THE SITUATION CALCULUS AND
HEHNER'S PROGRAMMING THEORY:
HARMONIZATION THROUGH REIFICATION

by

Patrick Prémont

A thesis submitted in conformity with the requirements
for the degree of Master of Science
Graduate Department of Computer Science
University of Toronto

© Copyright by Patrick Prémont 1997
The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author’s permission.

L’auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L’auteur conserve la propriété du droit d’auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.
Abstract

Acquiring the general ability to reason formally about the effect of actions is a task of great importance. It is at the heart of knowledge representation in artificial intelligence, and formal programming methods in software engineering.

This shared goal leads to similarities in the formalisms developed in the two fields. Here we explore the similarities between the situation calculus from artificial intelligence and a programming theory by Eric C. R. Hehner. These similarities provide a bridge between the two formalisms which may be used to share results. By making some modifications to the formalisms, such as reifying some of their concepts, we widen this bridge and enable a more harmonious exchange between the two fields.
Acknowledgments

I would like to thank my supervisors: Ray Reiter, for interesting discussions about the situation calculus and this work, and Eric Hehner, especially for his wonderful course on his programming theory, which gave me a strong appreciation for the importance of logic in programming.

I acknowledge the financial support of the National Science and Engineering Research Council of Canada, and I am very grateful for the love and support extended by my family and friends.
# Contents

1 Introduction .................................................. 1
   1.1 Motivation ................................................. 1
   1.2 Strategy and Thesis Overview .......................... 2

2 The Situation Calculus and Hehner's Theory ............... 5
   2.1 Hehner's Theory of Programming ....................... 5
   2.2 The Situation Calculus ................................. 6
       2.2.1 GOLOG ............................................. 7
       2.2.2 Variants of the Situation Calculus ............... 8

3 Comparison of the Two Formalisms .................................. 9
   3.1 Logic and Syntax .......................................... 9
       3.1.1 Translation ........................................ 10
       3.1.2 Foundational Axioms of the Situation Calculus ........ 11
   3.2 States and Fluents ....................................... 13
       3.2.1 Referring to Parts of the State ................. 13
       3.2.2 States and Situations ........................... 15
   3.3 Actions .................................................... 16
       3.3.1 Logical Nature of Actions in the Situation Calculus .......... 16
       3.3.2 Logical Nature of Actions in Hehner's Theory ........... 17
       3.3.3 Specification of the Effect of Actions ............ 20
   3.4 Time and Concurrency .................................. 35

4 Increasing the expressiveness ................................. 38
### 4. Reifying Expressions

#### 4.1 Reifying Actions

#### 4.2 Reifying Expressions

- 4.2.1 Benefits
- 4.2.2 Axiomatization of reified expressions

#### 4.3 Reasoning with multiple representations

#### 4.4 Using situations in Hehner’s theory

#### 4.5 Recursion and consistency

### 5. Summary and Future Work
Chapter 1

Introduction

1.1 Motivation

The ability to reason skillfully about actions and their effect is of undeniable importance. The use of formalisms can increase this ability considerably. Here we consider particularly general logical formalisms which may be used to reason about actions in many different contexts.

One field where such formalisms are of interest is knowledge representation in artificial intelligence (AI). There the problem of designing machines able to reason about the world, in all its generality, leads to the development of general formalisms to reason about actions and their effect, such as the situation calculus. This calculus has been part of artificial intelligence for a long time [McCarthy63], but has recently received more interest as a powerful foundation for the development of serious applications such as the control of robots by rational agents\(^1\).

Another field is software engineering, where theories of programming are developed to help programmers create programs, reason about their properties, and prove that they meet the desired specifications. A particularly simple yet general formalism is Hehner's theory of programming [Hehner93]. Its general approach is also used by other researchers interested in programming from formal specifications [Morgan90]. Such a formalism is

\(^1\)Work in this area continues in the Cognitive Robotics Group at the Department of Computer Science of the University of Toronto. For an example see [Lespérance94].
programs may be used to describe all sorts of dynamic systems.

