Using Abstraction and Nondeterminism to Plan Reaction Loops

David J. Musliner
Institute for Advanced Computer Studies
The University of Maryland
College Park, Maryland 20742
musliner@umiacs.umd.edu

Abstract

By looping over a set of behaviors, reactive systems use repetition and feedback to deal with errors and environmental uncertainty. Their robust, fault-tolerant performance makes reactive systems desirable for executing plans. However, most planning systems cannot reason about the loops that characterize reactive systems. In this paper, we show how the structured application of abstraction and nondeterminism can map complex planning problems requiring loop plans into a simpler representation amenable to standard planning technologies. In the process, we illustrate key recipes for automatically building predictable reactive systems that are guaranteed to achieve their goals.

Introduction

The uncertainty inherent in real-world domains has proven problematic for traditional AI planning technologies that rely on complete, accurate, and deterministic world models. In response, reactive systems (e.g., Agre & Chapman 1987; Firby 1987) have become popular because they can deal with the uncertainties of real-world domains. The primary advantage of reactive systems is that they do not make predictions based on a world model, and thus they avoid potential failures due to inadequate models. Instead, reactive systems rely on repeatedly executing simple "persistent" behaviors until feedback indicates that their goals have been achieved. Unlike traditional open-loop AI plans, these reactive systems address environmental uncertainty and the possibility of execution-time failures by implementing repeated, feedback-based, closed-loop behaviors.

One major problem with most reactive systems is that they are difficult to design (and usually hand-coded), so that their behaviors are not necessarily logically correct or timely. There is no assurance that these systems will choose an appropriate action for a given situation, or that the selected action will be executed quickly enough to meet domain-imposed deadlines. Furthermore, because the design and construction process is not automated, building new reactive systems for different domains requires lengthy human interactions. Several researchers have recognized that traditional AI planning systems might be used to automate the *ad hoc* process of designing reactive systems (e.g., Schoppers 1990), leading to significant advances in performance predictability and rapid system adaptation. However, most AI planning systems are unable to plan in domains that involve the type of repetition (looping) characteristic of reactive systems. Furthermore, planners do not usually create reactions, but rather they generate plans as a fixed sequential (or partially ordered) set of distinct actions.

In this paper we describe the use of abstraction and nondeterminism to allow planners to generate looping reactive plans, thus addressing a critical problem for hybrid planning/reaction systems. Essentially, we give cookbook recipes for transforming the complexities of a domain into an abstract form so that reactive plans are suitable and classical planners are useful. Some of the abstraction techniques are novel, some are not: our primary contribution is in showing how they can be combined in routine ways to make planners handle loops and generate reactive behaviors that can be guaranteed to operate in a logical and timely fashion.

We begin by introducing a simple, intuitive example to illustrate the first abstraction technique and how it can be useful in the automatic planning of loops. We then present a brief overview of the Cooperative Intelligent Real-Time Control Architecture (CIRCA) (Musliner, Durfee, & Shin 1993), a system designed to automatically plan and execute reactive behaviors. We then provide a more detailed example showing how CIRCA uses abstraction, nondeterminism, and an "abstract" time representation to plan reactive loops. Despite the extensive domain abstractions used to make planning feasible, CIRCA's automatically-constructed reactive systems are guaranteed to accomplish their goals and preserve the system's safety. Thus CIRCA combines the positive potential of both planning systems (knowledge-based deliberation, provably logical behavior) and reactive systems (rapid, fault-tolerant feedback loops).

This work was supported in part by the National Science Foundation under Grants IRI-9209031 and IRI-9158473, by a NSF Graduate Fellowship, and by the Arpa/Rome Laboratory Planning Initiative (F30602-93-C-0039). David Musliner is also affiliated with the UM Institute for Systems Research (NSF Grant NSFD CDR-88003012).

```
OPERATOR precise-hammer-blow
  PRECONDS: ((arm-raised T) (nail-height ?X))
  POSTCONDS: ((arm-raised nil)
              (nail-height (max 0 (- ?X 1.2))))
```

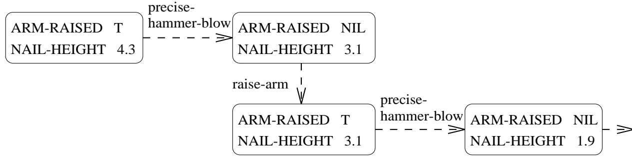

Figure 1: A fictitious hammering operator and the plan that might result.

