AAAI Spring Symposium on Lessons Learned from Implemented Software Architectures for Physical Agents, p. 127-134

On the Design of Structured Circuit Semantics

Jaeho Lee

Artificial Intelligence Laboratory
The University of Michigan
1101 Beal Avenue
Ann Arbor, Michigan 48109–2110
jaeho@eecs.umich.edu

Abstract

Structured Circuit Semantics (SCS) is a formal semantics of robotic languages that can be used to explicitly represent the control behavior of various robotic control systems. Our approach to the design of SCS has been to incorporate the most essential features of other reactive plan execution systems in a compact, yet rich formal semantics. A formal specification of a reactive plan is essential for us to be able to generate a plan, reason about it, and communicate it among possibly multiple agents. Major design decisions include how to handle continuous actions and how to represent various agent behaviors such as nondeterministic behavior, best-first behavior, and persistent behavior. Metalevel reasoning and multiple threads of execution are also important design factors. Currently our effort is directed toward a semantics of multiagent robotic languages.

1 Introduction

The Procedural Reasoning System (PRS) (Ingrand, Georgeff, and Rao 1992) is a general purpose reasoning system particularly suited for use in domains in which there are predetermined procedures for handling the situations that might arise. We have been using our implementation of PRS, called the University of Michigan Procedural Reasoning System (UM-PRS), for robotic applications in unpredictable domains. The advantage of this system is that it allows robotic vehicles to pursue long-term goals by choosing pieces of relevant procedures depending on the changing context, rather than blindly following a prearranged plan. Specifically, UM-PRS has been used to control a real outdoor vehicle that chooses its behavior based on what it senses from its environment (Lee, Huber, Durfee, and Kenny 1994).

Our UM-PRS implementation and applications to outdoor vehicle control motivated the development of Structured Circuit Semantics (SCS)[1] as the need for clean execution semantics arose. SCS is a formal semantics of robotic languages that can be used to explicitly represent the control behavior of various robotic control systems. SCS extends the circuit semantics notion of Teleo-Reactive (T-R) programs (Nilsson 1992; Nilsson 1994) into a richer, yet compact, formal semantics. It also provides a basis for constructing new robotic systems with more understandable semantics which can be tailored to particular domain needs.

In this paper, we briefly describe our SCS-based agent architecture and discuss our design decisions involved with both SCS and the agent architecture.

2 Agent Architecture

The SCS, a general semantics for robotic control systems, can be directly transformed into the SCS language which is interpreted and executed by an interpreter. We have recently completed an implementation of an SCS-based agent architecture written in C++ to meet the needs of an efficient real-time robotic control. The complete system consists of the SCS interpreter, the world model, effectors and sensors as shown in Figure 1.

This effort is directed at supplanting our previous implementation of PRS, UM-PRS, with a more general plan execution system that can be tailored to the control needs of our application domain.

The SCS language has many useful built-in actions such as arithmetic operations, world model access and update operations, and general matching operations. In this section, we will describe major components of the SCS plan execution system.

Expressions

The SCS language supports that expressions that can be evaluated at runtime. *Values* and *variables* are expressions. A value can be either an integer number,

[0]This research was sponsored in part by ARPA under contract DAAE-07-92-C-R012.

[1]Readers can see Appendix C or refer to (Lee and Durfee 1994) for the details of SCS.

