associated with each AgentSpeak(F) program during the translation process; i.e. the labels given as input to the translator form the finite set of agent labels over which the quantifiers are defined. The only unusual operator in this language is (Does l a), which holds if the agent denoted by l has requested action a and that is the next action to be executed by the environment. An AgentSpeak(F) atomic formula *at* is used to refer to what is *actually* true of the environment. In practical terms, this amounts to checking whether the predicate is in the data structure where the percepts are stored by the environment. Recall that we earlier gave formal semantics to the other modalities above.

In our previous work, the language also allowed the use of any valid Boolean expression in the model specification language of the model checker being used. This can be handy for users experienced in the use of model checkers, but note that they do not operate at the level of the BDI system but on the model generated in the input language of the model checker, thus being of a much lower level of abstraction. Therefore, such expressions could in fact lead to different verification results in a sliced system compared with the original (unsliced) system, if the low level PROMELA/Java expression referred to internal variables that were affected by, e.g. the number of plans in the system. To avoid using cumbersome conditionals in the proofs, and given that it is not important (nor elegant) for our approach to use them, we have now excluded their use from specifications altogether. Also in previous work the language admitted quantification over finite sets of agent (labels) to be used with the four main modalities. This could be dealt with easily in the proofs we give later, but would make them unnecessarily long and again these are not essential

constructs (they can always be expanded to formulæ that do not use them), so we omit them here.

The concrete syntax used in the system for writing formulæ of the language above is also dependent on the underlying model checker. Before we pass the LTL formula on to the model checker, we translate Bel, Des and Int into predicates accessing the AgentSpeak(F) data structures modelled in the model checker's input language (according to the definitions in the previous section). The Does modality is implemented by checking the first action in the environment's data structure where agents insert the actions they want to see executed by the environment process. The first item in such a data structure is the action to be executed next by the environment (as soon as it is scheduled for execution).

2.5 *Slicing logic programs*

One of the earliest papers to discuss slicing for logic programs is that of Zhao *et al.* [56]. The paper presents a graph-theoretical representation of a concurrent logic program, which can be used for slicing. An arc-classified digraph called a *Literal Dependence Net* (LDN) is used to represent four types of dependencies of concurrent logic programs: control, data, synchronization and communication dependencies. Later on, a backward slicing algorithm for Prolog was presented by Schoenig and Ducassé [42]. They propose an algorithm that can do slicing with greater precision than the approach in [56]. Slicing is done at the level of arguments of predicates, so slices are subsets of the clauses of the original programs where also some predicate arguments may have been replaced by *anonymous variables*. Slicing in the context used by those authors is intended at debugging, software maintenance and understanding, and so on. So the more details of a program can be eliminated, the better. As we shall argue later, removing arguments individually is not particularly relevant with respect to slicing for model checking. Another difference between those two approaches is that the work in [42] is intended to produce *executable slices*, which for those software engineering tasks mentioned above, is quite important. Again, this is not particularly relevant in our context here as we aim to use the sliced program for verification only.

An approach similar to Schoenig and Ducassé's was introduced by Vasconcelos and Aragão [49]; both approaches apply slicing to Prolog programs at the level of predicate arguments, and generate *executable* slices. An advantage of the work by Vasconcelos and Aragão is that they proved correctness of their slicing algorithms. Also, they explicitly mention an implementation of their algorithm and all the necessary preparation (e.g. mode annotations and generation of dependence graphs).

The approaches by Schoening and Ducassé [42] and Vasconcelos and Aragão [49] both work for Prolog programs. Although AgentSpeak is quite similar to Prolog in many respects, which suggests that we might be able to base our algorithm on those approaches, an AgentSpeak plan has in fact essentially the same structure of a guarded clause. Zhao *et al.* [56] proposed a slicing algorithm that is specific to Guarded Horn Clauses, so their approach is a better candidate as a basis for ours. Again, we do not need to generate executable slices (an important motivation in both [42] and [49]), as we are only interested in preserving the truth of certain properties of the system. Further, the graph-based approach in [56] provides a very clear algorithm. These are the reasons why we have chosen to base our approach on theirs.

More recently, Zhao *et al.* [57] extended their approach, using what they call an Argument Dependence Net. They use the same principles as in their previous work, but refine the program representation to have annotations on dependencies at the level of arguments rather literals. For our

purposes here, Zhao's early work suffices, as we do not need slicing at the level of arguments. Instead of a slice for a particular variable, as usual in approaches related to software engineering, we here aim to remove whole plans based on their influence on the truth of a whole predicate (under certain modalities).

Note that neither [57] nor [42] prove the correctness of their slicing methods. A recent paper by Szilágyi *et al.* [44] presents both static and dynamic slicing techniques for constraint logic programs which they prove to produce slices with respect to variables in the program. Constraint logic programs generalize logic programs, thus in principle their approach could be used for our purposes too. Although they also present a static technique, they concentrate on dynamic slices (see also [43]), by defining slices of sets of constraints, mapping them to slices of the program's proof tree, and finally mapping those to slices of the program itself. Their static slicing technique is very elegant, but does not take into consideration all the details that Zhao *et al.* do, and thus would generate less efficient slices. Also, in our context (slicing for model checking), it is important to produce static, rather than dynamic, slices.

Thus, in this article we adopt the technique presented in [56] as a basis for our slicing algorithm for AgentSpeak. Note that their work is intended for concurrent logic programs, where body literals are AND processes, different clauses of a procedure are OR processes, shared variables relate to process communication and synchronization, etc. However, all such dependencies apply to any logic program, as the authors of that paper observe themselves. Although we are not dealing with concurrent logic programs of this kind, the reader may consider the terms used in their algorithm (such as 'communication dependencies') as metaphors for dependencies that we also have to deal with.

2.6 *Generating literal dependence graphs*

Here, we summarize the approach presented in [56], which will be used as a basis for the algorithm we introduce in Section 3. It is heavily based on two representations of a logic program. The first, called an *And/Or Parallel Control-Flow Net* (CFN), is an arc-classified digraph (directed graph) where control-flow dependencies are annotated. The second is called a *Definition-Use Net* (DUN), and contains annotations on data dependencies.

In a CFN, vertices are used to represent the heads, guards and each literal in the bodies of the clauses in the program. Execution arcs (both AND-parallel and OR-parallel) as well as sequential control arcs are used to denote control flow information. The generation of such a CFN can be understood informally from the rules presented in Figure 3; observe in particular how literals in the body of a clause generate AND-parallel arcs (Figure 3b) and how alternative clauses for the same literal generate OR-parallel arcs (Figure 3c).[2] Note that, as we will be dealing with slicing sets of AgentSpeak plans (each plan having the same structure as a guarded clause), we have not reproduced here the rules given in [56] for unit clauses and goal clauses, as these are not relevant for our slicing algorithm.

As noted above, we also need to annotate a logic program (based on the approach used in concurrent logic programming) with data, synchronization and communication dependencies among literals. For this, another structure is needed, the so called DUN. Its definition requires four functions: D determines the variables *defined* at each vertex, U determines the variables *used* at each vertex, S determines the set of channel variables *sent* at each vertex and R determines the set of channel

[2] Both OR- and AND-parallelism represent different types of opportunities for automatically exploring parallelism in logic programs. What is important here is that they represent control-flow dependencies between literals in a logic program

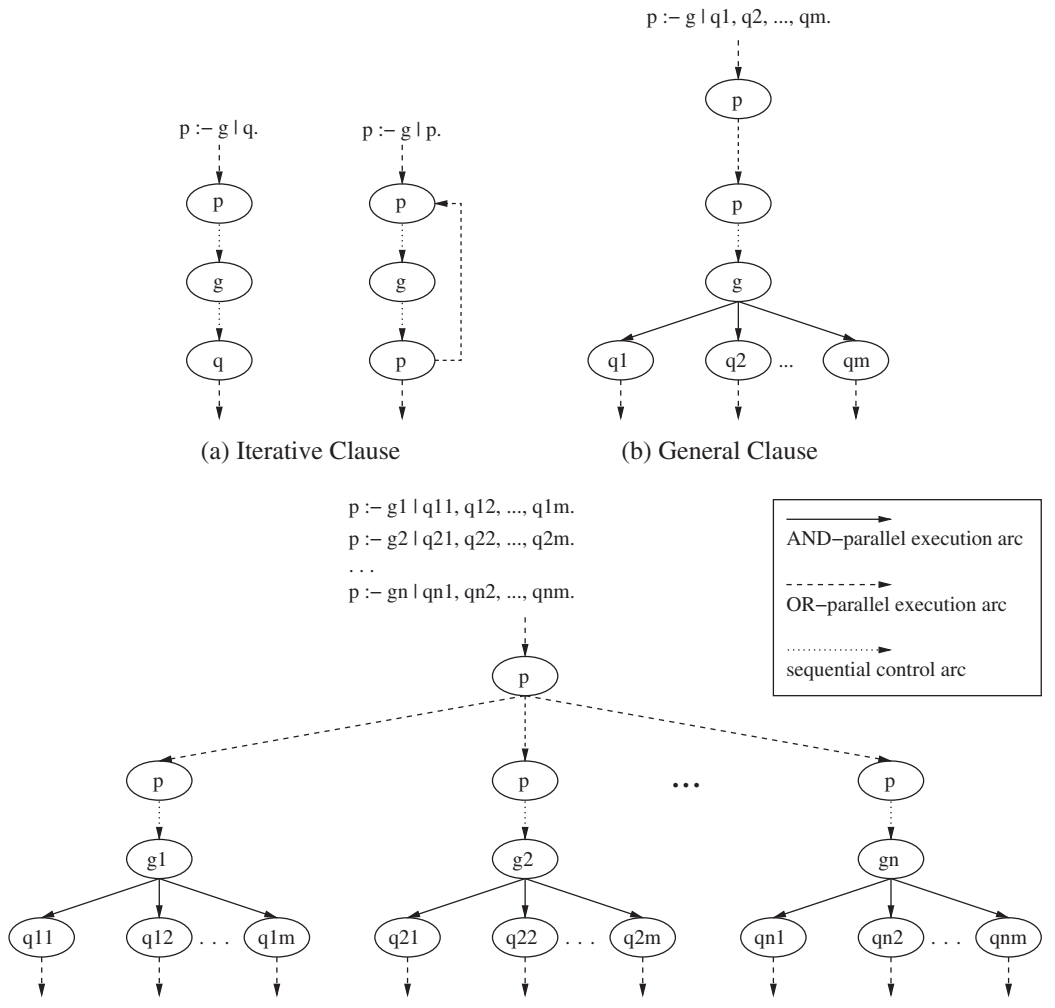


FIGURE 3. CFN generation rules [56].

variables *received* at each vertex. Functions D and U are determined by *mode inference* (Zhao *et al.*, in their later work, use the approach proposed in [27]); mode inference for logical variables is done by abstract interpretation.

