Structured Circuit Semantics for Reactive Plan Execution Systems

Jaeho Lee and **Edmund H. Durfee***
Department of EE and CS
University of Michigan
Ann Arbor, MI 48109
{jaeho,durfee}@eecs.umich.edu

Abstract

A variety of reactive plan execution systems have been developed in recent years, each attempting to solve the problem of taking reasonable courses of action fast enough in a dynamically changing world. Comparing these competing approaches, and collecting the best features of each, has been problematic because of the diverse representations and (sometimes implicit) control structures that they have employed. To rectify this problem, we have extended the circuit semantics notion of teleo-reactive programs into richer, yet compact semantics, called structured circuit semantics (SCS), that can be used to explicitly represent the control behavior of various reactive execution systems. By transforming existing systems into SCS, we can identify underlying control assumptions and begin to identify more rigorously the strengths and limitations of these systems. Moreover, SCS provides a basis for constructing new reactive execution systems, with more understandable semantics, that can be tailored to particular domain needs.

Introduction

The realization that agents in dynamic, unpredictable environments should consider the evolving state of the environment when making decisions about actions to take to pursue their goals, has led to a plethora of systems for reactive plan execution, including PRS (Ingrand, Georgeff, & Rao 1992), Universal Plans (Schoppers 1987), Teleo-Reactive Programs (Nilsson 1992; 1994), and RAPs (Firby 1989; 1992), among others. The challenge faced by a researcher who needs to incorporate a reactive plan execution system into a larger endeavor is determining how to decide among these candidate systems. For example, in a project to develop a system for controlling and coordinating outdoor robotic vehicles (Lee *et al.* 1994), which reactive plan execution system is right for the job?

A primary difficulty in answering this question is that many of the fundamental capabilities of and assumptions behind reactive plan execution systems are not easily discernible, being tied up in descriptions of procedures and interpreters, which are in turn expressed in system-specific ways. One goal of the work we describe here, therefore, is to develop a means for formally specifying reactive plan execution systems so as to cast a variety of these systems into a single framework, thereby allowing us to more readily identify and compare the capabilities and assumptions of each. With such tools in hand, moreover, we are working toward devising an interpreter for our formalism that will allow us to easily implement appropriate reactive plan execution systems with precisely the characteristics needed by a particular domain.

Our new formalism, called *Structured Circuit Semantics* (SCS), extends the Circuit Semantics of Teleo-Reactive Programs to be powerful enough to encompass the representation capability of many reactive planning systems. In this paper, we briefly review circuit semantics as a starting point for our extensions, and point out some limitations of circuit semantics that make it inappropriate for a task like that of controlling outdoor robotic vehicles. We then present our SCS formalism to overcome these limitations, and demonstrate the power of SCS through a simple assembly problem that demands reactive and robust plan execution. Finally, we analyze SCS as a general reactive plan specification language by comparing SCS with other reactive plan execution systems. We conclude this paper with discussions on implementation issues and extensions to applications involving multiple agents.

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, combining 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 re-

*This work was sponsored, in part, by ARPA under contract DAAE-07-92-C-R012.

sponsive 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 rules (from (Nilsson 1994)):

$K_1 \rightarrow a_1;\ K_2 \rightarrow a_2;\ \cdots;\ K_i \rightarrow a_i;\ \cdots;\ K_m \rightarrow a_m;$

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).

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.

Limitations of the T-R programs

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, in the outdoor robotic vehicle domain, suitable reactive execution appears to require a language with a richer circuit semantics than is embodied in T-R programs (or in many other reactive execution systems, for that matter).

Execution Cycle

In an ideal reactive system with circuit semantics, the conditions are *continuously* being evaluated and, when appropriate, their associated actions are *continuously* being executed. Real electrical circuits, however have a natural, characteristic frequency that leads to cycles of execution, and these same cycles occur in reactive execution systems, corresponding to the perception-cognition-action cycle. Traditionally, reactive systems have concentrated on increasing this frequency by, for example, reducing the time needs of cognition, but this cycle cannot be completely eliminated.

Circuit semantics represents two different kinds of actions— *energized* and *ballistic*. Energized actions are those that must be sustained by an enabling condition to continue operating; ballistic ones, once called, run to completion (Nilsson 1992). We argue that the energized actions can be implemented using ballistic actions by making the perception-cognition-action frequency 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 energized actions *must* be mapped down to the ballistic actions anyway. Our definition of atomic actions to be described later is based on this argument.

