Slicing Agent Programs for more Efficient Verification[⋆]

Michael Winikoff[1], Louise Dennis[2], and Michael Fisher[2]

[1] University of Otago, New Zealand
michael.winikoff@otago.ac.nz
http://infosci.otago.ac.nz/michael-winikoff
[2] Liverpool University, UK
{l.a.dennis,mfisher}@liverpool.ac.uk

Abstract. Agent programs are increasingly used as the core high-level decision-making components within a range of autonomous systems and, as the deployment of such systems in safety-critical scenarios develops, the need for strong and trustworthy *verification* becomes acute. Formal verification techniques such as model-checking provide this high level of assurance yet they are typically both complex and slow to deploy. In this paper we introduce, develop and evaluate a *program slicing* technique that significantly improves the efficiency of such verification, hence providing more effective routes to the assurance of safety, reliability, and ethics in autonomous systems.

Keywords: Formal Verification · Program Analysis · Agent-Oriented Programming Languages

1 Introduction

The study of agent programming languages is becoming increasingly important not just from an academic viewpoint but because agent programs now play a central role in many *autonomous systems*. The need for transparency and explainability, in particular, is leading to the development of *hybrid agent architectures* for autonomous systems whereby a *rational agent* [26] provides the core high-level decision-making capabilities within the autonomous systems architecture. This approach leads naturally to clarity, flexibility and verifiability [15,12]. Specifically, a rational agent is not only able to take independent decisions but has explicit notions of the *motivations* that lead it to select one option over another. The predominant model of rational agency, and one that we follow here, is that of BDI ('Beliefs', 'Desires', and 'Intentions') [7], in which the agent's assessments about the state of the world (and itself) are captured as *beliefs*, the agent's long-term motivations are captured as *desires*, while the agent's immediate motivations are captured as *intentions* [20].

There is a wide range of programming languages that use the idea of rational agency, often the BDI approach, as their central model, for example AgentSpeak [21], Jason [5], GOAL [16], GWENDOLEN [11]), and others [2,3]. As these become deployed in increasingly sophisticated and complex scenarios, there is increased need for much

[⋆] Work supported in the UK by EPSRC through the projects EP/L024845, EP/R026084, and EP/R026092.

greater assurance through *verification and validation*. Although the most common approach to software verification is through *testing*, [1], Winikoff *et al.* [23,25] show how assurance of agent programs cannot feasibly be carried out using traditional software testing, leading us to *formal verification*.

Formal verification is a mathematically well-founded process for proving that a specification given in formal logic matches the system in question. For a specific logical property, ϕ, there are many different approaches to this [14,9,6], ranging from deductive verification against a logical description of the system ψ_S (i.e., $\vdash \psi_S \Rightarrow \phi$) to the algorithmic verification of the property against a model of the system, M_S (i.e., $M_S \models \phi$). The latter has been extremely successful in Computer Science and Artificial Intelligence, primarily through the *model checking* approach [8]. This takes a description of the system in question, capturing all possible executions, and then checks the logical property against this description (and, hence, against all possible executions).

If (rational) agents are to be used at the core of increasingly sophisticated autonomous systems, it therefore seems natural to explore the model checking of these agent programs. There have been several developments in this direction [19,17], with the most well-developed being that of AJPF/MCAPL [12,15]. This verification approach has been used throughout a range of work tackling applications in autonomous aircraft, spacecraft and road vehicles [22,18,10,12], where a (central) rational agent is verified using model-checking in order to assess all high-level decision-making.

While very useful, such formal verification can be extremely slow, even for relatively small programs [12]. Around a decade ago, there was initial work by Bordini *et al.* [4] aiming to improve the efficiency of agent program model checking by using *slicing*. The basic idea (see Figure 1) is that instead of model-checking a property ϕ with respect to program π situated in environment $\mathcal{E}$, we instead model-check a *sliced program* π'. The sliced program is a simplified version of π where (some) parts of the program that do not affect the truth of ϕ have been removed. This can result in a program π' that is smaller and substantively faster to model check. For example, Bordini *et al.* [4] found a 61% reduction in run-time to check a particular property.

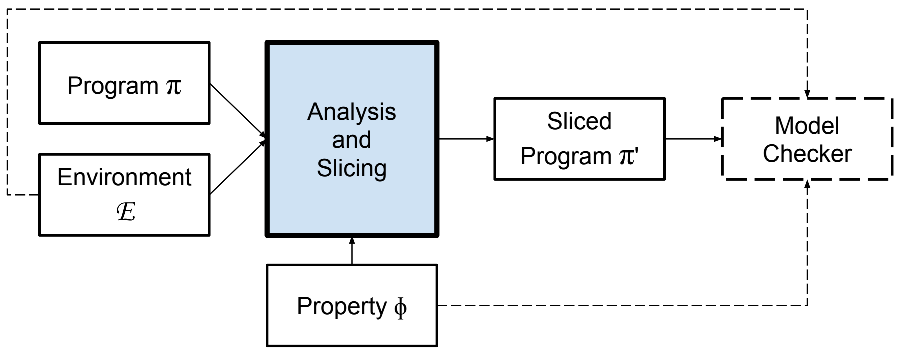


Fig. 1: High-level view of the process

This paper advances the 2009 paper, updating the algorithm for the contemporary verification framework and proposing a new, and improved, slicing method. We begin by briefly introducing required background material, including reviewing the slicing algorithm of Bordini *et al*. We then present the new slicing method, provide evaluation results, and conclude with a brief discussion of future work (which includes a formal proof of corectness).

2 Background

We briefly review required background, including the specific BDI language we use, GWENDOLEN [11], the mapping from a BDI program to a graph structure, and the slicing method proposed by Bordini *et al* [4].