A form of control dependence in a concurrent logic program occurs when clauses share the same head literal. This is called *selective control dependence* in [56]. Its definition uses the CFN to determine whether two literals are directly selective-control dependent. Two vertices can be also *directly data dependent*. Zhao *et al.* use the DUN to define a data-dependence relation between literals. *Synchronization* in concurrent logic programs relates to two types of dependencies in logic programs in general: dependencies between the guard (or the head literal if the guard is empty) and the body literals, or between body literals that share logical variables. Similarly, *communication* in concurrent logic programming captures data dependencies between literals in different clauses.

The definition of LDN is then an arc-classified digraph containing all four types of dependencies mentioned above (control, data, synchronization, and communication).

A *static slicing criterion* is defined in [56] as a pair, $\langle l, V\rangle$, where l is a literal in the program and V is a set of variables that appear in l. The *static slice* $SS(l, V)$ of a logic program given a static slicing criterion $\langle l, V\rangle$ is the set of all literals in that program which possibly affect the execution of l and/or affect the values with which variables in V are instantiated. Interestingly, once the LDN of a logic program is built, a static slice can be determined simply by solving a reachability problem in the LDN arc-classified digraph. In the algorithm we present in the next section, we will use an LDN to represent control-flow and data dependencies between literals in a logic program; the details of *how* the LDN is created are quite involved. Therefore, for further details and formal definition of the LDN structure and its construction, we refer the interested reader to [56].

3 Slicing AgentSpeak

In contrast to the slicing approaches mentioned above, in our work we are interested in property-based slicing. That is, instead of finding a slice for a particular variable of a particular literal in a logic program, we here need to be able to slice the agent system specification based on a given property. Slicing should be such that the result of subsequent model checking of the sliced system for that property will give identical results to the checking of that property on the original system. The aim is, or course, to make model checking more efficient for the sliced system by reducing its state space.

We use the same dependence annotations as in the usual form of slicing logic programs in an algorithm for slicing AgentSpeak given a certain specification that a multi-agent system is supposed to satisfy. The algorithm is presented next, then we give some abstract, illustrative examples; we also give proof sketches for correctness and complexity results.

3.1 Stages of a slicing method for AgentSpeak

In our approach, a system is specified as a set of AgentSpeak programs, one for each agent, and an abstract representation of the environment. The environment is abstractly represented as a set of *initial facts* ('fact' here is being used with the same meaning as in Prolog terminology), determining the initial state of the environment, and a set of rules stating which facts are changed when agents execute each particular action (or spontaneously in case of 'dynamic environments'). Note that changes in the sate of the environment may then come to alter agents' beliefs through perception of the environment.

The environment dynamics is thus abstractly represented by a set of rules with one agent action in the left-hand side and a sequence of possible percept changes, in the form of addition or deletion of predicates, in the right-hand side. Syntactically, this is represented as, for example, '$a_1 \rightarrow +p_1; -p_2$.'. Recall that, in AgentSpeak notation, $+p$ means the addition of belief p and $-p$ means its deletion; similarly, in an abstract environment description, we use this to describe changes in the state of the environment, characterized by a set of environment facts which determine the percepts that agents acquire when perceiving the environment. In the case of dynamic environments, rules can have an empty left-hand side, to denote that those environment-state changes can happen at any time, regardless of whether agents are executing any actions or not. Importantly, for our purposes we assume that common aspects of multi-agent systems such as the change in beliefs due to inter-agent communication, or changes in beliefs caused by faulty perception, are incorporated as appropriate non-determinism in the given representation of the environment.

As well as the system specification, the property for which a slice will be obtained (and will later be used for model checking) also needs to be given. This is specified in the BDI logic defined in Section 2.4. The input to an AgentSpeak slicer is thus a finite set of AgentSpeak programs A, the abstract environment E and the property P for which the slice is to be obtained. Our slicing method then works in three stages, as described below.

Stage I: at this stage, the LDN for the system is created, according to the algorithm by Zhao *et al.* [56], discussed in Section 2.5. When matching literals in different parts of the programs, the AgentSpeak notations such as '**+**', '**−**', '**!**' and '**?**' are considered to be part of the predicate symbol. The only extra care to be taken in such matching is that a $\mathbf{!}g$ in the body of a plan should match a $\mathbf{+!}g$ in the triggering events of plans—refer to rule **AchvGl** (then further to rules $\mathbf{SelEv_1}$, $\mathbf{Rel_1}$, $\mathbf{Appl_1}$, **SelAppl** and **IntEv**) of the operational semantics to see that, when a course of action in the body of a plan has an achievement goal g (i.e. $\mathbf{!}g$) to be achieved, this will generate an event for a goal addition (i.e. $\mathbf{+!}g$) which, when selected in a later reasoning cycle, will be matched with a plan whose triggering event is $\mathbf{+!}g$ (in order to attempt to achieve that goal).
Initially, an LDN is created for each individual AgentSpeak program. Then the environment LDN and the various agent LDNs are connected as follows:

(1) In the environment specification, for each rule, edges are added from the left-hand side to each percept change in the right-hand side.
(2) Create edges from action predicates in the plan bodies (of all agents) to the left-hand side of the environment rules. In the case of environment rules with empty left-hand sides, we have to create links from at least one node in the body of each plan in the system to the beliefs in those rules, as these belief changes can always happen regardless of what actions are performed.
(3) For each percept change within the environment's initial facts, or in the right-hand side of environment rules, create edges from it to all matching triggering events in the plans of all agents.

An example system specification and its corresponding LDN is shown in Figure 4. In the figure, most plan contexts (i.e. guards) are omitted for the sake of clarity.
In order to make the algorithm for the next stage clearer, we introduce the following terminology for the nodes of the LDN created for the individual AgentSpeak programs. We call a t-node any node of the LDN that was created for the triggering event[3] of a plan, a c-node any node created from literals in the context of the plan and b-node any node created from body literals.

Stage II: once the LDN is created, at Stage II plans are marked according to Algorithm 1. It takes, as input: the system specification (`System`), i.e. the set of AgentSpeak programs A and the environment representation E; the LDN generated for the previous stage (`LDN`); and the property that one intends to later model check (`Property`).

Stage III: at this stage, a 'slice' of the system is obtained by simply deleting all plans that were *not marked* in Stage II. If it happens that all plans of an agent are deleted, then the whole agent can be safely removed from the system, as that agent will have no effect in checking whether the system satisfies the given property. We also remove a goal appearing in the body of a plan if all relevant plans for that goal were deleted at this stage.

[3] Recall that a plan's triggering event is equivalent to the head of a Guarded Horn Clause, and a plan's context is equivalent to the guard of the clause.

ALGORITHM 1 Marking plans given `System`, `LDN`, `Property` (Stage II of the AgentSpeak Slicing Method)

for all subformula f of `Property` with Bel, Des, Int, or Does modalities or an AgentSpeak atomic formula **do**
  for all agent ag in the `System` **do**
    for all plan p in agent ag **do**
      let te be the node of the `LDN` that represents the triggering event of p
      if $f = (\mathsf{Bel}\ ag\ b)$ **then**
        for all b-node b_i labelled $+b$ or $-b$ in ag's plans, or in the facts and right-hand side of rules in the *Environment* **do**
          if b_i is reachable from te in `LDN` **then**
            mark p
      if $f = (\mathsf{Des}\ ag\ g)$ **then**
        for all b-node g_i labelled $!g$ in ag's plans **do**
          if g_i is reachable from te in `LDN` **then**
            mark p
      if $f = (\mathsf{Int}\ ag\ g)$ **then** {note t-node below, rather than b-node}
        for all t-node g_i labelled $!g$ in ag's plans **do**
          if g_i is reachable from te in `LDN` **then**
            mark p
      if $f = (\mathsf{Does}\ ag\ a)$ **then**
        for all b-node a_i labelled a in ag's plans **do**
          if a_i is reachable from te in `LDN` **then**
            mark p
      if f is an AgentSpeak atomic formula b not in the scope of the modalities above {meaning b is true of the *Environment*} **then**
        for all node b_i labelled $+b$ or $-b$ in the facts and right-hand side of rules in the *Environment* **do**
          if b_i is reachable from te in `LDN` **then**
            mark p

3.2 *Examples*

For the example shown in Figure 4, and `Property` $= \Diamond(\mathsf{Des}\ ag1\ g2)$, all plans are marked after checking for reachability from each of the nodes representing the triggering events of all plans to the only instance of $!g2$ in the body of $ag1$'s plans. As all plans are marked, this means that for this particular set of programs and given property, slicing would not eliminate *any* part of the original code. Now consider a similar example, in which only the body of $ag2$'s last plan is changed from `a1` to `a3`, as shown in Figure 5.

For this second abstract example, and the same `Property` $= \Diamond(\mathsf{Des}\ ag1\ g2)$, Table 1 shows which plans are marked after checking reachability from each of the nodes representing the triggering events of all plans to the only instance of $!g2$ in the body of $ag1$'s plans. In the table, a plan is referred to by its triggering event, which is in this particular example is unambiguous.