A Simple Example

In his early work on representing plan loops, Drummond (1985) illustrates a plan for hammering a nail by repeatedly raising and lowering a hammer, never terminating. In this section, we introduce our approach to planning loops using this same example, with two significant differences: first, our hammering plan is automatically generated, and second, the plan will terminate when the nail is driven flush.

Consider first the way in which a (hypothetical) traditional planning system might address the hammering problem, as illustrated in Figure 1. Here the **precise-hammer-blow** operator must specify exactly how far down it drives the nail, and the final plan yields a world model that enumerates all the possible heights the nail may protrude.

The hammering domain illustrates several critical aspects of looping plans that make them difficult for planners. First, uncertainty may make it impossible to specify an operator's effects so precisely, and hence impossible to predict exactly how many loop iterations will be necessary to achieve the goal. In such uncertain domains, the termination conditions of the loop can only be determined during the actual execution of the loop. However, traditional planning operator representations (e.g., STRIPS add/delete lists (Nilsson 1980)) cannot represent an operator whose effects are not fully deterministic[1]. Another major problem is that, even if we could predict how many iterations are necessary, a classical planning system would need to enumerate at least one state (and probably many) for each of the iterations. This is obviously undesirable, since it exacerbates the state-space explosion already experienced by classical planners. To address these problems and allow a planner to derive a compact plan without considering innumerable states, we apply two forms of abstraction[2].

[1]A deterministic operator implements a fixed mapping of an input to a unique corresponding output. A nondeterministic operator implements a completely uncertain (or random) mapping from an input to one of a set of possible outputs.

[2]We use "abstraction" in a general sense to mean the omission of detail. This usage conforms nicely with intuition, as well as with (Wilkins 1988), in that our abstract models match larger sets of possible worlds than less-abstract models.

```
ACTION hammer-blow
  PRECONDS: ((arm-raised T))
  POSTCONDS: (((arm-raised nil) (nail-flush T))
              ((arm-raised nil) (nail-flush nil)))
```

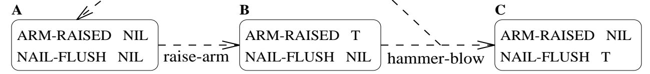

Figure 2: A nondeterministic transition, and the resulting dynamically-terminated loop.

First, we must abstract away the details in the domain representation that cause problems with the state-space explosion. In the hammering domain, the height of the nail is the domain feature that changes on each loop of the plan, so it is the culprit, the "counting" variable. We remove this counting effect by abstracting the height feature to two critical values: either the nail is flush or it is not.

Since the height of the nail has been abstracted away to a binary value, the effects of the **hammer-blow** operator must be similarly abstracted. In the process, the excessive precision associated with the continuous-valued operator is abstracted away, and nondeterminism is used to represent the resulting uncertainty. Figure 2 shows the abstracted operator, whose effects are now represented as a nondeterministic transition either to **(nail-flush T)** or back to **(nail-flush nil)**.

So now we have the world model shown in Figure 2, which more accurately reflects the uncertainty of the real world: the nail is initially sticking out above the surface, and we can keep hitting it until it is finally flush with the surface, at which time we will move out of the loop and into state **C**. Summarizing the techniques used thus far, we have the following recipe:

Recipe 1: Eliminating Counting Variables

1. Create a binary variable with **T** and **nil** states corresponding to the critical "some or none" transition of the counting variable.
2. Modify the increment operator to lead to the **T** state of the binary variable.
3. Modify the decrement operator to be nondeterministic, leading to either of the binary variable's states.

