Modeling and Programming Devices and Web Agents*

Sheila A. McIlraith

Knowledge Systems Laboratory, Department of Computer Science
Stanford University, Stanford CA 94025-9020
sam@ksl.stanford.edu

Abstract. This paper integrates research in robot programming and reasoning about action with research in model-based reasoning about physical systems to provide a capability for modeling and programming devices and web agents, which we term *model-based programming*. Model-based programs are reusable high-level programs that capture the procedural knowledge of how to accomplish a task, without specifying all the device- and web-service-specific details. Model-based programs must be instantiated in the context of a model of a specific device/web service and state of the world. The instantiated programs are simply sequences of actions, which can be executed by an appropriate agent to control the behavior of the system. The separation of control and model enables reuse of model-based programs across classes of related devices and services whose configuration changes as the result of replacement, redesign, reconfiguration or component failure. Additionally, the logical formalism underlying model-based programming enables verification of properties such as safety, program existence, and goal achievement. Our model-based programs are realized by exploiting research on the logic programming language Golog, together with research on representing actions and state constraints in the situation calculus, and modeling physical systems using state constraints.

1 Introduction

We are seeing a tremendous increase in the prevalence of physical devices with embedded digital controllers. Such devices range from smart children's toys to smart photocopiers and buildings, power distribution systems, and autonomous spacecraft. We are likewise seeing the web evolve from an information-based service provider to a network populated by programs, sensors and other devices. It is predicted that in the next decade, computers will be ubiquitous, that many device will have some sort of computer inside them, and that many business services will be agent-enabled and delivered over the web. Designing reliable software for these devices and web agents is often a complex task. In our research, we examine formal techniques to model, diagnose, test, and more recently, to program physical devices and web agents that are controlled by digital computers.

In this paper we integrate and extend research on the agent programming language Golog (e.g., [4]) and reasoning about action in the situation calculus with research in

* A preliminary version of this work was presented at the Tenth International Workshop on Principles of Diagnosis, June, 1999.

modeling physical systems and model-based reasoning about physical systems (e.g., [7–9]) to provide a new capability for developing device and web agent software which we term *model-based programming*[1][2]. Model-based programs are generic, reusable high-level programs that capture the generic procedural knowledge of *how to* accomplish a task for a class of devices, such as isolating valve leaks in spacecraft, without specifying all the device-specific details, such as turning off valve-54 before valve-93. Model-based programs (MBPs) are deductively instantiated in the context of a rich device-specific model of device structure, behavior and function, and state. The instantiated programs are simply sequences of actions, which are performed to realize the program.

The merits of model-based programming come from the exploitation of models of system behavior and from the separation of those models from high-level procedural knowledge about how to perform a task for *classes* of like devices. Model-based programs are written at a sufficiently high level of abstraction that they are amenable to reuse in the face of device reconfiguration, replacement, redesign or component failure. Also, they are easier to write than traditional control programs for devices, ridding the engineer/programmer of keeping track of the potentially complex details of a system design, with all its subcomponent interactions. Finally, because of the logical foundations of model-based programming, certain properties of model-based programs such as safety, program existence and goal achievement can be verified, and/or simply enforced in the generation of program instances.

In this paper, we argue that the situation calculus [6, 14] and the logic programming Golog [4] together provide a natural formalism for model-based programming. To develop the models for our model-based programming paradigm, we take as our starting point: a set of state constraints in first-order logic, that can describe the structure and behavior of a physical device; and a set of actions. We appeal to a solution to the frame and ramification problems in the situation calculus in order to provide an integrated representation of our physical device and the actions that affect it. This representation scheme is the critical enabler of our model-based programming capability. It provides the representation scheme for declarative encoding of the *model* for our model-based programs. With a representation for our models in hand, we introduce the notion of a model-based program, show how to exploit Golog to specify model-based programs, and show how to generate program instances from the program and the model using deductive machinery. Finally, we show how the logical formalism underlying model-based programming enables verification of certain properties such as safety, program existence, and goal achievement. We conclude with a brief discussion of related work.

2 Model-Based Programming

Model-based programming comprises two components:

A Model which provides an integrated representation of the structure and behavior of the complex physical system being programmed, the operator or controller actions

[1] The term *model-based programming* did not originate with us (e.g., [16, 18], etc.) We each use the term differently.

[2] As a result of space limitations, we restrict our discussion to devices, leaving further explicit discussion of web agents to a longer paper.

that affect it, and the state of the system. The model dictates the language for the program, and is often shared over a class of like devices.

A Program which describes the high-level procedure for performing some task, using the operator or controller actions.