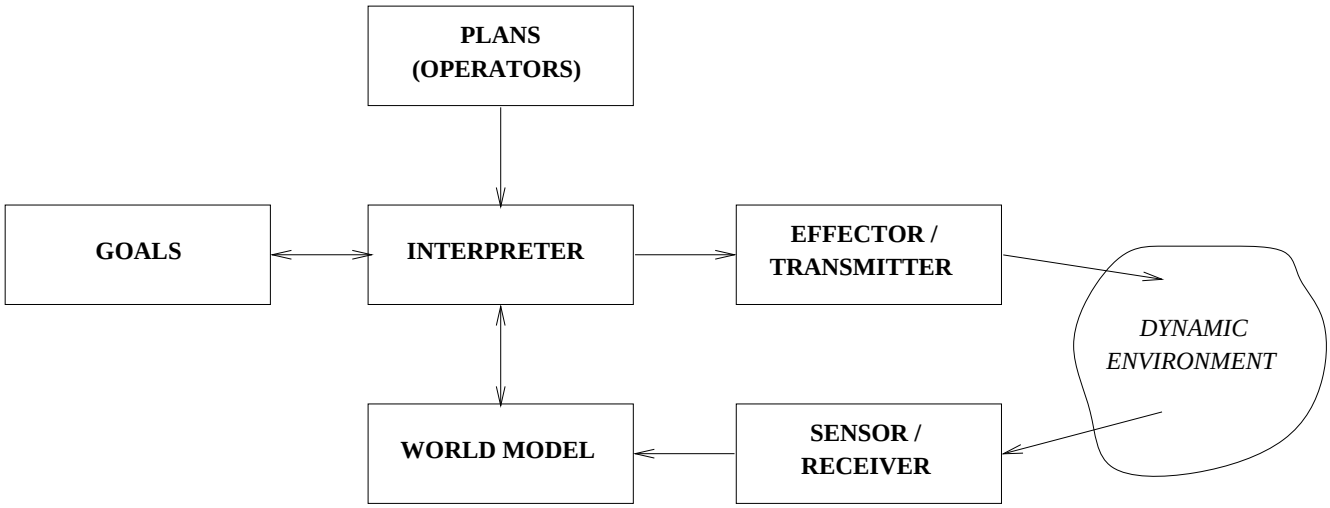

Figure 1: SCS Plan Execution System

a floating point number, an address (an integer), or a string. A variable can be bound to any value. A *function* is also an expression. The syntax of function expressions follows the Lisp function format closely. A function expression consists of a pair of parentheses where the first element within is the name of a function and the rest of the elements are arguments to the function.

World Model

The world model holds the facts that represent the current state of the world. Information stored in the world model includes state variables, sensory information, conclusions from deduction or inferencing, modeling information, etc. In our implementation, the world model is a database of facts that are represented as relations. Initial facts are asserted at the beginning of a program by the user, and additional facts can be either asserted, retracted, or updated by the SCS program.

Goals

Goals in the SCS plan execution system are the world states that the system is trying to achieve or maintain. A goal can be either a top-level goal which controls the system's highest order behavior, or a subgoal activated by the execution of a step.

SCS Plans

An SCS Plan is a predefined sequence of steps (Appendix C) of how to satisfy a specific task or goal. A step can be defined with a list of arguments (local variables) that are bound when the step is called. The defined step is called and expanded accordingly at runtime. It may be viewed as a plan schema which is instantiated with the bindings of the arguments. Typically, the actions in a step constitute either primitive actions or subgoals to be achieved. A primitive action is a behavior or activity that can be executed directly. Any other type of action represents (1) a goal that needs to be achieved or maintained, (2) a query, (3) a test, or (4) an assertion or retraction of world model information.

The Interpreter

The SCS interpreter controls the execution of the entire system. It also acts as the runtime stack for the system. It keeps track of the progress of each high-level goal and all of the subgoals. The interpreter initiates, suspends, resumes, and cancels the execution of plans in much the same way as an operating system does. It maintains information about what steps are currently active, the success or failure of steps, as well as what step is to be executed next.

3 Design Issues

Our approach to SCS has been to incorporate the essential and best features of other reactive plan execution systems. As a result, most of the features of SCS can be accomplished in other systems using either implicit or explicit mechanisms. Nevertheless, our semantics and the corresponding system try to capture those features in an explicit, simple, and clean way to allow SCS plans to be generated by a generative planner. In this section, we list our major decisions in designing the semantics and the agent architecture.

Continuous Action

Actions in physical domains tend to be continuous actions. Thus, the conditions for actions must be *continuously* evaluated and, when appropriate, their associated actions should be *continuously* executed. However, real systems have a natural, characteristic frequency that leads to cycles of execution. These cycles also occur in reactive execution systems, typically corresponding to the perception-cognition-action cycle. Traditionally, reactive systems have concentrated on increasing the frequency of this cycle by, for example, reducing the time needs of cognition, but this cycle cannot be completely eliminated.