With the abstract **nail-flush** feature and the corresponding operator, the state-space problem has been addressed and the model in Figure 2 represents the need for a loop which repeats until a dynamic termination condition holds. Note that the **hammer-blow** action is not sufficient by itself, because we do not want to hit the nail every time we are holding up the hammer. To build a reaction that would yield the state-space behavior shown in Figure 2, *we still need a planner* to decide which of the various applicable operators should actually be executed in any particular world

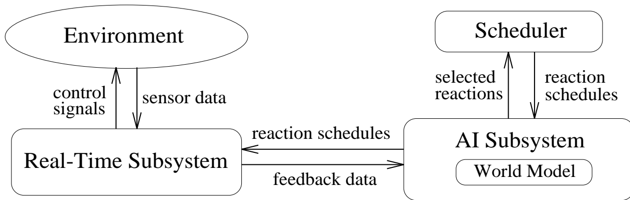

Figure 3: Overview of CIRCA.

```
TAP hammer-blow
  TEST: (and (nail-flush nil) (arm-raised T))
  ACTION: hammer-blow
  MAX-PERIOD: 2 seconds

SCHEDULE: (raise-arm hammer-blow) repeat
```

Figure 4: A trivial example TAP & TAP schedule for the hammering domain.

state. The planner must decide that we should invoke the **hammer-blow** action only when the nail is not yet flush (i.e., in state **B** only, not in state **C**). In the next section, we briefly describe how CIRCA is able to perform this type of planning (despite the resulting state-space loops), and how the system addresses the final problem of the hammering domain: actually representing a looping plan.

Overview of CIRCA

As illustrated in Figure 3, CIRCA consists of three subsystems operating in parallel (Musliner, Durfee, & Shin 1993). The AI Subsystem (AIS) acts as a planning system, reasoning about a model of the domain and deriving appropriate reaction plans. These plans are sent to the Scheduler module, along with timing constraints expressing how frequently each reaction must be executed. The Scheduler tries to build a cyclic schedule of reactions that will meet all the timing constraints. If a schedule is found, the planned reactions can be sent to the Real-Time Subsystem (RTS) for execution. The RTS executes previously-derived plans while the AIS and Scheduler are cooperatively developing a new plan; each reaction plan is designed to keep the system safe (avoiding failures), so that the search-based planning performed by the AIS is isolated from the ongoing real-time deadlines of the environment.

CIRCA's reactive plans are built as schedules of Test-Action Pairs (TAPs). As shown in Figure 4, each TAP is an annotated production rule consisting of a test expression, an action, and a timing constraint on how frequently the TAP must be executed. When executing a TAP, the RTS evaluates the test expression and, if it returns true, the RTS executes the corresponding action. TAPs differ from other reactive mechanisms such as RAPs (Firby 1987) in two fundamental ways: first, TAPs are automatically generated by CIRCA's planning system, and second, TAPs specify how frequently they must be executed in order to meet domain deadlines. CIRCA's Scheduler module uses the TAP timing requirements when it builds TAP schedules that are themselves loops; Figure 4 shows a simple schedule for the hammering domain, which oscillates between the **raise-arm** and **hammer-blow** TAPs.

The world model and planning algorithm that the AIS uses to develop TAP plans are detailed in (Musliner, Durfee, & Shin 1994). For our purposes, it is sufficient to understand that the model is a modified state/transition graph in which states correspond to complete descriptions of the world (modulo some level of abstraction), and three types of transitions represent the ways the world can change. *Temporal transitions* represent time and ongoing processes. The timing behavior of a temporal transition is related to the rate of the process it represents: for example, the process of consuming a jar of salsa will take some minimum amount of time to complete, depending on the rate of consumption. *Event transitions* represent occurrences outside the agent's control, while *action transitions* represent the intentional actions of TAPs. CIRCA can control the timing behavior of action transitions by setting the timing constraints of TAPs. For example, CIRCA can build a TAP that executes at least once every minute, to ensure that a new jar of salsa is opened within two minutes after the last jar is finished.

