A Situation Calculus Model of Hypertext

Richard Scherl and Michael Bieber

Department of Computer and Information Sciences
New Jersey Institute of Technology
Newark, NJ 07102-1982 U.S.A.
email: scherl@cis.njit.edu bieber@cis.njit.edu

Fabio Vitali

Computer Science Department
University of Bologna
Italy
email: fabio@cs.unibo.it

Abstract

We utilize the situation calculus to develop a logical model of hypertext systems. The work builds upon the earlier work of Bieber and Kimbrough in the logical modeling of hypertext systems. In our presentation, a particular version of the situation calculus (which includes a language for programming complex actions) developed for the modeling of dynamic worlds and for the control of robotic agents (as studied in artificial intelligence) is used to represent the dynamics of a hypertext system. We argue that the formulation besides being of interest in itself has a number of advantages over other methods of formalizing hypertext systems.

1 Introduction

Hypertext and related systems are becoming ubiquitous. A wide variety of systems have been built. Each of these systems [7]:

> provides its users with the ability to create, manipulate, and/or examine a network of information-containing nodes interconnected by relational links.

There is both a database of interconnected pieces of information and facilities for navigating and modifying this database.

Bieber and Kimbrough [1] have developed an initial logic model of a generic hypertext system and then built upon that model a notion of generalized hypertext. Generalized hypertext uses logical inferencing rather than manual (human) specification to create links, nodes, and impose views. In addition to serving as the foundation for generalized hypertext, the logic model also proved useful for the specification and coding of the generalized hypertext system Max [1]. We feel that logic models of hypertext systems can in general play a useful role in the comparison of various hypertext systems, as well as in the specification and coding of specific systems, much in the same way as a specification in a formal language such as *Z*. Once inferencing is added, as in generalized hypertext, the logic model becomes essential.

But the logic model developed by Bieber and Kimbrough [1] suffers from a major drawback. It does not have a means of representing the dynamic aspects that are essential to a hypertext systems. It really is only a logic model for certain elements of the hypertext system. The central aspect, the changes that occur as links are traversed, is left outside of the model. In the situation calculus model developed here, it is possible given an axiomatization to ask whether or not a particular sentence must be true after the execution of a particular sequence of actions.

The modeling of dynamic worlds has been studied in artificial intelligence where the common sense reasoning needed by a robotic agent is a major concern. A wide variety of formalisms have been developed to capture this sort of common sense reasoning about change. Researchers have struggled with a number of difficult problems such as the frame[1] problem [19].

One of the oldest formalisms for representing dynamically changing worlds is the situation calculus [12]. Recently it has been enjoying a revival because its expressiveness is much richer than what had been commonly believed [5, 14], and it has proven useful both as a method for specifying and for implementing robotic agents [10, 9, 8, 15].

The situation calculus provides a formalism for reasoning about actions and their effects on the world. Axioms are used to specify the prerequisites of actions as well as their effects, that is, the fluents that they change. In general, it is also necessary to provide frame axioms to specify which fluents remain unchanged by the actions. In the worst case this might

[1] This is the problem of having to add extra axioms called frame axioms to specify the intuitively obvious common facts about what we know from common sense does not change after an action occurs.

require an axiom for every combination of action and fluent. Recently, Reiter [15] (generalizing the work of Haas [6], Schubert [18] and Pednault [13]) has given a set of conditions under which the explicit specification of frame axioms can be avoided. Under these circumstances a relatively simple solution suffices. We utilize the formulation of Reiter in this paper.

Building upon Reiter's work, the Cognitive Robotics group at the Univesity of Toronto [10, 9, 8] has further developed the situation calculus to both model a robotic agent in a dynamic world and to develop a high-level programming language called GOLOG for declaratively defining complex actions (such as iteration, conditionals, and loops) that are built upon the basic primitive actions of the situation calculus. Thus specification and implementation are accomplished in a unified framework.

This does not mean that all problems are solved. But as discussed in [10, 9, 8] there is an implemented interpreter for GOLOG. Thus a specification in GOLOG can be executed by this interpreter. This interpreted code can then serve as the basis for compilation into an efficient program, or can serve as a specification language that has the added advantage of being executed in an interpreted mode.

In this paper, we utilize both the situation calculus and GOLOG to represent the dynamics of a hypertext system. Even though the situation calculus was initially developed and studied with A.I. problems in mind, it and the approach to the frame problem has been applied to the formalization of software systems. For example it has been used to formalize the evolution of a database under the effect of an arbitrary sequence of update transactions [17]. Additionally, the approach has been used as the foundations for a language for software specification [2]

Here we explore another domain, hypertext, in which the simple solution to the frame problem ([15]) proves to be sufficient for most aspects of the problem. Additionally, we feel that the formulation besides being of interest in itself has a number of advantages over other methods of formalizing hypertext systems. It preserves all of the advantages of the Bieber and Kimbrough [1] logic model in that one can easily specify aspects of generalized hypertext.

Also, the approach based on the situation calculus gives us automatically a formalism for representing and reasoning about context. We wish to model applications that determine which links to display based on the dynamic situation of the user interacting with the database. A number of hypertext systems incorporate such features. For example, in Trellis [4] it is possible to control the navigational possibilities available to the user, and to let him/her access and activate links based on the previous steps that the user has taken. Context is also utilized in [20, 3].

We utilize the situation calculus to develop a logical model of a core sort of hypertext system, very close to the basic hypertext of [1]. It does not, for example, have all of the features of the Dexter model [7]. These can all be readily added. For example, the only links considered in this paper are binary and we do not consider composite nodes.

