Modeling Complex Systems in the Situation Calculus: A Case Study Using the Dagstuhl Steam Boiler Problem

T. G. Kelley
Department of Computer Science
University of Toronto
Toronto, Canada M5S 1A4
email: tgk@cs.toronto.edu

Abstract

We take advantage of the results of (Rei96) to show that the situation calculus is a powerful and practical modeling language. The paper provides a brief overview of the concurrent temporal situation calculus and how it is used to specify physical behavior. It then presents the Dagstuhl steam boiler problem as an example of a complex physical system of interest in the real world. The problem was the focus of the Dagstuhl meeting, "Methods for Semantics and Specification", whose goal was to develop criteria by which to compare advantages and drawbacks of formal methods for practical applications. The paper presents the situation calculus specification of the focus of the problem, the steam boiler controller. It then discusses the theoretical foundation of a PROLOG technology simulator, which, together with the specification, form an implementation of the controller. The paper concludes with an evaluation of the situation calculus solution to the problem, using the criteria which emanated from the Dagstuhl meeting.

Introduction

The situation calculus language (MH69) has received much attention from the Cognitive Robotics Group at the University of Toronto in recent years. The language is showing tremendous promise as a formal framework for modeling the dynamical worlds encountered in real life.

The challenges facing such a framework are numerous. For example, it must facilitate the representation of time, continuous processes, actions performed by agents with "free will", actions performed by Nature, non-deterministic actions of chance, knowledge-producing actions and the mental state of agents, concurrent or simultaneous actions, etc. The framework must also be conducive to various types of reasoning, including prediction, planning, diagnosis, and hypothetical reasoning. A single formal theory of action and time that satisfies these conditions is the ongoing objective of the Cognitive Robotics Group.

Encouraging progress towards this long-range objective using the situation calculus as the framework has been achieved (Rei91; Pin94; LRL+96; LLL+96; SL93). This paper concentrates specifically on the results of (Rei96) which make it possible to formally model the behavior of physical systems as complex as a steam boiler controller. These physical systems (the toilet of (Kel96) is another example) involve time, continuous processes, and simultaneous natural actions (those dictated by the laws of physics). The situation calculus of (Rei96) provides a knowledge representation framework that is conducive to the specification and simulation of such systems, while explicitly embodying a solution to the frame and qualification problems.

As a specification language, the situation calculus boasts many desirable properties. For example, a situation calculus model of a physical system is a truly logical specification of that system. Hence, items of interest, such as behaviors of parameters, are *logical consequences* of the specification. This feature of the situation calculus is clearly conducive to formal verification. Furthermore, the foundational axioms of the situation calculus provide a firm theoretical foundation for a situation calculus-based PROLOG simulator of situation calculus specifications. The behavioral properties of a situation calculus specification can be obtained automatically by directly executing the specification on the simulator. *Hence, a specification in situation calculus form is also an implementation whose behavioral properties are automatically formally verified against the specification.*

To illustrate that the situation calculus is a practical modeling language, we have formalized the controller specification for the Dagstuhl steam boiler (Abr94). The original text from which the specification is derived was written by LtCol. J. C. Bauer for the Institute for Risk Research of the University of Waterloo, and submitted as a competition problem to be solved by the participants of the International Software Safety Symposium organized by the Institute for Risk Research. The Dagstuhl steam boiler problem, solved in this paper, stems from that original text. The problem was the focus of the Dagstuhl meeting, "Methods for Semantics and Specification", whose goal was to develop criteria by which to compare advan-

tages and drawbacks of formal methods for practical applications. Hence, the problem is ideal for exhibiting the features of the situation calculus. In this paper, I present an evaluation of the situation calculus solution to the problem, using the criteria which emanated from the Dagstuhl meeting (ABL95).

Situation Calculus Ontology

The situation calculus is designed to formalize the behavior of dynamically changing worlds. Intuitively there are two facets to the ontology itself: 1) distinguishing between different courses of action, and 2) determining the state of the world after different courses of action. There are two additional facets of the situation calculus: 3) axioms which specify which courses of action can happen, and 4) axioms which specify the results of courses of action.

Naming Courses of Action

The mechanism for all change in such worlds is one or more agents, perhaps including Nature, performing named, instantaneous, *actions*. *Situations* are histories of concurrent action occurrences, each denoting a different possible evolution of the world. A concurrent action is a set of simultaneous simple actions. The constant symbol S_0 denotes the initial situation in which no actions have yet occurred. Other than S_0, all situations have names of the form, $do(\alpha, \sigma)$, intuitively meaning the result of doing action α in situation σ. Actions are denoted by functions, with time t being the last parameter. For example, consider the situation

$$do(\{stop\_talking(T_2)\},$$
$$do(\{begin\_walking(T_1), begin\_talking(T_1)\}, S_0),$$

which denotes the world history in which an agent begins walking and talking at time T_1, and then stops talking at time T_2. The agent would not be talking in this situation, but she would still be walking. The first concurrent action performed is $\{begin\_walking(T_1), begin\_talking(T_1)\}$, which is the set consisting of two simple actions $begin\_walking(T_1)$, and $begin\_talking(T_1)$. Given any situation, the order of action occurrences is obtained by scanning the situation from right to left. So, this situation denotes a world history corresponding to the action sequence

$$[\{begin\_walking(T_1), begin\_talking(T_1)\},$$
$$\{stop\_talking(T_2)\}].$$

The State of the World

The state of a world resulting from a certain course of action is determined by the values of *fluents*.