To build plans, CIRCA begins with a set of goal descriptions, a set of initial world states, and a set of transition descriptions that detail the types of events, actions, and processes possible in the world. The planning algorithm pushes the initial states onto a stack and then performs a modified STRIPS-like depth-first search for a plan that satisfies all the system's goals. On each planning loop iteration, the top state is popped off the stack and all applicable event and temporal transitions are applied, generating new reachable states that are pushed onto the stack. The planner uses a multi-step lookahead heuristic to choose the best action for the current state, generates the states that result from the selected action, and then repeats the planning loop. Chronological backtracking is initiated if the planner cannot find a good plan (e.g., if it cannot avoid a catastrophic failure state).

To illustrate the planning process, consider again the nailing domain example in Figure 2. If state **A** is the initial condition given to the planner, it will choose to apply the **raise-arm** action, generating state **B**. In state **B**, when the **hammer-blow** action transition is applicable, the planner will project forward both of the action's possible postconditions, and will recognize that it may lead to the desired state **C**, where **(nail-flush T)** holds. Thus the **hammer-blow** action will be chosen correctly to accomplish the task. Projecting forward along the other branch of the nondeterministic postconditions, the planner will also realize that the action transition may loop back onto state **A**. Since an action has already been selected for that state, no further planning is necessary. Thus the nondeterminism poses no difficulty, and CIRCA can easily plan looping behaviors with dynamic termination conditions.

The repetition itself is inherent in all of CIRCA's plans, because they are implemented not as traditional sequential plans but as reactive TAP plans. The RTS continually loops over the schedule of TAPs, repeatedly testing their applicability conditions and executing their actions whenever appropriate. Thus, if the world model contains a loop (i.e., the planner thinks the world may re-enter a state it has been in before), the TAP form of the control plan already ensures that the state will be recognized and appropriate action taken, as many times as necessary. The planner does not need to perform any additional reasoning to accommodate repeated behaviors.

A More Complex Example

Several aspects of the hammering domain make it particularly simple, including the lack of events and temporal transitions (processes), the lack of timing requirements such as deadlines, and the simple goal of achievement. To extend beyond those limitations, we introduce the "grocery stocking" domain, in which an agent must never run out of a particular grocery item (say, salsa). The agent must develop a plan that coordinates opening new jars of salsa, putting salsa on the shopping list when stock runs low, and going grocery shopping to replenish the stock. There are several tough problems hidden in this seemingly simple domain, including plan loops, a counting variable, and a special type of goal.

However, before we address these problems with the abstraction techniques described above, we must first utilize a different form of abstraction called *indexical features* (Agre & Chapman 1987). This technique is used to avoid the enumeration problems that result from individuating specific objects in the environment. For example, if the planner distinguished between individual salsa jars (e.g., **jar21** and **jar22**) it would have to know all the possible jar names ahead of time, or else it would need the ability to generate new names, and the state space would be infinite.

To avoid this problem, we encode the environment using indexical features, which refer to objects by their relationship to our agent. For example, we can use a feature **have-open-salsa** to indicate that a jar of salsa is currently being consumed, but the specific identity of that jar need never be established. Indexical features thus abstract away from the identity of objects, but they do so in a slightly unusual fashion. The mapping of individual objects to their "classification" by indexical features is dynamic, changing as objects move through the world. So the salsa jar that is open at one time may be different than the jar open at another time, but the agent's representation will not indicate any difference.

