State-Space Reduction Techniques in Agent Verification*

Rafael H. Bordini[1†] Michael Fisher[1] Willem Visser[2] Michael Wooldridge[1]

[1] University of Liverpool, Liverpool L69 3BX, U.K. {bordini,michael,mjw}@csc.liv.ac.uk

[2] RIACS/NASA Ames Research Center, CA 94035, USA. wvisser@email.arc.nasa.gov

Abstract

We have developed a set of tools to allow the use of model-checking techniques for the verification of systems directly implemented in an agent-oriented programming language. The success of model checking as a verification technique for large systems is dependent partly on its use in combination with various state-space reduction techniques. An important example of such techniques is property-based slicing. This paper introduces an algorithm for property-based slicing of AgentSpeak multi-agent systems. We apply our approach to the AgentSpeak code for a scenario inspired by routine tasks of autonomous Mars rovers, and explain how slicing reduces the search space in theory. We consider experiments on such scenarios, and initial results show a significant reduction in the state space, thus indicating that this approach can have an important impact on the practicality of agent verification.

1. Introduction

As multi-agent systems come to be used in increasingly critical applications, the need to provide tools supporting their *verification* — showing that a system is correct with respect to its stated requirements — becomes even stronger. We have developed model checking techniques and tools for verifying systems implemented in AgentSpeak(L) [4, 2]. The AgentSpeak(L) BDI logic programming language was created by Rao [7], and is one of the few languages for programming 'intelligent' multi-agent systems. While we have developed these tools, based on translating AgentSpeak(L) programs into either Java (for use with the JPF model checker) or Promela (the input language for the SPIN model checker), practical experience has shown that, as with conventional programs, the verification of multi-agent programs suffers from the *state explosion* problem. In carrying out model checking on a concurrent system, the whole state space of the system must be represented in some way — and this state space is typically enormous.

A key technique used in simplifying the analysis of conventional programs is that of *slicing*. The basic idea behind program slicing is to eliminate details of the program that are not relevant to the analysis in hand [10]. Thus, in our case, since we wish to verify some property, the idea is to eliminate parts of the program that do not affect that property; this is called *property-based slicing*. Although slicing techniques have been successfully used in conventional programs to reduce the state-space required, these standard techniques 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. In short, this is what we describe in this paper: a new agent-based slicing algorithm and its application to model-checking AgentSpeak.

The remainder of this paper is organised as follows. The next section provides a brief overview of our approach to model checking agent systems. Section 3 gives the necessary background on slicing for logic programming languages. We then introduce our AgentSpeak slicing algorithm, give an illustrative example, and outline a proof of correctness in Section 4. Section 5 discusses a case study on an autonomous Mars rover scenario.

2. Model Checking AgentSpeak

A first key step in our research was to restrict AgentSpeak(L) to finite state systems: AgentSpeak(F), a finite state version of AgentSpeak(L), was first described in [2]. 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 tools for model checking that are available.

An AgentSpeak agent (ag) is created by the specification of a set of beliefs (bs), which is a set of ground predicates, and a set of plans (ps). AgentSpeak(L) distinguishes two types of goals (g): *achievement goals* and *test goals*.

* Work at Liverpool supported by an EC Marie Curie Fellowship under contract number HPMF-CT-2001-00065.

† Currently at the Department of Computer Science, University of Durham, DH1 3LE, U.K. E-mail: R.Bordini@durham.ac.uk.

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

Figure 1. Concrete Syntax of AgentSpeak(F).

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. A *test goal* states that the agent wants to test whether the associated predicate is one of its beliefs. A *triggering event* (te) defines which events may initiate the execution of plans. There are two types of triggering events: those related to the *addition* ('**+**') and *deletion* ('**-**') of mental attitudes (beliefs or goals).