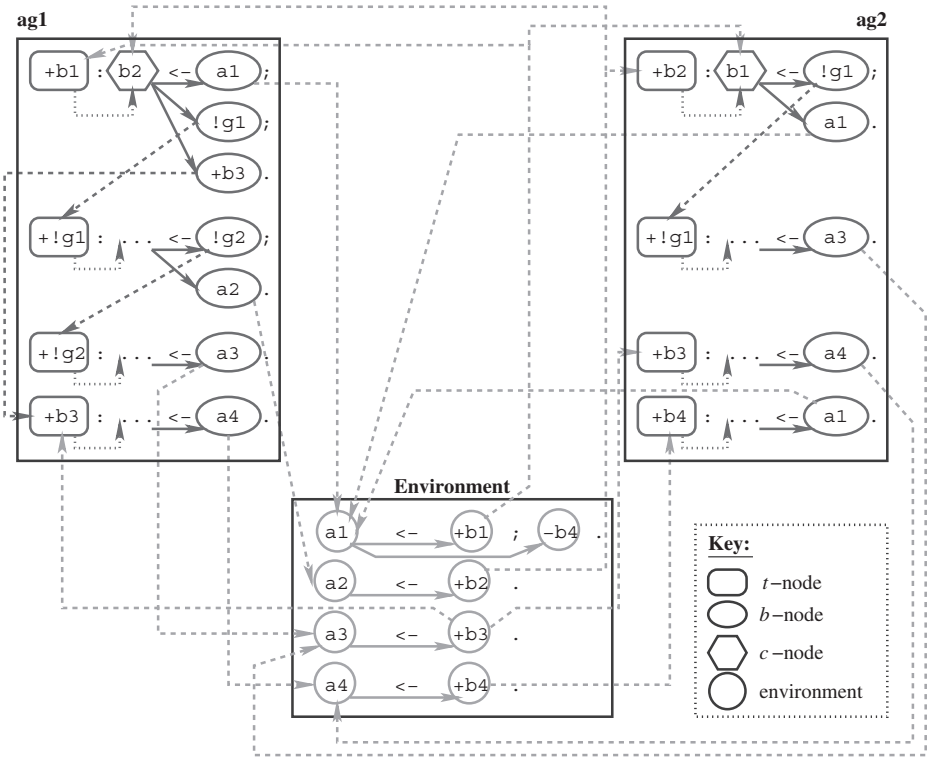


FIGURE 4. Abstract example I.

From the table, we see that, in the slice produced by our algorithm, only the plans with triggering events $+b1$ and $+!g1$ remain for $ag1$, and only plan $+b2$ remains for $ag2$. Model checking for this property can be done on the translation to Java or SPIN from this particular slice of the system.

Although it may be counter-intuitive that a plan for $+!g2$ is left out of the slice even though $g2$ appears in the property, that is correct according to the interpretation we have given to the Des modality (see Definition 2.6). By that definition, to desire g an agent does not need a plan for it; having g as an achievement goal in the body of any plan is all that is necessary for g to (possibly) become desired. For g to be intended rather than desired, then a plan for it is necessary (in practice, an applicable plan). So, counter-intuitive though it may be, although $g2$ (with Des) appears in the property, the only plan for it (i.e. having $g2$ in its triggering event) is left out of the slice generated using that property as the slicing criterion.

It may seem, at first sight, that the difference in the algorithm for Des and Int modalities has no impact on the generated slice. However, consider the example in Figure 6.

For this example and property (Des $ag1$ $g1$), both the plan with $+!g1$ as triggering event, and the one with $+b2$ for triggering event can be eliminated. This is not the case for property (Int $ag1$ $g1$)

3.3 Correctness and complexity

In this section, we prove correctness and give broad complexity results for our slicing algorithm. We first make clear what we mean by correctness of the slicing algorithm, in the following definition.

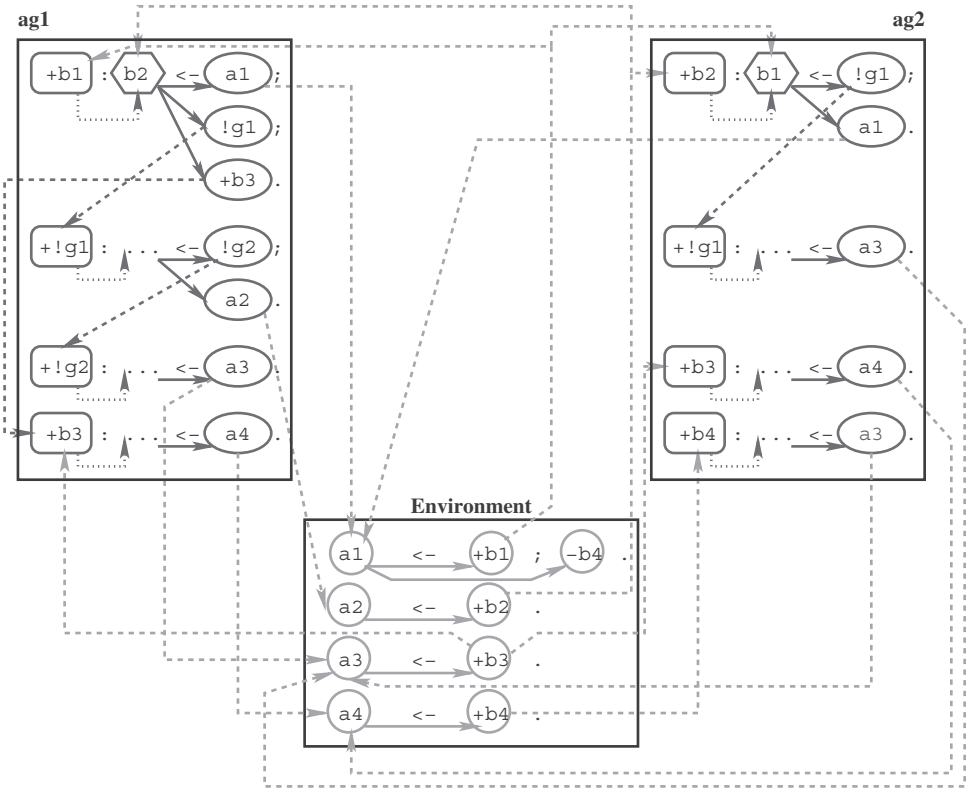


FIGURE 5. Abstract example II.

TABLE 1. Marked plans in example 2 after stage II of the algorithm

***ag*1's plans**		***ag*2's plans**	
$+b1$	$\surd$	$+b2$	$\surd$
$+!g1$	$\surd$	$+!g1$	
$+!g2$		$+b3$	
$+b3$		$+b4$	

Recall that a system in our approach is a set of AgentSpeak programs A situated in an environment E; the slicing algorithm takes A, E and a property P (which is later to be model checked) as arguments and returns A', a set of AgentSpeak programs that are sliced down from A. As usual in model checking, $\mathcal{M} \models \varphi$ means that the initial states of the system of which $\mathcal{M}$ is a model satisfy formula φ. We use A, E to refer to the model of the multi-agent system defined by A and E, and we use $A, E \models_s \varphi$ to say that φ is true at state s of the A, E model.

DEFINITION 3.1 (Slicing correctness)
An AgentSpeak slicing algorithm σ is *correct* if for any finite set of AgentSpeak programs A, abstract environment E, property P and $A' = \sigma(A, E, P)$, we have that $A, E \models P$ if and only if $A', E \models P$.

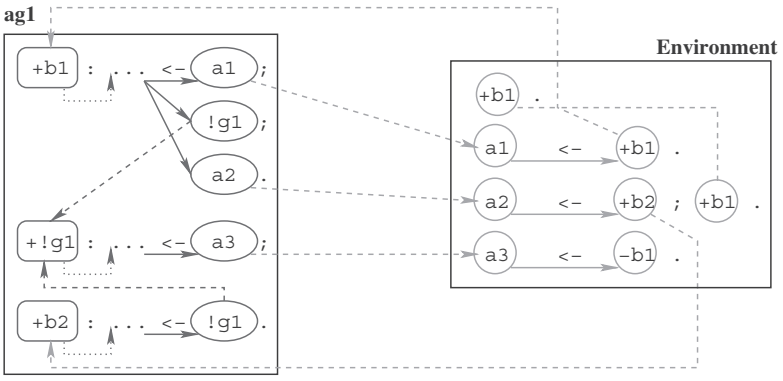


FIGURE 6. Abstract example III.

In order to prove that our algorithm is correct according to the definition above, we depend on the LDN constructed in Part I which captures all dependencies between any plans in the multi-agent system (i.e. the plans of all agents). Recall that, initially, the algorithm determines that the LDN for *each agent individually* is generated according to the algorithm given in [56] (see Section 2.5), each agent being viewed as a Guarded Horn Clause program. The correctness of our algorithm actually relies on the lemma below which is a corollary of the following conjecture.

CONJECTURE 3.2
The algorithm in [56], which uses reachability in a LDN's directed graph to generate slices of logic programs, is correct.

LEMMA 3.3
If there is a path in the LDN's directed graph from vertex v_1 to v_2, then there exists an execution path in the logic program that generated the LDN in which the execution of the literal represented by v_1 (and the instantiation of its variables) can lead to the execution (and affect the instantiation) of the literal represented by v_2.

Note that the above lemma would be a corollary of a correctness theorem of the slicing algorithm based on an LDN (see Section 2.5) and given the fact that a *slice*, defined as the set of all literals that can affect the execution or the contents of the variables in the literal given in the slicing criteria, is determined through a reachability problem to that particular literal in the LDN. We did not find in the literature a correctness proof for the particular algorithm we used here (hence we leave it as a conjecture), but there are such proofs for similar algorithms, e.g. the one in [49]. Therefore, although we here rely on this conjecture, this is not unreasonable as we could have used those alternative algorithms, which would provide us with the same dependency information, but with known correctness results. The choice for this particular approach, despite the lack of correctness results, was primarily because it yields a much clearer presentation of the ideas in this article.