Relational fluents are denoted by predicate symbols taking a situation term as their last argument. These relations represent what is true about the world after carrying out the course of action specified by their situation argument. For example, the fluent, $happy(p, s)$ might mean that person p is happy in s. Note that technically a situation is not a state, but a history of action occurrences; so in this context, we should take "p is true *in* s" to mean, "p is true *after carrying out, in order, all and only the actions specified by* s".

Functional fluents are functions whose value varies from situation to situation. For example, $constant\_position(ball, s)$ might denote the real-valued constant position of a ball in situation s.

Continuous processes are represented using functional fluents. The important idea, due to Pinto (Pin94), is that although a continuous process involves continuous change in the values of one or more parameters, the values of the parameters can be modeled by equations which do not change in a particular situation. We say that the *behavior* of the parameter is constant.

Consider a ball's position, which varies with time in s. The $position(ball, s)$ fluent has as its value a function of time. Also consider the function,[1] $val(f, t)$, which takes as arguments a function of time, f, and a time, t, and whose value is the value of f at t. We take $f(t)$ to be an abbreviation for $val(f, t)$, and we could write[2] $position(ball, s)(t) = x_0 - 1/2gt^2$, meaning that the value of the position function of the ball at time t in situation s is $x_0 - 1/2gt^2$. The position function in a certain situation might be defined on the entire real line, as it is in this case, but it is only relevant to the model on some half-open interval: $[start(s), \infty)$, or $[start(s), start(do(c, s)))$ for some c, where $start(do(c, s))$ is defined to be $time(c)$, and $time(c)$ denotes the time at which c occurs.

In the situation calculus, all change must be the result of some action occurrence, and functional fluents are consistent with this. In the case of the ball, although the ball's position varies in s, its position follows a single function in s. The behavior represented by the $position(x, s)$ fluent remains unchanged until some action, perhaps $catch(x, t)$, takes place to change it.

Specifying Which Courses of Action Can Happen

Precondition axioms determine the conditions under which an action is possible. A formalization of a world includes one precondition axiom for each action. For example, the precondition axiom for the $bounce(ball, t)$ action states that the ball bounces if it is falling and it is at the floor.

Consider the function,[3] $\frac{d}{dt}(f)$, which takes a function of time, f, as its argument and has as its value the function which is the time rate of change of f. We

[1] In principle, this function could be axiomatized.

[2] In this paper, lower case Roman characters denote variables. Also, free variables are implicitly universally prenex quantified.

[3] In principle, this function could be axiomatized.

write $\frac{d}{dt}(position(ball, s))(t) < 0$ to say that the value of the time rate of change of the position function of the ball at time t in s is less than 0. In other words, the ball is falling at time t. Hence, the precondition axiom for the $bounce(ball, t)$ action is

$$Poss(bounce(ball, t), s) \equiv t \geq start(s) \wedge position(ball, s)(t) = 0 \wedge \frac{d}{dt}(position(ball, s))(t) < 0 \quad (1)$$

where $Poss(a, s)$ means that action a is possible in situation s.

Natural actions, such as $bounce(ball, t)$, are a special case: when a natural action can possibly occur at time t, it *does* occur, unless some other action, perhaps $catch(ball, t)$, occurs sooner. In worlds where no agent has "free will", all actions are natural. Intuitively, an agent has "free will" if it is impossible to predict what actions the agent will perform, or, at least, *when* the agent will perform some action. Nature is taken to be characterized by all of the scientific laws that scientists strive to know, and to the extent that those laws exist and do not change, Nature has no "free will".

In worlds where all actions are natural, if certain conditions are met, it is possible to simulate the evolution of the world. Simulation is possible when in every situation it is possible to determine whether and when actions will occur. In effect, the world evolves deterministically according to the laws of Nature (e.g. Newton's laws). When this is the case, there is only one legal path through the tree of situations.

The Results of Courses of Action

The ways in which the values of fluents are affected by action occurrences are determined by *successor state axioms*. A formalization of a world includes one successor state axiom for each fluent. The axiom specifies all individual conditions under which the fluent will change, and how the fluent changes under those conditions.

Consider a world where a ball can bounce, and it can be caught, and no other actions can affect it. The successor state axiom for the $position(ball, s)$ fluent would state that if the concurrent action c is possible in s, then the value of $position(ball, do(c, s))$ depends upon what simple actions are in c. If c contains a *bounce* and not a *catch*, the ball's velocity reverses. If c contains a *catch*, the ball's position becomes constant where it is caught. If c contains neither a *bounce* nor a *catch*, the behavior of the ball's position remains unchanged.

The axiom is

$$\begin{aligned} &Poss(c, s) \supset position(ball, do(c, s)) = f \equiv \\ &\quad [(\exists t')bounce(ball, t') \in c \wedge catch(ball, t') \notin c \wedge \\ &\quad (\forall t) f(t) = 0 - \frac{d}{dt}(position(ball, s))(t')t - gt^2] \\ &\quad \vee \\ &\quad [(\exists t')catch(ball, t') \in c \wedge \\ &\quad (\forall t) f(t) = position(ball, s)(t')] \\ &\quad \vee \\ &\quad [(\forall t)bounce(ball, t) \notin c \wedge catch(ball, t) \notin c \wedge \\ &\quad f = position(ball, s)] \end{aligned} \quad (2)$$

where the surface on which the ball bounces is taken to be at position 0, and $g > 0$ is the acceleration due to gravity.

Successor state axioms such as this embody Reiter's (Rei91) solution to the frame problem.