In [7], a hypertext system is divided into three layers. These are the *run-time* layer, the *storage* layer, and the *within-component* layer. The run-time layer consists of the mechanisms that enable the user to interact with the hypertext system. The storage layer is the network of nodes and the links between them. The internal structure of the nodes is captured by the within-component layer. In this paper nothing more will be said about the within-component layer as we will not be considering composite nodes.

In the next section, the situation calculus background is given. Then, in Section 3, a model of the core hypertext system is developed. This discussion includes the representation of the storage layer, and the basic actions needed by the run-time layer. But it does not include a representation of the overall operation of the run-time layer, because that requires a discussion of the complex actions that form the basis of GOLOG.

In Section 4 complex actions and the GOLOG programming language are discussed and in Section 5 these features are used to specify the run-time component. At this point, the model can be used to reason about what is true after the execution of a particular sequence of actions and to represent the hypertext system as an ongoing process, a read-evaluate-print loop.

Section 6 covers the problem of reasoning about global properties of the axiomatization. These are properties that hold in any state of the system. Establishing these properties requires justifying the use of inductive reasoning over situations.

Section 7 discusses the issues involved in adding notions from generalized hypertext. Finally, in Section 8, the paper is summarized and future work is discussed.

2 The Situation Calculus: A Language for Specifying Dynamics

The situation calculus (following the presentation in [15]) is a first-order language for representing dynamically changing worlds in which all of the changes

are the result of named *actions* performed by some agent. For example

$$\text{DROP}(x)$$

represents the action of dropping some object x. Terms are used to represent states of the world–i.e. *situations*. If α is an action and s a situation, the result of performing α in s is represented by $do(\alpha, s)$. The constant S_0 is used to denote the initial situation. Relations whose truth values vary from situation to situation, called *fluents*, are denoted by predicate symbols taking a situation term as the last argument. For example, $\text{BROKEN}(x, s)$ means that object x is broken in situation s. Functions whose denotations vary from situation to situation are called *functional fluents*. They are denoted by function symbols with an extra argument taking a situation term, as in $\text{POSITION}(robot, s)$, i.e., the position of the *robot* in s.

It is assumed that the axiomatizer has provided for each action $\alpha(\vec{x})$, an *action precondition axiom* of the form given in 1, where $\pi_\alpha(\vec{x}, s)$ is a formula specifying the preconditions for action $\alpha(\vec{x})$.

$$\text{POSS}(\alpha(\vec{x}), s) \equiv \pi_\alpha(\vec{x}, s) \tag{1}$$

An action precondition axiom for the action DROP is given below.

$$\text{POSS}(\text{DROP}(x), s) \equiv \text{HOLDING}(x, s) \tag{2}$$

The axiom states that the drop action is possible if and only if the agent is holding an object.

The core of the method of axiomatization is the construction of *successor state axioms*. Their general form is given below:

$$\begin{aligned} &\text{POSS}(a, s) \rightarrow [F(\vec{x}, do(a, s)) \equiv \\ &\gamma_F^+(\vec{x}, a, s) \vee (F(\vec{x}, s) \wedge \neg\gamma_F^-(\vec{x}, a, s))] \end{aligned} \tag{3}$$

Similar successor state axioms may be written for functional fluents. A successor state axiom is needed for each fluent F, and an action precondition axiom is needed for each action a.

Reiter[15] shows how to derive a set of *successor state axioms* of the form given in 3 from the usual positive and negative effect axioms, a completeness assumption[2], and the restriction that there are no ramifications, i.e., indirect effects of actions[3]. Often it is possible to code axioms directly in the form of successor state axioms. That is what we will do in the next section of this paper.

Given such an axiomatization, one can give the axiomatization of the initial situation $\mathcal{F}_0$, the axiomatization of the hypertext system $\mathcal{F}_{ss}$ and then ask whether or not the axiomatization entails that a particular sentence G will be true after the execution of a particular sequence of actions (contained in s_{gr}).

$$\mathcal{F}_0 \bigcup \mathcal{F}_{ss} \models G(s_{gr})$$

Methods for efficiently automating such queries are discussed in [15].

3 Hypertext

Following [1], a *hypertext* consists of an arbitrary number of interrelated *nodes*, *links*, and *buttons*. *Nodes* are objects that are declared in a data base and, when displayed, are represented as text on the screen. *Links*, which describe relationships between pairs of nodes[4] (called the source and sink), are also declared in a data base.

The essential fluents and their intended interpretations are as follows:

1. $\text{NODE}(x, y, z, s)$ x is a node with content expression y and semantic type z in situation s.
2. $\text{LINK}(u, v, w, x, y, z, s)$ u is a link from source node v to sink node w with semantic type x, operation type y, and either display mode or procedure identifier z in situation s.
3. $\text{BUTTON}(x, y, z, s)$ x is a button representing link y with context expression z in situation s.
4. $\text{WINDOW}(s)$ denotes that which is displayed in the main window in situation s (a functional fluent).
5. $\text{CURRENT\_NODE}(x, s)$ x is the current node in situation s.
6. $\text{POP\_UP\_WINDOW}(s)$ denotes the display in the pop_up_window in situation s (a functional fluent).

Consider the following simple hyperdocument taken from [1]. All of these declarations are facts declared to be true at the initial situation S_0.

[2] Reiter[15] also discusses the need for unique name axioms for actions and situations.

[3] This last condition can be ensured by prohibiting state constraints, i.e., sentences that specify an interaction between fluents. An example of such a sentence is $\forall s \text{P}(s) \equiv \text{Q}(s)$. If an action a made $P(do(a, s))$ true, then the action would have the indirect effect of making $Q(do(a, s))$ true as well since the constraint specifies that Q must have the same truth value as P in every situation.

