Contextual Reasoning in the Verification of PRS Agent Programs

Wayne Wobcke
Intelligent Systems Research Group
BT Laboratories, MLB 1/PP 12
Martlesham Heath, Ipswich Suffolk IP5 3RE
United Kingdom

Abstract

Although software agents are becoming more widely used, methodology for constructing agent programs is poorly understood. In this paper, we take a step towards specifying and proving correctness for a class of agent programs based on the PRS architecture, Georgeff and Lansky (1987), one of the most widely used in industrial settings. We view PRS as a simplified operating system, capable of running concurrently a series of plans, each of which at any time is in a state of partial execution. The PRS system is construed as using a simplified interrupt mechanism that enables it, using information about goal priorities, to "recover" from various contingencies so that the blocked plans can be resumed and eventually successfully completed. We develop a simple methodology for PRS program construction, then present a formalism combining dynamic logic and context-based reasoning that can be used to reason about such PRS plans.

1. Introduction

In this paper, we take a step towards specifying and proving correctness for a class of agent programs based on the PRS architecture, Georgeff and Lansky (1987). PRS and its successor dMARS are two of the most widely used architectures for building agent systems, and have been used in air traffic management, business process management and air combat modelling, Georgeff and Rao (1998). PRS is a type of *rational* agent architecture, by which is meant that it is based on taking seriously the notion of intention, e.g. as expounded by Bratman (1987).

We take the view that PRS is a kind of simplified operating system, capable of running concurrently a series of plans, each of which at any time is in a state of partial execution. The system is operating in an environment which is dynamically changing, and the job of the interpreter is to monitor these changes and respond to them in such a way that the plans can succeed in achieving their goals. It does this, we contend, by use of a simplified interrupt mechanism that enables the system, using information about goal priorities, to "recover" from various contingencies so that the blocked plans can be resumed and eventually completed. The job of the programmer is to specify plans that can be invoked to deal with every contingency that can occur. If it is possible to recover from every contingency, the system can be guaranteed to achieve its preset goals.

We develop a formalism that can be used to reason about PRS programs viewed in this way, without claiming that this is the *only* way that PRS can be viewed. Our formalism is based on dynamic logic, and thus construes programs as state transition functions (but where in Computer Science, the states are internal machine states, our states are external world states). This work follows on from similar applications of dynamic logic in agent settings, e.g. Singh (1994). In section 2, we review the PRS architecture and present a simple method for constructing PRS agent programs. We give a formalism for reasoning about PRS program construction based on our methodology in section 3, and illustrate the use of the formalism with a simple correctness proof.

2. PRS Agent Programs

PRS (*Procedural Reasoning System*) was initially described in Georgeff and Lansky (1987). Basically, PRS agent programs (as I will call them) are collections of plans, officially called Knowledge Areas (KAs). These plans are essentially the same as standard plans in the Artificial Intelligence literature, in that they have a *precondition* (a condition under which the plan can be executed), an *effect* (a condition which successful execution of the plan will achieve), and a *body* (a collection of subactions which when successfully executed will achieve the effect). The body of a plan is very similar to a standard computer program, except that there can be subgoals of the form *achieve g*, meaning that the system should achieve the goal *g* in whichever way is convenient: these are the analogues of procedure calls. In addition, PRS plans have a *context* (a condition that must be true when each action in the plan is initiated), a *trigger* (a condition that indicates when the interpreter should consider the plan for execution), a *termination condition* (a condition indicating when the plan should be dropped), and a *priority* (a number indicating how important the plan's goal is to achieve). The trigger is important in dynamic settings: when there are a numhelps the interpreter to find the "best" way of achieving the goal, given the current execution context. Note that due to unforeseen changes in the world, the execution context of a goal cannot always be predicted at planning time. The priority of each plan enables the system to determine which plan to pursue given limited resources (usually a plan with the highest priority is chosen for execution). Thus the use of triggers embodies a kind of "forward-directed" reactive reasoning, whereas goal reduction embodies a kind of "backward-directed" goal-driven reasoning.