The Dagstuhl Steam Boiler

In this section, we attempt merely to introduce the Dagstuhl steam boiler problem in enough detail for the purposes of this paper, rather than reproduce the specification. See (Abr94) for the complete specification.

The Dagstuhl steam boiler system consists of the following units:

- the steam boiler with a water level which is kept preferably within the normal operating range, but certainly must be kept within the safe range,
- a device to measure the quantity of water in the steam boiler (denoted by *water_level*),
- four pumps which are either off or on to provide the steam boiler with water (we use $pump(i)$ to denote the ith pump),
- one controller to supervise each pump (four in all), reporting on its water flow (we use $pump\_control(i)$ to denote the ith pump controller),
- a device to measure the quantity of steam exiting the steam boiler (denoted by *steam_rate*),
- an operator desk, from which a $STOP$ message can be sent to the controller,
- a message transmission system used for all communication between the controller and the steam boiler components.

We attempt to keep our notation consistent with the original problem description (Abr94). Messages are denoted by functions and constants with upper case names. Mandatory messages must be present in every transmission. There are roughly four classes of messages:

Control messages : The controller sends messages to the physical units to direct their actions, and the physical units send messages to the controller to indicate the actual state of the steam boiler.

The control messages sent by the controller to the physical units are

- $VALVE$: sent in *initialization* mode to request the opening and then the closure of the valve for evacuation of water from the steam-boiler,

- $OPEN\_PUMP(n)$: sent to activate $pump(n)$,
- $CLOSE\_PUMP(n)$: sent to deactivate $pump(n)$.

The control messages sent by the physical units to the controller are

- $PUMP\_STATE(n, b)$: sent to indicate that the state of $pump(n)$ is b, which may be 0 or 1, meaning open or closed, mandatory,
- $PUMP\_CNTL\_STATE(n, b)$: sent to indicate that the flow of water from $pump(n)$ is b, which may be 0 or 1, meaning there is flow or there is no flow, mandatory,
- $LEVEL(v)$: sent to indicate that the level of water in the boiler is v, mandatory,
- $STEAM(v)$: sent to indicate that steam is exiting the boiler at rate v, mandatory.

Fault Detection and Repair messages : When the controller infers that a component is defective (a pump claims to be operating, but there is no water flow, for example), the controller sends a fault detection message. When the a defective component has been repaired, a repair message is sent to the controller.

The controller sends the following fault detection messages:

- $PUMP\_FAIL\_DETN(n)$: sent (until acknowledgement is received) to indicate that the controller has detected the failure of $pump(n)$.
- $PUMP\_CNTL\_FAIL\_DETN(n)$: sent (until acknowledgement is received) to indicate that the controller has detected the failure of $pump\_control(n)$,
- $LEVEL\_FAIL$: sent (until acknowledgement is received) to indicate that the controller has detected the failure of the water level measuring unit,
- $STEAM\_FAIL$: sent (until acknowledgement is received) to indicate that the controller has detected the failure of the unit that measures the rate of steam exiting the boiler.

The controller receives the following repair messages:

- $PUMP\_REPD(n)$: sent (until acknowledgement is received) to indicate that $pump(n)$ has been repaired,
- $PUMP\_CNTL\_REPD(n)$: sent (until acknowledgement is received) to indicate that $pump\_control(n)$ has been repaired,
- $LEVEL\_REPD$: sent (until acknowledgement is received) to indicate that the water level measuring unit has been repaired,
- $STEAM\_REPD$: sent (until acknowledgement is received) to indicate that the unit that measures the rate of steam exiting the boiler has been repaired.

Acknowledgement messages : The physical units send the controller a corresponding acknowledgement message for each fault detection message. For example, $PUMP\_FAIL\_ACK(n)$ is sent to the controller to acknowledge the receipt of a $PUMP\_FAIL\_DETN(n)$ message. Similarly, the controller sends the physical units a corresponding acknowledgement message for each repair message. For example, $PUMP\_REPD\_ACK(n)$ is sent to the physical units to acknowledge a $PUMP\_REPD(n)$ message.

Other messages : There are also messages that have somewhat administrative purposes.

The controller sends the following administrative messages to the physical units:

- $PROGRAM\_READY$: sent (until acknowledgement is received) in *initialization* mode to indicate that the controller is ready to assume control of the steam boiler,
- $MODE(m)$: sent to to indicate that the program's current mode of operation is m (see below for the modes of operation), mandatory,
- $EMERGENCY\_STOP$: sent to immediately transfer control of the steam boiler to the operators.

The physical units send the following administrative messages to the controller:

- $STOP$: when the controller receives this message three times in a row, the program goes into *emergency_stop* mode,
- $STEAM\_BOILER\_WAITING$: sent in *initialization* mode to trigger the start of the program,
- $PHYSICAL\_UNITS\_READY$: sent in *initialization* mode to acknowledge a $PROGRAM\_READY$ message.

A solution to the steam boiler problem is an implementation of a controller program that will keep the water level in the boiler within the normal operating range by receiving and sending messages through the message system. If for some reason the water level threatens to go outside of the safe operating range, the controller should immediately transmit an $EMERGENCY\_STOP$ message, and halt.

The operation of the program is cyclic. Every five seconds, the program receives the incoming messages from the steam boiler components. It then computes the set of messages that should be sent out to the components, and transmits those messages. In each cycle, all messages are assumed to be received or transmitted simultaneously.