[4] The links are directional — pointing from the source node to the sink node.

NODE(1,['If the','Soviet Union','is to be competitive', 'we must hit inside curveball', '_', button(1)],QUOTATION,S_0)

NODE(2,['Quotation of the Day', 'The New York Times','August 16, 1989'], DESCRIPTION,S_0)

LINK(1,1,2,more_information,display, full_window,S_0)

BUTTON(1,1,'Aleksei L. Nilolov',S_0)

BUTTON(0,0,'Exit',S_0)

CURRENT_NODE(1,S_0)

This database declares nodes 1 and 2. Link 1 has node 1 as a source node and node 2 as a sink node. Button 1 is a button for link 1 and is displayed with node 1. Node 1 is set to be initially the current node. Finally, button 0 is there so that the user can signal to exit and terminate the session.

The available commands are:

1. TRAVERSE_SINK(x, y) moves the current node from x to y.
2. TRAVERSE_SOURCE(x, y) moves the current node from y to x.
3. DISPLAY_NODE_ATTRIBUTE(x) displays in a pop-up window the attribute of the node x.
4. DISPLAY_LINK_ATTRIBUTE(x) displays in a pop-up window the attribute of the link x.
5. MAKE_DISPLAY_TEXT(x) displays in a window the text encoded in content expression x.
6. CREATE_NODE(x, y, z) adds to the database node x with content expression y and semantic type z.
7. DELETE_NODE(x) deletes from the database node x.
8. CREATE_LINK(u, v, w, x, y, z) adds to the database link u from source node v to sink node w with semantic type x, operation type y, and either display mode or procedure identifier z.
9. DELETE_LINK(u) deletes from the database link u.

The action MAKE_DISPLAY_TEXT creates the display in the window. Here we do not give a full specification of what appears in the window. But it is required that the action properly displays all buttons in the content of the node and also button 0 is always displayed. Also, we assume that there can only be one main window and one pop_up_window displayed at a time.

The following is the successor state axiom for the predicate CURRENT_NODE.

$$\begin{aligned}\text{POSS}(a, s) \rightarrow [\text{CURRENT\_NODE}(x, do(a, s)) \equiv \\ a = \text{TRAVERSE\_SINK}(y, x) \vee \\ a = \text{TRAVERSE\_SOURCE}(x, y) \vee \\ (\neg(a = \text{TRAVERSE\_SINK}(y, x) \vee \\ a = \text{TRAVERSE\_SOURCE}(y, x)) \\ \wedge \text{CURRENT\_NODE}(x, s))]\end{aligned} \tag{4}$$

Sentence 4 specifies that the current node in a particular situation that results from performing an action is either the sink node on a link if the action was a TRAVERSE_SINK action, or the source node on a link if the action was a TRAVERSE_SOURCE action or otherwise the current node in the situation prior to the action.

The axioms[5] capturing the possibility conditions for the two actions TRAVERSE_SINK and TRAVERSE_SOURCE are as follows:

$$\begin{aligned}\text{POSS}(\text{TRAVERSE\_SINK}(y, x), s) \equiv \\ \text{CURRENT\_NODE}(y, s) \\ \wedge \exists u\ \text{LINK}(u, y, x, \_, \_, \_, s)\end{aligned} \tag{5}$$

$$\begin{aligned}\text{POSS}(\text{TRAVERSE\_SOURCE}(x, y), s) \equiv \\ \text{CURRENT\_NODE}(y, s) \wedge \\ \exists u\ \text{LINK}(u, x, y, \_, \_, \_, s)\end{aligned} \tag{6}$$

Sentence 5 specifies that the TRAVERSE_SINK action from y to x is possible in a particular situation s if and only if y is the current node and there exists a link from y to x. Sentence 6 specifies that the TRAVERSE_SOURCE action from y to x is possible in a particular situation s if and only if y is the current node and there exists a link from x to y.

The following is the successor state axiom for WINDOW.

$$\begin{aligned}\text{POSS}(a, s) \rightarrow [\text{WINDOW}(do(a, s)) = x \equiv \\ a = \text{MAKE\_DISPLAY\_TEXT}(x) \vee \\ (a \neq \text{MAKE\_DISPLAY\_TEXT}(y) \\ \wedge \text{WINDOW}(s) = x)]\end{aligned} \tag{7}$$

[5] We note that for the sake of notational perspicuity we are using the underscore as in Prolog's anonymous variable. No logical generality is lost by this, for an equivalent formula can always be had by replacing each anonymous variable with a unique variable and universally quantifying over the entire formula.

Some object x is displayed in the main window if either the previous action was a MAKE_DISPLAY_TEXT(x) action or x was already in the window and the action was something other than one that displayed text in the window.

The following is the specification of the possibility condition for the action MAKE_DISPLAY_TEXT.

$$\begin{aligned}&\textsc{Poss}(\textsc{make\_display\_text}(x), s) \equiv \\ &\quad \exists z\, \textsc{current\_node}(z, s) \wedge \textsc{node}(z, x, \_, s)\end{aligned} \tag{8}$$

The displayed material must be the content expression of the current node.