These two fields and formalisms are essentially concerned with the same problem: to obtain a formal understanding of the effect of actions in order to automatically create plans or programs that perform the appropriate sequence of actions to achieve given goals. In the case of software engineering the requirement that programs be created automatically is often relaxed, admitting a measure of human intervention, but even then the amount of automation or regularity in a design method is considered a measure of the quality of the method.

Since they were created to address similar problems, we expect to find many similarities between the two formalisms.

Our goal here is to contribute to the two fields by enabling productive exchanges between them (by relating their formalisms) and by increasing the expressiveness of the two formalisms.

1.2 Strategy and Thesis Overview

To enable exchanges between the fields we expose the equivalences, similarities and differences between the two formalisms. By modifying the formalisms, without changing their fundamental characteristics, we can increase their similarities and harmonize them so that work done in either formalism can be more easily or trivially translated to the other formalism. Some of the changes will be performed at the level of the logic and the syntax, but the most interesting changes will be the reification of concepts found in the formalisms. Reification replaces extra-logical concepts by first class objects which may then be described by terms within the logic. This strategy may be visualized using the figure that follows, where the points S and H represent the situation calculus and Hehner's theory, respectively.
The changes performed at the level of the logic and the syntax consist in expressing the situation calculus in the logic of Hehner's theory. This takes us from point S to point S'. This transition involves the reification of logical concepts such as booleans. Then, through successive reifications, the two formalisms are brought together until they are almost identical. This takes us from S' and H to S'' and H'. By composing and reversing some of these transformations, we can create a path between S and H that is almost complete. The path is complete when the results that are to be moved between the formalisms do not rely on the rare differences that remain between S'' and H', and do not violate the reasonable assumptions the underlie the transformations.

Increasing the expressiveness of the formalisms will mostly be a result of the reifications. As a result of the availability of new objects, it will become possible to express new statements within the logic.

In chapter 2 we briefly introduce the two formalisms. In chapter 3 we study, compare and occasionally modify their various aspects. Section 3.1 studies the logic and syntax and effects the transition from S to S' in the figure above. The other sections of chapter 3 study states, fluents, actions, time and concurrency while effecting a good part of the transitions from S' and H to S'' to H'. In chapter 4 we complete the reification of programs and examine some of its applications. We also present an application of reified fluents that allows reasoning with multiple representations, possibly at different levels of abstraction.
more work is required.
Chapter 2

The Situation Calculus and Hehner's Theory

We first provide a brief introduction to the two formalisms since the reader may not be familiar with both. Additional information about the formalisms will be provided progressively in the following chapters.

2.1 Hehner's Theory of Programming

Hehner's theory of programming [Hehner93] provides axioms and proof rules which allow us to express and reason about states, programs and specifications.

In this formalism, states are represented as lists. Although it would be possible to refer to components of the state by their indexes in the list, a convention is established that allows us to refer to the components through distinct names called state variables.

Specifications are represented as boolean expressions that relate initial and final states. The boolean expression of a specification is true if and only if the transition from the initial state to the final state is permitted according to the specification.

Programs are implemented specifications, which may be executed on a computer. The boolean expression of a program is true of an initial state and a final state if and only if the execution of the program may result in the transition between these two states.

So the expression of a program characterizes possible transitions and the expression
respects a specification, it is sufficient to prove that all possible transitions are permitted, or in other words, that the boolean expression of the program logically implies the boolean expression of the specification. This constitutes refinement of the specification by the program. Refinement proofs are an important application of the theory.

The theory includes a large number of axioms that define objects such as booleans, numbers, characters, bunches, sets, lists, strings, functions, stacks and queues. We do not use all of these objects here, and we do not reproduce the axioms for objects and operators since most of them have standard semantics. We briefly describe bunches and the special relationship between lists and functions as it becomes necessary in the following sections.

2.2 The Situation Calculus

The situation calculus is a first order language that has terms to represent actions and situations, and functions to represent fluents.