Components of the steam boiler can become defective. Defective components determine (for the most part) the operation mode of the controller. The operation modes and their corresponding conditions (mostly) are:

initialization : the steam boiler is not yet operating;

normal : no component is defective;

degraded : any component except the water level monitor is defective;

rescue : the water level monitor is defective, but the steam monitor is not. In *rescue* mode, the controller estimates upper and lower bounds on the water level, based on the dynamics of the boiler. If the controller calculates that either bound will be outside the safe operating range at the next cycle, then the water level is considered to be outside the safe range.

emergency_stop : the water level and steam monitors are both defective, or the water level is threatening to go outside the safe range.

When the program detects that a component is faulty, it sends an appropriate message to the operator desk. After the component is repaired, an appropriate message is sent to the controller to inform it of the repair.

The Situation Calculus Language

The instantiation of McCarthy's (MH69) situation calculus language used in this paper to formalize the steam boiler controller is due to Reiter (Rei96), largely influenced by Pinto's (Pin94) work on concurrency and continuous processes. The language has the following ontology:

- a sort *situation*, and a distinguished situation constant symbol S_0.
- a sort *time* ranging over the reals.
- a sort *action* of *simple* actions. All actions are instantaneous, and are denoted by a family of functions that take a parameter in the last argument position denoting the time of the action's occurrence. Variables a, a', etc. are used for simple actions.
- a sort *concurrent* of concurrent actions which are sets of simple actions. Variables c, c', etc. are used for concurrent actions.
- a function symbol $time : action \rightarrow \Re$, where $time(a)$ denotes the time of the action a.
- a function symbol $start : situation \rightarrow \Re$, where $start(s)$ denotes the start time of the situation s.
- a function symbol $do : action \times situation \rightarrow situation$.
- The predicate symbol $Poss$, where $Poss(a, s)$ means that the simple action a is possible in situation s (and similarly for any concurrent action c).
- The predicate symbol $<$, where $s < s'$ means that s' is reachable from s through the execution of a sequence of possible actions (simple or concurrent).
- The foundational axioms for the concurrent temporal situation calculus, provided in (Rei96), which are generalizations of those provided in (LR94) and (Rei93) for the nonconcurrent situation calculus. These axioms include unique names axioms for situations, a definition for $<$, a coherency criterion for concurrent actions, and an induction axiom.

Axiomatizing Application Domains

Levesque *et al.* (LRL+96) list the general types of axioms required to formalize an application domain in the situation calculus. In particular, our axiomatization consists of the following axioms:

- For each *simple* action A, a single action precondition axiom of the form

$$Poss(A(\vec{x}, t), s) \equiv start(s) \leq t \wedge \Phi(\vec{x}, t, s)$$

  where $\Phi(\vec{x}, t, s)$ is any first order formula with free variables among $\vec{x}$, t, and s whose only term of sort *situation* is s.

- For each fluent f (except for *defined fluents*, which are defined in terms of other fluents), a single successor state axiom. The form of a successor state axiom for a relational fluent is

$$Poss(c, s) \supset f(\vec{x}, do(c, s)) \equiv$$
$$\gamma_f^+(\vec{x}, c, s) \vee f(\vec{x}, s) \wedge \neg\gamma_f^-(\vec{x}, c, s)$$

  where $\gamma_f^+(\vec{x}, c, s)$ and $\gamma_f^-(\vec{x}, c, s)$ denote the conditions under which c, if performed in s, results in $f(\vec{x}, do(c, s))$ becoming true and false, respectively. For a functional fluent, the form of the successor state axiom is

$$Poss(c, s) \supset f(\vec{x}, do(c, s)) = y \equiv$$
$$\gamma_f(\vec{x}, y, c, s) \vee y = f(\vec{x}, s) \wedge \neg(\exists y')\gamma_f(\vec{x}, y', c, s)$$

  Here, $\gamma_f(\vec{x}, y, c, s)$ is a first order formula whose free variables are among $\vec{x}, y, c, s$. In the case of Successor State Axiom 2, $\gamma_{position}(\vec{x}, f, c, s)$ is

$$[(\exists t')bounce(ball, t') \in c \wedge catch(ball, t') \notin c \wedge$$
$$(\forall t)f(t) = 0 - \tfrac{d}{dt}(position(ball, s))(t')t - gt^2]$$
$$\vee$$
$$[(\exists t')catch(ball, t') \in c \wedge$$
$$(\forall t)f(t) = position(ball, s)(t')]$$

- Unique names axioms for the primitive actions.
- Axioms describing the initial situation.
- The foundational axioms mentioned in the previous section.

Formalizing the Steam Boiler Specification

The first step in implementing a steam boiler controller with the situation calculus is to reconcile the inherently procedural operation of a steam boiler controller with the inherently declarative nature of the situation calculus.

The issue is the following. The axiomatizer cannot completely specify what the behavior of the controller will be after it is put into service, because the information comprising the incoming messages that will be received at run-time is not available to her at the time of specification. She cannot write down, for example, that at time t, the controller will receive message m, and do x.

Knowledge-producing actions are required to handle this issue properly (see, for example, (LLL+96; SL93)). The receipt of a message would be represented by the execution of a knowledge-producing action. The axiomatizer would be able to write down that at time t, the controller will receive a message, and if the message is m_1, the controller will do x_1, but if the message is m_2, the controller will do x_2, etc.

In order to keep the particular implementation presented in this paper as simple as possible, we do not use knowledge producing actions. We deal with the issue of incoming messages non-logically, and in order to not completely betray one of the main benefits of the situation calculus approach to the steam boiler problem, namely, that the the approach is logical, we must be careful about these non-logical properties.