2.1 The Model

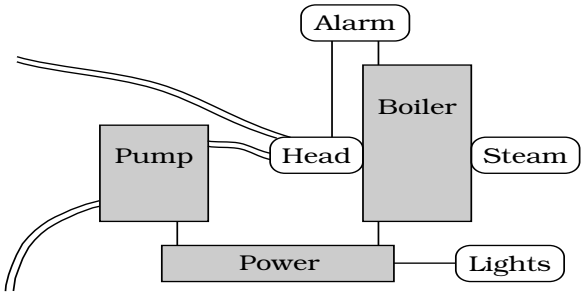
The first step towards achieving our vision of model-based programming is to define a suitable representation for our models. In this section we demonstrate that the situation calculus will provide a suitable language for declarative specification of our device models. Model-based reasoning often represents the structure and behavior of physical systems as a set of state constraints in first-order logic. The first challenge we must address is how to integrate operator or controller actions into our representation, in order to obtain an integrated representation of our system. To do so, we appeal to a solution to the frame and ramification problems proposed in [7, 10], that automatically compiles a situation calculus theory of action with a a set of state constraints. We begin with a brief overview of the situation calculus.

The Situation Calculus The situation calculus language we employ to axiomatize our domains is a sorted first-order language with equality. The sorts are of type $\mathcal{A}$ for primitive *actions*, $\mathcal{S}$ for *situations*, $\mathcal{F}$ for fluents, and $\mathcal{O}$ for everything else, including domain objects ([14]). We represent each action as a (possibly parameterized) first-class object within the language. Situations are simply sequences of actions. The evolution of the world can be viewed as a tree rooted at the distinguished initial situation S_0. The branches of the tree are determined by the possible future situations that could arise from the realization of particular sequences of actions. As such, each situation along the tree is simply a history of the sequence of actions performed to reach it. The function symbol *do* maps an action term and a situation term into a new situation term. For example, $do(turn\_on(Pmp, S_0))$ is the situation resulting from performing the action of turning on the pump in situation S_0. The distinguished predicate $Poss(a, s)$ denotes that an action a is possible to perform in situation s (e.g., $Poss(turn\_on(Pmp), S_0)$). Thus, $Poss$ determines the subset of the situation tree consisting of situations that are possible in the world. Finally, those properties or relations whose truth value can change from situation to situation are referred to as *fluents*. For example, the property that the pump is on in situation s could be represented by the fluent $on(Pmp, s)$. The situation calculus language we employ in this paper is restricted to primitive, determinate actions. For the present, our language does not include a representation of time or concurrency.

The Representation Scheme Our representation scheme automatically integrates a set of state constraints, such as the ones found in a typical model-based reasoning system description, SD [2] with a situation calculus theory of action to provide a compiled representation scheme. We sketch the integration procedure in sufficient detail to be replicated. We illustrate it in terms of an example of a power plant feedwater system.

The system consists of three potentially malfunctioning components: a power supply (Pwr); a pump (Pmp); and a boiler (Blr). The power supply provides power to

both the pump and the boiler. The pump fills the header with water (wtr_enter_head), which in turn provides water to the boiler, producing steam. Alternately, the header can be filled manually (Man_Fill). To make the example more interesting, we take liberty with the functioning of the actual system and assume that once water is entering the header, a siphon is created. Water will only stop entering the header when the siphon is stopped. The system also contains lights and an alarm, and it contains people. The plant is occupied at all times unless it is explicitly evacuated. Finally we have stipulated certain components of the plant as vital. Such components should not be turned off in the event of an emergency.



Power Plant Feedwater System

This system is typically axiomatized in terms of a set of **state constraints**. The following is a representative subset.[3] All formulae are universally quantified with maximum scope, unless otherwise noted.

$$\neg AB(Pwr) \wedge \neg AB(Pmp) \wedge on(Pmp) \supset wtr\_enters\_head$$
$$on(Man\_Fill) \supset wtr\_enters\_head$$
$$\neg wtr\_enters\_head \wedge on(Blr) \supset on(Alarm)$$
$$AB(Blr) \supset on(Alarm)$$
$$\ldots$$
$$\neg(on(Pmp) \wedge on(Man\_Fill))$$
$$Pwr \neq Pmp \neq Blr \neq Aux\_Pwr \neq Alarm \neq Man\_Fill$$

We also have a situation calculus action theory. One component of our theory of action is a set of **effect axioms** that describe the effects on our power plant of actions performed by the system, a human or nature. The effect axioms take the following form:

$$Poss(a, s) \wedge \text{conditions} \supset fluent(\boldsymbol{x}, do(a, s)).$$

Effect axioms state that if $Poss(a, s)$, i.e. it is possible to perform action a in situation s, and some conditions are true, then *fluent* will be true in the situation resulting from doing action a in situation s, i.e. the situation $do(a, s)$. The following are typical effect axioms.

$$Poss(a, s) \wedge a = turn\_on(Pmp) \supset on(Pmp, do(a, s))$$
$$Poss(a, s) \wedge a = blr\_blow \supset AB(Blr, do(a, s))$$

[3] Note that for simplicity, this particular set of state constraints violates the no-function-in-structure philosophy. This characteristic is not in any way essential to our representation.