Our treatment of correctness relies on a particular approach to the construction of PRS programs. We do not claim that the formalization applies to any PRS program. Consider designing a plan to achieve a particular goal g. We give the following intuitive picture as to how this might be done, taking for now the simple case in which it is assumed that there are no calls to subgoals and no contingencies that arise during execution. Recall that each plan has an associated context, a condition that must be true throughout the plan's execution. It seems natural to start by determining a collection of possible initial states S, then proceed by dividing this set into subsets of states S_i such that for each subset S_i, it is possible to define a single plan P_i that can achieve g without leaving the states in S_i (except possibly at the end of the plan, when g itself is true). The subsets S_i need not be disjoint, but their union should equal S. The next task is to define formulae c_i that characterize the S_i, meaning that each c_i is true of all the states in S_i but not true of any state not in S_i (because P_i does not work in these states)—this is not necessarily straightforward! The correctness of the plan P_i can be expressed as the dynamic logic formula $c_i \Rightarrow [P_i]g$, and proven so using standard techniques. Furthermore, the formula $c_1 \vee \cdots \vee c_n$ (assuming there are a finite number of contexts $1, \cdots, n$) characterizes the set of initial states S, and the assumption that S contains all the possible initial states is expressed by the formula $\Box(c_1 \vee \cdots \vee c_n)$. From this and the correctness of the individual plans, it follows that $([P_1]g \vee \cdots \vee [P_n]g)$.

Now consider designing plans to respond to "procedure calls", i.e. to satisfy subgoals of the form *achieve* g occurring in a plan P. The plan will have a context c that characterizes a set of states S: each state in S is one in which P achieves its goal, assuming that all the calls to *achieve* g succeed. Note first that, not only should the procedure achieve g, but it should also maintain the context c. Thus the context of the subprogram should entail c. Now we may proceed as above, decomposing S into subsets S_i, characterizing those subsets by context formulae c_i, and defining plans P_i with contexts c_i that achieve g. The priorities of each subplan P_i should be at least that of P. It is apparent that by repeating this process for calls to achieve subgoals within subprograms, the programmer defines a hierarchy of contexts by continually partitioning the original set of states S. For each subprogram P_i with set of S and context c, we have that $S_i \subseteq S$ and $c_i \vdash c$. That is, the hierarchy forms a partial order on sets of states with the ordering inherited from set inclusion. Note that, as above, from a collection of formulae $c_i \Rightarrow [P_i]g$ for $i = 1, \cdots, n$, it follows that $c \Rightarrow ([P_1]g \vee \cdots \vee [P_n]g)$, where $c \equiv c_1 \vee \cdots \vee c_n$.

The next stage is to consider the contingencies that can arise while executing a plan P. The purpose of the contingency plans is, whenever possible, to restore the context of the original plan c (or if this is impossible, to cause the original plan to be dropped by achieving $\neg c$). However, it is not necessary that *each* contingency plan achieve c: below, we give a simple example where executing a *sequence* of contingency plans restores c. Moreover, a contingency plan need not directly achieve c; rather it can block the original plan from being executed until c is true. This is also illustrated in the example below. Even so, it seems natural to start with a set of states S that defines when the contingency occurs, and to divide this set into subsets for each of which a contingency plan can be defined. The priority of any contingency plan must be greater than that of the original plan to ensure that the contingency plan is chosen by the interpreter for execution in preference to the original plan. Any subgoals in the contingency plan can be handled as described above. Now by repeating this process, i.e. by defining contingencies to handle contingencies, the programmer also defines a hierarchy of contexts, but in contrast to that defined for subprograms, this is a hierarchy of exceptions. That is, if P_i is a contingency plan with set of states S_i and context c_i, for a plan P with set of states S and context c, then there is no necessary relationship between S_i and S, nor between c_i and c. The whole design process stops when there are no remaining contingencies to consider.