In SCS, atomic actions are the basic unit of actions. An atomic action is guaranteed to terminate within a bounded time and cannot be interrupted. We argue that continuous actions can be implemented using atomic actions by making the execution cycle higher than or equal to the characteristic frequency of the agent's environment. In fact, if the agent is to be implemented using conventional computer systems, the continuous actions *must* be mapped down to the discrete atomic actions anyway.

Clearly defining an execution cycle, therefore, goes hand-in-hand with defining atomic actions. Without a characteristic execution cycle, a continuously running system could take control from any of its actions, even if those actions are incomplete. For example, consider a program of production rules whose conditions are to be evaluated on both sensory input and stored internal state information. During the execution of an action that is supposed to make several changes to the internal state, an earlier condition in the production rule is satisfied, and the original action is abandoned, possibly rendering the internal state inconsistent. In pathological cases, the system could become caught in an oscillation between zero or more actions.

Of course, such interruptions could be avoided by augmenting the conditions such that they will not change truth value at awkward times, but this implicitly institutes an execution cycle and atomicity, which should be more efficiently and explicitly represented.

Nondeterministic Behavior

The **do any** construct in our SCS formalism can specify multiple equally good actions for the situation (see Appendix C). One of the actions is nondeterministically chosen at runtime and executed. If a chosen action fails, another action within the construct is nondeterministically selected and tried until one of them succeeds.

We argue that the capability of specifying a nondeterministic choice of action is important in reactive systems because generating a total ordering of the actions at design time can be difficult and can lead to overly rigid runtime performance. Actions that appear equally good at design time will have an order imposed on them arbitrarily. This could result in the system repeatedly choosing one action that is currently ineffective because it happens to appear earlier than a peer action. Instead, the system should be able to leave collections of actions unordered and try them nondeterministically.

Best-First Behavior

One of the reasons for demanding reactivity is to be sensitive to the way the utility of an action may vary with specific situations and to choose the applicable action that is best *relative* to the others. Since which action is best relative to the others depends on the runtime situation, the selection cannot be captured in a static ordering of actions. In a system with metalevel reasoning capability such as the PRS, the relative utility of competing actions is determined by the metalevel conflict resolving mechanism. In some production systems, static system-wide conflict resolution methods are used.

A more general answer is to introduce a dynamic utility-based (cost based in dual) selection among candidate actions. In SCS, the **do best** serves this purpose. Each action in the **do best** construct has an associated utility function as well as an energizing condition[2]. Each action with a satisfied condition competes or bids by submitting its expected utility, and the highest bidder is selected. The need for utility-based conflict resolution will become clearer with the discussion of metalevel reasoning in this section.

Persistent Behavior

The success of an action can be measured in terms of whether it had the desired effect at the desired time on the environment. Because of the characteristic frequency of the system, even sustained actions (such as keeping a vehicle centered on the road) can be viewed as sequences of atomic actions (such as repeatedly checking position and correcting heading). Thus, since an atomic action can have an effect on the environment, determining whether that effect was achieved is important part of controlling the execution of further actions.

The energizing conditions associated with an action must check for effects of the action, such that failure naturally leads to the adoption of a different action. However, because there might be a variety of subtle

[2]For the definition of the energizing condition, see Appendix C

effects on the environment that an action would cause that would indicate failure, and because embedding these in the energizing condition could be inefficient and messy (non-modular), it is useful to allow actions to return information about success and failure. If actions can return failure information, constructs can respond to this information, allowing a broader range of reactive (exception handling) behavior. In SCS, several different constructs encode different responses to action failures to provide a variety of reactive execution behaviors.

Currently, the failure semantics tells only whether or not to persistently retry the failed step, but it could be generalized to condition-based persistence which would limit the number of retries.

Metalevel Reasoning

Metalevel knowledge is different from application-level knowledge in that the former is knowledge about the latter. For example, in PRS, metalevel Knowledge Areas decide which procedure (Knowledge Area) to run when more than one of the procedures are applicable. We introduce the approach of embedding metalevel decision into the situation rather than keeping them separate and argue the advantage of doing so in reactive plan execution systems.

