Model-Based Programming using Golog and the Situation Calculus

Sheila A. McIlraith
Knowledge Systems Laboratory
Stanford University
Stanford CA 94025
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 new capability for model-based reasoning systems, 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 system-specific details. Model-based programs must be instantiated in the context of a model of a specific system/device and state of the world. The instantiated programs are simply sequences of actions, which can be performed by an appropriate agent to control the behavior of the system. The separation of control and model enables reuse of model-based programs for devices 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

Much of the research in model-based reasoning about complex physical systems has aimed to combine a task-independent, declarative representation of the behaviour of some physical system, such as a circuit or an electromechanical device, with a set of task-specific reasoning mechanisms (e.g., abductive reasoning, consistency-based reasoning) in order to perform a particular task such as diagnosis. The models[1] often capture the behavior and structure of the physical system and are commonly represented as a set of state constraints.

One virtue of the model-based approach is ease of modification and reuse. Unlike its precursor, the expert system, model-based reasoning systems enable a model of the device to be created once for multiple applications, and since the representations are declarative, they are easily modified. An arguable shortcoming of typical model-based reasoning systems is that their declarative representations do not enable easy representation of the procedural knowledge and heuristics of experts on *how* to perform a particular task.

In this paper, we are motivated by a growing class of physical systems that have received less attention from the model-based reasoning community – the class of complex physical systems that are controlled by a human operator, or by an embedded controller. The question we pose ourselves is: *How can we exploit rich declarative models to facilitate programming complex physical systems*, maintaining the virtues of a model-based approach, while addressing its shortcomings with respect to representing procedural knowledge. We answer this question by integrating research from robot programming and reasoning about action with research in model-based reasoning about complex physical systems.

The main contribution of this paper is to provide a new capability to the model-based reasoning community – *model-based programming*[2] Model-based programs are reusable high-level programs that capture the procedural knowledge of how to accomplish a task, without specifying all the system-specific details. They are called model-based because, on their own, they are too abstract to be executed. They must be instantiated in the context of a model of a specific system/device and state of the world. The instantiated programs are simply sequences of actions, which can subsequently be performed by an appropriate agent to control the behavior of the system. This separation of control and model within model-based programming enables reuse of model-based programs for devices whose configuration and hence models changes as the result of e.g., device replacement, redesign, reconfiguration or device component failure.

To position our model-based programming work, it is also worthwhile to discuss what model-based programming is *not*. Model-based programming is not deductive program synthesis (e.g., (Smith & Green 1996), (Manna & Waldinger 1987)), nor is it model-based program synthesis from software reuse libraries (e.g., (Stickel *et al.* 1994)). Whereas deductive program synthesis uses deductive machinery to synthesize a program from a specification, model-based pro-

[1] Throughout this paper, the term *model* is used in the model-based reasoning or engineering sense, not in the semantic sense, unless otherwise noted.

[2] The term *model-based programming* did not originate with us. E.g., (Williams & Nayak 1996), (Stickel *et al.* 1994), etc. each use the term with different meanings from ours.

gramming *starts* with a program, and uses models and deductive machinery to simply fill in some details. Model-based programming is also not the derivation of decision trees from models (e.g., (Price *et al.* 1996)). One can view the generation of decision trees from models as inducing some sort of procedural knowledge relating to a task, but these "procedures" are system induced as opposed to user-defined; they are specific to the one model used to generate them; and they lacks the structure provided by model-based programming language constructs. Finally, model-based programming is not planning. There is some relationship to planning which we will discuss in this paper, but a model-based program is truly a program complete with typical program structure such as while loops and if-then-else's.

In (McIlraith 1998) we presented some very preliminary thoughts on using the situation calculus to develop generic procedures for model-based computing. Here we provide a much richer development of the related idea of model-based programming including discussion of how to verify properties of programs within our formalism. In this paper, we argue that the situation calculus and the logic programming Golog together provide a natural formalism for model-based programming. We take as our starting point two separate entities: a set of state constraints in first-order logic, that can describe the structure and behavior of a physical system; and a set of actions. In Section 2.1 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. With a representation for our models in hand, Section 2.2 introduces the notion of a model-based program, shows how to exploit Golog to specify model-based programs, and shows how to generate program instances from the program and the model using deductive machinery. In control applications, it is often desirable to be able to prove properties of programs. In Section 3 we show how the logical formalism underlying model-based programming enables verification of properties such as safety, program existence, and goal achievement. We conclude with a brief discussion of experimentation, related work, discussion and summary.