For example, consider designing a simplified program for an aircraft to take off. Assume the basic takeoff plan can be defined, and succeeds provided the runway is free. That is, the condition $\neg$*runway_free* is a contingency. Two plans are defined to deal with this contingency, differing in their context of application. In one, the plane is on the runway and must be diverted; in the other, the plane is not on the runway and simply waits. Note the subtleties in even this program: the *divert* plan does not restore the original context *runway_free*, but changes the context to $\neg$*on_runway* so that the *wait* plan is invoked. Also, the *wait* plan may be repeatedly invoked until its trigger is false; when this is the case, the context of the *takeoff* plan is true, so this plan can be resumed. It only remains to assign priorities to the plans such that the contingency plans have higher priority than the *takeoff* plan (this can be done in any way convenient, so just let the priority of *takeoff* be 10 and the priority of *divert* and *wait* be 20). The final program is shown in Figure 1. Boxes indicate contexts and an arrow from one context to another indicates that the first handles a contingency that can arise while the second is executing.

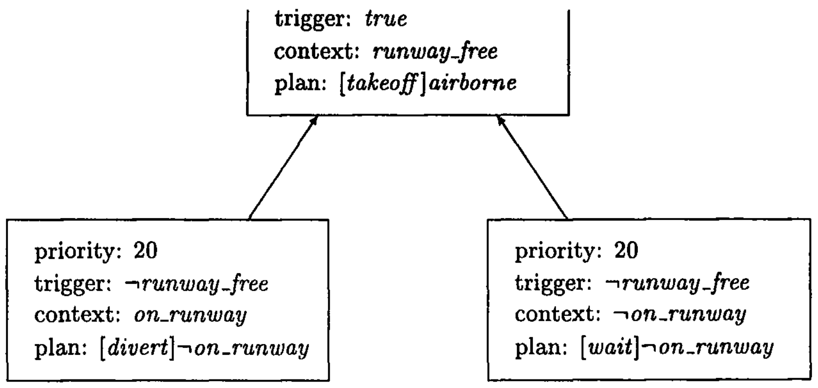

Figure 1: Takeoff Program

3. Correctness of Agent Programs

We now present a formalism that can be used to reason about PRS programs constructed as in section 2. The essence of the formalism is the combination of dynamic logic and context-based reasoning, Wobcke (1989). The technical formalism is related to labelled deductive systems, Gabbay (1996), in which each formula is assigned a label, signifying a context in which the formula is true. For a formula representing the correctness of a plan, the label can be identified with the possible execution contexts of the plan, and hence the label also indirectly represents a set of assumptions under which the plan can be proven correct.

The formal language we use is based on propositional dynamic logic, Pratt (1976). The semantics of dynamic logic is based on binary state transition relations. More precisely, an interpretation M consists of a modal frame F and a valuation on atomic proposition symbols V. The frame F consists of a nonempty set of states S together with a binary relation R_π on S for each program term π. The valuation V is a mapping from the set of atomic proposition symbols to the power set of S.

Satisfaction at a state s in an interpretation M is defined as follows.

$M \models_s A$ iff $s \in V(A)$ for A an atomic formula
$M \models_s \neg A$ iff $M \not\models_s A$
$M \models_s A \wedge B$ iff $M \models_s A$ and $M \models_s B$
$M \models_s A \vee B$ iff $M \models_s A$ or $M \models_s B$
$M \models_s A \Rightarrow B$ iff $M \not\models_s A$ or $M \models_s B$
$M \models_s [\pi]A$ iff for all t such that $R_\pi(s,t)$, $M \models_t A$

Finally, there are a number of constraints on the R_π to ensure that each reflects the operational semantics of the program construction operations.

$R_{\alpha;\beta} = R_\alpha \circ R_\beta = \{(s,t) : \exists u(R_\alpha(s,u) \text{ and } R_\beta(u,t))\}$
$R_{\alpha\cup\beta} = R_\alpha \cup R_\beta$
$R_{\alpha^*} = R^*_\alpha$ (the transitive closure of R_α)
$R_{A?} = \{(s,s) : M \models_s A\}$

It can be shown, Goldblatt (1992), that the following axiom schemata and rule are sound and complete with respect to the above semantics (that includes the constraints on the R_π).