2.1 BDI Programming Languages & Gwendolen

The common core of BDI languages is that an agent program is a collection of *plans*. Each plan $t : c \leftarrow s_1; \ldots ; s_n$ comprises a *trigger* t (e.g. posting a sub-goal, denoted $+!g$, or a change to the agent's beliefs, denoted $+b$ or $-b$), a *context condition* c that indicates in which situation the plan is applicable, and a *plan body*. The plan body is a sequence of steps s_i (or more generally a program). Steps include belief updates (adding a belief $+b$ or removing a belief $-b$), testing conditions ($?c$), posting sub-goals $!g$ (but $+!g$ in GWENDOLEN), and actions a.

The core execution cycle is that when a trigger t' is posted, all the *relevant* plans (those whose triggers t unify with t') are collected. A relevant plan is *applicable* if its context condition currently holds. An applicable plan is selected, and that plan's body is executed. Execution is interleaved with processing of other incoming percepts/events and with parallel execution of other plans, in response to other triggers. Details vary between languages (e.g. see Winikoff [24]). We assume the common practice of considering relevant plans in sequential order.

Many BDI languages incorporate a failure handling mechanism where if a step fails (e.g. an action's preconditions are not met), the plan body it is a part of fails. Failure handling then considers the trigger for that plan, and seeks to use alternative plans to handle it. This is done by re-posting the trigger, and using other plans to handle it.

In the remainder of this paper we use the GWENDOLEN notation, where a plan is written: "$t : \{c\} \leftarrow s_1, \ldots , s_n;$". Context conditions are logical combinations ("," denotes conjunction, and "~" denotes negation) of beliefs ("B") and goals being pursued ("G"). The notation $\mathsf{perf}(a)$ denotes performing an action, and $^*c$ means "wait for condition c". Figure 2 shows a simple GWENDOLEN program implementing a cruise control, taken from the GWENDOLEN distribution. Note that the annotation [achieve] indicates an achievement goal, which can be explained in terms of the following[3] "meta-plan": $+!achieve(G) : \{\mathsf{B}\ G\}.\ \ +!achieve(G) : \{\tilde{}\ \mathsf{B}\ G\} \leftarrow +!G; +!achieve(G)$, i.e. keep trying $+!G$ until G is believed.

It is important to note that although the presentation in this paper uses the GWENDOLEN notation, the language features are very similar to those of other BDI languages,

[3] An empty plan body is indicated by eliding the "$\leftarrow$".

```
:Reasoning Rules:
can_accelerate :− safe, ~ driver_accelerates, ~ driver_brakes;

:Initial Goals:
at_speed_limit [achieve]

:Plans:
+! at_speed_limit [achieve] : {B can_accelerate}
  ← perf(accelerate), wait;
+! at_speed_limit [achieve] : {~B safe} ← ∗safe;
+! at_speed_limit [achieve] : {B driver_accelerates}
  ← ∗~driver_accelerates;
+! at_speed_limit [achieve] : {B driver_brakes}
  ← ∗~driver_brakes;
+at_speed_limit: {B can_accelerate, B at_speed_limit}
  ← perf(maintain_speed);
−at_speed_limit: {~G at_speed_limit [achieve],
  ~B at_speed_limit} ← +! at_speed_limit[achieve];
−safe: {~B driver_brakes, ~B safe} ← perf(brake);
+driver_accelerates: {B safe, ~B driver_brakes,
  B driver_accelerates} ← perf(accelerate);
+driver_brakes: {B driver_brakes} ← perf(brake);
```

Fig. 2: GWENDOLEN program for cruise control.

and changing this paper to apply to another BDI language would require only two very minor changes[4].

An agent exists in an *environment*, which needs to be modelled for verification purposes. Whereas Bordini *et al.* model the environment using a collection of plans that connect each action to its post-conditions, we instead follow Dennis *et al.* [12] and do not define a direct link between actions and their post-conditions. Instead, we define a collection of possible exogenous belief updates that can occur. This representation is more realistic, since often, in real domains, there is a delay between an action being commenced, and the effects of that action manifesting. Additionally, the effects of an action are not usually guaranteed. For example, performing an accelerate action does not necessarily result in being at the speed limit, instead, at some future point the sensors may indicate that the car has reached the speed limit. We define a set of *exogenous belief updates*, $\mathcal{B}$, for instance, for the cruise control example the set of relevant exogenous belief updates is $\mathcal{B} = \{\mathsf{at\_speed\_limit}, \mathsf{safe}, \mathsf{driver\_accelerates}, \mathsf{driver\_brakes}\}$. For each $b \in \mathcal{B}$ a percept $+b$ or $-b$ can occur at any time.

[4] Specifically, there is a minor change to the graph construction, noted in a later footnote, and the definition of a belief link would change very slightly (replacing "*" with "?").

For verification, we define a simple language based on the property specification language used in MCAPL/AJPF:

$$\psi ::= \mathsf{Bel}\ a\ p \mid \mathsf{Des}\ a\ p \mid \mathsf{Int}\ a\ p \mid \mathsf{Does}\ a\ p \tag{1}$$

$$\varphi\ ::\ \ \psi \mid \varphi \wedge \varphi \mid \phi \vee \varphi \mid \neg\varphi \mid \varphi \rightarrow \varphi \mid \Box\varphi \mid \Diamond\varphi \tag{2}$$

The full semantics of this language is given by Dennis *et al.* [12]. It is based on Propositional Linear Temporal Logic (PLTL) [13] defined over program traces where the expressions in (2) are as standard in presentations of PLTL and $\mathsf{Bel}\ a\ p$ means that p appears in the belief base of agent a, $\mathsf{Des}\ a\ p$ means that p appears in the goal base of agent a, $\mathsf{Int}\ a\ p$ means that p appears in the goal base of agent a *and also* that a plan has been selected to handle the goal, and $\mathsf{Does}\ a\ p$ means that agent a has executed $\mathsf{perf}(p)$.