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 that affect it, and the state of the system. The model dictates the language for the program.

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 find a suitable representation for our models. In this section we demonstrate that the situation calculus will provide a suitable language for this task. 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 (McIlraith 1997), 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.

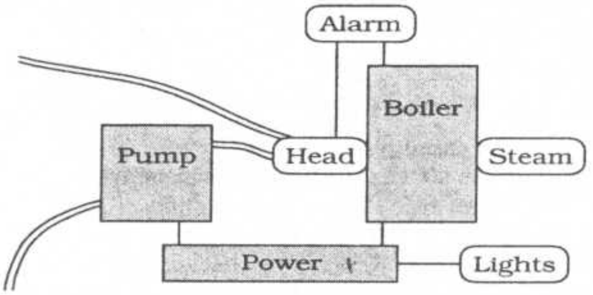
2.1.1 The Situation Calculus

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 $\mathcal{D}$ for everything else, including domain objects ((Lin & Reiter 1994), (Reiter 1998)). 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.

2.1.2 The Representation Scheme

Our representation scheme automatically integrates a set of state constraints, such as the ones found in a typical model-based diagnosis system description, SD (de Kleer, Mackworth, & Reiter 1992) 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]

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

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 general form:

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

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.

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

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

$$Poss(a, s) \supset nec\_conditions$$

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.

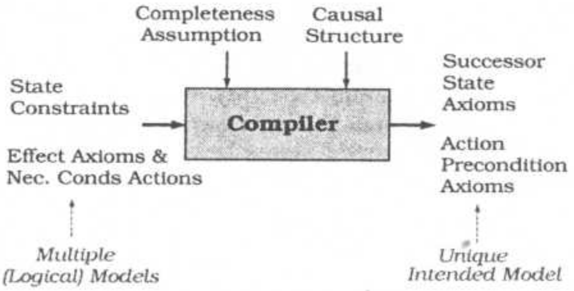
$$Poss(blr\_blow, s) \supset on(Blr, s)$$
$$Poss(blr\_fix, s) \supset \neg on(Blr, s)$$

We now have axioms describing the constraints on the system state, and also axioms describing the actions that affect system state. Unfortunately, these axioms on their own 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 further described in (McIlraith 1997).



The Compilation Process

The resulting example axiomatization is provided below. We will refer to this collection of axioms as a **situation calculus domain axiomatization**. It comprises:

- successor state axioms,
- action precondition axioms,
- axioms describing the initial situation, S_0,
- unique names for actions and domain closure axioms for actions (not included), and
- the foundational axioms of the situation calculus which are domain independent (not included) (Reiter 1998).

The first element of the domain axiomatization after compilation is the set of **successor state axioms**. Successor state axioms are of the following general form.

$Poss(a, s) \supset [fluent(do(a, s)) \equiv$
an action made it true
$\vee$ *a state constraint made it true*
$\vee$ *it was already true*
$\wedge$ *neither an action nor a state constraint made it false*]

[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.

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.

Here is the complete set of successor state axioms for our example.

$$\begin{aligned} &Poss(a,s) \supset [on(Pmp, do(a,s)) \equiv \\ &\quad a = \mathit{turn\_on}(Pmp) \\ &\quad \vee (on(Pmp,s) \wedge a \neq \mathit{turn\_off}(Pmp))] \end{aligned} \tag{1}$$

$$\begin{aligned} &Poss(a,s) \supset [on(Aux\_Pwr, do(a,s)) \equiv \\ &\quad a = \mathit{turn\_on}(Aux\_Pwr) \\ &\quad \vee (on(Aux\_Pwr,s) \wedge a \neq \mathit{turn\_off}(Aux\_Pwr))] \end{aligned} \tag{2}$$

$$\begin{aligned} &Poss(a,s) \supset [on(Blr, do(a,s)) \equiv \\ &\quad a = \mathit{turn\_on}(Blr) \\ &\quad \vee (on(Blr,s) \wedge a \neq \mathit{turn\_off}(Blr))] \end{aligned} \tag{3}$$