In addition to effect axioms our theory also has a set of **necessary conditions for actions** which are of the following general form:

$$Poss(a, s) \supset \text{nec\_conditions}$$

These axioms say that if it is possible to perform action a in situation s then certain conditions (so-called nec_conditions) must hold in that situation. The following are typical necessary conditions for actions.

$$Poss(blr\_blow, s) \supset on(Blr, s)$$
$$Poss(blr\_fix, s) \supset \neg on(Blr, s)$$

We now have axioms describing the constraints on the system state, and also axioms describing the actions that affect system state. Unfortunately, these axioms collectively yield unintended interpretations. That is, there are unintended (semantic) models of this theory. This happens because there are several assumptions that we hold about the theory that have not been made explicit. In particular,

Completeness Assumption: we assume that the axiomatizer has done his/her job properly and that the state constraints, effect axioms and necessary conditions for actions capture all the elements that can affect our system.

Causal Structure: we assume a particular causal structure that lets us interpret how the actions interact with our state constraints, i.e. how effects are propagated through the system, and what state constraints preclude an action from being performed. The causal structure must be acyclic.

We make these assumptions explicit and compile our assumptions, state constraints and theory of action into a final model-based representation. The compilation process is semantically justified and fully described in [10]. The resulting example axiomatization is provided below. We will refer to this collection of axioms as a **situation calculus domain axiomatization** and together with foundational axioms of the situation calculus, Σ [14] they form a situation calculus theory, $\mathcal{D}$.

- successor state axioms, $\mathcal{D}_{SS}$,
- action precondition axioms, $\mathcal{D}_{ap}$,
- axioms describing the initial situation, $\mathcal{D}_{S_0}$,
- unique names for actions, $\mathcal{D}_{una}$,
- domain closure axioms for actions, $\mathcal{D}_{dca}$.

The first element of the domain axiomatization after compilation is the set of **successor state axioms**, compiled form the effect axioms and state constraints under the assumptions above. Successor state axioms are of the following general form.

$$\begin{aligned} Poss(a, s) \supset [fluent(do(a, s)) &\equiv \text{an action made it true} \\ &\vee \text{ a state constraint made it true} \\ &\vee \text{ it was already true} \\ &\quad \wedge \text{ neither an action nor a state constraint made it false}] \end{aligned}$$

I.e., if it is possible to perform action a in situation s, then $fluent$ will be true in the resulting situation if and only if an action made it true, a state constraint made it true, or it was already true and neither an action nor a state constraint made it false.

The set of *intermediate* successor state axioms for our example:

$$Poss(a,s) \supset [on(Pmp, do(a,s)) \equiv a = turn\_on(Pmp) \\ \vee (on(Pmp,s) \wedge a \neq turn\_off(Pmp))] \quad (1)$$

$$Poss(a,s) \supset [on(Aux\_Pwr, do(a,s)) \equiv a = turn\_on(Aux\_Pwr) \\ \vee (on(Aux\_Pwr,s) \wedge a \neq turn\_off(Aux\_Pwr))] \quad (2)$$

$$Poss(a,s) \supset [on(Blr, do(a,s)) \equiv a = turn\_on(Blr) \\ \vee (on(Blr,s) \wedge a \neq turn\_off(Blr))] \quad (3)$$

$$Poss(a,s) \supset [on(Alarm, do(a,s)) \equiv a = turn\_on(Alarm) \vee AB(Blr, (do(a,s)) \\ \vee (\neg wtr\_enter\_head(do(a,s)) \wedge on(Blr, do(a,s))) \\ \vee (on(Alarm,s) \wedge a \neq turn\_off(Alarm)] \quad (4)$$

$$Poss(a,s) \supset [AB(Blr, do(a,s)) \equiv a = blr\_blow \vee (AB(Blr,s) \wedge a \neq blr\_fix)] \quad (5)$$

$$Poss(a,s) \supset [AB(Pwr, do(a,s)) \equiv a = pwr\_failure \\ \vee (AB(Pwr,s) \wedge a \neq turn\_on(Aux\_Pwr) \\ \wedge \ a \neq pwr\_fix)] \quad (6)$$

$$Poss(a,s) \supset [AB(Pmp, do(a,s)) \equiv a = pmp\_burn\_out \\ \vee (AB(Pmp,s) \wedge a \neq pmp\_fix)] \quad (7)$$

$$Poss(a,s) \supset [on(Man\_Fill, do(a,s)) \equiv a = turn\_on(Man\_Fill) \\ \vee (on(Man\_Fill,s) \wedge a \neq turn\_off(Man\_Fill))] \quad (8)$$

$$Poss(a,s) \supset [wtr\_enter\_head(do(a,s)) \equiv on(Man\_Fill, do(a,s)) \\ \vee (\neg AB(Pwr, do(a,s)) \wedge \neg AB(Pmp, do(a,s)) \\ \wedge \ on(Pmp, do(a,s))) \\ \vee wtr\_enter\_head(s) \wedge a \neq stop\_siphon] \quad (9)$$