Sentence 9 is the successor-state axiom for what is displayed in the pop_up_window.

$$\begin{aligned}&\textsc{Poss}(a, s) \rightarrow [\textsc{pop\_up\_window}(do(a, s)) = x \equiv \\ &\quad (\exists y\, a = \textsc{display\_link\_attr}(y) \wedge \\ &\qquad \textsc{link}(y, \_, \_, x, \_, \_, s)) \vee \\ &\quad (\exists y a = \textsc{display\_node\_attr}(y) \wedge \\ &\qquad \textsc{node}(y, \_, x, s)) \vee \\ &\quad (\neg(a = \textsc{display\_link\_attr}(y) \vee \\ &\qquad a = \textsc{display\_node\_attr}(y)) \\ &\qquad\quad \wedge \textsc{Pop\_up\_window}(s) = x))]\end{aligned} \tag{9}$$

The following are the two axioms defining the possibility conditions for the actions DISPLAY_LINK_ATTR and DISPLAY_NODE_ATTR.

$$\begin{aligned}&\textsc{Poss}(\textsc{display\_node\_attr}(y), s) \equiv \\ &\quad \textsc{current\_node}(y, s) \wedge \exists x\, \textsc{button}(x, y, \_, s)\end{aligned} \tag{10}$$

The command DISPLAY_NODE_ATTR(y) is possible if and only if y is the current node and is represented by a button.

$$\begin{aligned}&\textsc{Poss}(\textsc{display\_link\_attr}(y), s) \equiv \\ &\quad \exists z\, \textsc{current\_node}(z, s) \wedge \\ &\qquad (\textsc{link}(y, z, \_, \_, \_, \_, s) \vee \textsc{link}(y, \_, z, \_, \_, \_, s)) \\ &\qquad \wedge \exists x\, \textsc{button}(x, y, \_, s)\end{aligned} \tag{11}$$

The command DISPLAY_LINK_ATTR(y) is possible if and only if z is the current node, y is a link with z as either the source or a the sink node, and there is a button representing y.

We also need successor state axioms for the fluents LINK and NODE.

$$\begin{aligned}&\textsc{Poss}(a, s) \rightarrow [\textsc{Node}(x, y, z, do(a, s)) \equiv \\ &\quad a = \textsc{create\_node}(x, y, z) \vee \\ &\quad (a \neq \textsc{delete\_node}(x) \wedge \textsc{Node}(x, y, z, s))]\end{aligned} \tag{12}$$

$$\begin{aligned}&\textsc{Poss}(a, s) \rightarrow [\textsc{Link}(u, v, w, x, y, z, do(a, s)) \equiv \\ &\quad a = \textsc{create\_link}(u, v, w, x, y, z) \vee \\ &\quad (a \neq \textsc{delete\_link}(u) \wedge \\ &\qquad \textsc{Link}(u, v, w, x, y, z, s))]\end{aligned} \tag{13}$$

The following[6] are the four axioms defining the possibility conditions for the actions CREATE_NODE, DELETE_NODE, CREATE_LINK, and DELETE_LINK.

$$\begin{aligned}&\textsc{Poss}(\textsc{create\_node}(x, y, z), s) \equiv \\ &\quad \neg(\exists u, v, w\, \textsc{Node}(u, v, w, s) \wedge u = x)\end{aligned} \tag{14}$$

$$\begin{aligned}&\textsc{Poss}(\textsc{delete\_node}(x), s) \equiv \\ &\quad \exists u, y, z\, \textsc{node}(u, y, z, s) \wedge \\ &\qquad \neg\exists v, w\, \textsc{link}(v, w, u, \_, \_, \_, s) \wedge \\ &\qquad\quad u = x\end{aligned} \tag{15}$$

$$\begin{aligned}&\textsc{Poss}(\textsc{create\_link}(x, v, w, x, y, z), s) \equiv \\ &\quad \neg(\exists u, n, o, p, q, r\, \textsc{Link}\ (u, n, o, p, q, r, s) \\ &\qquad \wedge u = x)\end{aligned} \tag{16}$$

$$\begin{aligned}&\textsc{Poss}(\textsc{delete\_link}(x), s) \equiv \\ &\quad \exists u, n, o, p, q, r\, \textsc{Link}\ (u, n, o, p, q, r, s) \\ &\qquad \wedge u = x\end{aligned} \tag{17}$$

One can not create a link or node that already exists. Furthermore one can not delete a node if there is a link that points to it. This is requirement is designed to prevent dangling links – see Section 6. Additionally, one can only delete a link if it exists.

We have not axiomatized a session log – a history of the commands executed since the current session was initiated. This is necessary to model backtracking, a common feature of hypertext systems. Note that the name of each situation contains within it a history of all of the commands executed since S_0. Thus backtracking can easily be added utilizing the situation itself as a session log.

4 GOLOG: Complex Actions

The actions discussed in the previous section are primitive and determinate. They are like primitive computer instructions (e.g. assignment). We need complex actions to construct a program that performs a series of complex actions, tests predicates for their truth values, and then performs other actions depending on the results of the test. An example of such a program is the run-time environment. This set of complex action expressions forms a programming language that is called GOLOG (alGOl in LOGic) [10].

Complex actions could be treated as first class entities, but since the tests that appear in forms like **if**

[6] We are assuming some method for specifying an arbitrary link or node as in the Dexter reference model [7].

ϕ **then** δ_1 **else** δ_2 involve formulas ϕ, this means that we must reify fluents and formulas. Moreover, it is necessary to axiomatize the correspondence between these reified formulas and the actual situation calculus formulas. This results in a much more complex theory.

Instead complex action expressions are treated as abbreviations for expressions in the situation calculus logical language. They may be thought of as macros that expand into the genuine logical expressions. A particular execution sequence of a complex action expression will be a sequence of situation calculus primitive actions.