Further, we need to show that the construction of the LDN for the whole multi-agent system maintains this property which is conjectured to follow from the algorithm on an LDN of an individual agent. In a multi-agent system, agents' actions change the environment and agents perceive changes in the environment. The rules in the abstract representation of the environment determine how actions (or non-deterministic moves of the environment) affect properties perceived by the agents. It is only

through this mechanism that one agent program can affect the particular plans executed in another. The combination of the individual LDNs in the algorithm is achieved in three steps:

(1) simply connect the two parts of each rule, in the obvious way;
(2) ensure that all actions in an agent are appropriately connected to some environment rule (its left-hand side specifically)—this step also takes care of non-deterministic moves of the environment; and
(3) connect the environment facts changed by actions to all relevant plans (specifically the plan triggers) in the agents.

We now have to show that the main part of our algorithm (Part II, and its finalization in Part III), which uses the LDN structure discussed above, is correct in the sense of Definition 3.1. This will be done by showing that the model of a system before slicing and the model of the system resulting from our slicing technique are *stuttering equivalent* with respect to the labelling of atomic formulæ used within the slicing criterion—this is appropriate for our purpose because our property specification is built on top of LTL without the 'next time' operator and it is known that any such temporal formula is *invariant under stuttering* (i.e. it has the same truth value in models that are stuttering equivalent). We introduce the main notions below, but for details on stuttering equivalence, see [11, p. 146], where stuttering equivalence is used to prove correctness of partial order reduction algorithms; in particular, see Theorem 9 on that page, which shows that any LTL formula without the next-time operator is invariant under stuttering.

Let a *block* be defined as segments of sequential states in a path (of a Kripke structure) where all the states are identically labelled.[4] Two infinite paths are stuttering equivalent if they can be partitioned into infinitely many such blocks, so that for each block in one path there is a corresponding block—possibly of different length—in the other path that is labelled equivalently. Generalizing this to models, we have that two models are stuttering equivalent if they have the same set of initial states and for every path in one of the models starting in an initial state there is a stuttering equivalent path in the other model, and vice-versa.

In particular, the correctness of our slicing algorithm is a corollary of a theorem showing that the model obtained by slicing is stuttering equivalent to the original model *with respect to labelling that is relevant for the slicing criterion*. Figure 7 shows an example of such stuttering equivalent structures (for the time being, ignore the annotations '(1)' and '(2)' in the diagram). In order to obtain the stuttering equivalence result, we first prove five lemmata for each of the basic cases of formulæ of the property specification language in our approach. In the lemmata, to make the presentation clearer we do not consider explicitly the existence of multi-agent communication and faulty perception[5] in the interpretation of AgentSpeak agents; i.e. beliefs are only changed from the changes determined by the environment rules, and goals derive from such changes (rather then, e.g. requests from other agents). In the lemmata below, for a given formula φ, let A' be $\sigma(A,E,\varphi)$ for a finite set of AgentSpeak programs A and environment E, where σ is Algorithm 1 (i.e. our slicing algorithm).

LEMMA 3.4 (Stuttering equivalence with respect to (Bel *ag b*))
Models A,E and A',E, where $A'=\sigma(A,E,\varphi)$ and (Bel *ag b*) is a subformula of φ, are stuttering equivalent with respect to the labelling of (Bel *ag b*) atomic fomulæ.

[4] Recall that the labelling function in a Kripke structure determines the atomic propositions which are true in that state.

[5] Note that we can avoid explicit communication and issues related to faulty perception because this can be assumed as having been represented in the rules given to describe the environment (which, recall, allow for non-determinism). It is straightforward, for example, for one to use the abstract specification of the environment to model beliefs that are added from actions that intuitively represent inter-agent communication.

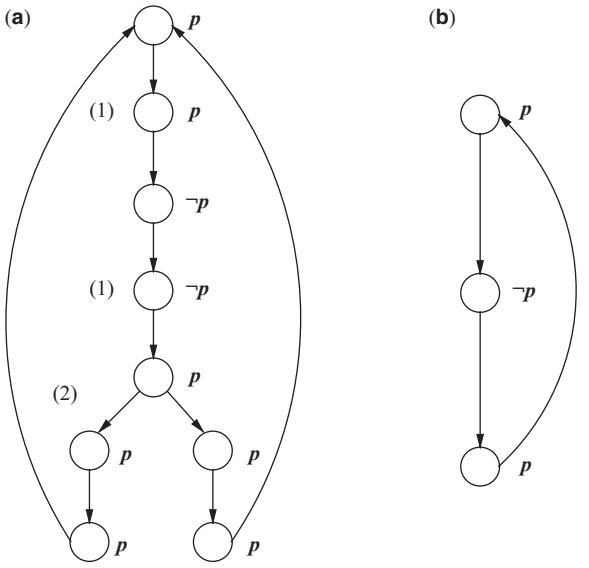


FIGURE 7. Two stuttering equivalent models with respect to an atomic proposition p. (**a**) Model of a system before slicing. (**b**) Possible model after slicing.

PROOF. A belief formula (Bel ag b) can only become true (respectively, false) under two circumstances: (i) when $+b$ (respectively, $-b$) appears in the body of one of ag's plans (for $ag \in A$), or (ii) by belief update based on the agent's perception of the environment, as determined by the rules in E. The first case is justified as follows. Considering the meaning of the Bel modality as per Definition 2.4, we see that a formula (Bel ag b) can only become true or false by b being removed or added to the belief base (i.e. the ag_{bs} component of a configuration of the transition system giving the operational semantics; see Section 2.2). From an analysis of the semantic rules, it is clear that the only rules of the semantics which change that component are **AddBel** (respectively, **DelBel**), and they do so whenever $+b$ or $-b$ appear in the body of a plan, precisely as referred to in case (i) above. Case (ii) is based purely on the understanding of how an agent perceives the environment and its abstract representation in E, which contain rules with $+b$ and $-b$ formulæ on their right-hand side.

Cases (i) and (ii) happen precisely at points in the program represented by nodes b_i in Algorithm 1, to which reachability is checked from each plan's triggering event (the head of the plan, which connects the remainder of the plan in the graph) whenever (Bel ag b) is a subformula of property P. Then, given Lemma 3.3, all plans that can lead the program to such control points, or affect the values bound to variables used in such parts of the programs, have paths in the LDN that reach nodes b_i, and hence are marked in the loop at line 1 of the algorithm. As these plans are marked in Stage II, they are *not* removed in Stage III of the algorithm, and are therefore kept in A'.

This effectively means that in all execution paths in model A,E, any state transition from a state where a formula (Bel ag b) is true to another state where it is false (or vice-versa) also exists in A',E. Therefore, what any removed plan did in the original model was to possibly increase the length of a block in which all states are labelled consistently for (Bel ag b) being either true or false, by changing the truth of other atomic properties other than (Bel ag b). As the definition of stuttering

equivalence is precisely that the length of equally labelled blocks is irrelevant, we have that, with respect to atomic properties (Bel $ag\ b$) specifically, A,E and A',E are stuttering equivalent. ■

In the lemmata below, whenever we say the proof is similar to the proof of Lemma 3.4, we refer exclusively to case (i), as case (ii) in that proof is specific to beliefs.

LEMMA 3.5 (Stuttering equivalence with respect to (Des $ag\ g$))
Models A,E and A',E, where $A'=\sigma(A,E,\varphi)$ and (Des $ag\ g$) is a subformula of φ, are stuttering equivalent with respect to the labelling of (Des $ag\ g$) atomic fomulæ.

PROOF. A formula (Des $ag\ g$), according to Definition 2.6, can only become true when $!g$ appears in the body of ag's plans, as can be seen in Rule **AchvGl**, and false when the chosen plan (which is directly reachable and therefore not removed) is finished (Rules **ClrInt**). The remainder of the proof is similar to the proof of Lemma 3.4, considering reachability to b-nodes g_i in the loop at line 1 of the algorithm (i.e. nodes representing body literals that are labelled $!g$). ■

LEMMA 3.6 (Stuttering equivalence with respect to (Int $ag\ g$))
Models A,E and A',E, where $A'=\sigma(A,E,\varphi)$ and (Int $ag\ g$) is a subformula of φ, are stuttering equivalent with respect to the labelling of (Int $ag\ g$) atomic fomulæ.

PROOF. A formula (Int $ag\ g$), according to Definition 2.5, becomes true when a plan with triggering event $+!g$ is added to the set of intentions, in Rule **IntEv**. The remainder of the proof is similar to Lemma 3.5, considering reachability to t-nodes g_i in the loop at line 1 of the algorithm (i.e. nodes representing triggering events that are labelled $+!g$). ■

LEMMA 3.7 (Stuttering equivalence with respect to (Does $ag\ a$))
Models A,E and A',E, where $A'=\sigma(A,E,\varphi)$ and (Does $ag\ a$) is a subformula of φ, are stuttering equivalent with respect to the labelling of (Does $ag\ a$) atomic fomulæ.

PROOF. The effect of an agent performing an action in the shared environment, is to change the state of the environment. An agent performing an action is abstractly represented in the semantics of the programming language by Rule **Action**; it is assumed that the agent's overall architecture, including its effectors, will take care of actually executing actions in A. In the abstract representation of the environment used here, whenever an agent executes an action, this triggers the left-hand side of an environment rule, and it is precisely at this point in a trace of the system that a formula (Does $ag\ a$) is true. This proof is then similar to the others, considering b-nodes a_i in the loop at line 1 of the algorithm (i.e. nodes representing the left-hand side of environment rules that are labelled a). ■

LEMMA 3.8 (Stuttering equivalence with respect to environment facts b)
Models A,E and A',E, where $A'=\sigma(A,E,\varphi)$ and AgentSpeak atomic formula b is a subformula of φ, are stuttering equivalent with respect to the labelling of AgentSpeak atomic formulæ b.