2.2 Mapping to a Graph Structure

We map each program clause of the form $t : c \leftarrow s_1, \ldots, s_n$ to a graph that has a trigger node[5] t, a context node c, and step nodes s_i $(1 \leq i \leq n)$. We assume that each node has a unique ID (which allows the occurrence of, for instance, a step such as $\mathsf{perf(accelerate)}$ in multiple places in the program to be represented by multiple nodes with unique identifiers, but the same name). We denote the name of a node as $\widehat{N}$ where N is the node's unique ID.

We then represent the plan's structure by defining *basic edges* (denoted $A \rightarrow B$) from c to s_1 and from each s_i to s_{i+1}. We also define *numbered edges* (denoted $A \overset{n}{\rightarrow} B$) from t to each plan's context condition. The initial mapping is extended with *triggering edges* (denoted $A \dashrightarrow B$) from step s_i to trigger t where[6] $\widehat{s_i} = \widehat{t}$). Figure 3 shows the graph plan structure corresponding to a simple program[7]. In the figure, the trigger node is a rectangle, step nodes are ovals, and context nodes are hexagons. The dashed line indicates a triggering edge, and the numbers are numbered edges (as explained above).

Space precludes a detailed discussion of the renaming required to handle multiple agents. Briefly, each agent's plans and beliefs are renamed apart.

2.3 Original Slicing Method

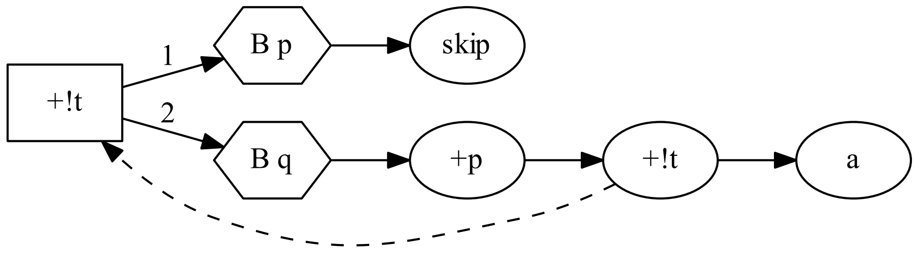
The approach of Bordini *et al.*, which is inspired by a slicing algorithm for a concurrent logic programming language [27], comprises three stages: (i) create a *literal dependence net* (LDN); (ii) mark nodes relevant to checking the property of interest ϕ [4, Algorithm 1, Page 1405]; and (iii) remove any plans that are not marked, yielding π'.

Building the LDN: The LDN takes, as a starting point, the basic mapping described in the previous subsection. It modifies this by: (i) instead of linking each step to the next

[5] There is one slight difference between our mapping and that used by Bordini *et al*: we do not have a trigger node for each plan, instead we use a single common trigger node for plans that share the same trigger.

[6] For AgentSpeak we would need to adjust this slightly, since the trigger node for a sub-goal is named $+!g$ but the step is named $!g$.

[7] The program was constructed to illustrate features of the graph representation.



```
+! t : {B p} ← skip;                                    1
+! t : {B q} ← +p, +!t, perf(a);                        2
```

Fig. 3: Simple program and the corresponding graph

step ($s_i \to s_{i+1}$), it instead has a link from the context condition directly to each step ($c \to s_i$ for each $s_i, 1 \leq i \leq n$); and (ii) it adds *belief links*, representing dependencies via the belief base: there is a link from each belief update to any context condition (or test) that depends on that belief[8]. Figure 4 shows the graph constructed for the same simple example. As before, a dashed line is a triggering edge. A dotted edge denotes a belief link. Observe that information about the order of steps, e.g. in the second plan body, is not preserved.

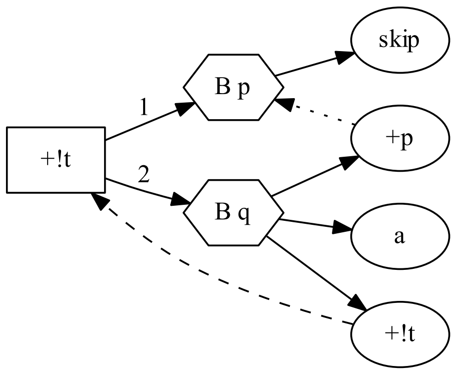


Fig. 4: Simple graph showing original slicing method's construction

Marking Nodes: Nodes are marked to indicate whether they affect the property ϕ being verified. The marking process considers every plan. Let te be the node corresponding to the trigger of the plan[9]. If the property of interest contains Bel $ag\ p$, then the plan is

[8] Handling reasoning rules requires additional complexity.

[9] We actually use the context condition, since we do not have a unique trigger node for each plan.

marked iff there is a node with name $+p$ or $-p$ that is reachable from te in the LDN. For $\mathsf{Des}\ ag\ g$ the plan is marked iff there is a *step* node with name $+!g$ reachable from te, for $\mathsf{Int}\ ag\ g$ it is marked iff a *trigger* node with name $+!g$ is reachable from te, and for $\mathsf{Does}\ ag\ p$ it is marked iff a *step* node with name $\mathsf{perf}(p)$ is reachable from te. The difference between $\mathsf{Des}\ ag\ p$ and $\mathsf{Int}\ ag\ p$ is that the latter requires a path to a trigger node.