First of all, we claim that most metalevel decisions use very domain-specific knowledge and utility functions which are local to the situation rather than global and abstract. In other words, there are very few general syntactic metalevel decisions which effectively resolve conflicts. For example, in PRS, the metalevel decision procedures which lie outside of the application-level procedures need specific conditions to isolate the specific situation and associated actions to apply. On the other hand, if such a decision making mechanism can be attached at the application-level steps, we can remove the redundant situation assessment conditions. It is still arguable that having one single metalevel decision procedure is better than placing multiple copies of the same decision procedures everywhere it is needed. If that is the case, the metalevel decision procedure can define as a function or macro to be invoked wherever a specific decision is needed.

Furthermore, if the plan execution system is supposed to get reactive plans from a generative planner (including a human), we conjecture that it is much harder to generate metalevel decisions than a procedure with embedded metalevel decisions.

Another important concern in reactive plan execution systems is the predictable response time. With the separated metalevel decision, the metalevel deliberation time can be arbitrarily long and unpredictable.

As mentioned before, the SCS **do best** constructor supports the idea of embedded metalevel decisions. In particular, when this constructor is wisely nested with other constructors, the effect of multi-level metalevel reasoning can be achieved effectively.

Multiple Threads of Execution

A thread of execution consists of a execution environment such as variable bindings and a invocation structure. The need for multiple threads of execution arises when there are multiple goals to pursue and thus the need to keep multiple foci of attention.

At one extreme, multiple threads of execution can be avoided if the execution environment is reestablished whenever the system switches its attention to a new environment. At the other extreme, every attempted thread of execution can be saved for potential future uses. The tradeoff between these two extremes depends on the stability of the world in which the agent is working. If the world is so dynamic that the saved threads do not accurately reflect the state of the world, then there is no need to save multiple threads of execution. If the world is static enough that the saved execution environments accord with the world most of the time, then it is advantageous to save multiple threads.

The approach in our agent architecture toward multiple threads of execution stands in the middle of these two extremes. We have implemented an explicit wait-resume mechanism. Thus, the interpreter is able to have multiple threads of execution only under explicitly specified conditions and to switch among the threads only at designated points of execution. We claim this scheme is sufficient for reactive plan execution systems because focus-switching decision points can be planned ahead by the generative planner. If the decision for multi-threading should be made at runtime, that decision making process could be combined with the above mentioned embedded metalevel decision mechanism.

4 Related Work and Future Work

A variety of languages for robotic systems have been developed in recent years, including PRS, Teleo-Reactive Programs, Universal Plans (Schoppers 1987), and RAPs (Firby 1989; Firby 1992). Each of these approaches attempts to solve the problem of taking reasonable courses of action fast enough in response to a dynamically changing environment. These competing approaches have diverse representations and (sometimes implicit) control structures. In developing SCS, we have attempted to collect the essential and best features of the robotic reactive plan execution system.

Because SCS embodies circuit semantics (see Ap-

pendix B), previous comparisons (Nilsson 1992; Nilsson 1994) between T-R programs and reactive plan execution systems such as SCR (Drummond 1989), GAPPS (Pack Kaelbling 1988), PRS (Georgeff and Lansky 1987), and Universal Plans (Schoppers 1987), are applicable here as well. In this section, we concentrate on comparisons which specifically deal with SCS.

As illustrated in the previous section, the **do first** construct and the capability of defining a step covers the circuit semantics of T-R programs. Universal Plans also fit easily within SCS through the nested use of the **do when** construct. The real power of SCS over T-R programs or Universal Plans is manifested when the simple SCS constructs interact in various ways.

Situated Control Rules (SCR) are constraints for plan execution that inform an independently competent execution system that it can act without a plan, if necessary. The plan simply serves to increase the system's goal-achieving ability. In other words, SCR alone is *not* a plan execution system, and its rules are not executable. This does not preclude, however, developing an integrated system where one component generates SCRs which, in turn, are automatically compiled into a SCS program to execute on another component. The semantics of SCS would make such compilation possible, although the SCR formulation has weaknesses that must be overcome. These weakness include the facts that (1) SCR does not consider variable binding, (2) SCR has no hierarchical execution structure (function call or recursion), and (3) SCR has no runtime reasoning.