$$Poss(a,s) \supset [lights\_out(do(a,s)) \equiv AB(Pwr, do(a,s)) \\ \wedge \neg on(Aux\_Pwr, do(a,s))] \quad (10)$$

$$Poss(a,s) \supset [steam(do(a,s)) \equiv (wtr\_enter\_head(do(a,s)) \wedge \neg AB(Pwr, do(a,s)) \\ \wedge \ \neg AB(Blr, do(a,s)) \wedge on(Blr, do(a,s)))] \quad (11)$$

$$Poss(a,s) \supset [occupied(do(a,s)) \equiv (occupied(s) \wedge a \neq evacuate)] \quad (12)$$

These are *intermediate* successor state axioms because they can be further compiled by substituting other intermediate successor state axioms for fluents relativized to situation

$do(a, s)$ on the righthand side of the equivalence connective. For example, Axiom (5) can be substituted into Axiom (4).

Additionally, there is a set of **action precondition axioms** that capture the necessary and suffi cient conditions for actions. They are a compilation of the necessary conditions for actions and the state constraints under the assumptions above. They are of the form:

$$Poss(a, s) \equiv \text{ nec\_conditions} \wedge \text{ implicit conditions from state constraints}$$

For example,

$$Poss(blr\_blow, s) \equiv \neg wtr\_enter\_head(s) \wedge on(Blr, s) \quad (13)$$
$$Poss(pmp\_burn\_out, s) \equiv on(Pmp, s) \quad (14)$$
$$Poss(blr\_fix, s) \equiv \neg on(Blr, s) \quad (15)$$
$$Poss(turn\_off(Alarm), s) \equiv$$
$$(wtr\_enter\_head(s) \vee \neg on(Blr, s)) \wedge \neg AB(Blr, s) \quad (16)$$
$$Poss(turn\_on(Man\_Fill), s) \equiv \neg on(Alarm, s) \wedge \neg on(Pmp, s) \quad (17)$$
$$Poss(turn\_on(Pmp), s) \equiv \neg on(Man\_Fill, s) \quad (18)$$
$$Poss(turn\_on(Alarm), s) \equiv true \quad (19)$$
$$\ldots$$

Our axiomatization will specify what is known of the **initial situation** of the world. This will include the truth value of some fluents in the initial situation S_0. E.g.,

$$vital(Pwr) \wedge vital(Aux\_Pwr) \wedge vital(Alarm) \quad (20)$$
$$\neg vital(Pmp) \wedge \neg vital(Blr) \wedge \neg vital(Man\_Fill) \quad (21)$$
$$\neg on(Pmp, S_0) \wedge \neg on(Blr, S_0) \wedge \neg on(Man\_Fill, S_0) \quad (22)$$
$$\neg AB(Pwr, S_0) \wedge \neg AB(Pmp, S_0) \wedge \neg AB(Blr, S_0) \quad (23)$$
$$\neg on(Aux\_Pwr, S_0) \wedge on(Pwr, S_0) \wedge occupied(S_0) \quad (24)$$

It will also include the state constraints relativized to the initial situation. E.g.,

$$\neg AB(Pwr, S_0) \wedge \neg AB(Pmp, S_0) \wedge on(Pmp, S_0) \supset wtr\_enter\_head(S_0) \quad (25)$$
$$on(Man\_Fill, S_0) \supset wtr\_enter\_head(S_0) \quad (26)$$
$$wtr\_enter\_head(S_0) \wedge \neg AB(Pwr, S_0) \wedge \neg AB(Blr, S_0) \wedge on(Blr, S_0)$$
$$\supset steam(S_0) \quad (27)$$
$$\neg wtr\_enter\_head(S_0) \wedge on(Blr, S_0) \supset on(Alarm, S_0) \quad (28)$$
$$AB(Blr, S_0) \supset on(Alarm, S_0) \quad (29)$$
$$\ldots$$

We have demonstrated that the situation calculus provides a suitable representation for the model-based programming models.

Definition 1 (Model). *A model-based programming model, M is a situation calculus domain axiomatization on the situation calculus language $\mathcal{L}$.*

We henceforth refer to the model of our power plant feedwater example as M_{SD}.

2.2 The Program

With the critical model representation in hand, we must now find a suitable representation for our model-based programs. Further, we must find a suitable mechanism for instantiating our model-based program with respect to our models. We argue that the logic programming language, Golog and theorem proving provide a natural formalism for this task. In the subsection to follow, we introduce the Golog logic programming language and its exploitation for model-based programming.

Golog Golog is a high-level logic programming language developed at the University of Toronto (e.g., [4]). Its primary use is for robot programming and to support high-level robot task planning (e.g., [1]), but it has also been used for agent-based programming (e.g., meeting scheduling). Golog provides a set of extralogical *constructs* for assembling *primitive actions* defined in the situation calculus (e.g., $turn\_on(Blr)$ or $stop\_siphon$ in our power plant example) into *macros* that can be viewed as complex actions, and that assemble into a program.