This is done by defining a predicate Do as in $Do(\delta, s, s')$ where δ is a complex action expression. $Do(\delta, s, s')$ is intended to mean that the agent's doing action δ in situation s leads to a (not necessarily unique) situation s'. The inductive definition of Do includes the following cases:

- $Do(a, s, s') \stackrel{\text{def}}{=} \text{Poss}(a, s) \wedge (s' = do(a, s))$ — simple actions
- $Do(\phi?, s, s') \stackrel{\text{def}}{=} \phi[s] \wedge (s = s')$ — tests
- $Do([\delta_1; \delta_2], s, s') \stackrel{\text{def}}{=} \exists s''(Do(\delta_1, s, s'') \wedge Do(\delta_2, s'', s'))$ — sequences
- $Do([\delta_1 | \delta_2], s, s') \stackrel{\text{def}}{=} Do(\delta_1, s, s') \vee Do(\delta_2, s, s')$ — nondeterministic choice of actions
- $Do((\Pi x)\delta, s, s') \stackrel{\text{def}}{=} \exists x Do(\delta, s, s')$ — nondeterministic choice of parameters
- $Do(\textbf{if } \phi \textbf{ then } \delta_1 \textbf{ else } \delta_2, s, s') \stackrel{\text{def}}{=}$

  $$(\phi[s] \rightarrow \text{Do}(\delta_1, s, s')) \wedge (\neg\phi[s] \rightarrow \text{Do}(\delta_2, s, s'))$$

- $Do(\delta^*, s, s') \stackrel{\text{def}}{=}$ — nondeterministic iteration

  $$\begin{array}{l} \forall P[(\forall s_1\, P(s_1, s_1) \rightarrow \\ \quad \forall s_1, s_2, s_3[P(s_1, s_2) \wedge Do(\delta, s_2, s_3) \\ \qquad \rightarrow P(s_1, s_3)] \\ \quad \rightarrow P(s, s')] \end{array}$$

- $Do(\textbf{while } \phi \textbf{ do } \delta, s, s') \stackrel{\text{def}}{=}$

  $$\begin{array}{l} \forall P(\\ \quad (\forall s_1\, \neg\phi[s_1] \rightarrow P(s_1, s_1)) \wedge \\ \quad (\forall s_1, s_2, s_3\, (\phi[s_1] \wedge Do(A, s_1, s_2) \wedge \\ \qquad P(s_2, s_3)) \\ \quad \rightarrow P(s_1, s_3)) \\ \qquad) \rightarrow P(s, s') \end{array}$$

The notation $\phi[s]$ means that a situation argument is added to all fluents in ϕ, if one is missing. The definition of while loops could be simplified by utilizing the definition of nondeterministic iteration. Note that the definition of while loops and nondeterministic iteration have to appeal to second order logic.

Second order logic is also need to define the procedure construct:

- (Recursive) procedure P with formal parameters $x_1, \ldots, x_n$ and actual parameters $t_1, \ldots, t_n$ and body α is defined as follows:

  $$\begin{array}{l} Do([\textbf{proc} \\ \quad P(x_1, \ldots, x_n)\ \alpha\ \textbf{end}](t_1, \ldots, t_n), s_1, s_2) \stackrel{\text{def}}{=} \\ \quad (\forall P)\{(\forall x_1, \ldots, x_n, s_1', s_2') \\ \qquad [P(x_1, \ldots, x_n, s_1', s_2') \equiv Do(\alpha, s_1', s_2')] \\ \qquad\quad \rightarrow P(t_1, \ldots, t_n, s_1, s_2)\}. \end{array}$$

Executing procedure P on actual parameters $t_1, \ldots, t_n$ takes you from s_1 to s_2 iff $(t_1, \ldots, t_n, s_1, s_2)$ is in the smallest set of tuples $(x_1, \ldots, x_n, s_1', s_2')$ such that executing α on $x_1, \ldots, x_n$ takes you from s_1' to s_2'.

In [10], a full discussion of these definitions as well as a Prolog interpreter for GOLOG may be found. Even though the correct definition of the complex actions does in certain cases use second-order logic, the interpreter is very simple.

5 Run-time Layer

The complex action macros of GOLOG can now be used to define the run-time layer. But we first need some additional actions. These are:

- RIGHTCLICK(x)
- MIDDLECLICK(x)
- LEFTCLICK(x)

The intended meaning of RIGHTCLICK(x) is that the user has clicked the button x with the right side of the mouse indicating that the request is for a traverse action (either traverse to source or traverse to sink). The intended meaning of LEFTCLICK(x) is that the user has clicked the button x with the left side of the mouse indicating that the request is to display the link attributes. A MIDDLECLICK(x) action is an indication that the user wants to have the node (current node) attributes displayed. These actions are really exogenous actions in that they are not performed by the program (agent), but rather by the outside world (user of the program). The choice of which one occurs is external and not part of the logic model.

Additionally, we need the following action:

- DEACTIVATE(x)

After executing the action requested by the user, the system will need to deactivate the button with this action.

Also needed are three fluents to indicate the status of the button.