$$\begin{aligned} &Poss(a,s) \supset [on(Alarm, do(a,s)) \equiv \\ &\quad a = \mathit{turn\_on}(Alarm) \vee AB(Blr,(do(a,s)) \\ &\quad \vee (\neg \mathit{wtr\_enter\_head}(do(a,s)) \wedge on(Blr, do(a,s))) \\ &\quad \vee (on(Alarm,s) \wedge a \neq \mathit{turn\_off}(Alarm)] \end{aligned} \tag{4}$$

$$\begin{aligned} &Poss(a,s) \supset [AB(Blr, do(a,s)) \equiv \\ &\quad a = \mathit{blr\_blow} \vee (AB(Blr,s) \wedge a \neq \mathit{blr\_fix})] \end{aligned} \tag{5}$$

$$\begin{aligned} &Poss(a,s) \supset [AB(Pwr, do(a,s)) \equiv a = \mathit{pwr\_failure} \\ &\quad \vee (AB(Pwr,s) \wedge a \neq \mathit{turn\_on}(Aux\_Pwr) \\ &\quad\quad \wedge\ a \neq \mathit{pwr\_fix})] \end{aligned} \tag{6}$$

$$\begin{aligned} &Poss(a,s) \supset [AB(Pmp, do(a,s)) \equiv \\ &\quad a = \mathit{pmp\_burn\_out} \\ &\quad \vee (AB(Pmp,s) \wedge a \neq \mathit{pmp\_fix})] \end{aligned} \tag{7}$$

$$\begin{aligned} &Poss(a,s) \supset [on(Man\_Fill, do(a,s)) \equiv \\ &\quad a = \mathit{turn\_on}(Man\_Fill) \\ &\quad \vee (on(Man\_Fill,s) \wedge a \neq \mathit{turn\_off}(Man\_Fill))] \end{aligned} \tag{8}$$

$$\begin{aligned} &Poss(a,s) \supset [\mathit{wtr\_enter\_head}(do(a,s)) \equiv \\ &\quad on(Man\_Fill, do(a,s)) \\ &\quad \vee (\neg AB(Pwr, do(a,s)) \wedge \neg AB(Pmp, do(a,s)) \\ &\quad\quad \wedge\ on(Pmp, do(a,s))) \\ &\quad \vee \mathit{wtr\_enter\_head}(s) \wedge a \neq \mathit{stop\_siphon}] \end{aligned} \tag{9}$$

$$\begin{aligned} &Poss(a,s) \supset [\mathit{lights\_out}(do(a,s)) \equiv \\ &\quad AB(Pwr, do(a,s)) \wedge \neg on(Aux\_Pwr, do(a,s))] \end{aligned} \tag{10}$$

$$\begin{aligned} &Poss(a,s) \supset [\mathit{steam}(do(a,s)) \equiv \\ &\quad (\mathit{wtr\_enter\_head}(do(a,s)) \wedge \neg AB(Pwr, do(a,s)) \\ &\quad\quad \wedge\ \neg AB(Blr, do(a,s)) \wedge on(Blr, do(a,s)))] \end{aligned} \tag{11}$$

$$\begin{aligned} &Poss(a,s) \supset [\mathit{occupied}(do(a,s)) \equiv \\ &\quad (\mathit{occupied}(s) \wedge a \neq \mathit{evacuate})] \end{aligned} \tag{12}$$

Observe that these successor state axioms can be further compiled by substituting successor state axioms for fluents relativized to situation $do(a, s)$. For example, among other axioms, Axiom (5) could be substituted into Axiom (4).