Many reactive systems use indexical features to avoid the difficulties of establishing symbol grounding and "object permanence" through sensing (e.g., determining that the jar you leave in the refrigerator is **jar21**, and that it is the same one you find there the next day). Although common among reactive systems, indexicality is rare among traditional planning systems, which usually name objects individually.

```
ACTION open-new-jar
  PRECONDS:  ((have-salsa-in-stock T))
  POSTCONDS: ( ((have-open-salsa T)
               (have-salsa-in-stock T))
              ((have-open-salsa T)
               (have-salsa-in-stock nil)) )
  MAX-DELAY: 5 minutes

ACTION put-salsa-on-list
  PRECONDS:  ()
  POSTCONDS: ((salsa-on-list T))
  MAX-DELAY: 1 minute

ACTION go-shopping-and-get-salsa
  PRECONDS:  ((salsa-on-list T))
  POSTCONDS: ((have-salsa-in-stock T)
              (salsa-on-list nil))
  MAX-DELAY: 1 hour

TEMPORAL finish-salsa-jar
  PRECONDS:  ((have-open-salsa T))
  POSTCONDS: ((have-open-salsa nil))
  MIN-DELAY: 2 days

TEMPORAL starve-without-salsa
  PRECONDS:  ((have-open-salsa nil))
  POSTCONDS: ((failure T))
  MIN-DELAY: 8 hours

GOALS: ((failure nil))

INITIAL STATE: ((salsa-on-list nil)
                (have-open-salsa T)
                (have-salsa-in-stock T))
```

Figure 5: Example domain description for the salsa-stocking problem.

Recipe 2: Eliminating Named Objects

1. Replace non-indexical state features with indexical, agent-oriented features.
2. Modify related operators.

Because the agent may stock up on salsa, a completely accurate model of the problem would have to include a variable indicating exactly how many jars are in stock at any time. We have already seen how such counting variables can cause problems with state-space enumeration and overly-precise operators. Therefore, we apply Recipe 1 to convert the counting variable into a binary feature.

The salsa domain also introduces a different type of goal: a goal of avoidance (never run out of salsa), rather than a goal of achievement (make the nail flush)[3]. Along with the new type of goal comes the complexity of representing time and ongoing processes in the

[3] Goals of avoidance (e.g., avoid `(out-of-salsa T)`) might also be thought of as goals of negated maintenance (e.g., maintain `(not (out-of-salsa T))`).

world. To make sure that the agent does not starve from lack of salsa, the planner must reason about the relative speeds and frequencies of shopping trips, salsa consumption, and other activities.

While there have been many forays into temporal representations for planners (e.g., Allen 1983), none have focused on the sort of repeated, long-term behaviors we are interested in producing. Instead, most temporal logic systems focus on maintaining partial ordering constraints among time intervals, for non-looping plans. Plan loops would pose severe problems for these approaches, in part because the duration of a loop may not be determined until runtime. Instead, we introduce an abstracted form of time information that is simple to manipulate, yet allows CIRCA to build reactive plans that are guaranteed to meet domain deadlines.

Figure 5 shows a sample set of CIRCA transition descriptions for the salsa domain. We have applied the previously-described abstraction techniques to eliminate the stock-counting variable, instead using the binary variable **have-salsa-in-stock**. The action of buying more stock now simply sets **have-salsa-in-stock** to **T**, and removing a jar from stock has a nondeterministic outcome, either leaving some stock, or not. The process of salsa consumption is represented by the temporal transition **finish-salsa-jar**, indicating that the agent takes at least two days to consume a jar. The goal of avoidance is expressed by the **starve-without-salsa** temporal transition, which indicates catastrophic failure will occur if the agent has no open jar of salsa for eight hours.

These latter temporal transitions embody our recipe for temporal abstraction: rather than representing detailed information about the rate at which a process proceeds (which may vary with domain features (e.g., menus, time of day)), we abstract that information to a single *worst-case* number. For temporal transitions, this is the shortest possible time until the transition to a new state might occur. For action transitions, the worst case is the longest possible time until the action will occur. These worst-case values can then be used to derive the rates at which various reactions must be executed in order to achieve their goals. In the salsa domain, we must ensure that the **starve-without-salsa** transition to failure is never allowed to happen. The basic idea is to build a TAP that executes frequently enough that some action will definitely be taken before that temporal transition to failure occurs, *preempting* failure and leading instead to a more desirable state. For example, CIRCA may decide that it must execute a TAP implementing the **open-new-jar** action at least once every 7 hours, to avoid starving from lack of salsa. Note that this does not mean that a new jar will be opened that frequently, but rather that the system will check to see if a new jar *should* be opened.