$[\alpha;\beta]A \Leftrightarrow [\alpha][\beta]A$
$[\alpha \cup \beta]A \Leftrightarrow ([\alpha]A \wedge [\beta]A)$
$[\alpha^*]A \Rightarrow (A \wedge [\alpha][\alpha^*]A)$
$[\alpha^*](A \Rightarrow [\alpha]A) \Rightarrow (A \Rightarrow [\alpha^*]A)$
$[A?]B \Leftrightarrow (A \Rightarrow B)$
$[\alpha](A \Rightarrow B) \Rightarrow ([\alpha]A \Rightarrow [\alpha]B)$
If $\vdash A$ infer $[\alpha]A$

We define a formal language LPDL (*Labelled Propositional Dynamic Logic*) for use in context-based reasoning. The atomic formulae of LPDL are of the form $l : A$, where l is drawn from a given set of *labels* and A is a formula of propositional dynamic logic. These atomic formulae can be combined using the propositional connectives $\neg$, $\wedge$, $\vee$ and $\Rightarrow$. An LPDL interpretation I is an assignment of a dynamic logic interpretation M_l to each label l. Satisfaction of LPDL formulae is defined as follows.

$I \models l : A$ iff $M_l \models A$ for A an atomic LPDL formula
$I \models \neg A$ iff $I \not\models A$
$I \models A \wedge B$ iff $I \models A$ and $I \models B$
$I \models A \vee B$ iff $I \models A$ or $I \models B$
$I \models A \Rightarrow B$ iff $I \not\models A$ or $I \models B$

It is straightforward to axiomatize LPDL, given an axiomatization of dynamic logic. LPDL contains all instances of propositional calculus axioms and modus ponens obtained by replacing an atomic proposition symbol by an LPDL formula, and the following axiom schemes in which l stands for any possible label.

$l : A$ for A an axiom of propositional dynamic logic
$l : (A \Rightarrow B) \Rightarrow (l : A \Rightarrow l : B)$
$l : A \Rightarrow \neg(l : \neg A)$
$l : A \Rightarrow l : \Box A$

The reasoning behind the process of designing PRS programs described in section 2 is essentially one of combining context-based reasoning for reasoning between contexts with dynamic logic for reasoning within ous contexts that need to be considered. First, each plan is associated with a labelled context that corresponds to its execution context, as illustrated for the takeoff plan in Figure 1. Second, each contingency is itself associated with a labelled context corresponding to the execution contexts of the set of plans that may be invoked to deal with the contingency. Finally, each priority level is associated with a labelled context corresponding to the execution contexts of the set of plans of that priority—these are plans that can possibly compete with each other for selection by the interpreter for execution.

A proof of correctness of a PRS program proceeds in stages, mirroring the design process. First, standard techniques are used to show correctness of plans and subprograms that execute in a single context. These proofs are all on the assumption that execution never leaves the assigned context, except possibly at the end of the plan when the goal is achieved. Next, reasoning between contexts is used to infer that all contingencies that arise during the execution of any plan can be successfully met. Any such proof of correctness is therefore reliant on the programmer's having identified the range of possible contingencies to any plan. Finally, conclusions about lower level plans are "lifted" to higher level contexts, and the process repeated until the top level plans are reached. We present three rules corresponding to these types of inference. The soundness of these inference rules follows from properties of the PRS interpreter.

A proof begins with assumptions about the lowest level plans in the hierarchy, and proceeds inductively according to the structure of the context hierarchy, as indicated in the plan in Figure 1. The required assumptions all mean that there are no exceptions arising at the lowest level (highest priority) plans, and all have the following form.

$$c : \Box((\mathit{context} \vee \mathit{goal}) \wedge \neg \mathit{termination})$$

We need $\mathit{context} \vee \mathit{goal}$ rather than just *context* because of the technical complication arising from the fact that the final state in the plan's execution may not satisfy the context formula (it satisfies the goal formula). We envisage, therefore, that the proof of correctness for the plan involves verifying that the goal formula is false after execution of each subaction in the plan, except possibly at the final state.

The Contingency Rule is used to infer that all contingency plans achieve some goal g. Here $\mathsf{Achieves}(g)$ is a special formula intended to indicate that the agent has a plan or plans that achieve g, and knows in which context to execute which plan. Here $l_1, \cdots l_n$ are any finite number of context labels.