In addition to the successor state axioms there is a set of **action precondition axioms** that capture the necessary and sufficient conditions for actions. They are of the form:

$$\begin{aligned} &Poss(a,s) \equiv \mathit{nec\_conditions} \\ &\quad \wedge \textit{ implicit conditions from state constraints} \end{aligned}$$

For example,

$$Poss(\mathit{blr\_blow}, s) \equiv \neg \mathit{wtr\_enter\_head}(s) \wedge on(Blr,s) \tag{13}$$

$$Poss(\mathit{pmp\_burn\_out}, s) \equiv on(Pmp,s) \tag{14}$$

$$Poss(\mathit{blr\_fix}, s) \equiv \neg on(Blr,s) \tag{15}$$

$$\begin{aligned} &Poss(\mathit{turn\_off}(Alarm), s) \equiv \\ &\quad (\mathit{wtr\_enter\_head}(s) \vee \neg on(Blr,s)) \wedge \neg AB(Blr,s) \end{aligned} \tag{16}$$

$$\begin{aligned} &Poss(\mathit{turn\_on}(Man\_Fill), s) \equiv \\ &\quad \neg on(Alarm,s) \wedge \neg on(Pmp,s) \end{aligned} \tag{17}$$

$$Poss(\mathit{turn\_on}(Pmp), s) \equiv \neg on(Man\_Fill,s) \tag{18}$$

$$Poss(\mathit{turn\_on}(Alarm), s) \equiv \mathit{true} \tag{19}$$

$$\ldots \tag{20}$$

Our axiomatization will have some knowledge regarding the **initial situation** of the world. This will include what is known of the truth value of predicates and fluents relativized to S_0, for example:

$$\mathit{vital}(Pwr) \wedge \mathit{vital}(Aux\_Pwr) \wedge \mathit{vital}(Alarm) \tag{21}$$

$$\neg \mathit{vital}(Pmp) \wedge \neg \mathit{vital}(Blr) \wedge \neg \mathit{vital}(Man\_Fill) \tag{22}$$

$$\mathit{occupied}(S_0) \tag{23}$$

$$\neg on(Pmp, S_0) \wedge \neg on(Blr, S_0) \wedge \neg on(Man\_Fill, S_0) \tag{24}$$

$$\neg AB(Pwr, S_0) \wedge \neg AB(Pmp, S_0) \wedge \neg AB(Blr, S_0) \tag{25}$$

$$\neg on(Aux\_Pwr, S_0) \wedge on(Pwr, S_0) \tag{26}$$

It will also include the state constraints relativized to the initial situation. We repeat only a few here for illustration purposes.

$$\begin{aligned} &\neg AB(Pwr, S_0) \wedge \neg AB(Pmp, S_0) \wedge on(Pmp, S_0) \\ &\quad \supset \mathit{wtr\_enter\_head}(S_0) \end{aligned} \tag{27}$$

$$on(Man\_Fill, S_0) \supset \mathit{wtr\_enter\_head}(S_0) \tag{28}$$

$$\begin{aligned} &\mathit{wtr\_enter\_head}(S_0) \wedge \neg AB(Pwr, S_0) \wedge \neg AB(Blr, S_0) \\ &\quad \wedge on(Blr, S_0) \supset \mathit{steam}(S_0) \end{aligned} \tag{29}$$

$$\neg \mathit{wtr\_enter\_head}(S_0) \wedge on(Blr, S_0) \supset on(Alarm, S_0) \tag{30}$$

$$AB(Blr, S_0) \supset on(Alarm, S_0) \tag{31}$$

$$\ldots \tag{32}$$

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.

2.2.1 Golog

Golog is a high-level logic programming language developed at the University of Toronto (Levesque *et al.* 1997). Its primary use is for robot programming and to support high-level robot task planning (e.g., (Burgard *et al.* 1998)), but is 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 $(\exists component)$.ON($component$) do
    TURN_OFF($component$)
endWhile
```

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

We leave detailed discussion of Golog to (Levesque *et al.* 1997) 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

loop	**while** ϕ **do** δ
procedure	**proc** $P(\vec{v})$ δ **end**

A Golog **program** is in turn comprised of a sequence of procedures.

Each of the programming constructs listed above is simply a macro, equivalent to a situation calculus formula. 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. 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.

$$Do(a, s, s') \doteq Poss(a[s], s) \wedge s' = do(a[s], s)\text{[4]}$$

$$Do([\delta_1 ; \delta_2], s, s') \doteq (\exists s^*).(Do(\delta_1, s, s^*) \wedge Do(\delta_2, s^*, s'))$$

$$Do((\pi x)\delta(x), s, s') \doteq (\exists x).Do(\delta(x), s, s')$$

Definitions of the rest of the complex actions can be found in (Levesque *et al.* 1997) 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
    $\forall(x)$[VITAL($x$) $\vee$ OFF($x$)]? |
        $(\pi x)$[[ON($x$) $\wedge$ $\neg$ VITAL($x$)]?; 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                                                    (33)
```

[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.