Because CIRCA only deals with a single worst-case timing value for each action and temporal transition, the process of manipulating this timing information is fairly simple. However, by retaining enough information to plan preempting reactions that deal with the domain's worst-case situations, this abstraction method still allows CIRCA to build TAP plans with guaranteed behavior. Summarizing, we have:

Recipe 3: Simplifying Time

1. Encode temporal transitions (external processes) with a minimum time to completion.
2. Encode action transitions (desired activities) with a maximum time to completion.
3. The only useful relation between these timed transitions is preemption.

In Figure 5, we have expressed the goal of avoidance via the **starve-without-salsa** temporal transition to failure. We can vary the precise meaning of the goal by altering the transition's timing parameter. For example, if the goal is "absolutely never run out of salsa," we can set the transition's timing delay to zero, so that as soon as there is no more salsa, failure occurs. Alternatively, if the goal is "never run out of salsa for more than eight hours," then the transition delay will be eight hours, and the agent will have a somewhat easier time dealing with the problem. Figure 6 shows a domain model for the latter case, in which the planner has reasoned about the rate of salsa consumption and the time until "salsa starvation" sets in, and it has decided when it must go shopping. In this case, the planner has found that it is acceptable to allow the agent to empty its stock of salsa entirely, even finishing off the last open jar before going shopping to avoid "salsa starvation." If the TAPs built for this reaction plan are approved by the Scheduler, the plan is feasible, and CIRCA can guarantee to avoid failure through starvation.

Thus CIRCA illustrates two of the desirable features of a hybrid planning/reacting system: first, CIRCA's reactive plans are automatically generated, so they are provably logical and timely; and second, the system can adapt to new domains using its planner. For example, suppose that the starvation transition's delay is shorter, and the agent can not be sure that it could go shopping quickly enough after all the salsa is consumed to avoid starvation. In that case, CIRCA's planner would find that the former plan is untenable, and it would backtrack to try a different approach. As shown in Figure 7, the planner could decide to go shopping as soon as the last jar of salsa is opened, rather than waiting until it has been consumed. In this way, CIRCA can reason about the timing constraints on its behavior and build goal-oriented reaction plans despite uncertainty, abstraction, and the loops in the domain model.

Conclusion

We have illustrated the use of several forms of abstraction to simplify complex planning domains, and make their looping behavior amenable to classical planning techniques. Using nondeterministic operators, indexical features, and worst-case timing values, CIRCA is able to automatically build reaction plans that are

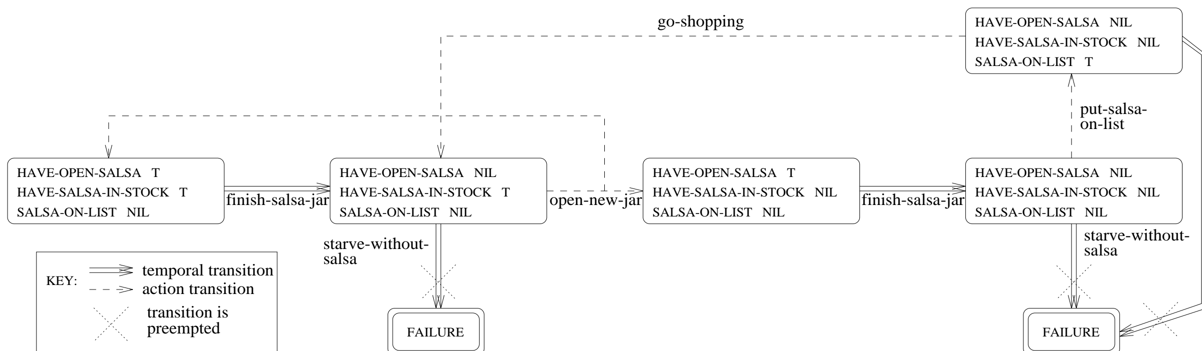

Figure 6: One possible world model of the salsa domain, after the planner has derived actions to avoid failure. In this case, the shopping need only be done after all the salsa has been consumed.