The RAP system (Firby 1989; Firby 1992) is very similar in flavor to SCS. RAP's intertwined conditional sequences enable the reactive execution of plans in a hierarchical manner. As with other systems, its basic difference with SCS is that it lacks circuit semantics. RAP also lacks several features of SCS including failure semantics (when a RAP method fails, it assumes the robot is in the same state as before the method was attempted), and the ability to resume a method from the suspended point of the method. SCS has clear failure semantics which specify what to do and where to start. Another limitation of the RAP interpreter is that methods lack the runtime priority information expressed by utility functions in SCS. RPL (McDermott 1991; McDermott 1992) extends RAPs by incorporating *fluents* and a `FILTER` construct to represent durative conditions, but these are much more compactly and intuitively captured in SCS.

PRS deserves special mention because a major motivation in developing SCS has been our need for formally specifying the PRS plan representation and its execution model. A formal specification of a reactive plan is essential in order for us to be able to generate the plan, reason about it, and communicate it among multiple agents. In the PRS perspective, SCS can be interpreted as a formalism for the PRS execution model using circuit semantics. In particular, the metalevel reasoning capabilities of PRS introduce a wide variety of possible execution structures. So far, we have been able to express much of PRS's utility-based metalevel decision making in SCS using the **do best** construct. Note that these constructs can be nested to arbitrary depth, corresponding to multiple metalevels in PRS. Decisions at lower levels can affect higher-level decisions through failure semantics and changes to the internal state, while utility calculations guide choices from higher to lower levels.

We are currently working on formally specifying the content of PRS metalevel knowledge areas to more rigorously capture it in SCS. With this modified system, not only will we have a more flexible plan execution system, but also one with clear semantics to support inter-agent communication and coordination in dynamic environments.

A Responses to Questions

Coordination

How should the agent arbitrate/coordinate/cooperate its behaviors and actions? Is there a need for central behavior coordination?

Most behavior coordination is already dictated by the conditions of actions and the semantics of SCS constructors. However, this does not mean that there is no need for a central behavior coordinator. The central coordinator dynamically assesses the current situation, checks conditions associated with each action, and interprets the semantics of the nested constructors. Furthermore, multiple threads of execution, nondeterministic behavior, and utility-based behavior all require central behavior coordination.

Interfaces

How can human expertise be easily brought into an agent's decisions? Will the agent need to translate natural language internally before it can interact with the world? How should an agent capture mission intentions or integrate various levels of autonomy or shared control? Can restricted vocabularies be learned and shared by agents operating in the same environment?

There are two basic forms of interface between the agent and human. The first form is a direct interface through agent plans. Tasks are directly assigned to the agent in terms of executable plans. The second form is an indirect interface through goals and facts. Goals or

facts can be received from other agents or recognized via sensors or plan recognition module.

Structural

How should the computational capabilities of an agent be divided, structured, and interconnected? What is the best decomposition/granularity of architectural components? What is gained by using a monolithic architecture versus a multi-level, distributed, or massively parallel architecture? Are embodied semantics important and how should they be implemented? How much does each level/component of an agent architecture have to know about the other levels/components?

The computational capabilities of an agent are divided into atomic actions in SCS (Section 3). Atomic actions are an agent's primitive capabilities. The structured SCS constructors interconnect atomic actions to form a hierarchical and recursive nesting of the actions. The agent architecture itself does not impose any restrictions on the granularity of the atomic actions. The granularity is, however, constrained by the application domain and the hardware on which the software agent is built.

Once the set of basic capabilities is defined in terms of atomic actions, SCS constructors allow us to exercise various conceptual agent architectures in the range between a monolithic architecture and a multi-level, parallel architecture.

Performance

What types of performance goals and metrics can realistically be used for agents operating in dynamic, uncertain, and even actively hostile environments? How can an architecture make guarantees about its performance with respect to the time-critical aspect of the agent's physical environment? What are the performance criteria for deciding what activities take place in each level/component of the architecture?

The performance metrics are subjective measure in that they depend upon the goals of the agent and intentions of the designer. In order to make guarantees about its performance, the agent needs a value system reflecting the subjective performance metrics as close as possible in the form of the utility function. The utility function must have the cost aspect of the agent's decision process as well as the time-critical aspect of the agent's physical environment.