PROOF. Within a property φ, an AgentSpeak atomic formula (a predicate not in the scope of any modality) represents something that is (objectively) true of the environment, rather than from the point of view of a particular agent. In the abstract representation of the environment we are considering, this can only be the case if that predicate appears (as a literal 'addition') in the right-hand side of one of the environment rules. This proof is similar to the others, considering reachability to nodes representing the right-hand side of environment rules that are labelled $+b$, and line 1 of the algorithm. ■

Using the five lemmata for each base case of the inductive definition of our property specification language (see Definition 2.7), we can now establish the following theorem about our slicing algorithm (σ) presented in Section 3.1. Below, LTL_{-X} refers to the subset of the well-known LTL excluding

the next-time operator ($\boldsymbol{X}$) [11]. Note that our property specification language has the five types of atomic formulæ referred in the five lemmata and is exactly as standard LTL$_{-\boldsymbol{X}}$ *wff* built on top of those atomic formulæ.

THEOREM 3.9 (Generating stuttering equivalent models)
For any formula P of the property specification language, A, E and A', E are stuttering equivalent with respect to the labelling of atomic formulæ within P, where $A' = \sigma(A, E, P)$.

PROOF. Follows immediately from Lemmatas 3.4–3.8 which cover each type of atomic property in formula P and the fact that the labelling of a (Kripke) model for an LTL formula determines precisely the truth of the atomic propositions, as well as the fact that the slicing algorithm considers all atomic subformulæ of P in turn, regardless of its connectives and LTL operators. That is, removed plans did not affect the labelling of *any* of the atomic subformulæ of P. ■

The correctness of the whole slicing algorithm is a corollary of the above algorithm given that, for two stuttering equivalent structures M and M', LTL$_{-\boldsymbol{X}}$ formula f, and every initial state s, it is known that $M \models_s \boldsymbol{A}f$ if, and only if, $M' \models_s \boldsymbol{A}f$ (see Corollary 2 in [11, p. 147]), where $\boldsymbol{A}$ is the branching time operator meaning that f is true in all paths starting from s. Recall that our language is based on LTL$_{-\boldsymbol{X}}$ and that LTL model checking implicitly checks if the given property is true in all paths starting at an initial state of the system.

COROLLARY 3.10 (AgentSpeak slicing algorithm correctness)
The slicing algorithm introduced here is correct in the sense of Definition 3.1. That is, for A' such that $\sigma(A, E, P) = A'$, $A, E \models P$ if and only if $A', E \models P$.

Next, we consider the complexity of our slicing algorithm. We define the size of an input to the slicing algorithm, m, to be $m = p + i + v + |\varphi|$, where p is number of plans in the original AgentSpeak programs, i is the maximum number of predicates in any one plan/rule, v is the maximum number of variables per plan/rule and $|\varphi|$ is the size of the property φ to be checked (i.e. the number of subformulæ of φ). We have:

THEOREM 3.11 (Complexity of the AgentSpeak slicing algorithm)
There is an AgentSpeak slicing algorithm with complexity $O(m^3)$.

PROOF. Consider the complexity of the three stages of the slicing algorithm.

Stage I: the graph for the LDN generated at stage I of the algorithm has n vertices, where $n = O(p \cdot i)$. The number of edges in the LDN graph is again $O(n)$. The construction of the LDN involves assessing the mode information of the variables, which is at most p steps for each variable, and then creating the graph structure by matching predicate symbols. Since each predicate is a node in the LDN, then this gives the number of steps for construction as $O(p \cdot v \cdot n^2)$.
Stage II: the time for marking plans in stage II, according to Algorithm 1, is linear on $p \cdot |\varphi| \cdot n$. This can be easily seen from the loops in lines 1, 2 and 3, and the inner loops in either 6, 10, 14, 18 or 22, depending on the case.
Stage III: this stage is linear in p (one pass through all of them, just deleting the ones not marked).

Recalling that $n = O(p \cdot i)$, we can see that the overall time complexity is at most $p^3 \cdot i^2 \cdot v + p^2 \cdot i \cdot |\varphi| + p$, and is thus $O(m^3)$. ■

Given that agents do not tend to have very large numbers of plans, then this shows that slicing with our algorithm can be done relatively efficiently, and is thus in theory worth doing before model checking. In particular, since the core problem in model checking is often the space requirements

rather than the time taken, it is useful to apply slicing if only to reduce the size of the state space by a small amount.

We next present a case study in which we experimentally showed that slicing could provide significant improvement on the space and time required for model checking that application.

4 Autonomous Mars Rover: a case study on intra-agent plan slicing

4.1 Scenario

The development of autonomous rovers for planetary exploration is an important aim of the research on 'remote agents' carried out at space agencies [35]. We illustrate our slicing technique with an abstract version of a Mars exploration scenario, characterizing a typical day of activity of rovers such as Sojourner. The ideas used here for creating such scenario were mainly taken from [52] (and to a lesser extent from [3]).

A Martian day is called 'sol' and the instructions sent to the rover and collected data transmitted from it are recorded by day since landing on the planet. Thus, 'sol 22' refers to the 22nd day of activity of the rover on Mars. The scenario described here is inspired by the description given in [52] of a sequence of instructions sent to Sojourner on sol 22.

(1) Back up to the rock named Soufflé.
(2) Place the arm with the spectrometer on the rock.
(3) Do extensive measurements on the rock surface.
(4) Perform a long traverse to another rock.

In this particular sol operation, it turned out that the rover did not position itself correctly to approach the rock with the spectrometer arm. The misplaced spectrometer meant that no useful data were collected and that particular rock could not be visited again, hence a science opportunity was lost. This is an example mentioned in [52] where more flexibility in the software that controls exploration rover is required.

The scenario used here is also inspired by the following extract from that paper:

> 'As an example of flexible execution, consider the following plan for a day's traverse: the rover is to the south of a small ridge, trying to head generally north. The up-linked primary plan specifies the following course of action:
> - Travel north to the top of the ridge
> - Choose between the options:
>   - Nominal option, highest utility (precondition: there must be a path)
>   * Continue to the north
>   * Down-link to ground at sundown
>   - Contingent option, lower utility
>   * Move back down the ridge
>   * Travel east scanning for a pass
>   * Down-link to ground at sundown'

The paper also mentions that the rover is given a plan to make it especially attentive to 'green patches' on rocks. These are likely to represent an interesting science opportunity and so the rover should always give priority to examining such rocks if they turn up on its way to another target. Computer graphics software embedded in the rover does all the work of finding paths for the rover to reach a certain target. The navigation software available in a testbed for Mars rovers is described in [3].

A final thing to consider for our example scenario is that the batteries installed in the rover only work when there is sunlight, so all science activities are restricted by the amount of energy stored during the day. The rover must make sure all collected data are transmitted back to earth before it runs out of energy supply. Thus, other activities should be interrupted if carrying them out will mean the rover will not have enough energy to down-link collected data back to Earth.

Although we try, in the code below, to account for a greater flexibility for exploration rovers (as aimed for in [52]) in aspects such as making sure the rover is correctly positioned before activating the spectrometer, note that we here describe an abstract scenario based on general ideas of what goes on with a rover in a day of operation. Planning for such remote agents is a lot more complicated, and resources (computational or otherwise) that can be used in an actual rover is greatly limited. With this in mind, we stress that we do not aim here to provide a realistic program for a remote agent. However, it is interesting to note how adequate the constructs of agent-oriented programming based on BDI notions are for describing some of the activities of such agents. This makes the code below an interesting example on which to apply our slicing technique.

4.2 AgentSpeak code

In this section, we present the AgentSpeak code for this abstract rover scenario described above. Each plan is annotated with a label (prefixed with '`@`') so that we can refer to them in the text that follows. Note that this is the code for an autonomous agent, not a multi-agent system. However, the simple BDI logic we use is for multi-agent systems and thus the modalities need to refer to one particular agent. We use *amr* (an acronym for Autonomous Mars Rover) to refer to the autonomous agent whose code is below.

The code begins with two plans that have been up-linked by ground staff for that particular day of operation on Mars. In [50], the operational semantics of AgentSpeak was extended to account for speech-act-based communication. Among the few illocutionary forces that were considered in that paper, two are of particular interest here: *TellHow* can be used to inform new plans to an agent, and *Achieve*, with the usual KQML semantics, in practical terms (for AgentSpeak agents) creates an internal event adding an achievement goal with the predicate in the message content. This provides a very high-level approach to allow a ground operations team to send specific instructions to the rover.

4.2.1 Newly communicated plans

The plans labelled `sol22` are the ones that are communicated to the rover on day 22 of the mission; they are specific for what the ground team wants the rover to do on that day. To do that, they communicate the plans with a *TellHow* illocutionary force in a message, and send another message, of type *Achieve*, telling to rover to achieve a state where `sol(22)` has been accomplished. Therefore, the rover will have to make use of the plan with triggering event `+!sol(22)`, that the ground team sent to it. The other plan sent by the ground team for that day's tasks is to be used by the rover if it encounters any obstacles in traversing to the position where the rock the ground team wants to examine is.

```
@sol22_1
+!sol(22) : true
   <-   .dropDesires(traverse(X));
        !traverse(north, top_ridge);
        !examine(souffle).
@sol22_2
+!alternative_travel(north, top_ridge) : true
```

```
   <-    !move_to(south, bottom_ridge);
         !move_to_scanning_path(east, souffle).
```

All other plans are the ones that would be present in the agent's plan library before it was sent to Mars. They are general plans, that provide the rover with the ability to autonomously react to important things observed in its environment.