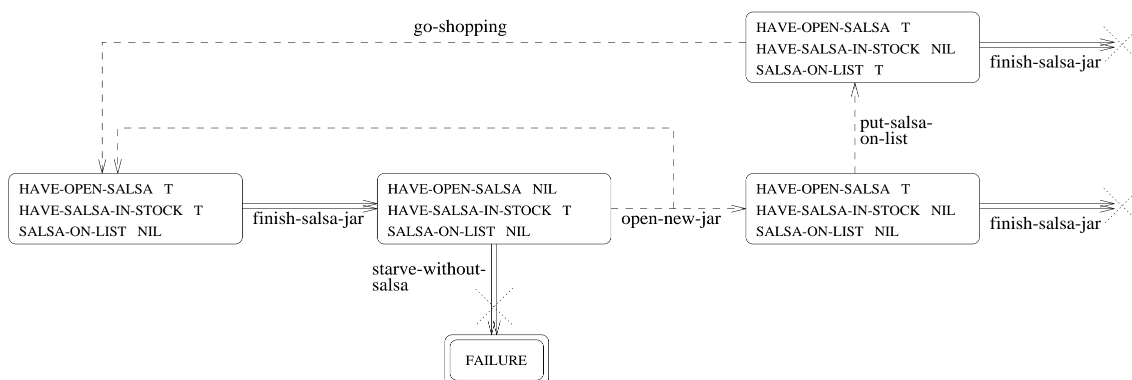

Figure 7: Another possible domain model, in which the agent must put salsa on the shopping list as soon as the stockpile is empty.

guaranteed to "do the right thing, by the right time." These aspects of provably logical and timely behavior make CIRCA's hybrid approach to planning and reaction more flexible and rigorous than previous systems.

There are several obvious extensions to the abstraction techniques we have described. For example, the replacement of counting variables with binary features can be generalized to the use of finite-range abstract features with a larger set of operators. In the salsa domain, if the time to finish a jar and starve was less than the shopping time, a useful encoding of the domain would have a trinary feature that could represent when only one jar remains, at which time the system would need to go shopping.

Currently, CIRCA requires the human system designer to make its representation decisions, such as how to map a counting variable into an abstract feature. However, given mapping patterns of the sort described here, and an ability to recognize critical state distinctions, it seems clear that an automated system should be able to derive useful and appropriate abstract representations for complex domains. Future work, then, might focus on developing additional recipes for abstraction, and extracting rules for when the recipes are useful and appropriate.

References

Agre, P. E., and Chapman, D. 1987. Pengi: An implementation of a theory of activity. In *Proc. National Conf. on Artificial Intelligence*, 268–272.

Allen, J. F. 1983. Maintaining knowledge about temporal intervals. *Communications of the ACM* 26(11):832–843.

Drummond, M. 1985. Refining and extending the procedural net. In *Proc. Int'l Joint Conf. on Artificial Intelligence*, 528–531.

Firby, R. J. 1987. An investigation into reactive planning in complex domains. In *Proc. National Conf. on Artificial Intelligence*, 202–206.

Musliner, D. J.; Durfee, E. H.; and Shin, K. G. 1993. CIRCA: A cooperative intelligent real-time control architecture. *IEEE Trans. Systems, Man, and Cybernetics* 23(6):1561–1574.

Musliner, D. J.; Durfee, E. H.; and Shin, K. G. 1994. World modeling for the dynamic construction of real-time control plans. To appear in *Artificial Intelligence*.

Nilsson, N. J. 1980. *Principles of Artificial Intelligence*. Tioga Press, Palo Alto, CA.

Schoppers, M. 1990. Automatic synthesis of perception driven discrete event control laws. In *Proc. 5th IEEE Int'l Symposium on Intelligent Control*, 410–416.

Wilkins, D. E. 1988. *Practical Planning: Extending the Classical AI Planning Paradigm.* Morgan Kaufmann.