Psychology

Why should we build agents that mimic anthropomorphic functionalities? How far can/should we draw metaphoric similarities to human/animal psychology? How much should memory organization depend on human/animal psychology?

In our applications, the physical agents are supposed to be populated among humans and interact with humans. The agent does not need to function in an anthropomorphic way. Nonetheless, if the derived behavior of the agent looks similar to human psychology, it will have a definite advantage in effectively interfacing with humans. At least, the agent should not surprise humans in their shared environment with unexpected behavior.

Learning

How can a given architecture support learning? How can knowledge and skills be moved between different layers of an agent architecture?

In SCS, the **do best** constructor allows a dynamic utility-based selection among candidate actions. Each action in the **do best** construct has an associated utility function which suggests its expected utility. In the current implementation, the interpreter selects the action with highest utility. In a future general implementation, we may incorporate a *discount factor* for each action's utility into the interpreter. The discount factor, a value between 0 and 1, is initially set to 1. The discount factor is multiplied by the evaluated value of the utility function to select the action to do next. While the utility function is a *static* measure function defined at the design time, the discount factor is an *adaptive* counterbalance.

The agent updates the value of discount factor for each action over the repeated courses of execution using reinforcement-based learning methods. In other words, upon executing an action, the *real* utility of actions to situations is measured based on the reinforcement received in response to the action. The real utility then updates the discount factor to reflects the applicability of the action.

B Circuit Semantics

When executing on a computational system, a program is said to have *circuit semantics* when it produces (at least conceptually) electrical circuits that are in turn used for control (Nilsson 1992). In particular, a teleo-reactive (T-R) sequence is an agent control program based on circuit semantics. T-R programs combine the notions of continuous feedback with more conventional computational mechanisms such as runtime parameter binding and passing, and hierarchical and recursive invocation structures.

In contrast with some of the behavior-based approaches, T-R programs are responsive to stored models of the environment as well as to their immediate sensory inputs (Nilsson 1994). In its simplest form, a T-R program consists of an ordered set of production

$$
\begin{aligned}
K_1 &\rightarrow a_1 \\
K_2 &\rightarrow a_2 \\
&\cdots \\
K_i &\rightarrow a_i \\
&\cdots \\
K_m &\rightarrow a_m
\end{aligned}
$$

Figure 2: Simple T-R Program

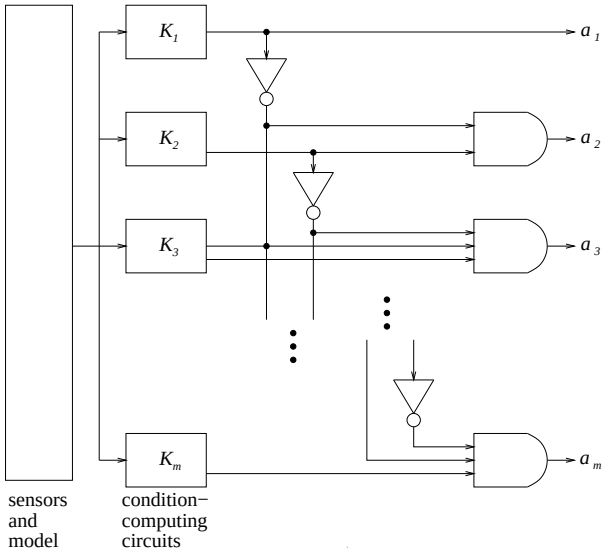

Figure 3: Corresponding Electrical Circuit (Nilsson 1992)

rules (Figure 2).

The K_i are conditions, and the a_i are actions. The interpreter scans the T-R sequence from the top until it finds a satisfied condition, and then executes the corresponding action. However, executing an action in this case might involve a prolonged activity instead of a discrete action. While the condition is the first true one, the action continues to be taken, so the T-R program can be continuously acting and evaluating whether to continue its current action (if it still corresponds to the first true condition) or to shift to another action (if the current action's condition is no longer satisfied or a condition earlier in the program becomes satisfied). Conceptually, the above T-R program produces the electrical circuit in Figure 3.