An AgentSpeak plan (p) has a *head* (the expression to the left of the arrow), which is formed from a triggering event (denoting the purpose of that plan), and a conjunction of belief literals (l) representing a *context* (ct). The conjunction of literals in the context must be a logical consequence of that agent's current beliefs if the plan is to be executed. A plan also has a *body* (h), which is a sequence of basic actions or (sub)goals that the agent has to achieve (or test) when the plan is triggered. Plan bodies are sequences of goals, belief updates (u), or *basic actions* that an agent is able to perform on its environment. Such actions are also defined as predicates, but with special predicate symbols (called *action symbols*) used to distinguish them.

The grammar in Figure 1 gives the concrete syntax of AgentSpeak(F). In the grammar, P stands for any predicate symbol, A for action symbols, and terms t_i are either constants or variables. As in Prolog, uppercase initial letters are used for variables, and lowercase for constants and predicate symbols (cf., Prolog atoms). Note that first order terms (cf., Prolog structures) are not allowed in the present version of AgentSpeak(F).

There are also some predefined *internal action* symbols, which are indicated by an initial '.' character. The action '.send' is used for speech-act based inter-agent communication, and is interpreted as follows. If an AgentSpeak(F) agent l_1 executes $\mathsf{.send}(l_2, \mathit{ilf}, at)$, 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. Other predefined internal actions are used for printing messages, and conditional and arithmetic operations, for example.

The main difference between AgentSpeak(F) and AgentSpeak(L) is that first order terms are disallowed. Other restrictions, which apply particularly when model checking is to be done with SPIN, are described in [2]. A multi-agent system is specified by the user as a collection of AgentSpeak(F) source files, one for each agent in the system. Various functions that are part of the interpretation of AgentSpeak(L) agents can be customised. Also, the user has to provide the environment in which the agents will be situated; this is written in the input language of the model checker itself, rather than AgentSpeak(F).

In the context of verifying multi-agent systems implemented in AgentSpeak, the most appropriate way of specifying the properties that the system satisfies (or does not satisfy) is by expressing them using a temporal logic combined with modalities for referring to agent's mental attitudes, in particular BDI logics [12, 8]. Such logics formalise the main concepts of the underlying BDI architecture used in reactive planning systems such as AgentSpeak agents. A way of interpreting the informational, motivational, and deliberative modalities of BDI logics for AgentSpeak(L) agents was given in [3] based on the operational semantics of AgentSpeak(L). In the present work, we use this framework for interpreting the BDI modalities in terms of data structures within the model of an AgentSpeak(F) agent given in the model checker's input language. This way, we can translate (temporal) BDI properties into LTL formulæ.

The logical property specification language for our model-checking approach is defined next. It is a simplified version of $\mathcal{LORA}$ [12], which is based on modal logics of intentionality, dynamic logic, and CTL*. In the 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. The main restriction of the language used here in comparison to $\mathcal{LORA}$ is that the Bel, Des, and Int modalities can only be applied to atomic formulæ (i.e., predicates as used in AgentSpeak).

Let pe be any valid boolean expression in the model specification language of the model checker being used, l be any agent label, x be a variable ranging over agent labels, and at and a be atomic and action formulæ defined in the AgentSpeak(F) syntax (see Figure 1), except with no variables allowed. Then the set of well-formed formulæ (*wff*) of this language is defined inductively as follows:

1. pe is a *wff*;
2. at is a *wff*;
3. $(\mathsf{Bel}\ l\ at)$, $(\mathsf{Des}\ l\ at)$, and $(\mathsf{Int}\ l\ at)$ are *wff*;
4. $\forall x.(M\ x\ at)$ and $\exists x.(M\ x\ at)$ are *wff*, where $M \in \{\mathsf{Bel}, \mathsf{Des}, \mathsf{Int}\}$ and x ranges over a finite set of agent labels;
5. $(\mathsf{Does}\ l\ a)$ is a *wff*;
6. 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 $(\varphi\ \mathcal{R}\ \psi)$;
7. nothing else is a *wff*.

In the syntax above, agent labels denoted by l, and over which variable x ranges, are the ones associated with each AgentSpeak(F) program during the translation process. That is, 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 $(\mathsf{Does}\ l\ a)$, 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 (rather than from the point of view, i.e., belief, of an agent). 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 formulæ into predicates accessing the AgentSpeak(F) data structures modelled in the model checker's input language. This is done in accordance with the definitions given in [3].