Contingency Rule:

$$\frac{l_1 : t \Rightarrow [\alpha_1]g, \ \cdots, \ l_n : t \Rightarrow [\alpha_n]g}{l_1 \cup \cdots \cup l_n : t \Rightarrow \mathsf{Achieves}(g)}$$

rule, to ensure that $l_1, \cdots l_n$ denote *all* the contexts that correspond to a plan dealing with the given contingency. Typically these plans are all at the same level of priority.

The Priority Rule is used to infer that out of all plans that have a given priority (the same priority as a contingency plan), the agent can achieve some goal. This rule is needed to ensure that the interpreter is still able to choose the correct plan(s) for execution when it must choose from a larger set of plans. The rule is as follows, assuming that $l_1, \cdots, l_n$ are all the plans that have priority p, and that p is also a new context label. It is intended that t denote a trigger for a contingency, and g the goal achieved by the contingency plans.

Priority Rule:

$$\frac{l_1 : t \Rightarrow \mathsf{Achieves}(g), \ \cdots, \ l_n : t \Rightarrow \mathsf{Achieves}(g)}{p : t \Rightarrow \mathsf{Achieves}(g)}$$

The Lifting Rule connects contingency plans to the higher level plans from which their contingency derives. We use the rule to infer that in the higher level context the contingency can be handled correctly. The statement of the rule assumes that the priority of the plan in context c is less than p. Again, t denotes a trigger and g the goal achieved by the contingency plans.

Lifting Rule:

$$\frac{p : t \Rightarrow \mathsf{Achieves}(g)}{c : \Box(t \Rightarrow g)}$$

Intuitively, while the trigger t is true and the goal g is not true, the agent's execution is in context p, hence all states in c satisfy the negation of $t \wedge \neg g$, i.e. $t \Rightarrow g$.

To illustrate the use of these rules in reasoning about PRS programs, consider the aircraft takeoff plan from Figure 1. We first assign labels (arbitrarily) to the execution contexts of the plans; let c correspond to *takeoff*, c_1 to *divert* and c_2 to *wait*. The reasoning starts at the leaves of the tree, where it is assumed there are no contingencies that arise duting the context of executing these plans. In this example, the assumptions that are needed are as follows.

$$c_1 : \Box(\mathit{on\_runway} \vee \neg \mathit{on\_runway}) \quad (1)$$

$$c_2 : \Box(\neg \mathit{on\_runway} \vee \neg \mathit{on\_runway}) \quad (2)$$

Assumption (1) is trivially true: it implies that there is no logical need for an exception to the *divert* plan. Intuitively this is because whenever such an exception could arise, the goal would already be true. However this does not preclude the possibility of plan failure for other reasons, and this could mean that further contingency plans are required. Assumption (2) is nontrivial: it states that when executing the *wait* plan, it is assumed that the plane is not on the runway. This would be false if it were possible for some event to cause the plane to become on the runway whilst waiting: this also could be reason for another contingency plan (perhaps have to be verified).

By constructing the proof, we aim to verify the following formula, which represents the assumption under which the *takeoff* plan should be proven correct.

$$c : \Box(\mathit{runway\_free} \vee \mathit{airborne}) \quad (3)$$

Forming the basis of the proof, we assume that the following formulae representing the correctness of the individual plans relative to their execution contexts can be proven using dynamic logic using the above assumptions. Each formula says that whenever the plan is initiated in a state in which its context is true, the plan achieves its goal.

$$c : \Box(\mathit{runway\_free} \Rightarrow [\mathit{takeoff}]\mathit{airborne}) \quad (4)$$

$$c_1 : \Box(\mathit{on\_runway} \Rightarrow [\mathit{divert}]\neg\mathit{on\_runway}) \quad (5)$$

$$c_2 : \Box(\neg\mathit{on\_runway} \Rightarrow [\mathit{wait}]\neg\mathit{on\_runway}) \quad (6)$$

As an aside, the following formulae can also be proven using standard dynamic logic.

$$c_1 : \neg\mathit{on\_runway} \Rightarrow [\mathit{wait}^*]\neg\mathit{on\_runway} \quad (7)$$

$$c_2 : \mathit{on\_runway} \Rightarrow [\mathit{divert}; \mathit{wait}^*]\neg\mathit{on\_runway} \quad (8)$$

These formulae indicate that both $[\mathit{wait}^*]$ and $[\mathit{divert}; \mathit{wait}^*]$ are possible plans in their respective these contexts, but note that it does not follow that these are the *only* plans that can be executed in these contexts.