In the context of our model-based representation, we can define a set of macros that is relevant to our domain or to a family of systems in our domain. The instruction set for these macros, the primitive actions, are simply the domain-specific primitive actions of our model-based representation. Hence, the macros or complex actions simply reduce to first-order (and occasionally second-order) formulae in our situation calculus language. The following are examples of Golog statements.

if AB(Pmp) **then** PMP_FIX **endIf**

while $(\exists x).$ON(x) **do** TURN_OFF(x) **endWhile**

proc PREVENTDANGER
  if OCCUPIED **then** EVACUATE **endIf**
endProc

We leave detailed discussion of Golog to [4, 14] and simply describe the constructs for the Golog language. Let δ_1 and δ_2 be complex action expressions and let ϕ and a be so-called **pseudo fluents/actions**, respectively, i.e., a fluent/action in the language of the situation calculus with all its situation arguments suppressed.

primitive action	a
test of truth	$\phi?$
sequence	$(\delta_1; \delta_2)$
nondeterministic choice between actions	$(\delta_1 \mid \delta_2)$
nondeterministic choice of arguments	$\pi x.\delta$
nondeterministic iteration	δ^*
conditional	**if** ϕ **then** δ_1 **else** δ_2 **endIf**
loop	**while** ϕ **do** δ **endWhile**
procedure	**proc** $P(v)$ δ **endProc**

A Golog **program** can in turn be comprised of a combination of procedures.

Golog also defines the abbreviation $Do(\delta, s, s')$. It says that $Do(\delta, s, s')$ holds whenever s' is a terminating situation following the execution of complex action δ, starting in situation s. Under Do, each of the programming constructs listed above is simply a macro, equivalent to a situation calculus formula.

Do is defined for each complex action construct. Three are defined below.

$$Do(a, s, s') \doteq Poss(a[s], s) \wedge s' = do(a[s], s)^4$$
$$Do([\delta_1; \delta_2], s, s') \doteq (\exists s^*).(Do(\delta_1, s, s^*) \wedge Do(\delta_2, s^*, s'))$$
$$Do((\pi x)\delta(x), s, s') \doteq (\exists x).Do(\delta(x), s, s')$$

Definitions of the rest of the complex actions can be found in [4] but their meaning should be apparent from the examples below. Before returning to our example, we define what we mean by a model-based program.

Definition 2 (Model-Based Program, δ for model M). *Given a model M in situation calculus language $\mathcal{L}$, δ is a model-based program for model M iff δ is a Golog program that only mentions pseudo actions and pseudo fluents drawn from $\mathcal{L}$.*

We begin by defining a rather simple looking procedure to illustrate the constructs in our language and to illustrate the range of procedures Golog can instantiate with respect to the example model, M_{SD}.

proc SHUTDOWN
  $\forall(x)[\text{VITAL}(x) \vee \text{OFF}(x)]? \mid$
    $(\pi x)[[\text{ON}(x) \wedge \neg \text{VITAL}(x)]?;\ \text{TURNOFF}(x)];$SHUTDOWN
endProc

The procedure SHUTDOWN directs the agent to turn off everything that isn't vital. If it is not the case that either everything is off or else it is vital, then pick a random thing that is on and that is not vital, turn it off and repeat the procedure until everything is either off or else it is vital.

From the simple procedures defined above, we can define the following model-based program that dictates a procedure for addressing an abnormal boiler.

if AB(Blr) **then**
  PREVENTDANGER; SHUTDOWN; BLR_FIX; RESTART[5]
end if (30)

This program on its own is very simple and seems uninteresting since it exploits little domain knowledge and thus doesn't capture many of the idiosyncrasies of the system. Instead, it illustrates the beauty of model-based programming. By using nondeterministic choice, the program need not stipulate which component to turn off first, but if there is a physical requirement to turn one component off before another, then

[4] Notation: $a[s]$ denotes the restoration of the situation arguments to any functional fluents mentioned by the action term a.

[5] Procedure not defined here.

it will be dictated in the model, M of the specific system, and when the model-based program is instantiated, M will ensure that the instantiation of the program enforces this ordering. This use of nondeterminism and exploitation of the model makes the program reusable for multiple different devices without the need to rewrite the program. It also saves the engineer/programmer from being mired in the details of the physical constraints of a potentially complex specific system.

It is important to observe that model-based programs are not programs in the conventional sense. While they have the complex structure of programs, including loops, if-then-else statements etc., they differ in that they are not necessarily deterministic. As such they run the gamut from playing the role of a procedurally specified plan sketch that helps to constrain the search space required in planning, to the other extreme where the model-based program provides a deterministic sequence of actions, much in the way a traditional program might. Unfortunately, planning is hard, particularly in cases where we have incomplete knowledge. Computationally, in the worst-case, a model-based program will further constrain the search space, helping the search engines hone in on a suitable sequence of actions to achieve the objective of the program. In the best place, it will dictate a unique sequence of actions.