Given[10] $\phi = \Box(\mathsf{Does}\ car\ \mathsf{accelerate} \rightarrow \mathsf{Bel}\ car\ \mathsf{safe})$, in the example above, the nodes of interest are those corresponding to the action $\mathsf{perf}(\mathsf{accelerate})$ or to a belief update that affects safe (but since this is updated exogenously, it does not appear in any plan). Therefore, the plans that are marked are those that have a path from their context condition to a perf(accelerate) node.

3 Improved Slicing Method

The slicing algorithm proposed by Bordini *et al.* [4] has a number of "missed opportunities" where it does not take into account information that is available. These include:

1. the ordering of execution, e.g. the sequence of steps, is not exploited, nor is the order of the plans, or to make use of knowledge about failure handling;
2. the initial goals are ignored, which means that even if a node cannot be reached in achieving these goals, it is still considered; and
3. the structure of ϕ is not considered: the algorithm for marking plans [4, Algorithm 1, Page 1405] considers only the presence of sub-formulae of the form $\mathsf{Bel}\ ag\ b$, $\mathsf{Des}\ ag\ b$, $\mathsf{Int}\ ag\ g$ and $\mathsf{Does}\ ag\ a$ (and atomic b), the logical structure of ϕ is not considered.

To illustrate where the original slicing algorithm misses out on useful distinctions consider a program that includes one plan with body "$?p; \mathsf{perf}(a)$"[11] where the action a is of interest (i.e. ϕ includes $\mathsf{Does}\ ag\ a$), and a second plan with body "$-p$". Now, the second plan is clearly important: its execution may prevent the first plan from progressing beyond its first step. However, it is possible that the second plan will never be executed before the first plan. The original slicing method does not take this into account.

Our proposed slicing process addresses the first two of these missed opportunities. Firstly, it constructs a dependency graph based on the language's semantics, including modelling failure handling, and handling GWENDOLEN language features. This allows sequencing information to be exploited in the slicing analysis. Secondly, we distinguish between parts of the agent program that are unreachable, and hence simply removed, and parts that are reachable, but can be compressed and simplified without affecting the verification outcome. Our analysis is therefore more fine-grained in that it allows *parts* of a plan to be removed. This turns out to provide a substantial efficiency gain.

The slicing process comprises four steps: constructing the dependency graph (Section 3.2), removing nodes that are not reachable (Section 3.3), marking nodes that are

[10] The '$\Box$' is the standard temporal operator meaning "at all points in the future".

[11] Where $?p$ tests whether p holds, failing if it does not. This differs from the GWENDOLEN construct $*p$ which suspends the plan until p becomes true.

incompressible (Section 3.4), and compressing nodes that are unmarked (Section 3.5). However, before we begin the process, we need to deal with certain language features, which we do by transforming them away.

3.1 Transforming the Program

A number of language features pose challenges, as the earlier slicing algorithm is not clear how they should be dealt with. Specifically, the earlier paper does not explain how the algorithm deals with these constructs, and how to deal with them is not obvious. For instance, the example program used by Bordini *et al.* included use of the `.dropDesires` built-in action, but it is not clear how this is handled in the LDN.

First feature: *achievement goals*. As noted earlier, an achievement goal in GWENDOLEN can be explained in terms of a "meta plan". For each achievement goal G we introduce meta-plans with trigger $+!achieve(G)$, and replace $+!G[achieve]$ with $+!achieve(G)$.

Second feature: *context conditions that test goals*. In GWENDOLEN a context condition can include not only tests of whether certain beliefs are held ("B p"), but also tests of whether a goal is held by the agent ("G g"). This poses a challenge, because the slicing analysis needs to know about the steps that can affect the truth of a context condition, but whereas changes that affect belief conditions are explicit, whether a goal is held by the agent is affected by the goal being posted or achieved, and this is not always explicit.

We therefore need to transform the program to make goal status changes explicit. We do this using beliefs, $goal\_G$, associated with each goal G, this mechanism requires us to associate changes to this belief both with the explicit posting of new goals and with the implicit removal of goals. This is achieved as follows. First, when a goal is posted, we also update the corresponding belief, i.e. we replace any $+!G$ with $+goal\_G, +!G$. Next, whenever a goal is dropped, we add an explicit $-goal\_G$. A goal is dropped when any one of its plans concludes, so given a goal that is tested for, we add to each of its plans a final step $-goal\_G$. Note that a special case is when a plan for G ends with a recursive sub-goal $+!G$: in this case the belief $goal\_G$ is not modified. Another special case is achievement goals, for which the goal is dropped only in the first meta-plan, and where we replace $+!G[achieve]$ with $+goal\_G, +!achieve(G)$. Finally, we replace the condition G G with B $goal\_G$.

The code below shows the transformed program for the example (showing only the parts that were changed). The changes are: (i) the achievement goal has been realised by adding a meta-plan and removing "[achieve]" (not shown); and (ii) the condition G at_speed_limit has been replaced with B goal_at_speed_limit, and that belief about a goal is updated in the first plan (where it is dropped) and in the last plan (where the goal is adopted). An additional change (not shown) is that the reasoning rule (line 2 of Figure 2) is handled by unfolding it: this replaces can_accelerate with its definition of "safe, ~ driver_accelerates, ~ driver_brakes".

```
+! achieve_at_speed_limit:{B at_speed_limit} ← −goal_at_speed_limit.
+! achieve_at_speed_limit : {~B at_speed_limit}
  ← +! at_speed_limit, +!achieve_at_speed_limit;
```

```
−at_speed_limit : {˜B goal_at_speed_limit, ˜B at_speed_limit}
  ← +goal_at_speed_limit, +! achieve_at_speed_limit;
```

Third feature: *explicitly dropping goals.* Explicit goal dropping ("$-!G$") is not used in the example. We deal with this not by transforming the program, but by having an additional case for generating belief links when constructing the graph (see end of Section 3.2).