3. Slicing Logic Programs

Perhaps the closest existing work to our goal of program slicing for AgentSpeak is that on slicing for logic programs, and so we here provide a brief introduction to previous work in this area. In one of the earliest papers on slicing logic programs, Zhao, Cheng, and Ushijima present a graph-theoretic representation of a concurrent logic program, which can be used for slicing [13]. An arc-classified digraph called *Literal Dependence Net* is used to represent four types of dependencies of concurrent logic programs: control, data, synchronisation, and communication dependencies. Subsequently, a backward slicing algorithm for Prolog was presented by Schoenig and Ducassé in [9] which is able to carry out slicing with greater precision than the approach in [13]. In [9], 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". More recently, Zhao, Cheng, and Ushijima extended their approach, using what they call an Argument Dependence Net [14]. They use the same principles as in their previous work, but refine the program representation to have annotations on dependence at the level of arguments rather literals.

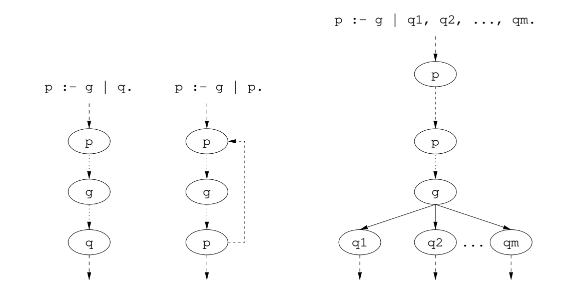
Slicing in the context of those authors is intended for debugging, software maintenance and understanding, and so on. Therefore, the more details of a program can be eliminated, the better. For our present purposes, Zhao's early work suffices, as we do not need slicing at the level of arguments. Instead of a slice for a particular variable, as usual in software engineering approaches, we here aim at removing plans (whole clauses) based on their influence on the truth of a predicate (rather than variable) that appears in a property specification (under certain modalities).

The approach of Schoening *et al.* works for Prolog programs. Although AgentSpeak is similar to Prolog in many respects, which would indicate that we might base our algorithm on [9], an AgentSpeak plan has the exact same structure of a guarded clause. The slicing algorithm proposed by Zhao *et al.* is specific to Guarded Horn Clauses, so their approach is a better candidate as a basis for ours. Besides, we do not need to generate *executable slices* (the main motivation in [9]), as we are only interested in preserving the truth of particular properties of the system.

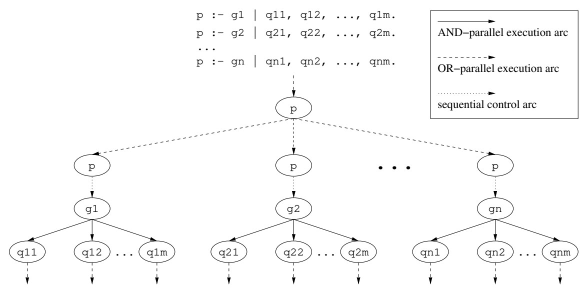
For these reasons, we chose to use the technique by Zhao *et al.* presented in [13] as a basis for our slicing algorithm for AgentSpeak. Note that their work is intended for concurrent logic programs, where body literals are AND processes, different clauses of a procedure are OR processes, shared variables relate to process communication and synchronisation, etc. However, all such dependencies apply to any logic program, as Zhao *et al.* observe themselves [13]. 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 to dependencies which must be dealt with in slicing any logic program (thus in our case too).