Indeed, what makes Golog ideal for model-based programming, is how Golog programs are instantiated with respect to a model.

Definition 3 (Model-Based Program Instance, A). *$\boldsymbol{A}$ is a model-based program instance of model M and model-based program δ iff $\boldsymbol{A}$ is a sequence of actions $[a_1, \ldots, a_m]$ such that*

$$M \models Do(\delta, S_0, do([a1, \ldots, a_m], S_0))^6.$$

Recall that the program itself is simply a macro for one or more situation calculus formulae. Hence, generation of a program instance can be achieved by theorem proving, in particular, by trying to prove $\exists s'.Do(\delta, S_0, s')$ from model M. The sequence of actions, $[a1, \ldots, a_m]$ constituting the program instance can be extracted from the binding for s' in the proof. We can see that in this context, the instantiation of a model-based program is related to deductive plan synthesis [3].

Returning to our example, instantiating the model-based program (30) with respect to our example model M_{SD}, which includes some constraints on the initial situation S_0 as defined in Axioms (20)–(24), terminates at the situation $do(evacuate, S_0)$. Consequently, the model-based program instance is composed of the single action $evacuate$. (All the other components of the system are off in the initial situation.) If the initial situation were changed so that all components that could be on at the same time were on, the proof of the program might return the terminating situation

$$do(turn\_off(Pmp), do(turn\_off(Blr), do(turn\_off(Alarm), do(evacuate, S_0))))$$

thus yielding the model-based program instance

$$evacuate;\ turn\_off(Alarm);\ turn\_off(Blr);\ turn\_off(Pmp).$$

To illustrate the power of Golog as a model-based programming language, imagine that our system is more complex than the one described by M_{SD}, that the pump must

[6] Notation: $do([a1, \ldots, a_m], S_0)$ abbreviates $do(a_m, (do(a_{m-1}, \ldots, (do(a_1, S_0)))))$.

be turned off after the boiler, and that before the boiler is turned off that there are valves that must be turned off. If this knowledge is contained in the model M_{SD2}, then this same simple model-based program, (30) is still applicable, but its instantiation will be different. In particular, to instantiate this model-based program, the theorem prover will pick a random nonvital component to turn off, but the preconditions to turn off that component may not be true, if so it will pick another, and another until it finally finds the correct sequence of actions that constitutes a proof, and hence a legal action sequence.

In this instance, an alternative to SHUTDOWN would be to exploit the knowledge of an expert familiar with the device, and to write a device-specific shutdown procedure, along the lines of the following, that captures at least some of this device-specific procedural knowledge.

proc NEWSHUTDOWN
SHUTVALVES; TURNOFF(Blr); TURNOFF(Pmp); TURNOFF($Alarm$)
endProc

proc SHUTVALVES
$\forall(x)$[VALVE(x) $\supset$ OFF(x)]? |
(πx)[[VALVE(x) $\wedge$ $\neg$ ON(x)]?; TURNOFF(x)]; SHUTVALVES
endProc

Indeed, in this particular example, writing such a program is viable, and NEWSHUTDOWN captures the expertise of the expert and in so doing, makes the the model-based instantiation process more efficient. Nevertheless, with a complex physical system comprised of hundreds of complex interacting components, correct sequencing of a shutdown procedure may be better left to a theorem prover following the complex constraints dictated in the model, rather than expecting a control engineer to recall all the complex interdependencies of the system.

This last example serves to illustrate that model-based programs can reside along a continuum from being underconstrained articulations of the goal of a task, to being a predetermined sequence of actions for achieving that goal. SHUTDOWN is situated closer to the goal end of the spectrum, whereas NEWSHUTDOWN is closer towards a predetermined sequence of actions.

3 Proving Properties of Programs

It is often desirable to be able to enforce and/or prove certain formal properties of programs. In our model-based programming paradigm, we may wish to verify properties of a model-based program we have written or of a program instance we have generated. We may also wish to experiment with the behavior of our model-based program by modifying aspects of our model M and seeing what effect it has on program properties. A special case of this, is modifying the initial situation S_0. Finally, rather than verifying properties, we may wish to actually generate program instances which enforce certain

properties. Since our model-based programs are simply macros for logical expressions, our programming paradigm immediately lends itself to this task.

Recall that model-based programs are generally written as generic procedures for *classes* of devices. Hence, an important first property to prove is that a program instance actually *exists* for a particular model-based program and specific device model. This proposition also shows that the program terminates [4].

Proposition 1 (Program Instance Existence). *A program instance exists for model-based program δ and model M iff*

$$M \models \exists s.Do(\delta, S_0, s).$$

Another interesting property is safety. Engineers who write control procedures often wish to verify that the trajectories generated by their control procedures do not pass through unsafe states, i.e., states where some safety property P does not hold.