3.2 Constructing the Dependency Graph

We define the dependency graph as follows. We start with the initial mapping of the program presented in Section 2.2. We then define a *control link* (denoted $A \Rightarrow B$) as existing between two nodes under any of the conditions below[12].

1. As previously, we have $A \Rightarrow B$ from a step A to the relevant trigger node B (formally: $A \Rightarrow B$ if $A \dashrightarrow B$).
2. Rather than linking a context condition to *all* steps in the plan body, we link the steps of the plan body in sequence. Additionally, we correctly capture sub-goals by not having a link from posting a sub-goal to the next step. Instead, the link to the next step is from where the sub-goal is achieved (i.e. the last step of each plan that achieves it).
   Specifically, this means that we have $A \Rightarrow B$ when there is a basic edge $A \rightarrow B$ from a context node or from a step other than a sub-goal (formally: $A \Rightarrow B$ if $A {\rightarrow} B \wedge (context(A) \vee (step(A) \wedge \neg subgoal(A)))$).
   In the case where $A' {\rightarrow} B$ and A' is a sub-goal, then instead of having a control link from A' to B, we find the plans that are triggered by A', and have links from the last step of each such plan to B. In other words, we also have $A \Rightarrow B$ if A is the last step of a plan that is triggered by $+!G$, and B is the next step after the posting of the goal G. (formally: $A \Rightarrow B$ if $step(A) \wedge (\neg subgoal(A)) \wedge last(A) \wedge getCaller(A, D) \wedge next(D, B)$, where $getCaller(A, D)$ is true when D is the identifier of a step that has the same name as the trigger node of the plan containing A, and $next(D, B)$ is true when B is the next step after D, taking account of control-flow returning from the end of a plan[13])
3. We capture plan order by linking from the trigger goal to only the *first* plan's context condition. Specifically $A \Rightarrow B$ when there is a basic edge numbered 1 from a trigger to the *first* context condition. (formally: $A \Rightarrow B$ if $A \xrightarrow{1} B$).
4. We capture failures, including in plans other than the first, by having failure links. We link each context condition that can fail (i.e. excluding a plan condition that is "`true`") to the next plan's context condition (if there is one). We also link each step in a plan body to the next plan's context condition, unless the plan step is one that cannot fail. Formally: $A \Rightarrow B$ if $\exists C, G : ((context(A) \wedge \widehat{A} \neq \text{“}true\text{”} \wedge G \xrightarrow{n} A) \vee (step(A) \wedge canFail(A) \wedge getContext(A, C) \wedge G \xrightarrow{n} C)) \wedge G \xrightarrow{n+1} B$

[12] We assume predicates $context(N)$, $step(N)$, $subgoal(N)$, as well as $last(N)$ (true iff N is the last step in a plan), $getContext(A, C)$ (true iff C is the context condition of the plan in which A appears), and $canFail(A)$ (true iff the step A can fail).

[13] Formally: $next(A, B) \equiv \exists D : A {\rightarrow} B \vee (last(A) \wedge getCaller(A, D) \wedge next(D, B))$ and $getCaller(A, D) \equiv getContext(A, C) \wedge G \xrightarrow{n} C \wedge step(D) \wedge \widehat{G} = \widehat{D}$.

In addition to control links, we also define *belief links* (denoted $A \Rightarrow_{\mathsf{B}} B$) between A and B if A updates a belief that can affect the truth of a context condition or test/wait B, where B contains the belief that A updates. While this definition does not cater for reasoning rules, we have: $A \Rightarrow_{\mathsf{B}} B$ if $step(A) \wedge (context(B) \vee (step(B) \wedge \widehat{B} = *C)) \wedge contains(\widehat{B}, getBelief(\widehat{A}))$ where $getBelief(+b) = getBelief(-b) = b$, and $contains(\phi, b)$ is true iff ϕ contains b.

Finally, as mentioned earlier, we need to also add belief links that relate to explicitly dropping a goal ("$-!G$"). Semantically, dropping a goal explicitly is problematic. This is because it creates a situation where the execution of a plan can be aborted at any point. Consider the simple program below. Suppose that $+!g1$ is being pursued. At any point in the execution of the first, or the second plan, a percept may update b, resulting in $g1$ being dropped, and its plan aborted. This means that there is a dependency between $-!g1$ and *every* node that follows on from the body of a plan to handle $+!g1$. We therefore define that there is also a belief link from $-!g$ to any node N such that $+!g \Rightarrow^* N$, where N is a step, $+!g$ is a trigger node, and $\Rightarrow^*$ denotes the transitive closure of $\Rightarrow$.

```
+! g1 ← +! g2.
+! g2 ← s1, s2.
+b ← −!g1.
```

We exclude belief links if there is no possibility that A can influence B. In other words, if B *cannot* occur after A, then we suppress the belief link from A to B. This makes sense because, in this situation, even though A can change b, which occurs in the condition of B, the change cannot occur before b is checked, and therefore there is no dependency. The definition of when B cannot occur after A is somewhat complex, and omitted due to space reasons. Roughly speaking, B and A must have the same initial goal or exogenous event (otherwise they occur in parallel), and there cannot be a control edge path from A to B.

Figure 5 shows the graph for the same simple program. The numbers on edges refer to the numbered items in Section 3.2, with 2' denoting the second case of the second numbered item.