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 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 deterministic sequence of actions.

Indeed, what is interesting and unique about Golog programs and what makes them ideal for model-based programming, is how they are instantiated with respect to a model.

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

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

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, $[a1, \dots , a_m]$ 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 (Green 1969).

Returning to our example, instantiating the model-based program (33) with respect to our example model M_{SD}, which includes some constraints on the initial situation S_0 as defined in Axioms (21)–(26), 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 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, (33) 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 system, and to write a system-specific shutdown procedure, along the lines of the following, that captures at least some of this system-specific procedural knowledge.

```
proc NEWSHUTDOWN
    SHUTVALVES;
    TURNOFF($Blr$);TURNOFF($Pmp$);TURNOFF($Alarm$)
endProc
proc SHUTVALVES
    $\forall(x)$[VALVE($x$) $\supset$ OFF($x$)]? |
      $(\pi x)$[[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 deterministic program for achieving that goal. SHUTDOWN is situated closer to the goal end of the spectrum, whereas NEWSHUTDOWN is closer towards a deterministic program.

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 properities 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.

An important first property to prove is that a program instance actually *exists* for a particular model-based program and model. This proposition also shows that the program terminates (Levesque *et al.* 1997).

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, $\vec{A} = [a_1, \ldots, a_m]$ of model-based program δ and model M enforces safety property $P(s)$ iff*

$$M \models Do(\delta, S_0, do(\vec{A}, S_0)) \supset P(\vec{A}, S_0).^6$$

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*

$$M \models (\forall s).Do(\delta, S_0, s) \supset P(\vec{\alpha}, S_0),$$

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

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, $\vec{A} = [a_1, \ldots, a_m]$ of model-based program, δ and model M achieves the goal $G(s)$ iff*

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

Again, by a simple variation on the above proposition, we can prove several other properties with respect to goal achievement. For example, we can prove that a model-based program is guaranteed to achieve its goal for every potential program instance.

Proposition 5 (Program Goal Achievement) *Let $G(s)$ be a first-order formula representing the goal of model-based program δ. δ and model M are guaranteed to achieve goal $G(s)$ iff*

$$M \models (\forall s).Do(\delta, S_0, s) \supset G(s).$$

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, 5 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 properities for all initial situations, but the associated propositions still hold.

Finally, exploiting Proposition 2 and Proposition 4, it is trivial to see how we can use theorem proving to generate a model-based program instance $\vec{A}$ that enforces the safety condition and/or achieves the program goal, rather than generating a model-based program instance and then verifying that it enforces/achieves certain properties. The sequence of actions comprising $\vec{A}$ will simply be the bindings for the terminating situation $(do(\vec{A}, S_0))$. This said, we note that if we have used Propositions 3 and 5 to show that all program instances of th model-based program enforce the safety condition and achieve the goal, then there is no need to stipulate these conditions when subsequently generating a program instance.

4 Related Work

The work presented here is related to several different research areas. In particular, this research is related in spirit only to work on plan sketches such as (Myers 1997). 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 in spirit to various types of program synthesis and model-based software reuse (e.g., (Smith & Green 1996), (Manna & Waldinger 1987), (Stickel *et al.* 1994)) and to model-based generation of decision trees (e.g., (Price *et al.* 1996)). We refer the reader back to Section 1 for a discussion of the relationship. As noted in Section 2.2, model-based programming is also related to deductive plan synthesis (e.g., (Green 1969)), since this provides part of the mechanism for program instantiation.

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

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

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 (Williams & Nayak 1996), and in particular to research charged with developing a model-based executive such as has recently been initiated by (Williams & Gupta 1999).

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 amenable to reuse.