Below, we summarise the approach presented in [13], which will be used in the algorithm we introduce in Section 4. It is heavily based on two representations of a logic program. The first, called *And/Or Parallel Control-Flow Net* (CFN), is an arc-classified digraph (directed graph) where control-flow dependencies are annotated. The second is called *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 CFN can be understood informally from the rules presented in Figure 2. Note that, as we will be dealing with slicing sets of AgentSpeak plans (each plan having the same structure of a guarded clause), we have not reproduced here the rules



(a) Iterative Clause (b) General Clause



(c) Procedure for `p`

Figure 2. CFN Generation Rules [13].

given in [13] for unit clauses and goal clauses, as these are not relevant for presenting 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, synchronisation and communication dependences among literals. For this, another structure is needed, the so called definition-use net (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 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 [5]); 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 [13]. 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. *Synchronisation* in concurrent logic programs relates to two types of dependences in logic programs in general: dependences 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 dependences between literals in different clauses. The definition of *Literal Dependence Net* (LDN) is then an arc-classified digraph containing all four types of dependencies mentioned above (control, data, synchronisation, and communication).

A *static slicing criteria* is defined in [13] 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.

4. Slicing AgentSpeak

In our approach, the system to be sliced is given by a set of AgentSpeak programs, one for each agent, and an abstract representation of the environment, stating which facts about the environment (which may then become agents' beliefs through perception of the environment) are changed, either by the agents' actions or spontaneously, in case of dynamic environments. The environment is thus abstractly represented by a set of rules with one action in the left-hand side and a sequence of possible belief changes in the form of addition or deletion of predicates. In case of dynamic environments, rules can have empty left-hand sides. The environment also has a set of facts (in Prolog terminology) representing what is initially true of the environment.

Besides the system components, 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 restricted BDI logic defined in Section 2. The input to the algorithm 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. The slicing algorithm 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*, as presented in Section 3. When matching literals in different parts of the programs, AgentSpeak notations such as '**+**', '**−**', '**!**', and '**?**' should be considered as part of the predicate symbol. The only extra care to be taken in such matching is that a **!**g in the body of a plan matches a **+!**g in the triggering events of plans.

Initially, an LDN is created for each individual AgentSpeak program. Then the environment's LDN is created and connected to the various agents' LDNs 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 matching 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 all actions to that rule, as these belief changes can happen at any time.)
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.

In order to make the algorithm shown in the next stage clearer, we introduce the following terminology for the nodes of the LDN created for the individual AgentSpeak codes. We call a t-node any node of the LDN that was created for the triggering event[1] 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.

An example system and its corresponding LDN is shown in Figure 3. In the figure, most plan contexts (i.e., guards) are omitted for the sake of clarity.

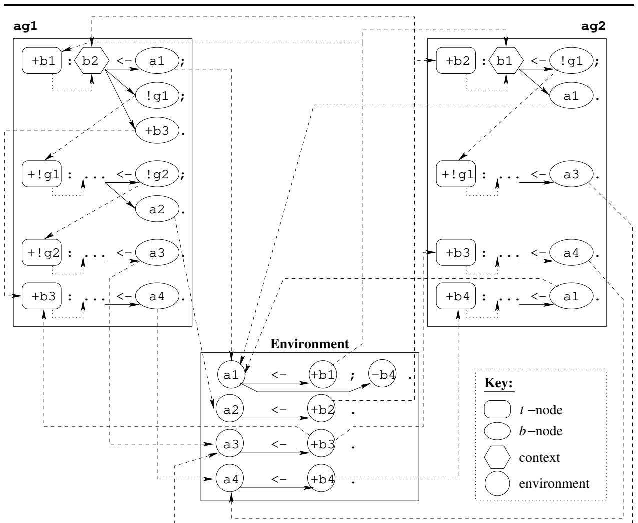


Figure 3. An Abstract Example.

Stage II Once the LDN is created, plans are marked according to Algorithm 1. It takes as input the system code (`System`), i.e., a set of AgentSpeak agent programs A and the *Environment* representation E; the LDN generated in the previous stage (`LDN`); and the property for which one intends to later model check (`Property`).

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

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

Stage III At this stage, a "slice" of the system is obtained by simply deleting all plans that were never marked throughout the execution of the algorithm 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 on whether the overall system satisfies the given property.