The actions, a_i, of a T-R sequence can be T-R sequences themselves, allowing hierarchical and recursive nesting of programs, eventually leading to actions that are primitives. In an executing hierarchical construction of T-R programs, note that a change of action at any level can occur should the conditions change. That is, all T-R programs in a hierarchy are running concurrently, in keeping with circuit semantics, rather than suspending while awaiting subprograms to complete.

C Structured Circuit Semantics

While T-R programs capture circuit semantics for reactive control in a very compact way, their compactness comes at the cost of representativeness for other domains. For example, the robotic manufacturing domain appears to require a language with richer circuit semantics than is embodied in T-R programs.

The basic unit in the structured circuit semantics is an *action*, a_i. Every action is atomic; it is guaranteed to terminate within a bounded time and cannot be interrupted. Atomic actions can be grouped to form other atomic actions, as in $(a_1; \cdots; a_n)$. In this case, all actions in the group are executed in sequence without being interrupted. Execution of an action usually changes the environment and/or internal state, including the world model, and returns either *success* or *failure*. The semantics of success and failure are especially important in some constructs such as **do any**, **do best**, and **do all** which will be explained below.

For generality, we can loosely define a *condition* as a function which returns TRUE or FALSE. When a condition returns true, the variables in the condition can be bound. More specifically, for our implementation we assume a pattern matching operation between condition patterns and specific relational information in the world model.

Once we have conditions and actions, we can then define various control constructs and their semantics. The purpose of most of the constructs is to wrap the actions and attach conditions to collections of actions, corresponding to the conditions (K_i) in T-R programs. As in the T-R programs' circuit semantics, the conditions are *durative* which means that they should be satisfied during the execution of the wrapped actions.

The constructs can be nested, and the attached conditions are dynamically stacked for checking. Whenever an atomic action is finished, the stack of conditions is checked from top to bottom (top conditions are the outmost conditions in the nested constructs).

If any condition is no longer satisfied, new choices of action at that level and those below are made.

A *step* is defined recursively as follows. In the construct descriptions below, $K_i, a_i, S_i, U_i, 1 \leq i \leq n$ are conditions, actions, steps, and utility functions, respectively.

• a_i is an step composed of a single atomic action. An action returns either success or failure and so does the step.

• $(a_1; \cdots; a_n)$ is an atomic step composed of atomic actions. The step fails if any of the actions fails.

• **do** $\{S_1;\ S_2;\ \cdots;\ S_n\}$ is a step that specifies a group of steps that are to be executed sequentially in the given order. The overall **do** step fails only as soon as the one of the substeps fails. Otherwise it succeeds. **do*** $\{\cdots\}$ has the same semantics as those of **do** except that, whenever a substep fails, it retries that substep until it succeeds. Thus **do*** itself never fails. This construct allows us to specify *persistent* behavior, and is particularly useful within the **do all** and **do any** constructs explained below.

• **do all** $\{S_1;\ S_2;\ \cdots;\ S_n\}$ is a step which tries to execute all steps in parallel (at least conceptually). If the agent can do only one step at a time, it nondeterministically chooses among those as yet unachieved. If any one of the steps fails, the whole **do all** fails immediately. This is similar to the semantics of the AND branch of the Knowledge Area in PRS. **do* all** is a variation of **do all** which tries failed substeps persistently, yet nondeterministically until all of them have succeeded.

• **do any** $\{S_1;\ S_2;\ \cdots;\ S_n\}$ is a step which selects nondeterministically one S_i and executes it. If that step fails, it keeps trying other actions until any of them succeeds. If every step is attempted and all fail, the **do any** step fails. This construct corresponds to the OR branch of the Knowledge Area in PRS. **do* any** is a variation of **do any** which keeps trying any action including the already failed steps until any of them succeeds.

• **do first** $\{K_1 \rightarrow S_1;\ \cdots;\ K_n \rightarrow S_n\}$ is a step which behaves almost the same as a T-R program. That is, the list of condition–step pairs is scanned from the top for the first pair whose condition part is satisfied, say K_i, and the corresponding step S_i is executed. The energizing condition K_i is continuously checked (at the characteristic frequency) as in T-R programs. The difference is that, if a step fails, the whole **do first** fails. To persistently try a step with satisfied conditions even if it fails (as in T-R programs), the **do* first** construct can be used.