5 Summary and Discussion

This paper synthesizes several subfields of AI, exploiting research in robot programming, reasoning about action, and model-based reasoning about physical systems. The particular class of physical systems driving this research is the class of systems that are controlled by an external agent such as a human or an embedded controller. The question we posed to ourselves was how we could exploit the benefits of rich declarative models, which have been used with great success in other aspects of model-based reasoning, to actually *program* physical devices. Clearly model-based programming supports programming a variety of tasks related to the diagnosis, control, maintenace and reconfiguration of physical systems. With the very recent exception of (Williams & Gupta 1999), none of the research on model-based reasoning about physical systems has addressed this issue. In that regard, this work is an important step.

The main contribution of this paper was to propose and provide a new capability for model-based reasoning about physical systems – model-based programming. 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 proposed 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.

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. 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. We hope to address some of these in future research. 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 (Levesque *et al.* 1997) 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. We have experimented with several applications, including a furnace leak test system (Probst 1996). Finally, not all physical system behavior can be expressed as logical state constraints. This is a weakness of our current model representation. We will need to extend our model representation language to include ODE's. We will do so by exploiting related work in the situation calculus (Pinto 1994).

6 Acknowledgements

I would like to acknowledge the Cognitive Robotics Group at the University of Toronto for their work on the development of Golog. I would also like to thank the reviewers of this paper for their excellent comments.

References

Burgard, W.; Cremers, A.; Fox, D.; Haehnel, D.; Lakemeyer, G.; Schulz, D.; Steiner, W.; and Thrun, S. 1998. The interactive museum tour-guide robot. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98)*. To appear.

de Kleer, J.; Mackworth, A.; and Reiter, R. 1992. Characterizing diagnoses and systems. *Artificial Intelligence* 56(2–3):197–222.

Green, C. C. 1969. Theorem proving by resolution as a basis for question-answering systems. In Meltzer, B., and Michie, D., eds., *Machine Intelligence 4*. New York: American Elsevier. 183–205.

Levesque, H.; Reiter, R.; Lesṕerance, Y.; Lin, F.; and Scherl, R. 1997. GOLOG: A logic programming language for dynamic domains. *The Journal of Logic Programming* 31:59–84.

Lin, F., and Reiter, R. 1994. State constraints revisited. *Journal of Logic and Computation* 4(5):655–678. Special Issue on Action and Processes.

Manna, Z., and Waldinger, R. 1987. How to Clear a Block: A Theory of Plans. *Journal of Automated Reasoning* 3:343–377.

McIlraith, S. 1997. A closed-form solution to the ramification problem (sometimes). In *Proceedings of the Workshop on Nonmonotonic Reasoning, Action and Change, Fifteenth International Joint Conference on Artificial Intelligence*.

McIlraith, S. 1998. Towards generic procedures for model-based computing. In *Proceedings of the Ninth International Workshop on Principles of Diagnosis*, 217–224.

Myers, K. 1997. Abductive completion of plan sketches. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-97)*, 687–693.

Pinto, J. 1994. *Temporal Reasoning in the Situation Calculus*. Ph.D. Dissertation, Department of Computer Science, University of Toronto, Toronto, Ontario, Canada. Also published as Technical Report, Dept. of Computer Science, University of Toronto (KRR-TR-94-1), Feb. 1994.

Price, C.; Wilson, M.; Timmis, J.; and C.Cain. 1996. Generating fault trees from fmea. In *Proceedings of the Seventh International Workshop on Principles of Diagnosis*, 183–190.

Probst, S. 1996. *Chemical Process Safety and Operability Analysis using Symbolic Model Checking*. Ph.D. Dissertation, Department of Chemical Engineering, Carnegie Mellon University.

Reiter, R. 1998. *KNOWLEDGE IN ACTION: Logical Foundations for Building Dynamic Systems*. In preparation.

Smith, D., and Green, C. 1996. Towards Practical Application of Software Synthesis. In *Proceedings of FMSP'96, the First Workshop on Formal Methods in Software Practice*, 31–39.

Stickel, M.; Waldinger, R.; Lowry, M.; Pressburger, T.; and Underwood, I. 1994. Deductive composition of astronomical software from subroutine libraries. In *Proceedings of the 12th Conference on Automated Deduction*.

Williams, B., and Gupta, V. 1999. Personal communication. Paper to appear in this proceedings.

Williams, B., and Nayak, P. 1996. Immobile robotics: AI in the new millenium. *AI Magazine* 16–35.