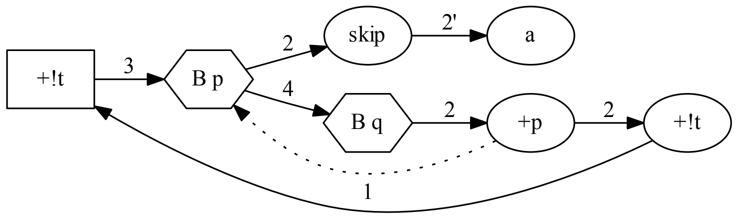


Fig. 5: Simple graph showing new slicing method's construction

3.3 Removing Unreachable Nodes

We analyse the graph to remove *unreachable* nodes. A node is *reachable* if there is a control link path to it from either the initial goal, or from an exogenous update. If neither of these is the case, then the node cannot be reached, and will never play a role in execution. It can therefore safely be removed from the graph. Note that since there is a control link path from each context condition to each step in its plan, if a step node is unreachable, that means that the whole plan is unreachable, and can be deleted, although a well-constructed program is unlikely to have unreachable plans.

3.4 Marking the Graph

Next, we mark nodes that play an essential role in determining the outcome of verification of the desired property ϕ. These are nodes that must be retained. Other atomic step nodes that remain unmarked are reachable, but can be *compressed* (i.e. replaced by a null-action "skip"). There are two ways in which a node can play an essential role, and hence be incompressible.

Firstly, a node A can directly affect the truth of ϕ (denoted $\mathit{directlyAffects}(A, \phi)$). For example, if ϕ includes the atomic property $\mathsf{Bel}\ a\ b$ then both $+b$ and $-b$ are marked as directly affecting ϕ. Similarly, following Bordini *et al.*, we define the same cases for desire, intention, and performing actions (see Section 2.3).

Secondly, a node can *indirectly* affect the truth of ϕ by changing a belief that can affect the subsequent execution. In order for this change to matter, it must occur before a subsequent node that directly affects ϕ. Changing a belief that affects subsequent execution corresponds to the notion of belief link, defined above, so this second case can be defined as $\mathit{indirectlyAffects}(A, \phi) \equiv \exists B, C : A \Rightarrow_{\mathsf{B}} B \wedge B \rightrightarrows^{*} C \wedge \mathit{directlyAffects}(C, \phi)$ where $B \rightrightarrows C \equiv B \Rightarrow C \vee B \Rightarrow_{\mathsf{B}} C$ and $\rightrightarrows^{*}$ is the usual "path of length 0 or more" operator. Note that in this case both A and B are marked: A can affect ϕ by changing a condition that affects the execution of B, and B can affect ϕ by allowing execution to take more than one possible path, depending on the condition modified by A.

We then propagate markings. The basic idea is that an unmarked step should be marked if it triggers a plan that can lead to a marked node. Formally we mark node A when $\mathit{step}(A) \wedge \mathit{trigger}(B) \wedge \widehat{A} = \widehat{B} \wedge B \Rightarrow^{*} C \wedge \mathit{marked}(C)$.

3.5 Compressing the Graph

Having marked the nodes that can affect ϕ, directly or indirectly, we can now simplify the program by "compressing" nodes that have not been marked. This is done via the following transformations, which are justified on semantic grounds. Note that since GWENDOLEN does not support disjunctions in context conditions, or the test step $?c$, the 3rd transformation cannot be done, and the 4th can only be done when $\bigvee c_i \equiv$ true, in which case no test is needed.

1. If a step is unmarked then replace it with the no-effect step "skip". Justification: If the step is unmarked, then its execution does not affect the verification property ϕ,

nor does it affect the future path of execution in a way that may affect ϕ. It therefore can be safely replaced with a "do nothing" step.

2. Replace "S , skip" or "skip , S" with just "S". This is a basic semantic equivalence, as long as the environment is *numerically ahistorical*, i.e. the result of performing an action does not depend on the number of actions performed. For example, consider an environment that includes a counter that is incremented each time an action a is performed. In this scenario, the result of executing $a; a$ is different to that of executing a. As long as the counter can play a role in the eventual truth of ϕ, we cannot compress or remove instances of a. Note that in verification one approach is to define an environment that at each point simply returns a nondeterministic subset of possible percepts [12]. This environment satisfies the assumption.
3. Any two adjacent plans which have bodies that are simply a single "skip" can be combined: $+!g : c_1 \leftarrow$ skip. $+!g : c_2 \leftarrow$ skip $\Rightarrow$ $+!g : c_1 \vee c_2 \leftarrow$ skip. This clearly preserves the execution semantics.
4. When a sub-goal has only relevant plans of the form "$+!g : c_i \leftarrow$ skip" then the plans can be deleted, and the sub-goal $+!g$ replaced with a simple test $?(\bigvee c_i)$ (if $\bigvee c_i$ is just "true" then $+!g$ can be replaced with "skip"). Again, this clearly preserves the semantics.
5. A plan triggered by an exogenous update that has a plan body that is just "skip" can be deleted, since, semantically, this has no effect: responding to an event by doing nothing is equivalent to ignoring the event.

Note that we only remove plans with empty ("skip") plan bodies if there are no other plans to handle that trigger. This differs from Bordini *et al*. The reason is that when considering failure handling, the presence of these plans can make a difference. For example, in Figure 6, if the context condition (lines 7 and 16) fails, then there is no alternative plan to attempt. In GWENDOLEN's semantics the failure to find *any* applicable plan forces the program into a tight loop in which perception is no longer polled[14]. The plans in lines 18-20 prevent this tight loop occurring.

Finally, note that while, for analysis purposes, we expand achievement goals using a meta-plan, when generating the final GWENDOLEN program we remove the meta-plans and go back to using achievement goals.

4 Evaluation

The previous section has presented the definitions of a new slicing analysis. Although this paper does not present a formal proof of correctness, we have explained along the way why the slicing algorithm works. In other words, we have given a sketch of correctness by construction. Further work includes a formal statement of correctness, along with a proof.