We define a predicate $input(m, t)$, which asserts that message m is ready to be received by the controller at time t. We handle incoming messages with a PROLOG `assert` statement, which updates the definition of the $input(m, t)$ predicate as messages are received. When executing a specification that is continually updated in this way, using the `assert` statement, the logical consequences actually used by the controller are all and only those logical consequences that would be used by the controller if all knowledge about future incoming messages were known and specified before the controller is put into service.

Another procedural aspect of a steam-boiler controller that conflicts with a purely declarative approach is the issue of outgoing messages. A situation calculus specification does not *do* anything, and it certainly does not send messages to a steam boiler. However, a situation calculus specification *can* be used by a theorem prover to derive logical consequences of the specification. The theorem prover would use the logical consequences of the specification to determine what messages should be transmitted to the steam boiler. We arrange things such that the theorem prover transmits to the steam boiler all and only those messages whose transmission is a logical consequence of the specification.

With these issues addressed, a situation calculus specification can be used to control a steam-boiler. The general operation of the situation calculus implementation is as follows:

1. Every five seconds, a set of messages are received from the steam boiler. For each of these messages, m, $Poss(receive(m, t), s)$ is now true at the current time t in the current situation s.
2. The concurrent action consisting of the complete set of $receive(m, t)$ actions is performed, and the values of the fluents are affected accordingly.
3. The new values of the fluents make more actions possible immediately. Since all the actions are natural, any action that is possible (and not prevented by an earlier action) is carried out.
4. The first actions to become possible are those that change the mode of operation of the controller, if any.
5. After the mode has stabilized, a complex action consisting of a set of $transmit(m, t)$ actions will be possible, and carried out.
6. The situation resulting from transmitting messages is stable, and the next actions to be possible are the actions to receive the messages in the next cycle.

In the following sections, we show what such a specification looks like. Rather than present the entire specification, we present enough of the specification to show the reader how such a specification can be constructed.

Actions

The following actions are sufficient to model the steam boiler:

- $receive(m, t)$: receive message m at time t
- $transmit(m, t)$: transmit message m at time t
- $switch\_to\_mode(m, t)$: switch to operation mode m at time t.

These are all natural actions.

Fluents

The fluents described in this section comprise a sufficient notion of the state of the system. The names of fluents are chosen to be consistent with the original steam boiler specification. The various quantities (e.g. the water throughput of a pump) are considered to be within an upper and lower bound, and each bound has three values associated with it: a *claimed* value, which is the corresponding sensor reading, a *calculated* value, which is calculated by the controller using its knowledge of the dynamics of the boiler, and a *best estimate*, which is the same as the claimed value if the sensor is not defective, and the same as the calculated value if the sensor is defective. Note that the upper and lower bounds on a quantity are both equal to the sensor reading of that quantity if the corresponding sensor is not defective. The fluents are

$mode(s) = z$: the controller's mode of operation is z, one of $\{initialization, normal, degraded, rescue, emergency\_stop\}$

$q(s) = z$: the last transmission contained a message claiming the quantity of water in the boiler was z

$qc1(s) = z$: z, a function of time, is the calculated lower bound ($qc2(s)$: upper bound) on the quantity of water in the boiler

$qa1(s) = z$: z is the best estimate of the lower bound ($qa2(s)$: upper bound) on the quantity of water in the boiler at the start of s

$v(s) = z$: the last transmission contained a message claiming the quantity of steam exiting the boiler was z

$vc1(s) = z$: z, a function of time, is the calculated minimum ($vc1(s)$: maximum) rate of steam leaving the boiler

$va1(s) = z$: z is the best estimate of the lower bound ($va2(s)$: upper bound) on the

$pc1(s) = z$: z is the calculated lower bound ($pc2(s)$: upper bound) on the (constant) throughput of the pumps.

$pa1(s) = z$: z is the best estimate of the lower bound ($pa2(s)$: upper bound) on the (constant) throughput of the pumps. rate of steam leaving the boiler at the start of s

$transmitted(m, s)$: message m was transmitted at the start of s

$received(m, t, s)$: message m was received at time t

$waiting\_begin\_flow(n, s) = z$: z is the number of transmissions received since pump n was turned on, and pump controller n has not yet confirmed the flow starting (if the pump has not been turned on, $z = -1$), similarly for $waiting\_stop\_flow(n, s)$

$transmission\_error(s)$: a transmission error occurred at the start of s (e.g. a mandatory message was missing from a transmission)

$defective(x, s)$: component x is defective

$steam\_boiler\_waiting(s)$: the steam boiler is waiting for the controller to indicate it is ready to come online

$program\_ready(s)$: program is ready to come on line

$physical\_units\_ready(s)$: the steam boiler is ready for control to begin

$pump\_state(n, b, s)$: $pump(n)$ has been shut off ($b = 0$), or turned on ($b = 1$)

$pump\_control\_state(n, b, s)$: the last transmission contained a message claiming that water from $pump(n)$ was ($b = 1$) or was not ($b = 0$) flowing

$valve\_open(s)$: the valve to let water drain out of the steam boiler is open

$waiting\_ack(m, s)$: the controller is waiting for acknowledgement message m

$send\_ack(x, s)$: the controller should send acknowledgement message m in the next transmission

Initial Situation

The situation is S_0 when the controller is turned on: $mode(S_0) = initialization$, $q(S_0) = 0$, $pa1(S_0)(t) = 0$, $pa2(S_0)(t) = 0$, $v(S_0)(t) = 0$, $\neg defective(x, S_0)$, etc.