- ActiveButtonRight(x)
- ActiveButtonMiddle(x)
- ActiveButtonLeft(x)

The successor state axioms for these fluents are relatively simple:

$$\begin{aligned}&\text{Poss}(a,s) \rightarrow [\text{ActiveButtonRight}(x, do(a,s)) \equiv \\ &a = \text{rightclick}(x) \vee \\ &(a \neq \text{deactivate}(x) \wedge \\ &\quad \text{ActiveButtonRight}(x,s))]\end{aligned} \tag{18}$$

$$\begin{aligned}&\text{Poss}(a,s) \rightarrow [\text{ActiveButtonLeft}(x, do(a,s)) \equiv \\ &a = \text{rightclick}(x) \vee \\ &(a \neq \text{deactivate}(x) \wedge \\ &\quad \text{ActiveButtonLeft}(x,s))]\end{aligned} \tag{19}$$

$$\begin{aligned}&\text{Poss}(a,s) \rightarrow [\text{ActiveButtonMiddle}(x, do(a,s)) \equiv \\ &a = \text{rightclick}(x) \vee \\ &(a \neq \text{deactivate}(x) \wedge \\ &\quad \text{ActiveButtonMiddle}(x,s))]\end{aligned} \tag{20}$$

These axioms specify that the button is active (in the appropriate fashion — right, left, or middle) if the previous action was a click (of the appropriate type — right, left, or middle), or if the previous action was not a deactivate action and the button was on (in the appropriate fashion) in the previous situation.

The following complex actions define the run-time environment. The procedure FIND_BUTTON determines what button is pressed and then calls the procedure PROCESS_COMMAND with the button as an argument and finally deactivates the button.

$$\begin{aligned}&\textbf{proc}\ \text{FIND\_BUTTON}(n) \\ &(\pi\, n)[(\text{ActivebuttonRight}(n) \vee \\ &\quad \text{ActiveButtonLeft}(n) \vee \\ &\quad\quad \text{ActiveButtonMiddle}(n))?; \\ &\quad \text{PROCESS\_COMMAND}(n); \\ &\quad \text{de\_activate}(n)] \\ &\textbf{end}.\end{aligned} \tag{21}$$

The procedure PROCESS_COMMAND takes a button as an argument. If the middle mouse key has been clicked, then the attributes of the current node are displayed. Otherwise, if the left mouse button has been clicked, the attributes of the link associated with the button are displayed. Otherwise, the procedure TRAVERSE is called with the link as an argument.

$$\begin{aligned}&\textbf{proc}\ \text{PROCESS\_COMMAND}(n) \\ &\quad \textbf{if}\ \text{ActiveButtonMiddle}(n) \\ &\quad\quad \textbf{then}\ (\pi\, u)[\text{Current\_node}(u)?; \\ &\quad\quad\quad \text{display\_node\_attr}(u) \\ &\quad\quad \textbf{else}\ [\textbf{if}\ \text{ActiveButtonLeft}(n) \\ &\quad\quad\quad \textbf{then}\ (\pi\, y)\text{Button}(n,y,z)?; \\ &\quad\quad\quad\quad \text{display\_link\_attr}(y) \\ &\quad\quad\quad \textbf{else}(\pi\, y)\text{Button}(n,y,z)?; \\ &\quad\quad\quad\quad \text{TRAVERSE}(y)]] \\ &\textbf{end}.\end{aligned} \tag{22}$$

The procedure TRAVERSE takes as an argument the link that is associated with the button that was activated. If the current node is the source node of that link, then a traverse to sink is performed. Otherwise a traverse to source is performed.

$$\begin{aligned}&\textbf{proc}\ \text{TRAVERSE}(x)\ (\pi\, y,z)[\text{Link}(x,y,z,\_,\_,\_)?; \\ &\quad \textbf{if}\ \text{Currentnode}(z) \\ &\quad\quad \textbf{then}\ \text{traverse\_source}(x) \\ &\quad\quad\quad \textbf{else}\ \text{traverse\_sink}(x)] \\ &\textbf{end}.\end{aligned} \tag{23}$$

The procedure CONTROL is the top level routine. It first calls OPEN_SESSION and then request. Note that request is not really definable within the logic model because it is basically a request for an exogenous action. But it is easily implemented as a prompt to the user to enter an action and then the execution of that action. Then the procedure loops as long as button 0 is not activated. This button is taken to represent the command to exit the program. Each time through the loop, first there is a call to FIND_BUTTON and then another request.

$$\begin{aligned}&\textbf{proc}\ \text{CONTROL} \\ &\quad \text{OPEN\_SESSION}; \\ &\quad \text{request}; \\ &\quad [\textbf{while}\ \neg\, \text{Active}(0)\ \textbf{do} \\ &\quad\quad \text{SERVE\_BUTTON}; \\ &\quad\quad \text{request}\ ]; \\ &\quad \textit{CLOSE\_SESSION} \\ &\textbf{end}.\end{aligned} \tag{24}$$

The procedure CLOSE_SESSION is quite simple. It calls the commands (undefined in our model) clearscreen and terminate.

$$\begin{aligned}&\textbf{proc}\ \text{CLOSE\_SESSION} \\ &\quad\quad \text{clearscreen}; \\ &\quad\quad \text{terminate} \\ &\textbf{end}.\end{aligned} \tag{25}$$

The procedure OPEN_SESSION, first performs clearscreen and then displays the current node.

$$\begin{array}{l}\textbf{proc}\ \text{OPEN\_SESSION}\\ \qquad \text{clearscreen};\\ \qquad (\pi\ n)[\textsc{Current\_node}(n)?;\\ \qquad \textsc{make\_display\_text}(n)]\\ \textbf{end}.\end{array} \tag{26}$$

Note that an interpreter exists for GOLOG code such as that presented here [10].

6 Induction

One may also want to reason about the global properties of a particular axiomatization. An example is to prove that dangling links will never occur given a particular axiomatization. A dangling link is a link where no sink node exists, maybe because someone has deleted it without updating the database. Dexter strongly prohibits this, but real-life implementations often ignore the issue.