Proposition 2 (Program Instance Safety). *Let $P(s)$ be a first-order formula representing the safety property. A program instance, $\boldsymbol{A} = [a_1, \ldots, a_m]$ of model-based program δ and model M enforces safety property $P(s)$ iff*

$$M \models Do(\delta, S_0, do(\boldsymbol{A}, S_0)) \supset P(\boldsymbol{A}, S_0).^7$$

By a simple variation on the above proposition, we can prove several stronger safety properties. For example, we can prove that a model-based program enforces the safety property for every potential program instance.

Proposition 3 (Program Safety). *Let $P(s)$ be a first-order formula representing the safety property. A model-based program, δ and model M enforce safety property $P(s)$ iff there is no situation s such that*

$$M \models \exists s.Do(\delta, S_0, s) \supset \neg P(\boldsymbol{\alpha}, S_0),$$

where for each situation variable $s = do([\alpha_1, \ldots, \alpha_n], S_0)$, $\boldsymbol{\alpha} = [\alpha_1, \ldots, \alpha_n]$.

A final property we wish to examine is goal achievement. Since our model-based programs are designed with some task in mind, we may wish to prove that when the program has terminated execution, it will have achieved the desired goal.

Proposition 4 (Program Instance Goal Achievement). *Let $G(s)$ be a first-order formula representing the goal of model-based program δ. A program instance, $\boldsymbol{A} = [a_1, \ldots, a_m]$ of model-based program, δ and model M achieves the goal $G(s)$ iff*

$$M \models Do(\delta, S_0, do(\boldsymbol{A}, S_0)) \supset G(do(\boldsymbol{A}, S_0)).$$

[7] Notation: $do(\boldsymbol{A}, S_0)$ is an abbreviation for $do(a_m, (do(a_{m-1}, \ldots, (do(a_1, S_0)))))$. $P(\boldsymbol{A}, S_0)$ is an abbreviation for $P(S_0) \wedge P(do(a_1, S_0)) \wedge \ldots \wedge P(do(\boldsymbol{A}, S_0))$.

There are many variants on these and other propositions, regarding properties of programs. For example, up until now, we have assumed that we have a fixed initial situation S_0, whose state is captured in our model, M. We can strengthen many of the above propositions by rejecting this assumption and proving Propositions 1, 3 for *any* initial situation. This can be done by replacing S_0 by initial situation variable s_0 and by quantifying, not only over s, but universally quantifying over s_0. Clearly, many programs will not enable the proof of properties for all initial situations, but the associated propositions still hold.

Finally, we can exploit automated reasoning techniques to prove some of these properties. In particular, for procedure and while-loop free programs, we can exploit regression [17] followed by theorem proving in the initial situation, as we do to prove other queries. We leave discussion of this topic to another paper.

4 Related Work

The work presented in this paper is related to several different research areas. In particular, this research is related in spirit only to work on plan sketches such as [11]. In contrast, plan sketches are instantiated through hierarchical substitution. Further, plan sketches generally don't exploit the procedural programming language constructs found in our model-based programming language. Model-based programming is also related to various types of program synthesis and model-based software reuse (e.g., [5, 15, 16]) and to model-based generation of decision trees (e.g., [13]). A subtle distinction is that whereas deductive program synthesis uses deductive machinery to synthesize a program from a specification, model-based programming *starts* with a program, and uses models and deductive machinery to simply fill in some details. Finally, model-based programming is related to planning and in particular to deductive plan synthesis (e.g., [3]). A Golog program, despite its if-then-else's and while loops, can be viewed as extra formulae that are added to the domain axiomatization, including formulae that define the terminating (or goal) situation. In so doing, these formulae reduce the search space required to search for or plan a sequence of actions.

Needless to say, model-based programming is intimately related to cognitive robotics, agent-based programming, and robot programming, particularly in Golog. This work drew heavily from the research on Golog. A major distinction in our work has been the challenge of dealing with large numbers of state constraints inherent to the representation of complex physical systems, and the desire to prove certain properties of our programs. In the first regard, our work is related to ongoing work at NASA on immobots [18], and in particular to research charged with developing a model-based executive.

Finally, this work is related to controller synthesis and controller programming from the engineering community. Comments on the distinction between model-based programming and program synthesis also hold for controller synthesis. With respect to controller programming, typical controller programming languages do not separate control from models. Hence, programs are system specific and not model based. As a consequence they are harder to write, much more brittle, and are not generally amenable to reuse across classes of devices.

5 Summary and Discussion

The main contribution of this paper was to propose and provide a capability for programming devices and web agents. Specifically: we envisaged the concept of model-based programming; proposed a representation and compilation procedure to create suitable models of physical systems in the situation calculus; proposed and demonstrated the effectiveness of Golog for expressing model-based programs themselves, and theorem proving as a model-based program instantiation mechanism. We also provided a set of propositions that characterized interesting properties of programs that could be verified or enforced within our model-based programming framework via regression and theorem proving.