4.1. An Abstract Example

For the example shown in Figure 3, 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 bodies 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 that same system except that the body of `ag2`'s last plan is changed from `a1` to `a3`. With this changed system, and the same `Property` $= \Diamond(\mathsf{Des}$ `ag1 g2`$)$, the plans that 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 are as follows: only the plans with

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

triggering events `+b1` and `+!g1` remain for `ag1`, and only plan `+b2` remains for `ag2` (plans are referred to by their triggering events, which is this particular example is unambiguous). Model checking for the property can be done on this particular slice of the system. While it may be counter-intuitive that a plan for `+!g2` is left out of the slice even though `g2` appears in the property, this is correct according to the interpretation we have given to the Des modality. By that definition, in order for an agent to desire g, no plan for that goal is required; 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 indeed required (in practice, an applicable plan). So, 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 system's slice that is generated when that property is used as slicing criterion.

4.2. Correctness Outline

In this section we provide the main ideas that are to be used in proving correctness results for our slicing algorithm. We first make clear what we mean by correctness of the slicing algorithm, in the following definition. Recall that, in our approach, a system is given by set of AgentSpeak agents A situated in an environment E; the slicing algorithm takes A, E and a property P as parameters and returns A', a set of AgentSpeak programs that are sliced down from A.

Definition 1 (Slicing Algorithm Correctness) *A slicing algorithm for AgentSpeak σ is correct if for any finite set of AgentSpeak programs A, abstract environment E, and property P, for A' such that $\sigma(A, E, P) = A'$, $A, E \models P$ if and only if $A', E \models P$.*

The proof that our algorithm is correct as per the definition above is based on five lemmas, one for each of the basic cases of formulæ of our property specification language. In the lemmas, we ignore the consequences of inter-agent communication[2] in the interpretation of AgentSpeak agents; that is, beliefs are only changed from perception of the environment, and goals derive from such changes (rather than, e.g., requests from other agents).

Lemma 1 (Belief subformula) *Formula $(\mathsf{Bel}\ ag\ b)$, with its definition given in [3], can only become true in regard to an AgentSpeak agent ag under two circumstances: (i) when $+b$ appears in the body of one of the agent's plans, or (ii) by belief revision from the agent's perception of the environment. Any plan in the system can make either (i) or (ii) happen if, and only if, it is marked by Algorithm 1 whenever $(\mathsf{Bel}\ ag\ b)$ is a subformula of property P.*

Proof (Sketch) The proof uses the inference rules that define the transition relation within the operational semantics of AgentSpeak, and the assumptions about an agent's belief revision process, to show that indeed only cases (i) and (ii) make such formulæ true, and that these cases 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). Then, assuming the LDN generation algorithm is correct, 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 to those nodes b_i, hence are reachable and marked in the loop at line 6 of the algorithm. □

We omit the remaining four lemmas, for subformulæ with Des, Int, and Does modalities, and AgentSpeak atomic formulæ (used to refer to facts about the environment), as they have similar enunciations (except that case (ii), regarding belief revision, is only relevant to Bel) and proofs, with reference to the particular reachability problem and the appropriate lines of the algorithm. From this, we may prove the correctness of the whole slicing algorithm as follows.

Theorem 1 (Slicing Algorithm Correctness) *The slicing algorithm for AgentSpeak introduced in this paper is correct in the sense of Definition 1.*

Proof (Sketch) By structural induction on the *wff* of the logic used to write the specifications, using the five lemmas that refer to the base cases. □

5. Autonomous Mars Rover: A Case Study on Intra-Agent Plan Slicing

The development of autonomous rovers for planet exploration is an important aim of the research on "remote agents" carried out at space agencies [6]. We illustrate our slicing technique with an abstract version of a Mars exploration scenario, characterising a typical day of activity of rovers such as Sojourner (in the Mars Pathfinder mission). The ideas used here for creating such scenario were mainly taken from [11] (and to a lesser extent from [1]).

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 [11] 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;

2 Note that we can assume there is no communication between agents without loss of generality. The abstract specification of the environment can be used to model beliefs that are changed by the execution of "special" actions representing the effects of inter-agent communication.