Now we need to start reasoning about contexts. Let $c_1 \cup c_2$ denote the context corresponding to the contingency $\neg\mathit{runway\_free}$. In the present example, the Contingency Rule enables the inference of the following formula from (5) and (6), which means that in every context associated with the contingency $\neg\mathit{runway\_free}$, the condition $\neg\mathit{on\_runway}$ is achieved.

$$c_1 \cup c_2 : \neg\mathit{runway\_free} \Rightarrow \mathsf{Achieves}(\neg\mathit{on\_runway}) \quad (9)$$

The Priority Rule is now used to infer the following formula, which means that the set of plans at priority 20 handle the contingency $\neg\mathit{runway\_free}$ correctly (recall that the contingency plans both have priority 20).

$$20 : \neg\mathit{runway\_free} \Rightarrow \mathsf{Achieves}(\neg\mathit{on\_runway}) \quad (10)$$

Finally, the Lifting Rule is used to prove the following formula, meaning that while the plane is attempting to take off, it is not on the runway unless the runway is free. This represents a "safety" condition that it is desirable to verify in this example.

$$c : \Box(\neg\mathit{runway\_free} \Rightarrow \neg\mathit{on\_runway}) \quad (11)$$

This means that the following formula holds at every state in context c, and so is a candidate for the context of the *takeoff* plan.

$$\mathit{on\_runway} \Rightarrow \mathit{runway\_free} \quad (12)$$

But (12) does not entail the plan's current context *runway_free*. However, it is apparent that (12) more that if it does not represent the context, there is no guarantee the plan will work (with the current plan, it is assumed, but not required, that the plane is always on the runway). It should therefore be possible to prove (4) using this weaker context assumption. Alternatively, the condition $\neg\mathit{on\_runway}$ could be added as a termination condition for the *takeoff* plan, so that (12) together with the negation of this condition imply the current context. In either case, the proof of correctness for the modified plan is now complete.

4. Conclusion

Our formalism for verifying PRS programs is based on dynamic logic and reasoning in context. The method presumes a simple hierarchical design process in which contingencies to plans are identified and contingency plans then defined to deal with them. Any proof of correctness is dependent on the programmer's having identified all the possible contingencies to any plan, and on the correctness of the contingency plans themselves. We do not claim that this is the only way to construct PRS agent programs, nor that this is the only way of applying formal methods to proving correctness for agent programs. The main lesson, we feel, is that any methodology for verifying agent programs must refer explicitly to the architecture of the system; the comparative simplicity of reactive systems perhaps explains why verification techniques for these systems are further advanced than those for more complex architectures.

References

Bratman, M.E. (1987) *Intention, Plans and Practical Reason.* Harvard University Press, Cambridge, MA.

Gabbay, D.M. (1996) *Labelled Deductive Systems. Volume 1.* Oxford University Press, Oxford.

Georgeff, M.P. & Lansky, A.L. (1987) 'Reactive Reasoning and Planning.' *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI-87)*, 677–682.

Georgeff, M.P. & Rao, A.S. (1998) 'Rational Software Agents: From Theory to Practice.' in Jennings, N.R. & Wooldridge, M.J. (Eds) *Agent Technology.* Springer-Verlag, Berlin.

Goldblatt, R. (1992) *Logics of Time and Computation. Second Edition.* Center for the Study of Language and Information, Stanford, CA.

Pratt, V.R. (1976) 'Semantical Considerations on Floyd-Hoare Logic.' *Proceedings of the Seventeenth IEEE Symposium on Foundations of Computer Science*, 109–121.

Singh, M.P. (1994) *Multiagent Systems.* Springer-Verlag, Berlin.

Wobcke, W.R. (1989) 'A Schema-Based Approach to Understanding Subjunctive Conditionals.' *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, 1461–1466.