The merits of model-based programming come from the exploitation of models of system behavior and from the separation of those models from high-level procedural knowledge about how to perform a task. Model-based programs are written at a sufficiently high level of abstraction that they are very amenable to reuse over classes of devices. Also, they are easier to write than traditional control programs, ridding the engineer/programmer of keeping track of the potentially complex details of a system design, with all its subcomponent interactions. Further, because of the logical foundations of model-based programming, important properties of model-based programs such as safety, program existence and goal achievement can be verified, and/or simply enforced in the generation of program instances.

There are several weaknesses to our approach at this time, The first is inherent in Golog – not all complex actions comprising our Golog programming language are first-order definable. Hence, in its general form, our model-based programming language is second order. However, as observed by [4] and experienced by the authors, first order is adequate for most purposes. The second problem is that the Prolog implementation of Golog relies on a closed-world assumption (CWA) which has suited our purposes, but is not a valid assumption in the general case. Finally, not all physical system behavior can be expressed as logical state constraints. This can be addressed by extending our model representation language to include ODE's (e.g., [12]).

6 Acknowledgements

We gratefully acknowledge the Cognitive Robotics Group at the University of Toronto for work on the development and implementation of Golog, and related research in the situation calculus. We also acknowledge NASA Grant NAG2-1337 and DARPA Grant N66001-97-C-8554-P00004 for partial funding of this work.

References

1. W. Burgard, A.B. Cremers, D. Fox, D. Haehnel, G. Lakemeyer, D. Schulz, W. Steiner, and S. Thrun. The interactive museum tour-guide robot. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98)*, pages 11–18, 1998.
2. J. de Kleer, A.K. Mackworth, and R. Reiter. Characterizing diagnoses and systems. *Artificial Intelligence*, 56(2–3):197–222, 1992.

3. C. C. Green. Theorem proving by resolution as a basis for question-answering systems. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*, pages 183–205. American Elsevier, New York, 1969.
4. H. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R. Scherl. GOLOG: A logic programming language for dynamic domains. *The Journal of Logic Programming*, 31:59–84, 1997.
5. Z. Manna and R. Waldinger. How to Clear a Block: A Theory of Plans. *Journal of Automated Reasoning*, 3:343–377, 1987.
6. J. McCarthy. Programs with common sense. In Marvin Minsky, editor, *Semantic Information Processing*, chapter 7, pages 403–418. The MIT Press, 1968.
7. S. McIlraith. Representing actions and state constraints in model-based diagnosis. In *Proceedings of the Fourteenth National Conference on Artifi cial Intelligence (AAAI-97)*, pages 43–49, 1997.
8. S. McIlraith. *Towards a Formal Account of Diagnostic Problem Solving*. PhD thesis, Department of Computer Science, University of Toronto, Toronto, Ontario, Canada, 1997.
9. S. McIlraith. Explanatory diagnosis: Conjecturing actions to explain observations. In *Proceedings of the Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR'98)*, pages 167–177, 1998.
10. S. McIlraith. A closed-form solution to the ramification problem (sometimes). *Artifi cial Intelligence*, 116(1–2):87–121, 2000.
11. K. Myers. Abductive completion of plan sketches. In *Proceedings of the Fourteenth National Conference on Artifi cial Intelligence (AAAI-97)*, pages 687–693, 1997.
12. J. Pinto. *Temporal Reasoning in the Situation Calculus.* PhD thesis, Department of Computer Science, University of Toronto, Toronto, Ontario, Canada, 1994.
13. C. Price, M. Wilson, J. Timmis, and C.Cain. Generating fault trees from fmea. In *Proceedings of the Seventh International Workshop on Principles of Diagnosis*, pages 183–190, 1996.
14. R. Reiter. *Knowledge in Action: Logical Foundations for Describing and Implementing Dynamical Systems*. 2000. In preparation. Draft available at http://www.cs.toronto.edu/~cogrobo/.
15. D. Smith and C. Green. Towards Practical Application of Software Synthesis. In *Proceedings of FMSP'96, the First Workshop on Formal Methods in Software Practice*, pages 31–39, San Diego, CA, January 1996.
16. M. Stickel, R. Waldinger, M. Lowry, T. Pressburger, and I. Underwood. Deductive composition of astronomical software from subroutine libraries. In *Proceedings of the 12th Conference on Automated Deduction*, 1994.
17. R. Waldinger. Achieving several goals simultaneously. In E. Elcock and D. Michie, editors, *Machine Intelligence 8*, pages 94–136. Ellis Horwood, Edinburgh, Scotland, 1977.
18. B. Williams and P. Nayak. Immobile robotics: AI in the new millenium. *AI Magazine*, pages 16–35, 1996.