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 was collected, and that particular rock could not be visited again, hence a science opportunity was lost. This is an example mentioned in that paper where more flexibility in the exploration rover control software is required.

That paper also describes a more flexible approach to sending instructions to a rover. This is not included here, but we took that description into account when modelling our application. They also mention that the rover is instructed to be especially attentive to "green patches" on rocks. This is likely to be an interesting science opportunity and so the rover should always give priority to examining such rocks if they turn up in its way to another target. Also, 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 is transmitted back to earth before it runs out of energy. Thus, other activities should be interrupted if carrying them out will mean the rover will not have enough energy to downlink collected data back to Earth.

Although we tried, in the AgentSpeak agent we developed for this application, to account for a greater flexibility for exploration rovers (as aimed in [11]) 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 one day of operation. Planning for such remote agents is a lot more complicated, as resources (computational or otherwise) that can be used in an actual rover are greatly restricted for technological and financial reasons.

The AgentSpeak code for this simplified Mars rover has 25 plans[3]. 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 remote agents such as the one discussed here. Thus, that code is an interesting example on which to apply our slicing technique, as described next.

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 property can be determined. This is similar to the motivation for removing clauses in traditional logic programs. It reduces the length of computations of individual intentions; note however that automata-theoretic model checking already avoid 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, yet another source of state space reduction can happen by slicing AgentSpeak(L) programs. Whenever all the plans that are used to handle particular external events can be removed, this greatly reduces the state space, given that 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. An alternative reduction would be to avoid the environment to generate such events in the first place (considering that they will not affect the property being verified anyway). Because the environment code is not usually in AgentSpeak(L), but is provided by the user, automatic slicing is less practical in this way. The user would have to remove, from its own code, the generation of the events that the algorithm would determine as safe to slice out.

We next show two examples of specifications that have been verified for the implemented system. In the specifications, *amr* is used to denote the agent (the *autonomous Mars rover*). An example of the first type of state-space reduction (reducing the path length of the computation associated with a particular intention), is as follows. Suppose the agent's original plan library did not include plans `r1–r4` (the ones which allow the rover to react to possible alternative targets, sundown, etc.). This would mean the agent would not, in any case, have more than a single intention at a time. Still, consider that the following property is to be checked (thus being our slicing criterion):

$$\Box((\mathsf{Does}\ amr\ \texttt{place\_spectrometer\_arm\_at(R)}) \rightarrow (\mathsf{Bel}\ amr\ \texttt{correctly\_positioned\_to\_examine(R)})) \quad (1)$$

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

Because plans `c1–c4` (the ones used for the agent to transmit back to the ground team all data it has gathered) can only become intended after some point in the execution where `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 (i.e., that level of detail of the intention execution is irrelevant for the given property). The generated slice for the above property does not include plans `c1–c4`.

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

3 Because of space limitation, we cannot include any of the AgentSpeak code used for the rover here. However, we have made it available on the Web at `http://www.durham.ac.uk/r.bordini/Publications/PubApp/AAMAS04-CR/amr.asl`.

$$\Box((\mathsf{Int}\ amr\ \texttt{transmit\_all\_remaining\_data(22)}) \rightarrow \Diamond\neg((\mathsf{Bel}\ amr\ \texttt{data(specData,souffle,22,\_)})\wedge \neg(\mathsf{Bel}\ amr\ \texttt{downlink(ground,specData,souffle,22)}))) \quad (2)$$

which means that in any execution path, 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 downlinked back to the ground team (this ensure, e.g., that the rover does not run out of batteries before it finishes the important task of transmitting all gathered data).

With the above slicing criterion, plan `r3` (a plan for reacting to possible ordinary targets; there is a separate one for reacting to rocks with green patches) 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 is being transmitted back to Earth, this could generate a large number of different system states in which the two focuses of attentions are being dealt with simultaneously by the rover.