4.2.2 Events that the rover should react to

The first four plans allow the rover to react to important events perceived from the environment. Plan `r1` makes sure that, as soon as sunset starts, the rover drops all other desires and intentions (anything it was doing, or was still to do), and transmit back to earth all data it has gathered during that day. After that, the agent can make an internal note to the fact that sol was complete. Plan `r2` allows the rover to react appropriately when it perceives 'green patches' on a rock nearby its location. The rover is instructed to always give preferences to such findings, as the scientists know that such green patches normally represent the most promising opportunities for scientific discoveries. When the rover is not already examining a rock, nor have a specific rock it intends to visit, any rock perceived is a possible target to move to; this is stated in plan `r3`. Plan `r4` is triggered when the agent perceives its batteries are low while it is examining a rock (and it has not yet started transmitting the data back to Earth). It should then stop the spectrometer, transmit all remaining data, and make an internal note saying that sol 22 is finished.

```
@r1
+sundown : true
   <-    ?today(Day);
         .dropAllDesires; // drop all except itself!
         !transmit_all_remaining_data;
         +sol(Day). // remember that today's task was accomplished

@r2
+green_patch(Rock)
   :     not battery_charge(low)
   <-    .dropDesires(traverse(X,Y));
         cg.get_target_directions(Rock, Direction, Position);
         !traverse(Direction, Position);
         !examine(Rock).

@r3
+possible_target(Rock)
   :     not .desire(traverse(X,Y)) & not .desire(examine(Z))
         & not running(spectrometer)
   <-    cg.get_target_directions(Rock, Direction, Position);
         !traverse(Direction, Position). // just to get a better look, let ground
                                         // staff decide if worth examining

@r4
+battery_charge(low)
   :     running(spectrometer)
   <-    stop(spectrometer);
         .dropDesire(transmit_all_remaining_data); // to avoid duplicating the goal
         !transmit_all_remaining_data;
         ?today(Day);
         +sol(Day). // remember that today's task was accomplished
```

4.2.3 Goals that the rover often has to achieve

Plans labelled `g1`–`g10` account for most of the rovers usual goals (e.g. traversing to a certain rock, finding alternative routes in case the normal route to that rock is obstructed, etc.).

```
@g1
+!traverse(Direction, Position) : true
   <-   cg.check_paths(Direction, Position); // this could be by comm with CG agent
        !travel(Direction, Position).

@g2
+!travel(Direction, Position)
   :    safe_path(Position) // updated by check_paths()
   <-   !move_to(Direction, Position).

@g3
+!travel(Direction, Position)
   :    not safe_path(Position)
   <-   !alternative_travel(Direction, Position).

@g4
+!move_to(Direction, Position) : at(Position) <- true.

@g5
+!move_to(Direction, Position)
   :    not at(Position) & safe_path(Position)
   <-   step_towards(Direction);
        cg.get_direction(Position, NewDirection); // update direction to Target!
                                                  // E.g., for a diagonal path.
        cg.check_paths(NewDirection, Position);
        !move_to(NewDirection, Position).

@g6
+!move_to(Direction, Position)
   :    not at(Position) & not safe_path(Position)
   <-   !travel(Direction, Position).

@g7
+!move_to_scanning_path(Direction, Target) : true
   <-   step_towards(Direction);
        cg.get_target_directions(Target, TDir, TPos);
        cg.check_paths(TDir, TPos);
        !follow_target(TDir, TPos, Direction, Target).

@g8
+!follow_target(Direction, Target, AltDir, AltTrgt)
   :    safe_path(Target)
   <-   cg.get_target_directions(Target, TDir, TPos);
        !move_to(Direction, TPos).

@g9
+!follow_target(Direction, Target, AltDir, AltTrgt)
   :    not safe_path(Target)
   <-   !move_to_scanning_path(AltDir, AltTrgt).
```

```
// default alternative travel
@g10
+!alternative_travel(Direction, Position) : true
   <-   cg.alternative_direction(Direction,NewDirection);
        !move_to_scanning_path(NewDirection, Position).
```

4.2.4 Science activities

Plans labelled `s1–s5` describe the science activities the rover has to perform (such as making sure it is correctly positioned before it turn on the spectrometer, collecting spectrometer data, etc.).

```
@s1
+!examine(Rock)
   :    correctly_positioned_to_examine(Rock)
   <-   place_spectrometer_arm_at(Rock);
        !extensive_measurements_on_surface_of(Rock).

@s2
+!examine(Rock)
   :    not correctly_positioned_to_examine(Rock)
   <-   !correctly_positioned_to_examine(Rock);
        !examine(Rock).

@s3
+!correctly_positioned_to_examine(Rock)
   :    not correctly_positioned_to_examine(Rock)
   <-   move_backwards_from(Rock);
        approach(Rock);
        !correctly_positioned_to_examine(Rock).

@s4
+!correctly_positioned_to_examine(Rock)
   :    correctly_positioned_to_examine(Rock)
   <-   true.

@s5
+!extensive_measurements_on_surface_of(Rock) : true
   <-   run_spectrometer(Rock).
        // ...
```

4.2.5 Communication

Finally, plans labelled `c1–c4` are used for communication.

```
@c1
+!transmit_all_remaining_data
   :    data(Type, Source, Time, Data) &
        not downlink(ground, Type, Source, Time)
   <-   !downlink(ground, Type, Source, Time);
        !transmit_all_remaining_data.
```

```
@c2
+!transmit_all_remaining_data : true <- true.

@c3
+!downlink(Agent, Type, Source, Time)
   :    turned_on(antenna)
   <-   ?data(Type, Source, Time, Data); // spectrometer output perceived by
                                         // sensing
        .send(Agent, tell, data(Type, Source, Time, Data));
        +downlink(Agent, Type, Source, Time).

@c4
+!downlink(Agent, Type, Source, Time)
   :    not turned_on(antenna)
   <-   turn_on(antenna);
        !downlink(Agent, Type, Source, Time).
```

It is interesting to note some of bugs that were found in the original AgentSpeak program during the model-checking exercises. The context of plan `r1` was originally empty. However, as a result of model checking, it was discovered that if sundown happens before the rover has finished gathering any significant data, it is not a good idea to drop all intentions. The context of plan `r3` initially had only `not .desire(traverse(X,Y))`; we found out that giving attention to possible targets in the other two situations should also be avoided. In plan `r4`, the line with the internal action `dropDesire` was later added as the battery charge could become low at a moment where the agent was already transmitting the gathered data, and having two parallel intentions for that same purpose obviously caused problems.

4.3 Slicing

Intuitively, there are two ways in which slicing particularly alleviates the state explosion of AgentSpeak programs. The first one is by removing plans that cannot affect the truth or otherwise of the formula in the slicing criterion, but would increase the length of a computation for an agent to handle particular events before the truth of the property can be determined. This is similar to the motivation for removing clauses in traditional logic programs. This form of state-space reduction resulting from our slicing method is marked (1) in Figure 7. Note however that automata-theoretic model checking already avoids expanding system states that are not necessary for finding a counter-example, which is a different situation.

Besides removing details of intermediate intention processing that are unnecessary for checking a certain property, another source of state-space reduction can happen when slicing AgentSpeak programs. Whenever all the plans that are used to handle particular external events can be removed, this greatly reduces the state space since, at any point during the computation associated with one intention, there are reachable states in which other intentions (other focuses of attention) are created to handle events that may have been generated by belief revision. Slicing out such plans eliminates all such branches of the computation tree. This form of state-space reduction is marked (2) in Figure 7.

An alternative way of making the reduction associated with events for which no plans become available would be to avoid the environment generating such events in the first place (considering that they will not affect the property being verified anyway). Because the environment representation is not usually AgentSpeak code, but is provided by the user, automatic slicing would be less practical

in this way. The user would have to remove, from their own code, the generation of the events that the algorithm would determine as safe to slice out.

An example of the first type of state-space reduction (the one which reduces the path length of the computation associated with a particular intention), is as follows. Suppose that the agent's original plan library did not include plans `r1–r4`. This would mean the agent would not, in any case, have more than a single intention at a time. Still, consider that the following is the property to be checked (and is thus our slicing criterion):

$$\begin{aligned}\Box((&\mathsf{Does}\ \mathit{amr}\ \texttt{place\_spectrometer\_arm\_at(R)})\rightarrow\\ &(\mathsf{Bel}\ \mathit{amr}\ \texttt{correctly\_positioned\_to\_examine(R)}))\end{aligned} \tag{1}$$

which means that whenever the rover performs the action of placing its spectrometer arm at a certain rock, it believes itself to be correctly positioned to examine that rock.

Because plans `c1–c4` can only become intended after `place_spectrometer_arm_at(R)` has already happened, there is no need to consider that part of the execution of the intention as it will not affect the property under consideration. Thus, the generated slice for the above property does not include plans `c1–c4`.

Note, however, that slicing does not always help reduce the state space. For example, consider the property[6]

$$\Diamond\left[\begin{array}{c}\left(\begin{array}{c}(\mathsf{Does}\ \mathit{amr}\ \texttt{place\_spectrometer\_arm\_at(R)})\\ \wedge\\ (\mathsf{Bel}\ \mathit{amr}\ \texttt{correctly\_positioned\_to\_examine(R)})\end{array}\right)\\ \vee\\ \texttt{battery==EmptyBattery}\end{array}\right]$$

Although it produces exactly the same slice as specification (1) above, no state-space reduction occurs in practical model checking. As mentioned previously, an advantage of automata-theoretic model checking is that some system states may not be generated in checking particular properties, and in this case the sliced out states coincide with those.

An example of the second type of state-space reduction (which avoids the generation of other focuses of attention in the agent that would not interfere with the property being checked) is:

$$\Box\left[\begin{array}{l}(\mathsf{Int}\ \mathit{amr}\ \texttt{transmit\_all\_remaining\_data(22)})\rightarrow\\ \Diamond\neg\left(\begin{array}{c}(\mathsf{Bel}\ \mathit{amr}\ \texttt{data(specData, souffle, 22, \_)})\\ \wedge\\ \neg(\mathsf{Bel}\ \mathit{amr}\ \texttt{downlink(ground, specData, souffle, 22)})\end{array}\right)\end{array}\right] \tag{2}$$

which means that, in any execution path[7], whenever the rover intends to transmit all remaining data back to Earth, some time after that there will be no data entry in its belief base for which there is not an associated belief saying that that particular piece of information has already been down-linked back to the ground team (this ensures, in particular, that the rover does not run out of power before it finishes the important task of transmitting all gathered data).

[6]Note that `battery==EmptyBattery` is a PROMELA Boolean expression that is used to check whether the robot has run out of battery charge.

[7]Recall that, in LTL model checking, the model checker ensures that the LTL formula in the given specification is true in the initial states of the system for each possible execution path of the system.