Precondition Axioms

In practice, it is possible to receive a message when that message arrives, and we know that messages will arrive approximately every five seconds. The PROLOG simulator that executes the situation calculus specification is responsible for controlling the timing of the transmit/receive cycle.

In theory, messages are received exactly when they arrive, and they arrive every five seconds (when t, which ranges over the reals, is equal to some integer that is a multiple of five). To ensure the program detects the absence of a transmission, we invent a message, *tick*, which is guaranteed to be received every five seconds. The value of the $transmission\_error(s)$ fluent is determined by considering what messages were received with each tick. The theorem prover (we use PROLOG) updates the definition of the $input(m, t)$ predicate (using the PROLOG **assert** statement) as messages arrive. The $input(m, t)$ predicate asserts that message m is ready to be received at time t.

The following axiom characterizes when messages are actually received (rather than only ready to be received):

$$Poss(receive(m, t), s) \equiv$$
$$t \geq start(s) \wedge (\exists i) t = 5i \wedge \quad (3)$$
$$\neg received(m, t, s) \wedge [input(m, t) \vee m = tick]$$

The precondition axiom for the $transmit(m, t)$ action is such that messages are transmitted exactly when they are appropriate for proper operation of the controller. The axiom ensures that the controller's mode has stabilized before any messages are transmitted. If the controller is in *emergency_stop* mode, only the message $MODE(emergency\_stop)$ should be transmitted. Also, is not possible to transmit a message that was just transmitted. These conditions are dictated by the following defined fluent:

$$transmit\_cond(m, t, s) \equiv$$
$$t \geq start(s) \wedge mode(s) \neq emergency\_stop \wedge$$
$$[(\forall m') \neg Poss(switch\_to\_mode(m', t), s)] \wedge$$
$$\neg transmitted(m, s)$$

In addition, the decision to switch the pumps on or off is characterized by the following defined fluent:

$$need\_pumps(s) \equiv$$
$$[qa1(s) < N_1 \wedge qa2(s) < N_1] \vee$$
$$[qa1(s) < N_1 \wedge N_1 < qa2(s) \wedge qa2(s) < N_2],$$

where N_1 and N_2 are the upper and lower limits, respectively, of the normal operating range for the water level in the boiler. Control strategies other than this one are possible; however, this one is proposed in (Abr94) as a reasonable candidate.

Given these two defined fluents, the precondition axiom for the $transmit(m, t)$ action is:

$$Poss(transmit(m, t), s) \equiv$$

$$
\begin{aligned}
& m = MODE(emergency\_stop) \wedge t \geq start(s) \wedge \\
& mode(s) = emergency\_stop \wedge \neg transmitted(m, s) \\
& \vee \\
& m = PROGRAM\_READY \wedge \\
& transmit\_cond(m, s, t) \wedge \\
& mode(s) = initialization \wedge program\_ready(s) \\
& \vee \\
& (\exists n)[m = OPEN\_PUMP(n) \wedge \\
& transmit\_cond(m, s, t) \wedge need\_pumps(s) \wedge \\
& [n = 1 \vee n = 2 \vee n = 3 \vee n = 4] \wedge \\
& \neg defective(pump(n), s) \wedge pump\_state(n, 0, s)] \\
& \vee \\
& \vdots \\
& m = STEAM\_REPD\_ACK \wedge \\
& transmit\_cond(m, s, t) \wedge \\
& send\_ack(STEAM\_REPD\_ACK) \qquad (4)
\end{aligned}
$$

Successor State Axioms

We present just the most interesting, representative successor state axioms.

The successor state axiom for the $mode(s) = m$ fluent is straightforward. The only action that can affect the mode of the program is a $switch\_to\_mode(m, t)$ action:

$$
\begin{aligned}
& Poss(c, s) \supset mode(do(c, s)) = m \equiv \\
& \quad (\exists t) switch\_to\_mode(m, t) \in c \vee \\
& \quad switch\_to\_mode(m', t) \notin c \wedge mode(s) = m \quad (5)
\end{aligned}
$$

The successor state axiom for $defective(x, s)$ characterizes the conditions under which a component is considered defective. It says,

- $pump(n)$ is defective if it changes state spontaneously, or if an $OPEN\_PUMP(n)$ or $CLOSE\_PUMP(n)$ message were sent in the previous cycle, but the flow has not yet started or stopped, respectively.
- $pump\_control(n)$ is defective if it changes state spontaneously, or if an $OPEN\_PUMP(n)$ or $CLOSE\_PUMP(n)$ message were sent in the previous cycle, but the flow has not yet started or stopped, respectively, and $pump(n)$ is not defective.
- $water\_level$, the water level measuring unit, is defective if it indicates a value that is outside the valid static limits (i.e. 0 and C), or if it indicates a value which is incompatible with the dynamics of the system.
- $steam\_rate$, the unit measuring the exit rate of steam, is defective if it indicates a value that is outside the valid static limits (i.e. 0 and W), or if it indicates a value which is incompatible with the dynamics of the system.