Situations are histories formed by the sequences of actions that have occurred since an initial situation. A special function do takes an action \(a\) and a situation \(s\), and returns the situation that results from the execution of \(a\) in situation \(s\). By iterating this process from situation \(s_0\) through a series of actions \(a_1, a_2, \ldots, a_n\), we obtain the situation \(s_n = do(a_n, do(a_{n-1}, do(\ldots do(a_1, s_0)\ldots)))\).

The state of a system in a situation is specified through fluents. A fluent is a function that takes a situation as one of its arguments. To describe the state in situation \(s\), one specifies the values taken by the different fluents in situation \(s\). For example:

\[
\begin{align*}
\text{waterLevel}(\text{tank1}, s) &= 2.3 \\
\text{waterTemperature}(\text{tank1}, s) &= 56.2 \\
&\vdots \\
\end{align*}
\]

A group of axioms, called Poss axioms, is used to determine whether or not an action may be performed from a situation. There is one Poss axiom for each action. For example, the specification of an action slide would require the user to say under which conditions
Using axioms that specify the effect of actions, it is possible to infer from the state of a system in the initial situation $s_0$, and from the sequence of actions $a_1, a_2, \ldots, a_n$, the state of the system in state $s_n$. The axioms that specify the effect of actions are called successor state axioms. There is one such axiom for each fluent, and it specifies the value of the fluent in the final situation by referring to the action that was executed, and to the value of fluents in the initial situation. For example a fluent broken could remain false until the corresponding fragile object is dropped through action drop. This would be expressed as the following successor state axiom:

$$Poss(a, s) \supset (\text{broken(do(a, s)))} \equiv a = \text{drop} \lor \text{broken(s)}).$$

### 2.2.1 GOLOG

In the situation calculus all actions are primitive, they are not expressed in terms of other actions. The formalism in itself does not provide a way of creating complex actions, or programs, from primitive actions.

To extend the expressiveness of the formalism to complex actions, a programming language called GOLOG [Levesque94] has been developed by the Cognitive Robotics Group at the Department of Computer Science of the University of Toronto.

Its semantics are defined through an extra-logical predicate $Do$ that takes a program, an initial situation and a final situation as arguments. The predicate is seen as a macro that expands to a situation calculus formula characterizing the relationship between the initial and final situations for a particular program. The formula holds if and only if the transition between the situations is one of the possible effects of the program.

Because the predicate $Do$ is not a predicate in the main logic, but only in the logic of the meta-level, GOLOG is not a simple extension of the situation calculus with new axioms, it rather defines a new layer on top of the situation calculus.
Within the context of the situation calculus ontology there are possible variations on the properties of situations, fluents, and actions.

For example in [Elkan96] fluents are not treated as functions but are given a first class status. They become objects on which functions may be applied and quantification may be performed within first-order logic; they are reified.

Other variants [Pinto94] [Reiter96] allow the simultaneous execution of many primitive actions, and introduce the concept of time in the situation calculus. A variant also includes the notion of agent [Lespérance96].

Later, when we refer to the situation calculus, it should be understood to mean the variant used in [Reiter91], which does not include the special features of the variants mentioned above.
Chapter 3

Comparison of the Two Formalisms

3.1 Logic and Syntax

To some extent, the differences between the two formalisms are due to their different logics and syntax. In order to perceive more clearly the more fundamental differences and the similarities, we adopt a standard logic and a standard syntax in which both theories will be expressed.

The situation calculus restricts itself to a many-sorted first order logic\(^1\) while the algebra of Hehner’s theory supports higher order logic. We standardize on the latter because of its greater generality. Also the situation calculus has significantly less constructs and therefore selecting the situation calculus logic would involve the translation of these numerous constructs to a more restrictive logic.

Translating the situation calculus formulae to another logic must be done with care to ensure that the semantics are preserved. However in this case we can proceed safely since the higher order logic in Hehner’s theory implements all of first order logic’s constructs such as functions, predicates and quantifiers, with their standard semantics. One concern that remains is the translation of sorts, which act as ranges for variables. But these too may be translated safely. Hehner’s theory provides bunches, which are the contents of sets and are used, together with the bunch inclusion predicate “:”, to specify the range of sorts.