Experiments were run on a machine with an MP 2000+ (1666 MHz) processor with 256K cache and 2GB of RAM (266 MHz). For specification (1), SPIN used 606MB of memory (1.18×10^6 states in the system) and took 86s to complete model checking. After slicing, numbers went down to 407MB ($945,165$ states) and 64s. This means a reduction of 25.6% on the time to model check, and a 33% reduction in memory usage. For specification (2), SPIN used 938MB of memory (2.87×10^6 states), and took 218s to complete. After slicing, this went down to 746MB (2.12×10^6 states) and 162s. This means a reduction of about 26% on the time to model check, and 21% on memory usage. Interestingly, using SPIN's built-in slicing algorithm does not reduce the state space at all.

6. Conclusions

We have presented a new algorithm for property-based slicing specifically targeted at AgentSpeak programs. Given a set of AgentSpeak programs forming a multi-agent system, we are now able to derive a second set of programs that has a smaller state space, yet is equivalent to the original one with respect to the property under consideration. This work forms part of our ongoing programme on the verification of multi-agent systems. Initial experimental results show a significant reduction in the state space, thus indicating that this approach can have an important impact on the practicality of automatic agent verification. An automatic AgentSpeak slicer based on this work is still being implemented, but initial complexity analysis shows there should not be an issue with the slicing time prior to model checking.

References

[1] J. Biesiadecki, M. W. Maimone, and J. Morrison. The Athena SDM rover: A testbed for mars rover mobility. In *6th Int. Symposium on AI, Robotics and Automation in Space (ISAIRAS-01), Montreal, Canada*, 2001.

[2] R. H. Bordini, M. Fisher, C. Pardavila, and M. Wooldridge. Model checking AgentSpeak. In *Proc. 2nd Int. Joint Conf. on Autonomous Agents and Multi-Agent Systems (AAMAS-2003), Melbourne, Australia*, pages 409–416, New York, NY, 2003. ACM Press.

[3] 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*, 2004. To appear.

[4] R. H. Bordini, W. Visser, M. Fisher, C. Pardavila, and M. Wooldridge. Model checking multi-agent programs with CASP. In *Proc. 15th Conf. on Computer-Aided Verification (CAV-2003), Boulder, CO*, LNCS 2725, pages 110–113. Springer-Verlag, Berlin, 2003. Tool description.

[5] M. Krishna Rao, D. Kapur, and R. Shyamasundar. Proving termination of GHC programs. In D. S. Warren, editor, *Proc. 10th Int. Conf. on Logic Programming, Budapest, Hungary*, pages 720–736, Cambridge, MA, 1993. MIT Press.

[6] 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(1–2):5–47, 1998.

[7] A. S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In *Proc. 7th Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW'96), Eindhoven, The Netherlands*, LNAI 1038, pages 42–55. Springer-Verlag, London, 1996.

[8] A. S. Rao and M. P. Georgeff. Decision procedures for BDI logics. *J. Logic and Computation*, 8(3):293–343, 1998.

[9] S. Schoenig and M. Ducassé. A backward slicing algorithm for Prolog. In *Proc. Third International Symposium on Static Analysis (SAS'96), Aachen, Germany*, LNCS 1145, pages 317–331. Springer-Verlag, 1996.

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

[11] R. Washington, K. Golden, J. Bresina, D. E. Smith, C. Anderson, and T. Smith. Autonomous rovers for mars exploration. In *Aerospace Conference, Aspen, CO*. IEEE, 1999.

[12] M. Wooldridge. *Reasoning about Rational Agents*. The MIT Press, Cambridge, MA, 2000.

[13] J. Zhao, J. Cheng, and K. Ushijima. Literal dependence net and its use in concurrent logic programming environment. In *Proc. Workshop on Parallel Logic Programming, held with FGCS'94, ICOT, Tokyo*, pages 127–141, 1994.

[14] J. Zhao, J. Cheng, and K. Ushijima. Slicing concurrent logic programs. In *Proc. 2nd Fuji Int. Workshop on Functional and Logic Programming, Shonan Village Center, Japan*, pages 143–162. World Scientific, 1997.