With the above slicing criterion, plan `r3` can be safely removed. Note that, although the slicing appears to be 'small' (i.e. just one plan is removed), a considerable reduction of the state space can ensue, depending also on how dynamic the environment is. If many possible targets are detected (and approached) during the time data are being transmitted back to Earth, this could generate a large number of different system states in which the two focuses of attention are being dealt with simultaneously by the rover.

An example of a slicing criterion for which the generated slice is the same as the original program (i.e. no plan is deleted) is:

$$\begin{array}{l}\Box(((\mathsf{Bel}\ amr\ \texttt{green\_patch(r1)})\vee\\ \quad((\mathsf{Bel}\ amr\ \texttt{sol(22)})\wedge(\mathsf{Bel}\ amr\ \texttt{possible\_target(r2)})))\\ \quad\rightarrow\\ \qquad\Diamond((\mathsf{Bel}\ amr\ \texttt{downlink(ground,specData,r1,22)})\vee\\ \qquad\quad(\mathsf{Bel}\ amr\ \texttt{downlink(ground,specData,r2,22)})))\end{array}$$

as every single plan of the agent is involved in determining whether the above property is true or not.

4.4 Experimental Results

We have not yet implemented the full slicing algorithm. However, in order to assess the efficacy of the slicing approach we *manually* sliced the above program (following the algorithm) and applied model checking both before and after slicing. Experiments were run on a machine with an MP 2000 + (1666 MHz) processor with 256 K cache and 2 GB of RAM (266 MHz). For specification (1), SPIN [24] used 606 MB of memory (1.18×10^6 states in the system) and took 86 s to complete model checking. Slicing improved this to 407 MB (945, 165 states) and 64 s. This gives a reduction of 25.6% on the time to model check, and a 33% reduction in memory usage. For specification (2), SPIN used 938 MB of memory (2.87×10^6 states), and took 218 s to complete checking. After slicing, this went down to 746 MB (2.12×10^6 states) and 162 s. This means a reduction of about 26% on the time to model check, and 21% on memory usage. Interestingly, SPIN's built-in slicing algorithm does *not* reduce the state space at all.

We have also ran a similar experiment using our approach to model checking AgentSpeak systems using JPF [51] rather than SPIN. We used a recent open source version of JPF [26], which has very sophisticated techniques allowing verification of Java programs whilst generating a much smaller state space than previous versions. This time the experiments were run on a 2.2 GHz, Dual Processor Pentium 4, with 1 GB RAM.

It is interesting to note how much smaller the state space generated is (and how much less memory is used) by JPF as compared with SPIN, although the latter is still (at least for this particular experiment) much faster. On the other hand, the latest version of JPF does *not* have inbuilt features to allow LTL verification, so the results of this experiment are for JPF being simply asked to visit the entire state space generated by the system. Again, in this experiment, we ran the model checker in the original system, and then for a sliced version of the system. The results for JPF were as follows. The *unsliced* system took 80 min to model check, generating 145 695 different states and using 69 MB of memory. The *sliced* system took 31 min to model check, generating 61 938 states and using 56 MB of memory. Therefore, for JPF, the reduction in the time to model check due to our slicing algorithm was 61%, although with only 19% of memory reduction.

Note that slicing does not affect in the same way the time taken to model check and the amount of memory required, as can be observed in the results above. Although slicing is likely to reduce the

number of states that the model checker will analyse, and therefore the time taken, the reduction in the amount of space required also depends on exactly which part of a state itself is sliced away. For example, if the slicing removes large data structures from each state then the memory reduction will be more than if it keeps each state intact, but slices some behaviours away. This means that, after slicing is applied, the reduction obtained in memory will not be directly correlated to the reduction in time taken to complete the model checking.

While this is just one case study, it serves as a compelling proof of concept that our slicing technique can significantly reduce the time/space requirements for practical model checking. We do not yet have an implementation for our slicing method, so for these experiments slicing was done by manually applying the algorithm to the AgentSpeak code. However, given our complexity results, it can be expected that slicing will typically be done reasonably efficiently.

5 Concluding Remarks

In this article, we have developed a slicing algorithm for AgentSpeak, and have provided both correctness and complexity results for this algorithm. The technique has been used in the formal verification of an agent-oriented program. The results of this case study indicate that our slicing technique has the potential to be of real practical benefit. Our approach also allows specifications to be written in a logic including agent-specific modalities.

For the experiments presented here, we manually generated the required slices using our algorithm. For a small example, as the one in this article, it is not difficult to ensure that the algorithm was applied correctly (it is certainly easier than to ensure correct implementation of the algorithm). However, the lack of implementation would prevent use of our slicing technique for large examples (particularly in industrial-scale software). One of our aims in future work is in fact to produce a fully fledged, user-friendly, open source implementation of our slicing method (which first would require an implementation of the algorithms for LDN generation).

Property-based slicing is a well-known state-space reduction technique used in practical software verification. More generally, slicing is widely used in software engineering [47], notably for program comprehension [31], but also testing [21], debugging [17], decomposition [18] and reuse [28]. To our knowledge, this is the very first slicing algorithm created for an agent-oriented programming language. Although it is very early for the use of slicing in agent programs, it is reasonable to expect that our property-based slicing technique will have significant use besides verification.

The fact that our approach uses a property specification language that includes BDI modalities should in principle greatly facilitate the use of slicing for software comprehension, for example. One of the main constructs in agent programming is that of a *goal* (i.e. a state of affairs the agent wishes to bring about). So, for example, the slice for property $\Diamond(\mathsf{Des}\ a\ g)$ would consist of only the plans (in agent a as well as the other agents) which can lead agent a to having the goal of achieving g. Intuitively, one can see that this would be a high-level approach for slicing agent programs for general use in software engineering, but this has never been tried in practice before, and therefore remains a speculation. One of our planned future projects is precisely to use our slicing technique for practical software comprehension, testing and reuse in industrial-strength agent-based software.

Another interesting question for further research is on the use of different slicing techniques than the one on which we based our approach. Unlike agent-oriented programming languages, for other more traditional programming languages a variety of slicing approaches exist. For example, dynamic slicing [44, 47] takes into account specific input values; this might be particularly interesting for future work on using slicing for testing and debugging agent programs. Amorphous slicing [22]

is a technique that allows not only the deletion of parts of the program but also syntactical changes in the parts of the program included in the slice, provided a projection of the semantics is preserved (in our case, the preserved semantics in question would be related to the specification we want to model check); it would be interesting to investigate the results of this approach for agent program slicing. Perhaps the alternative slicing approach that could be more easily combined with ours is Conditioned Slicing [9, 10], where a (first-order logic) formula is used to characterize possible inputs (rather than requiring the exact inputs as in dynamic slicing). For example, certain restrictions on inputs could be included in the specification that we use as slicing criterion in our approach. However, this all remains to be investigated.

Slicing algorithms are typically language dependent, and in the case of property-based slicing, dependent on both the programming language as well as the property specification language. Although our algorithm is specific to AgentSpeak and the 'shallow' BDI logic built on top of LTL_{-X} used in this article, the similarities between the leading agent-oriented programming languages [14] (many of which are based on logic programming) indicate that our work could at least serve as inspiration for slicing algorithms for other agent programming languages. This is another topic we hope will be investigated in further work in the area of multi-agent programming languages [4].

Acknowledgements

Many thanks to the anonymous reviewers for their detailed comments. Work partially supported by the EU through HPMF-CT-2001-00065.

References

[1] E. Allen Emerson. Temporal and Modal Logic. In *Handbook of Theoretical Computer Science*, J. van Leeuwen, ed., Vol. B, Ch. 16, pp. 997–1072. Elsevier Science, 1990.

[2] M. Benerecetti and A. Cimatti. Validation of multiagent systems by symbolic model checking. In *Proceedings of the 3rd International Workshop on Agent-Oriented Software Engineering (AOSE)*, Vol. 2585 of *Lecture Notes in Computer Science*, pp. 32–46. Springer, 2003.

[3] J. Biesiadecki, M. W. Maimone, and J. Morrison. The Athena SDM rover: a testbed for Mars rover mobility. In *Sixth International Symposium on AI, Robotics and Automation in Space (ISAIRAS-01), June, Montreal, Canada*, 2001.

[4] R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, eds. *Multi-Agent Programming: Languages, Platforms and Applications*. Number 15 in *Multiagent Systems, Artificial Societies, and Simulated Organizations*. Springer, 2005.

[5] R. H. Bordini, M. Fisher, W. Visser, and M. Wooldridge. Model checking rational agents. *IEEE Intelligent Systems*, **19**, 46–52, 2004.

[6] R. H. Bordini, M. Fisher, W. Visser, and M. Wooldridge. Verifying multi-agent programs by model checking. *Journal of Autonomous Agents and Multi-Agent Systems*, **12**, 239–256, 2006.

[7] R. H. Bordini, J. F. Hübner, and M. Wooldridge. *Programming Multi-Agent Systems in AgentSpeak Using Jason*. *Wiley Series in Agent Technology*. John Wiley & Sons, 2007.

[8] R. H. Bordini and Á. F. Moreira. Proving BDI properties of agent-oriented programming languages: the asymmetry thesis principles in AgentSpeak(L). *Annals of Mathematics and Artificial Intelligence*, **42**, 197–226, 2004. (Special Issue on Computational Logic in Multi-Agent Systems).

[9] G. Canfora, A. Cimitile, and A. De Lucia. Conditioned program slicing. In *Information and Software Technology, Special Issue on Program Slicing*, M. Harman and K. Gallagher eds., p. 40, Elsevier, 1998.

[10] D. Cheda and S. Cavadini. Conditional slicing for first-order functional logic programs. In *17th International Workshop on Functional and (Constraint) Logic Programming (WFLP-2008), Siena, Italy, 3–4 July*, 2008.

[11] E. M. Clarke Jr., O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.

[12] P. R. Cohen and H. J. Levesque. Intention is choice with commitment. *Artificial Intelligence*, **42**, 213–261, 1990.