• **do best** $\{K_1\ [U_1] \rightarrow S_1;\ \cdots;\ K_n\ [U_n] \rightarrow S_n\}$ is a step which evaluates U_i for each True K_i $(1 \leq i \leq n)$, and selects a step S_i which has the highest utility. If several steps have the highest utility, one of these is selected by the **do any** rules. The failure semantics is the same as that of the **do any** construct. The **do* best** step is similarly defined.

• **repeat** $\{S_1;\ S_2;\ \cdots;\ S_n\}$ works the same way as **do** does, but the steps are repeatedly executed. The **repeat*** step is also similarly defined.

The **do**, **do all**, **do any**, **do first**, **do best**, **repeat**, and their *-ed constructs may have following optional modifiers

• **while** K_0 : specifies the energizing condition K_0 to be continuously checked between each atomic action. The associated step is kept activated only while K_0 is true. For example, **do while** K_0 $\{\cdots\}$ does the **do** step as long as K_0 is true. Note that K_0 is an energizing condition of the associated step. Thus, if the energizing condition is not satisfied, the step does *not* fail, but just becomes deactivated. **until** K_0 is shorthand for **while** $\neg K_0$.

• **when** K_0 : specifies that the condition K_0 must be true before the associated step is started. That is, K_0 is only checked before execution, but not checked again during execution. For example, **do* all when** K_0 **while** K_1 $\{\cdots\}$ is a step which can be activated *when* K_0 is true, and *all* substeps of which will be *persistently* tried *while* K_1 is true. **unless** K_0 is shorthand for **when** $\neg K_0$.

References

Drummond, M. (1989). Situated control rules. In R. J. Brachman, H. J. Levesque, and R. Reiter (Eds.), *KR'89: Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning*, pp. 103–113. Toronto, Ontario, Canada: Kaufmann.

Firby, R. J. (1989, January). Adaptive execution in complex dynamic worlds. Technical Note YALE/DCS/RR #672, Department of Computer Science, Yale University.

Firby, R. J. (1992, June). Building symbolic primitives with continuous control routines. In J. Hendler (Ed.), *Artificial Intelligence Planning Systems: Proceedings of the First International Conference*, College Park, Maryland, pp. 62–68. Morgan Kaufmann.

Georgeff, M. P. and A. L. Lansky (1987). Reactive reasoning and planning. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, Seattle, Washington, pp. 677–682. (Also published in *Readings in Planning*, James

Allen, James Hendler and Austin Tate, editors, pages 729–734, Morgan Kaufmann, 1990.).

Ingrand, F. F., M. P. Georgeff, and A. S. Rao (1992, December). An architecture for real-time reasoning and system control. *IEEE Expert 7*(6), 34–44.

Lee, J. and E. H. Durfee (1994, July). Structured circuit semantics for reactive plan execution systems. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, Seattle, Washington, pp. 1232–1237.

Lee, J., M. J. Huber, E. H. Durfee, and P. G. Kenny (1994, March). UM-PRS: an implementation of the procedural reasoning system for multirobot applications. In *Conference on Intelligent Robotics in Field, Factory, Service, and Space (CIRFFSS '94)*, Houston, Texas, pp. 842–849.

McDermott, D. (1991, June). A reactive plan language. Technical Note YALEU/CSD/RR #864, Department of Computer Science, Yale University.

McDermott, D. (1992, December). Transformational planning of reactive behavior. Technical Note YALEU/CSD/RR #941, Department of Computer Science, Yale University.

Nilsson, N. J. (1992, January). Toward agent programs with circuit semantics. Technical Report STAN-CS-92-1412, Department of Computer Science, Stanford University.

Nilsson, N. J. (1994). Teleo-reactive programs for agent control. *Journal of Artificial Intelligence Research 1*, 139–158.

Pack Kaelbling, L. (1988). Goals as parallel program specifications. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, St. Paul, Minnesota, pp. 60–65.

Schoppers, M. J. (1987). Universal plans for reactive robots in unpredictable environments. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, Milan, Italy, pp. 1039–1046.