The problem is, given a situation calculus axiomatization, to determine whether a particular sentence is true in all possible situations. This demands reasoning by induction. The purpose of this section is to utilize the foundational axioms for the situation calculus [11, 16] so that properties such as the absence of dangling links may be proven for a particular axiomatization.

Up to this point we have implicitly been using a many-sorted language. The two domain independent sorts are *situation* and *action*. The sort *action* is for primitive actions. The unique situation constant symbol, S_0, is a member of the sort *situation*. Additional situations are generated by the binary function $do : action \times situation \rightarrow situation$.

The initial situation, S_0 is like the number 0 in Peano arithmetic. Unlike Peano arithmetic which has a unique successor function, we have a family of successor functions modeled by the binary function *do*. Lin and Reiter [11] have, on analogy with the Peano axioms for number theory, developed a set of foundational axioms for the situation calculus. The axioms are as follows:

$$S_0 \neq do(a, s) \tag{27}$$

$$do(a_1, s_1) = do(a_2, s_2) \rightarrow a_1 = a_2 \wedge s_1 = s_2 \tag{28}$$

$$\begin{array}{l}(\forall P).\{P(S_0) \wedge [(\forall a, s).[P(s) \rightarrow P(do(a, s))]]\}\\ \quad \rightarrow (\forall s)P(s)]\end{array} \tag{29}$$

Axiom (29) is a second order way of limiting the sort *situation* to the smallest set containing S_0, and closed under the application of the function *do* to an action and a situation. Any model of these axioms will have its domain of situations isomorphic to the smallest set $\mathcal{S}$ satisfying:

1. $S_0 \in \mathcal{S}$.
2. If $S \in \mathcal{S}$, and $A \in \mathcal{A}$, then $do(A, S) \in \mathcal{S}$, where $\mathcal{A}$ is the domain of actions in the model.

These axioms say that the tree of situations is really a tree. There are no cycles and there is no merging.

There are two additional axioms:

$$\neg s < S_0 \tag{30}$$

$$s < do(a, s') \equiv (Poss(a, s') \wedge s \leq s') \tag{31}$$

where $s \leq s'$ is shorthand for $s < s' \vee s = s'$. These five axioms are *domain independent*. They provide the basic properties of situations in any domain specific axiomatization of particular fluents and actions.

We want to reason about situations reachable by an executable sequence of actions. Intuitively, $s \leq s'$ holds if and only if there is a sequence of zero or more *executable* actions which lead from situation s to s'. An action is executable if the action's preconditions are true in the situation in which the action is to be performed.

One of the consequences of the foundational axioms is the following[7]

$$\begin{array}{l}(\forall P)[P(S_0) \wedge (\forall a, s)(P(s) \wedge S_0 \leq s \wedge Poss(a, s)\\ \qquad \rightarrow P(do(a, s))) \rightarrow\\ \qquad\quad (\forall s).S_0 \leq s \rightarrow P(s)].\end{array} \tag{32}$$

This principle of induction enables us to prove properties that hold over all situations that are possible.

Sentence 33 expresses the property that there are no dangling links.

$$\begin{array}{l}\forall\, s\ \exists\, u, v, w, x, y, z\ \textsc{Link}(u, v, w, x, y, z, s)\ \rightarrow\\ \qquad \exists\, x, m, n\ \textsc{Node}(x, m, n, s)\ \wedge\ x = w\end{array} \tag{33}$$

Using 32, one can prove that this sentence is an inductive consequence of the axiomatization presented in this paper.

7 Generalized Hypertext

We are attracted to the situation calculus primarily as a way to model the dynamic aspects of hypertext systems. In particular we are interested in hypertext support for information systems that dynamically generate document (node) content. Bieber and Kimbrough's model of generalized hypertext employs

[7] See [11, 16] for a full discussion.

logical mapping rules called bridge laws which dynamically (at run-time) map application components (application commands and the results of executing commands) to hypertext components (nodes, links and buttons). They illustrate their approach with Maxi, a generalized version of Max.

Following [1], we use predicates $\textsc{GhtNode}(x, y, s)$ to represent that x is a generalized node with content description y and $\textsc{GhtLink}(u, v, w, z, y, z, s)$ to represent that u is a generalized link. Sentence 34 adapts a bridge law for nodes from [1] to the approach being taken here.

$$\begin{aligned}\textsc{Model}(x.y, z) \ \wedge \ &\textsc{Source}(x, w) \wedge \\ &\textsc{Description}(x, v) \\ \rightarrow \textsc{GhtNode}(x, &[[\textsc{math-ex}, z], \\ &[\textsc{source}, w], \\ &[\textsc{descr}, v], \\ &[\textsc{type}, \textsc{n-model}], \\ &[\textsc{model-type}, y]]), \\ &S_0)\end{aligned} \tag{34}$$

In this example, the predicates SOURCE, MODEL, and DESCRIPTION all belong to the application domain of Maxi.

Sentence 35 adapts a bridge law for links from [1] to the approach being taken here.

$$\begin{aligned}\textsc{Model}(u, \_, \_) \ \wedge \ &\textsc{Available\_Commands}(u, z) \\ \rightarrow \textsc{GhtLink}(u, v, w, &[[\textsc{l-type}, \textsc{model}], \\ &[\textsc{owner}, \mathrm{TEFA}]], \_, z, S_0)\end{aligned} \tag{35}$$

Here again, the the predicates MODEL and AVAILABLE_COMMANDS both belong to the application domain of Maxi.

Note that the bridge laws here only operate on situation S_0. We are assuming that the application domain is static during the period that the hypertext system is operating. Eliminating this assumption would mean that we would want the S_0 in the bridge laws to be replaced by a universally quantified s. This would then be a state constraint and would in general violate the requirement for Reiter's simple solution to the frame problem. So, in this context, the frame problem returns.