$$
\begin{aligned}
& Poss(c, s) \supset defective(x, do(c, s)) \equiv \\
& \quad (\exists n) x = pump(n) \wedge \\
& \quad [[(\exists b, t) receive(PUMP\_STATE(n, b), t) \in c \wedge \\
& \quad (\exists a) pump\_state(n, a, s) \wedge a \neq b] \vee \\
& \quad pump\_state(n, 0, s) \wedge \\
& \quad waiting\_begin\_flow(n, s) = 1 \vee \\
& \quad pump\_state(n, 1, s) \wedge \\
& \quad waiting\_stop\_flow(n, s) = 1] \\
& \quad \vee \\
& \quad (\exists n) x = pump\_control(n) \wedge \\
& \quad \neg defective(pump(n), s) \wedge \\
& \quad [[pump\_control\_state(n, 0, s) \wedge \\
& \quad waiting\_begin\_flow(n, s) > 1 \vee \\
& \quad pump\_control\_state(n, 1, s) \wedge \\
& \quad waiting\_stop\_flow(n, s) > 1] \vee \\
& \quad (\exists a) pump\_control\_state(n, a, s) \wedge \\
& \quad waiting\_begin\_flow(n, s) = -1 \wedge \\
& \quad waiting\_stop\_flow(n, s) = -1 \wedge \\
& \quad (\exists b) pump\_state(n, b, s) \wedge a \neq b] \\
& \quad \vee \\
& \quad x = water\_level \wedge [q(s) < 0 \vee q(s) > C] \vee \\
& \quad q(s) < qc1(s)(time(c)) \vee q(s) > qc2(s)(time(c))] \\
& \quad \vee \\
& \quad x = steam\_rate \wedge [v(s) < 0 \vee v(s) > W] \vee \qquad (6) \\
& \quad v(s) < vc1(s)(time(c)) \vee v(s) > vc2(s)(time(c))]
\end{aligned}
$$

Defined Fluents

Defined fluents are fluents that are defined in terms of other fluents. We could write down a successor state axiom for any defined fluent. It would be a compilation of the successor state axioms of the fluents in the definition of the defined fluent that have a situation argument of the form $do(c, s)$. However, the resulting successor state axiom would be larger and more complicated than the defined fluent.

The fluent $qa2(s) = z$ means that z is the best estimate of the upper bound on the quantity of water in the boiler. If the water level detection equipment is not defective, the program accepts the value given by that equipment as the best estimate; otherwise, it uses the value calculated by $qc2(s)(t)$ at the time of the action which started the situation. The formula for $qa1$ is similar to this one for $qa2$:

$$
\begin{aligned}
& qa2(do(c, s)) = z \equiv \\
& \quad defective(water\_level, s) \wedge z = qc2(s)(time(c)) \vee \\
& \quad \neg defective(water\_level, s) \wedge z = q(do(c, s)) \qquad (7)
\end{aligned}
$$

The Situation Calculus Simulator

In this section we discuss a PROLOG technology simulator. The simulator can simulate a concurrent situa-

tion calculus specification like the steam boiler formalization presented in the previous section.

The PROLOG Simulator

A situation calculus model defines a tree of situations emanating from the distinguished situation S_0. Some of the situations in the tree correspond to *legal* situations, and some do not. A legal situation is consistent with the laws of Nature, in that a natural action must occur at the time dictated by natural laws governing the behavior of the system, unless the action is prevented from occurring by an earlier natural action. Reiter (Rei96) defines the $legal(s)$ predicate to formalize this principle:

$$\begin{aligned} &legal(s) \equiv \\ &\quad S_0 \leq s \wedge \\ &\quad (\forall a, c, s').natural(a) \wedge Poss(a, s') \wedge \\ &\quad do(c, s') \leq s \wedge a \notin c \supset time(c) < time(a). \end{aligned} \quad (8)$$

Here, $\leq$ is the ordering relation defined by the foundational axioms mentioned earlier. The *legal* predicate is instrumental in the implementation of a simulator, as will become clear.

A domain of discourse in which all actions are natural is said to comply with Reiter's (Rei96) Natural World Condition (NWC). This condition assures a deterministic simulation.

Another concept crucial to the implementation of a simulator is the notion of Reiter's (Rei96) Least Natural Time Points:

$$\begin{aligned} &lntp(s, t) \equiv \\ &\quad (\exists a)[natural(a) \wedge Poss(a, s) \wedge time(a) = t \wedge \\ &\quad (\forall a')[natural(a') \wedge Poss(a', s) \supset \\ &\quad time(a') \geq t]. \end{aligned} \quad (9)$$

Informally, the least natural time point is the earliest time at which any natural action can possibly occur in a situation. The Least Natural Time Point Condition ($LNTPC$) is the following:

$$(\forall s).(\exists a)[natural(a) \wedge Poss(a, s)] \supset (\exists t) lntp(s, t). \quad (10)$$

This condition states that every situation in which there is a possible natural action has a least natural time point. An example of a world where this condition fails is one in which we have $(\forall a).natural(a) \equiv (\exists x, t)a = B(x, t)$, where x ranges over the non-zero natural numbers, and $Poss(B(x, t), s) \equiv t = start(s) + 1/x$.

Reiter (Rei96) puts this all together with his foundational axioms for the concurrent temporal situation calculus and proves:

$$\begin{aligned} &LNTPC \wedge NWC \supset legal(do(c, s)) \equiv \\ &\quad legal(s) \wedge Poss(c, s) \wedge \\ &\quad (\forall a)[a \in c \equiv Poss(a, s) \wedge lntp(s, time(a))]. \end{aligned} \quad (11)$$

Formula 11 is the engine for the simulator. The simulator is a PROLOG procedure that takes a situation term s as an argument (initially S_0), prints its argument, constructs a set of actions c such that

$$(\forall a)[a \in c \equiv Poss(a, s) \wedge lntp(s, time(a))],$$

and recursively calls itself with $do(c, s)$. In so doing, the simulator follows the path of legal situations (there is only one path of legal situations when all actions are natural), simulating the evolution of the system.