We have written software that takes a representation of a transformed GWENDOLEN program, and implements the slicing method described in the previous section. Specifically, the program transformation (Section 3.1) is done manually, but the graph generation, reachability analysis, and marking are all automated. The final compression step

[14] This feature of the language is not common in BDI languages and it is possible that Bordini *et al.* had not come across such behaviour when they were designing their algorithm

(Section 3.5) is performed manually. The software also implements the original slicing analysis of Bordini *et al.*, for comparison purposes. It is important to appreciate that the parts of the process that have been implemented are the complex parts of the analysis, whereas the manual parts are simple local and compositional steps. It is also worth noting that the implementation was done by transliterating the formal definitions given earlier into Prolog. This means that it is easy to see that the implementation correctly captures these definitions. However, the implementation is not efficient. Developing an efficient implementation is future work.

We applied the slicing method to two programs from the GWENDOLEN distribution. Both programs have been verified. The first was selected initially since it is simpler than other verified GWENDOLEN programs, so was a good starting point. The second was selected as a representative larger, and more complex, program. There are not many verified GWENDOLEN programs, and slicing and timing more programs is future work.

Applying the slicing method to the cruise control example results in the sliced program shown in the bottom part of Figure 6. The middle part of the figure shows the plans resulting from applying the old slicing method. Comparing the two slicing methods, we observe: (1) That the original slicing method appears to be overly eager to slice away plans that are essential to the execution (e.g. the first meta-plan that terminates the recursion is sliced away [not shown in the Figure]); and (2) That when the original slicing method retains a plan, it retains the whole plan, whereas the improved method can simplify the plan (e.g. removing the wait comparing lines 8-9 with line 17).

Property $\phi = \Box(\mathsf{Does}\ car\ \mathsf{accelerate} \rightarrow \mathsf{Bel}\ car\ \mathsf{safe})$ was verified using AJPF[15] against the original GWENDOLEN program, the program sliced using the Bordini *et al.* method, and the program sliced using the new method. The original program (which, as shown in Figure 2, has 9 plans) took 12.388 seconds to verify (user+sys time), whereas the sliced programs took respectively 5.116 and 5.26 seconds to verify. As shown in Figure 6, the original slicing method slices away 6 plans, keeping 3, whereas the new slicing method keeps 6 plans, but is able to slice away parts of the plans' bodies.

We also manually analysed a larger program which manages the physical configuration of a collection of autonomous Low Earth Orbit (LEO) satellites [12, Section 4]. The LEO program, which has 35 plans, is in the GWENDOLEN distribution, and the property that we verified is theorem 18, which states that "if the planning process succeeds then either the agent eventually believes it is maintaining the position or it believes it has a broken thruster". For this program and property, slicing using the old method (yielding a program with 24 plans) makes no difference to model checking performance. However, the new method (also yielding a program with 24 plans) is able to reduce the execution from around 38 minutes (117554 states) to around 27 minutes (67670 states). The reason why the new method does substantially better is that it is able to remove parts of plans, which the old method is not able to do. For this program, this considerably reduces the search space for the model checker.

[15] On a 3.2 GHz Intel Core i5 iMac with 16 GB RAM running OSX 10.10.3.

```
:Reasoning Rules:
can_accelerate :− safe, ~ driver_accelerates, ~ driver_brakes;
:Initial Goals:
at_speed_limit [achieve]

:Plans: // Bordini slicing
+! at_speed_limit [achieve] : {B can_accelerate} ←
        perf(accelerate),
        wait;
−at_speed_limit: {~G at_speed_limit [achieve],
         ~B at_speed_limit} ← +! at_speed_limit[achieve];
+driver_accelerates: {B safe, ~B driver_brakes,
         B driver_accelerates} ← perf(accelerate);

:Plans: // New slicing
+! at_speed_limit [achieve] : {B can_accelerate} ←
        perf(accelerate);
+! at_speed_limit [achieve] : {~B safe}; // ← skip
+! at_speed_limit [achieve] : {B driver_accelerates} ;
+! at_speed_limit [achieve] : {B driver_brakes} ;
−at_speed_limit: {~G at_speed_limit [achieve],
         ~B at_speed_limit} ← +! at_speed_limit[achieve];
+driver_accelerates: {B safe, ~B driver_brakes,
         B driver_accelerates} ← perf(accelerate);
```

Fig. 6: Sliced GWENDOLEN program

5 Discussion

We have extended the work of Bordini *et al.* [4] by updating it for the GWENDOLEN framework, defining a graph that reflects the execution semantics of the language (including failure handling), and using a more precise slicing method that is able to simplify plan bodies. We emphasise that this paper provides a full and precise formalisation of the method, and that we have implemented the complex parts of the process. By contrast, Bordini *et al.* do not actually define the precise construction of the LDN, instead they refer to Zhao *et al*. Additionally, they did not implement the method, performing their slicing entirely by hand.

Future work includes:

1. Defining the belief link constraint "can influence", and extending to properly handle reasoning rules, which requires revising the definition of $contains$ to account for indirect effects via reasoning rules;
2. Exploiting the logical structure of ϕ, for instance, consider $\phi = \Box(\text{Does } ag\ a \rightarrow \text{Bel } ag\ b)$, i.e. whenever ag does a it must believe b. The original slicing algorithm considers any step that modifies b to be relevant, but the belief b is only relevant

when action a is done. So for instance, in the (special but not unusual) case where a only occurs as the first step of a plan whose context condition checks b, then no other plan is relevant to model checking ϕ;
3. Further evaluation, including investigating what characteristics of particular agent programs make them more or less likely to benefit from slicing, and implementing the transformation and compression steps;
4. Implementing an efficient algorithm[16], and analysing its complexity; and
5. Completing the proof of correctness.