[13] J. C. Corbett, M. B. Dwyer, H. John, and Robby. Bandera: a source-level interface for model checking Java programs. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000), 4–11 June, Limerick, Ireland*, pp. 762–765. ACM Press, 2000.

[14] L. A. Dennis, B. Farwer, R. H. Bordini, M. Fisher, and M. Wooldridge. A Common Semantic Basis for BDI Languages. In *Proceedings of the Seventh International Workshop on Programming Multiagent Systems (ProMAS)*, *Lecture Notes in Artificial Intelligence*. Springer, 2007.

[15] M. Fisher, R. H. Bordini, B. Hirsch, and P. Torroni. Computational logics and agents: a road map of current technologies and future trends. *Computational Intelligence*, **23**, 61–91, 2009.

[16] M. Fisher, D. Gabbay, and L. Vila, eds. *Handbook of Temporal Reasoning in Artificial Intelligence*, Vol. 1 of *Advances in Artificial Intelligence*. Elsevier Publishers, North Holland, 2005.

[17] M. A. Francel and S. Rugaber. The value of slicing while debugging. *Science of Computer Programming*, **40**, 151–169, 2001.

[18] K. B. Gallagher and J. R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, **17**, 751–761, 1991.

[19] L. Giordano, A. Martelli, and C. Schwind. Specifying and verifying interaction protocols in a temporal action logic. *Journal of Applied Logic*, **5**, 214–234, 2007.

[20] D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. MIT Press, 2000.

[21] M. Harman and S. Danicic. Using program slicing to simplify testing. *Software Testing, Verification and Reliability*, **5**, 143–162, 1995.

[22] M. Harman and S. Danicic. Amorphous program slicing. In *WPC '97: Proceedings of the 5th International Workshop on Program Comprehension (WPC '97)*, pp. 70–79. IEEE Computer Society, 1997.

[23] J. Hatcliff and M. B. Dwyer. Using the Bandera Tool Set to model-check properties of concurrent Java software. In K. G. Larsen and M. Nielsen, eds, *Proceedings of the 12th International Conference Concurrency Theory (CONCUR 2001), Aalborg, Denmark, 20–25 August*, Vol. 2154 of *Lecture Notes Computer Science*, pp. 39–58. Springer, 2001.

[24] G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.

[25] N. R. Jennings and M. Wooldridge (eds) Applications of agent technology. In *Agent Technology: Foundations, Applications, and Markets*. Springer, 1998.

[26] Java PathFinder. Available at `http://javapathfinder.sourceforge.net`. 2009.

[27] M. Krishna Rao, D. Kapur, and R. Shyamasundar. Proving termination of GHC Programs. In D. S. Warren, ed., *Proceedings of the Tenth International Conference on Logic Programming, 21–24 June, Budapest, Hungary*, pp. 720–736. MIT Press, 1993.

[28] F. Lanubile and G. Visaggio. Extracting reusable functions by flow graph-based program slicing. *IEEE Transactions Software Engineering*, **23**, 246–259, 1997.

[29] M. Ljunberg and A. Lucas. The OASIS air traffic management system. In *Proceedings of the Second Pacific Rim International Conference on AI (PRICAI-92)*, Seoul, Korea, 1992.

[30] J. W. Lloyd. *Foundations of Logic Programming*, 2nd edn. Springer, 1987.

[31] A. D. Lucia, A. R. Fasolino, and M. Munro. Understanding function behaviors through program slicing. In *4th International Workshop on Program Comprehension (WPC '96), March 29-31, 1996, Berlin, Germany*, pp. 9–10. IEEE Computer Society, 1996.

[32] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer, 1992.

[33] J. Mayfield, Y. Labrou, and T. Finin. Evaluation of KQML as an agent communication language. In M. Wooldridge, J. P. Müller, and M. Tambe, eds, *Intelligent Agents II—Proceedings of the Second International Workshop on Agent Theories, Architectures, and Languages (ATAL'95), held as part of IJCAI'95, Montréal, Canada, August 1995*, Number 1037 in *Lecture Notes in Artificial Intelligence*, pp. 347–360, Springer, 1996.

[34] A. Moreno and C. Garbay. Special issue on "Software Agents in Health Care". *Artificial Intelligence in Medicine*, **27**, 229–232, 2003.

[35] N. Muscettola, P. Pandurang Nayak, B. Pell, and B. C. Williams. Remote Agent: to boldly go where no AI system has gone before. *Artificial Intelligence*, **103**, 5–47, 1998.

[36] N. Osman and D. Robertson. Dynamic verification of trust in distributed open systems. In *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, pp. 1440–1445, 2007.

[37] G. Plotkin. A structural approach to operational semantics. *Technical Report DAIMI FN-19*. Department of Computer Science, Aarhus University, 1981.

[38] F. Raimondi and A. Lomuscio. Automatic verification of multi-agent systems by model checking via ordered binary decision diagrams. *Journal of Applied Logic*, **5**, 235–251, 2007.

[39] A. S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In W. Van de Velde and J. Perram, eds, *Proceedings of the Seventh Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW'96), 22–25 January, Eindhoven, The Netherlands*, Number 1038 in *Lecture Notes in Artificial Intelligence*, pp. 42–55. Springer, 1996.

[40] A. S. Rao and M. P. Georgeff. Decision procedures for BDI logics. *Journal of Logic and Computation*, **8**, 293–343, 1998.

[41] Robby, M. B. Dwyer, and J. Hatcliff. Bogor: an extensible and highly-modular model checking framework. In *Proceedings of the 11th ACM SIGSOFT Symposium on Foundations of Software Engineering 2003 held jointly with 9th European Software Engineering Conference, Fourth Joint Meeting ESEC/FSE 2003, Helsinki, Finland, 1–5 September*, pp. 267–276. ACM Press, 2003.

[42] S. Schoenig and M. Ducassé. A backward slicing algorithm for Prolog. In R. Cousot and D. A. Schmidt, eds, *Proceedings of the Third International Symposium on Static Analysis (SAS'96), Aachen, Germany, 24–26 September 1996*, Vol. 1145 of *Lecture Notes in Computer Science*, pp. 317–331. Springer, 1996.

[43] G. Szilágyi, T. Gyimóthy, and J. Maluszyński. Slicing of constraint logic programs. In M. Ducassé, ed., *Proceedings of the Fourth International Workshop on Automated Debugging (AADEBUG 2000), 28–30 August, Munich*. Computing Research Repository cs.SE/0010035, 2000.

[44] G. Szilágyi, T. Gyimóthy, and J. Maluszyński. Static and dynamic slicing of constraint logic programs. *Journal of Automated Software Engineering*, **9**, 41–65, 2002.

[45] SPIN: on-the-fly LTL model checking. Available at `http://spinroot.com/spin`.

[46] E. M. Tadjouddine, F. Guerin, and W. W. Vasconcelos. Abstracting and verifying strategy-proofness for auction mechanisms. In M. Baldoni, T. C. Son, M. B. van Riemsdijk, and M. Winikoff, eds, *Declarative Agent Languages and Technologies VI, 6th International Workshop, DALT 2008, Estoril, Portugal, May 12, 2008, Revised Selected and Invited Papers*, Vol. 5397 of *Lecture Notes in Computer Science*, pp. 197–214. Springer, 2009.

[47] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, **3**, 121–189, 1995.

[48] M. B. van Riemsdijk, F. S. de Boer, M. Dastani, and J.-J. C. Meyer. Prototyping 3APL in the Maude term rewriting language. In K. Inoue, K. Satoh, and F. Toni, eds, *Computational Logic in Multi-Agent Systems, 7th International Workshop, CLIMA VII, Hakodate, Japan, May 8-9, 2006, Revised Selected and Invited Papers*, Vol. 4371 of *Lecture Notes in Computer Science*, pp. 95–114. Springer, 2007.

[49] W. W. Vasconcelos and M. A. T. Aragão. Slicing knowledge-based systems: techniques and applications. *Knowledge-Based Systems*, **13**, 177–198, 2000.

[50] R. Vieira, A. Moreira, M. Wooldridge, and R. H. Bordini. On the formal semantics of speech-act based communication in an agent-oriented programming language. *Journal of Artificial Intelligence Research (JAIR)*, **29**, 221–267, 2007.

[51] W. Visser, K. Havelund, G. Brat, and S. Park. Model Checking Programs. In *Proceedings of the Fifteenth International Conference on Automated Software Engineering (ASE'00), 11-15 September, Grenoble, France*, pp. 3–12. IEEE Computer Society, 2000.

[52] R. Washington, K. Golden, J. Bresina, D. Smith, C. Anderson, and T. Smith. Autonomous Rovers for Mars Exploration. In *Aerospace Conference, 6–13 March, Aspen, CO*, Vol. 1, pp. 237–251. IEEE, 1999.

[53] W. Wobcke, M. Chee, and K. Ji. Model checking for prs-like agents. In S. Zhang and R. Jarvis, eds, *AI 2005: Advances in Artificial Intelligence, 18th Australian Joint Conference on Artificial Intelligence, Sydney, Australia, December 5-9, 2005, Proceedings*, Vol. 3809 of *Lecture Notes in Computer Science*, pp. 17–28. Springer, 2005.

[54] M. Wooldridge. *Reasoning about Rational Agents*. The MIT Press, 2000.

[55] B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen. A brief survey of program slicing. *SIGSOFT Software Engineering Notes*, **30**, 1–36, 2005.

[56] J. Zhao, J. Cheng, and K. Ushijima. Literal dependence net and its use in concurrent logic programming environment. In *Proceedings of the Workshop on Parallel Logic Programming, held with FGCS'94, ICOT, Tokyo, 15–16 December*, pp. 127–141, 1994.

[57] J. Zhao, J. Cheng, and K. Ushijima. Slicing concurrent logic programs. In T. Ida, A. Ohori, and M. Takeichi, eds, *Proceedings of the Second Fuji International Workshop on Functional and Logic Programming, Shonan Village Center, Japan, 1–4 November 1996*, pp. 143–162. World Scientific, 1997.

Received 1 May 2009