Here is the PROLOG code:

```
simulate(S) :-
    nl,nl,print(S),
    setof(A,lntp(S,A),C),
    simulate(do(C,S)).

lntp(S,A):-
    natural(A)),
    poss(A,S),
    time(A,T),
    not (natural(A_prime),
         poss(A_prime,S),
         time(A_prime,T_prime),
         T>T_prime).
```

The steam boiler controller version of this code needs to transmit outgoing messages and add information about incoming messages as the simulation proceeds, so the `simulate(S)` procedure becomes:

```
simulate(S) :-
    ( setof(A,lntp(S,A),C),
      transmit_messages(C),
      simulate(do(C,S))
      ;
      read(Incoming_messages),
      assert_input(Incoming_messages),
      simulate(S)).
```

Translating a Model to PROLOG

Clark's completion semantics for logic programming (Cla78) admit a translation from the situation calculus axioms to PROLOG clauses. The procedure is to simply make the implication in the axioms go only one way, and write down the clausal form. For example, the PROLOG equivalent of Axiom 5 is

```
mode(do(C,S),M):-
    poss(C,S),
    (member(switch_to_mode(M,T),C),
     ;
     not member(switch_to_mode(M_prime,T,C),
     mode(S,M)).
```

Evaluation of the Solution

In this section, we attempt to apply the evaluation criteria given in (ABL95) to this formalization and simulation. These criteria were formulated by participants at the Dagstuhl meeting to evaluate various solutions

to the steam boiler problem and to compare the specific merits and drawbacks of the formal methods used in those solutions.

The criteria consist of a series of questions about the solution. The following is a sample of the questions, and my own proposed answers.

- *Does the solution comprise a requirements specification?* Yes, since it is a translation of the informal requirements specification into first-order logic.
- *Is the requirements specification of the solution formal and rigorous?* Yes.
- *Does the solution comprise a functional design?* Yes.
- *Has the functional design of the solution been verified against the requirements specification?* The functional design of the solution *is* the requirements specification.
- *Does the solution comprise an architectural design?* Yes.
- *Has the architectural design of the solution been verified against the requirements specification?* The architectural design of the solution *is* the requirements specification.
- *Does the solution comprise an implementation of a control program?* Yes.
- *What are the comparable other solutions?* This is planned future work.
- *How much time has been spent on producing the solution?* Approximately four weeks.
- *How much preparation is need to become sufficiently expert of the used specification framework in order to be able to produce a solution to such a problem in that framework?* Unknown. A course on first-order logic is probably necessary, as well as some reading on the situation calculus.
- *What are the premises for a good understanding of the proposed solution?* Again, a course on first-order logic is probably necessary, as well as some reading on the situation calculus.

Acknowledgements

Ray Reiter provided useful comments and important corrections. I have also benefited from discussions with Javier Pinto.

References

Jean-Raymond Abrial, Egon Boerger, and Hans Langmaack. Preliminary report for the Dagstuhl-seminar 9523: Methods for semantics and specification.
Available via WWW at http://www.informatik.uni-kiel.de/~procos/dag9523/dag9523.html, 1995.

Jean-Raymond Abrial. Steam-boiler control specification problem. Distributed to the participants of the Dagstuhl Meeting, "Methods for Semantics and Specification", June 4-9, 1995. This paper, and its companion, "Additional information concerning the physical behavior of the steam boiler", are available via WWW at http://www.informatik.uni-kiel.de/~procos/dag9523/dag9523.html, August 1994.

K. L. Clark. Negation as failure. In Hervé Gallaire and Jack Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, New York, 1978.

T. G. Kelley. Reasoning about physical systems with the situation calculus. In *COMMON SENSE '96: the third symposium on logical formalizations of commonsense reasoning*, Stanford University, January 1996.

Yves Lespérance, Hector J. Levesque, Fangzhen Lin, Daniel Marcu, Raymond Reiter, and Richard B. Scherl. Foundations of a logical approach to agent programming. In M. Wooldridge, J. P. Müller, and M. Tambe, editors, *Intelligent Agents Volume II—Proceedings of the 1995 Workshop on Agent Theories, Architectures, and Languages (ATAL-95)*, pages 331–346. Springer-Verlag, Lecture Notes in Artificial Intelligence, 1996. To Appear.

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

H. J. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R. Scherl. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming, Special Issue on Reasoning about Action and Change*, 1996. To appear.

John McCarthy and Patrick Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*, pages 463–502. Edinburgh University Press, Edinburgh, Scotland, 1969.

Javier Andrés Pinto. *Temporal Reasoning in the Situation Calculus.* PhD thesis, University of Toronto, Toronto, Ontario, Canada, February 1994.

R. Reiter. The frame problem in the situation calculus: a simple solution (sometimes) and a completeness result for goal regression. In Vladimir Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pages 359–380. Academic Press, San Diego, CA, 1991.

R. Reiter. Proving properties of states in the situation calculus. *Artificial Intelligence*, 64:337–351, 1993.

R. Reiter. Natural actions, concurrency and continuous time in the situation calculus. In *COMMON SENSE '96: the third symposium on logical formalizations of commonsense reasoning*, Stanford University, January 1996.

Richard B. Scherl and Hector J. Levesque. The frame problem and knowledge-producing actions. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 689–695, Washington, DC, July 1993. AAAI Press/The MIT Press.