References

1. Ammann, P., Offutt, J.: Introduction to Software Testing. Cambridge University Press (2008)
2. Bordini, R.H., Dastani, M., Dix, J., El Fallah Seghrouchni, A. (eds.): Multi-Agent Programming: Languages, Platforms and Applications. Springer (2005)
3. Bordini, R.H., Dastani, M., Dix, J., El Fallah Seghrouchni, A. (eds.): Multi-agent Programming: Languages, Tools, and Applications. Springer (2009). https://doi.org/10.1007/978-0-387-89299-3
4. Bordini, R.H., Fisher, M., Wooldridge, M., Visser, W.: Property-based Slicing for Agent Verification. J. Log. Comput. **19**(6), 1385–1425 (2009). https://doi.org/10.1093/logcom/exp029
5. Bordini, R.H., Hübner, J.F., Wooldridge, M.: Programming multi-agent systems in AgentSpeak using Jason. Wiley (2007), ISBN 0470029005
6. Boyer, R.S., Moore, J.S. (eds.): The Correctness Problem in Computer Science. Academic Press, London (1981)
7. Bratman, M.E.: Intentions, Plans, and Practical Reason. Harvard University Press, Cambridge, MA (1987)
8. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press (2000), ISBN 978-0-262-03270-4
9. DeMillo, R.A., Lipton, R.J., Perlis, A.J.: Social Processes and Proofs of Theorems of Programs. ACM Communications **22**(5), 271–280 (1979)
10. Dennis, L., Fisher, M., Slavkovik, M., Webster, M.: Formal verification of ethical choices in autonomous systems. Robotics and Autonomous Systems **77**, 1–14 (2016). https://doi.org/10.1016/j.robot.2015.11.012
11. Dennis, L.A.: Gwendolen semantics: 2017. Tech. Rep. ULCS-17-001, University of Liverpool, Department of Computer Science (2017)
12. Dennis, L.A., Fisher, M., Lincoln, N.K., Lisitsa, A., Veres, S.M.: Practical verification of decision-making in agent-based autonomous systems. Automated Software Engineering **23**(3), 305–359 (2016). https://doi.org/10.1007/s10515-014-0168-9
13. Emerson, E.A.: Temporal and modal logic. In: van Leeuwen, J. (ed.) Handbook of Theoretical Computer Science, pp. 996–1072. Elsevier (1990)
14. Fetzer, J.H.: Program Verification: The Very Idea. ACM Communications **31**(9), 1048–1063 (1988)
15. Fisher, M., Dennis, L.A., Webster, M.: Verifying Autonomous Systems. ACM Communications **56**(9), 84–93 (2013)
16. Hindriks, K.V., de Boer, F.S., van der Hoek, W., Meyer, J.J.: Agent Programming with Declarative Goals. In: Intelligent Agents VII (Proc. 6th Workshop on Agent Theories, Architectures, and Languages). LNAI, vol. 1986, pp. 228–243. Springer (2001)

[16] We know that this can be done, because the slicing analysis only considers the static program structure, and involves defining links and checking for paths, which can be done efficiently.

17. Jongmans, S.S.T.Q., Hindriks, K.V., van Riemsdijk, M.B.: Model Checking Agent Programs by Using the Program Interpreter. In: Proc. 11th International Workshop on Computational Logic in Multi-Agent Systems (CLIMA). LNCS, vol. 6245, pp. 219–237. Springer (2010)
18. Kamali, M., Dennis, L.A., McAree, O., Fisher, M., Veres, S.M.: Formal Verification of Autonomous Vehicle Platooning. Science of Computer Programming **148**, 88–106 (2017). https://doi.org/10.1016/j.scico.2017.05.006
19. Lomuscio, A., Qu, H., Raimondi, F.: MCMAS: A Model Checker for the Verification of Multi-Agent Systems. In: Proc. 21st International Conference on Computer Aided Verification (CAV). LNCS, vol. 5643, pp. 682–688. Springer (2009)
20. Rao, A.S., Georgeff, M.P.: An Abstract Architecture for Rational Agents. In: Proc. 3rd International Conference on Principles of Knowledge Representation and Reasoning (KR). pp. 439–449 (1992)
21. Rao, A.S.: AgentSpeak(L): BDI Agents speak out in a logical computable language. In: de Velde, W.V., Perrame, J. (eds.) Agents Breaking Away: Proceedings of the Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW'96). Lecture Notes in Artificial Intelligence, vol. 1038, pp. 42–55. Springer (1996)
22. Webster, M., Cameron, N., Fisher, M., Jump, M.: Generating Certification Evidence for Autonomous Unmanned Aircraft Using Model Checking and Simulation. Journal of Aerospace Information Systems **11**(5), 258–279 (May 2014)
23. Winikoff, M., Cranefield, S.: On the Testability of BDI Agent Systems. Journal of Artificial Intelligence Research (JAIR) **51**, 71–131 (2014). https://doi.org/10.1613/jair.4458
24. Winikoff, M.: An AgentSpeak meta-interpreter and its applications. In: Third International Workshop on Programming Multi-Agent Systems (ProMAS). pp. 123–138. Springer, LNCS 3862 (post-proceedings, 2006) (2005). https://doi.org/10.1007/11678823_8
25. Winikoff, M.: BDI Agent Testability Revisited. Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS) **31**(5), 1094–1132 (2017), https://doi.org/10.1007/s10458-016-9356-2
26. Wooldridge, M., Rao, A. (eds.): Foundations of Rational Agency. Applied Logic Series, Kluwer Academic Publishers (Mar 1999)
27. Zhao, J., Cheng, J., Ushijima, K.: Literal Dependence Net and Its Use in Concurrent Logic Programming Environment. In: Proceedings of the workshop on parallel logic programming (held with FGCS'94). pp. 127–141 (1994)