Our future research concerns finding ways to fully model the interaction between the hypertext systems and the information systems application(s) it supports. This will involve handling the types of state constraints that arise in our intended applications, utilizing the work of of [11].

8 Conclusions and Future Work

We have developed a situation calculus model for hypertext systems. This model captures the dynamic aspects of hypertext – the changes that occur in moving from one situation to another when commands are executed. It has the advantage of being able to utilize an existing interpreter for the language in which the model is expressed.

Stand-alone hypertext systems and those providing hypertext services to other applications have many dynamic features, which most hypertext models do not adequately represent. Using the situation calculus could prove an effective tool for representing these aspects and may very well allow us to fully model such systems and implement them. We hope to continue this line of research and do this.

In particular, we can now add a new aspect to the hypertext models that have been proposed: that of the determination of hypertext features based on the context of the current session. Being able to formally specify the rules by which some links are to be activated or some content is to be included into a hypertext document is an important facility that can allow author to create hypertext that automatically adapt to the situation, the user's reactions, and the overall context of the navigation. External linkbases, dynamic computation-oriented information systems, and interfaces to expert systems could profit by this ability.

Acknowledgements

This research has been supported by the NASA JOVE faculty fellowship program, the New Jersey Center for Multimedia Research, the New Jersey Institute of Technology under SBR grant 421250, the New Jersey Department of Transportation, and the National Center for Transportation and Industrial Productivity at NJIT.

References

[1] Michael P. Bieber and Steven O. Kimbrough. On the logic of generalized hypertext. *Decision Support Systems*, 11:241–257, 1994.

[2] Alex Borgida, John Mylopoulos, and Raymond Reiter. On the frame problem in procedure specifications, 1993. presented at the Fifteenth International Conference on Software Engineering, Baltimore, Maryland.

[3] M.C. Buchanan and P.T. Zellweger. Specifying temporal behavior in hypermedia documents. In *ECHT '92 Proceedings*, pages 262–271, Milano, November 1992. ACM Press.

[4] R. Furuta and P.D. Stotts. Programmable browsing semantics in trellis. In *Hypertext '89 Proceed-*

ings, pages 27–42, Pittsburgh, November 1989. ACM Press.

[5] M. Gelfond, V. Lifschitz, and A. Rabinov. What are the limitations of the situation calculus? In *Automated Reasoning: Essays in Honor of Woody Bledsoe*, pages 167–179. Kluwer Academic Publishers, Dordrecht, 1996.

[6] A. R. Haas. The case for domain-specific frame axioms. In F. M. Brown, editor, *The Frame Problem in Artificial Intelligence. Proceedings of the 1987 Workshop*, pages 343–348. Morgan Kaufmann Publishers, Inc., San Mateo, California, 1987.

[7] Frank Halasz and Mayer Schwartz. The dexter hypertext reference model. *Communications of the ACM*, pages 30–39, 1994.

[8] Yves Lespérance, Hector Levesque, Fangzhen Lin, Daniel Marcu, Ray Reiter, and Richard Scherl. A logical approach to high-level robot programming — a progress report. Appears in *Control of the Physical World by Intelligent Systems*, Working Notes of the 1994 AAAI Fall Symposium, New Orleans, LA, November 1994.

[9] Yves Lespérance, Hector J. Levesque, F. Lin, Daniel Marcu, Raymond Reiter, and Richard B. Scherl. Foundations of a logical approach to agent programming. In *Proceedings of the IJCAI-95 Workshop on Agent Theories, Architectures, and Languages*, August 1995.

[10] Hector Levesque, Raymond Reiter, Yves Lespérance, Fangzhen Lin, and Richard B. Scherl. Golog: A logic programming language for dynamic domains. *Journal of Logic Programming*, 1997. to appear.

[11] Fangzhen Lin and Raymond Reiter. State constraints revisited. *Journal of Logic and Computation*, 4(5):655–678, 1994.

[12] J. McCarthy and P. 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, UK, 1969.

[13] E.P.D. Pednault. ADL: exploring the middle ground between STRIPS and the situation calculus. In R.J. Brachman, H. Levesque, and R. Reiter, editors, *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning*, pages 324–332. Morgan Kaufmann Publishers, Inc., San Mateo, California, 1989.

[14] J.A. Pinto. *Temporal Reasoning in the Situation Calculus*. PhD thesis, Department of Computer Science, University of Toronto, Toronto, Ontario, 1994. Available as technical report KRR-TR-94-1.

[15] Raymond 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.

[16] Raymond Reiter. Proving properties of states in the situation calculus. *Artificial Intelligence*, pages 337–351, December 1993.

[17] Raymond Reiter. On specifying database updates. *The Journal of Logic Programming*, pages 53–91, 1995.

[18] L.K. Schubert. Monotonic solution of the frame problem in the situation calculus: an efficient method for worlds with fully specified actions. In H. E. Kyberg, R.P. Loui, and G.N. Carlson, editors, *Knowledge Representation and Defeasible Reasoning*, pages 23–67. Kluwer Academic Press, Boston, Mass., 1990.

[19] Murray Shanahan. *Solving the Frame Problem: A Mathematical Investigation of the Common Sense Law of Inertia*. MIT Press, Cambridge, Massachusetts, 1997.

[20] F.W. Tompa, G.E. Blake, and D.R. Raymond. Hypertext by link-resolving components. In *Hypertext '93 Proceedings*, pages 118–130, Seattle, November 1993. ACM Press.