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 T-R program whose conditions are to be evaluated both against sensory input and on stored internal state information. While executing an action that is supposed to make several changes to the internal state, an earlier condition in the T-R program 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 (when "wedged") 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.

Non-Deterministic Behavior

In a T-R program, the condition-action pairs are ordered strictly and statically. However, generating a total ordering on the actions at design time might be difficult and can lead to overly rigid runtime performance. For example, actions that appear equally good at design time will have an order imposed on them nonetheless, possibly forcing the system into repeatedly taking 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 non-deterministically.

The **do any** construct in our formalism to be described in the following section can specify multiple equally good actions for the situation. One of the actions is chosen nondeterministically at run time and executed. If the chosen action fails, another action within the construct is again nondeterministically selected and tried until one of them succeeds. We can imagine a circuit component with one input and n outputs, which energizes one of its outputs whenever the component is energized. Note that, with work, this nondeterminism can be forced into T-R programs by, for example, having the condition of each equally good action include a match against some randomly assigned state variable that is randomly reassigned upon each action failure. However, once again, such machinations serve to implicitly implement a capability that should be explicit.

Best-First Behavior

One of the reasons for demanding reactivity is to be sensitive to the way utilities of actions 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 the static ordering of actions. Let's consider a simple three line example T-R program:

available(airplane) → fly;
available(car) → drive;
True → walk;

Suppose that the airplane is not available initially. The agent thus chooses to rent a car and drive. After driving 10 hours, the agent needs to drive only 30 more minutes to get to the destination; however, it discovers that it is passing an airport which has a plane available. According to the above T-R program, the agent switches to the flying action, even though it could be that, between dropping off the rental car, boarding the plan, taxiing, and so on, flying from here will take much longer than just completing the drive.

If the agent is to accomplish its goal in a timely way, the T-R program above needs additional conditions for taking the airplane to avoid this inappropriate transition. The question is *where* to put *what* conditions. For this three line program, it is not terribly hard to devise additional conditions. If we assume, for simplicity, that the function `tt` (travel-time) returns the time needed to get to the destination from the current situation, the T-R program becomes:

available(airplane) ∧ tt(airplane) < tt(car)
∧ tt(airplane) < tt(foot) → fly;
available(car) ∧ tt(car) < tt(foot) → drive;
True → walk;

But what if there are ten different ways to get to the destination? The lefthand side conditions of the T-R program might have to mention all the ways of traveling. Moreover, if a new way to travel were discovered, it could not be introduced into this T-R program without possibly affecting the conditions for other actions.

A more general answer is to introduce a *decision layer* above the *circuit layer*. The decision layer dynamically makes utility-based (cost based in dual) selections among candidate actions, conceptually energizing the "best" circuit. 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. Each action with a satisfied condition competes (or bids) by submitting its (expected) utility, and the highest bidder is selected.[1] This scheme is very similar to blackboard control mechanisms where each knowledge source proposes its utility, but differs from many blackboard systems in that the conditions and utilities for the actions are checked every cycle, as dictated by the circuit semantics. Using the **do best** construct, the above example will be represented as follows where `ttu` is a utility function of the travel time.

do best { available(airplane) [ttu(airplane)] → fly;
available(car) [ttu(car)] → drive;
True [ttu(foot)] → walk; }

Note that, in keeping with circuit semantics, we can map the decision layer into real circuitry as well, being realized as a circuit that controls other circuits which evaluate themselves dynamically. By having some number of decision layers (about decision layers), we can get the meta-levels of other reactive systems such as PRS.

Failure Semantics

The success of an action can be measured in terms of whether it had the desired effect at the desired time on the environment. As was argued previously, 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 might have a desired effect on the environment, determining whether that effect was achieved is important in controlling the execution of further actions. Effects can be checked for in the energizing conditions associated with an action, such that failure naturally leads to the adoption of a different action. However, because there might be a variety of subtle 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.

In T-R programs, if an action fails without changing any of the program's energizing conditions, the same action will be kept energized until the action eventually succeeds. 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.

Structured Circuit Semantics

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. As argued previously, sustained actions are typically repetitions of an atomic action.

Once all actions are defined, we can limit the (upper and lower) bound for the *perception-cognition-action* cycle. Atomic actions can also 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

[1] Tie-breaking is done similarly to the nondeterministic selection of actions. In fact, the **do any** construct is just a special case of the **do best** where the utility calculations always return identical values.

changes the environment and/or internal state (including the world model) and returns either *success* or *failure*. The semantics of success and failure are important in some constructs such as **do any**, **do best**, and **do all**.

For generality, we can loosely define a *condition* as a function which returns true of false, and when true can generate bindings for variables expressed in the condition.[2] We can then define various control constructs and their semantics. Some concepts are borrowed from the semantics of the Procedural Reasoning System (PRS) and a PRS implementation (UM-PRS (Lee *et al.* 1994)). Because of space limitations, we cannot describe the PRS architecture in detail here, but interested readers can refer to (Ingrand, Georgeff, & Rao 1992). The purpose of most of the constructs is to wrap the actions and attach energizing 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* and should be satisfied during the execution of the wrapped actions. The difference is that the conditions are checked only between atomic actions rather than continuously, providing us with clear semantics for the execution and feedback cycles, and avoiding the potential oscillation problems mentioned previously.

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.

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

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

$\diamond$ **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.

$\diamond$ **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.

$\diamond$ **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.

$\diamond$ **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.

$\diamond$ **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 TrueK_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.

$\diamond$ **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

$\diamond$ **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** $\neg K_0$.

$\diamond$ **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** $\neg K_0$.

[2] More specifically, for our implementation we assume a pattern matching operation between condition patterns and specific relational information in the world model. The details of this are beyond the scope of this paper.

free(1) ∧ free(2) ∧ available(B) → Lplace(B,2);
free(3) ∧ free(2) ∧ available(B) → Rplace(B,2);

free(1) ∧ free(3) ∧ available(A) → place(A,1);
free(3) ∧ free(1) ∧ available(C) → place(C,3);

free(1) ∧ ¬free(2) ∧ available(A) → place(A,1);
free(3) ∧ ¬free(2) ∧ available(C) → place(C,3).

Figure 1: T-R Program

We have described SCS as a general semantics for reactive plan execution systems. As a matter of fact, the semantics can be directly transformed into SCS Language (SCSL) which is interpreted and executed by an interpreter. We are currently implementing the *SCS Reactive Plan Execution System* in C++. The system consists of the SCSL interpreter and the world model. The SCSL has numerous other built-in actions including arithmetic operations, world model match and update, etc. In the SCSL, a step can be defined using the construct **define** with a list of arguments (local variables) that are bound when the step is called: **define** `step-name`($\$x_1$, $\$x_2$, $\cdots$, $\$x_m$) `step`. The defined step can be called, and expanded accordingly at run time.

Example: BNL Problem

To briefly illustrate how SCS can help clarify the implicit control semantics of a plan execution language, we here consider a simpler example than what we have encountered in the robotic vehicle domain, but which still exemplifies concerns in reactive and robust plan execution. In the BNL (B Not Last) problem (Drummond 1989), we are given a table on which to assemble three blocks in a row: block A on the left, at location 1; block B in the middle, at location 2; and block C on the right, at location 3. The blocks are not initially available for placement, and each block can be placed on the table once available (the exact means for moving blocks is immaterial). The only constraint on assembly is that *block* B *cannot be placed last:* once A and C are down, there is not enough room to squeeze in B since it must be swept in from the left or the right. We assume that a block, once placed, cannot be moved away again.

As in (Drummond 1989), we assume three predicates: `free`, `at`, and `available`; and four actions: `place(A,1)` (place block A at location 1), `place(C,3)` (place block C at location 3), `Lplace(B,2)` (sweep B in from the left), and `Rplace(B,2)` (sweep B in from the right). The example in Figure 1 is an T-R program that we generated to solve this problem.[3] This program solves the problem, but contains some implicit ordering preferences for placing B from the left and placing A first when A and C are both available. By translating the T-R program into SCS (figure 2), and comparing it to our own solution in SCS (figure 3), the implicit control ordering of T-R programs is explicitly seen,[4] highlighting how T-R programs cannot capture, in an explicit way, nondeterminism, which has been captured in the richer semantics of SCS.

[3]In (Nilsson 1992), a similar T-R program is presented, but it is for a simpler variation of the BNL problem.

```
do* first {
    free(1) ∧ free(2) ∧ available(B) → Lplace(B,2);
    free(3) ∧ free(2) ∧ available(B) → Rplace(B,2);
    free(1) ∧ free(3) ∧ available(A) → place(A,1);
    free(3) ∧ free(1) ∧ available(C) → place(C,3);
    free(1) ∧ ¬free(2) ∧ available(A) → place(A,1);
    free(3) ∧ ¬free(2) ∧ available(C) → place(C,3) }
```

Figure 2: Direct Translation to SCS

```
repeat while free(1) ∨ free(2) ∨ free(3) {
 do any {
  do while free(2) ∧ available(B) {
   do any { do when free(1) { Lplace(B, 2) };
            do when free(3) { Rplace(B, 2) }; }}
  do while free(1) ∧ free(3) {
   do any { do when available(A) { place(A, 1) };
            do when available(C) { place(C, 3) }; }}
  do while ¬free(2) {
   do any { do when available(A) { place(A, 1) };
            do when available(C) { place(C, 3) }; }}}}}
```

Figure 3: SCS Program for the General BNL Problem

Related Work and Future Work

Because SCS embodies circuit semantics, previous comparisons (Nilsson 1992; 1994) between T-R programs and reactive plan execution systems such as SCR (Drummond 1989), GAPPS (Pack Kaelbling 1988), PRS (Ingrand, Georgeff, & Rao 1992), and Universal Plans (Schoppers 1987), are applicable here as well. In this section, therefore, we concentrate on comparisons more specifically 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 are manifested when the simple SCS constructs interact in various ways.

Situated Control Rules (SCR) are constraints for plan execution that are used to 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

[4]The SCS constructs can often allow multiple such mappings. Thus, while program equivalence cannot always be detected syntactically, the implicit control information can be captured and compared explicitly.

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, such as (1) it does not consider variable binding, (2) it has no hierarchical execution structure (function call or recursion), and (3) it has no run-time reasoning.

The RAP system (Firby 1989; 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, as well as 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 enter a method from the method's middle. SCS has clear failure semantics and specifies what to do and where to start. Another limitation of the RAP interpreter is that methods lack run-time priority information, which is expressed by utility functions in SCS **do best**. RPL (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.

The central system of Sonja (Chapman 1990) uses a circuit description language, MACNET and arbitration macrology. Although it supports circuit semantics at the (boolean) gate level, the arbitration macrology allows only compilation time arbitration in a non-structured manner because it compiles down into MACNET circuitry just before the system runs. SCS can more generally capture reactive behavior required for applications such as playing video games (Sonja (Chapman 1990) and Pengi (Agre & Chapman 1987)) and for the traffic world scenario (CROS (Hendler & Sanborn 1987)).

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 for us to be able to generate it, reason about it, and communicate about 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 meta-level 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 meta-level decision making in SCS using the **do best** construct. Note that these constructs can be nested to arbitrary depth, corresponding to multiple meta-levels in PRS choosing the best method for choosing the best method for achieving a desired goal. 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.

Encouraged by the ability of SCS to capture explicitly the control structures of various plan execution systems, we are implementing *SCS Reactive Plan Execution System* in C++. This effort is directed at supplanting our previous implementation of PRS with a more general execution system that can be tailored to the control needs of our application domain. Toward this end, we are currently working on rigorously capturing in SCS the content of PRS meta-level knowledge areas. 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.

References

Agre, P. E., and Chapman, D. 1987. Pengi: An implementation of a theory of activity. In *AAAI-87*, 268–272.

Chapman, D. 1990. Vision, instruction and action. Tech. Report 1204, MIT AI Laboratory.

Drummond, M. 1989. Situated control rules. In *KR'89*, 103–113.

Firby, R. J. 1989. Adaptive execution in complex dynamic worlds. Tech. Note YALE/DCS/RR #672, Dept. of Computer Science, Yale University.

Firby, R. J. 1992. Building symbolic primitives with continuous control routines. In Hendler, J., ed., *Artificial Intelligence Planning Systems: Proc. of the First International Conference*, 62–68.

Hendler, J. A., and Sanborn, J. C. 1987. A model of reaction for planning in dynamic environments. In *Proc. of the DARPA Knowledge-Based Planning Workshop*, 24.1–24.10.

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

Lee, J.; Huber, M. J.; Durfee, E. H.; and Kenny, P. G. 1994. 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)*, 842–849.

McDermott, D. 1992. Transformational planning of reactive behavior. Tech. Note YALEU/CSD/RR #941, Dept. of Computer Science, Yale University.

Nilsson, N. J. 1992. Toward agent programs with circuit semantics. Tech. Report STAN-CS-92-1412, Dept. 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 *AAAI-88*, 60–65.

Schoppers, M. J. 1987. Universal plans for reactive robots in unpredictable environments. In *IJCAI-87*, 1039–1046.