\(^1\)Occasionally second order logic sentences are used, such as one of the foundational axioms presented in section 3.1.2. In this work we loosely refer to the logic of the formalism as first order, neglecting those rare exceptions.
Although we use a higher order logic, quantification over functions will not be used unless it is necessary. This should keep to a minimum our reliance on higher order logic and facilitate an eventual interpretation and use of our results in a first order logic, which could become necessary for performance reasons in a particular automated reasoning application.

Having selected the higher order logic, it becomes natural to standardize on the corresponding syntax. Also the syntax of Hehner's theory is compact, a valuable attribute when performing manually the complex proofs of properties of programs.

3.1.1 Translation

The translation of situation calculus terms and formulae to the logic and syntax of Hehner's theory involves the following modifications:

- Function applications are denoted by juxtaposition, which is left-associative. Parentheses are required only when a different association is intended. For example $do(a_2, do(a_1, s_0))$ becomes $do a_2 (do a_1 s_0)$.

In Hehner's theory functions are curried: functions of more than one argument are represented as unary functions. These take the first argument and then return functions that accept the rest of the arguments in the same incremental manner. So $do a s$ is equivalent to $(do a) s$. However this semantic difference in the nature of functions will not affect the validity of sentences translated from the situation calculus: within these sentences functions are always applied to all of their arguments.

- The logical implication symbol becomes $\Rightarrow$, and the logical equivalence symbol becomes equality $(\equiv)$. Equality represents logical equivalence because in Hehner's theory truth values (the booleans $T$ and $\bot$) are treated like any other object. Another consequence of the treatment of $T$ and $\bot$ as ordinary values is that predicates become simply functions that return boolean values. Note that the precedence of the operator $\equiv$ is the same as the precedence of operator $>$, not $\Rightarrow$, so $a = a \land b$
• For each sort a bunch is defined that consists of the same elements as the sort. Free variables of some sort become free variables constrained to values within the corresponding bunch. So \( do(a, s) = s' \), where \( s \) and \( s' \) are variables of sort situation and \( a \) is a variable of sort action, becomes \( do \ a \ s = s' \land a: act \land s: sit \land s': sit \). Bunches \( act \) and \( sit \) correspond to the sorts action and situation, respectively. These bunches must be axiomatized to contain the same elements as the corresponding sorts. The symbol \( \cdot \) designates a binary infix function that take two bunches and returns \( \top \) if all elements contained in the first bunch occur in the second bunch, and \( \bot \) otherwise.

A restriction such as \( a : act \) is not strictly sufficient to preserve the first order logic semantics: \( a \) must also be constrained to contain only one element. This may be asserted by conjoining \( \#a = 1 \), where \( \# \) is a function that returns the cardinality of a bunch. However both restrictions \( (a: act \land \#a = 1) \) may be omitted when it is obvious from the context that a variable is intended to designate an element of a sort.

Sorted quantification is translated similarly by specifying explicitly the range of quantification as the appropriate bunch. For example \( \forall a. E \) becomes \( \forall a: act \cdot E \). But here again the range specification may be omitted when it is obvious from the context that the variable ranges over a particular sort.

It should be noted that these transformations correspond to the reification of functions, booleans and sorts, respectively.

### 3.1.2 Foundational Axioms of the Situation Calculus

The foundational axioms of the situation calculus [Lin94] axiomatize the situations, the function \( do \) and an ordering relation \(< \) on situations. Using the conventions specified above, derived from Hehner's theory, the foundational axioms may be expressed as:
\textit{do: act} \rightarrow \textit{sit} \rightarrow \textit{sit} \quad (3.1)

\textit{S}_0: \textit{sit} \quad (3.2)

\textit{S}_0 \neq \textit{do a s} \quad (3.3)

\textit{do a}_1 \textit{ s}_1 = \textit{do a}_2 \textit{ s}_2 \Rightarrow \textit{a}_1 = \textit{a}_2 \land \textit{s}_1 = \textit{s}_2 \quad (3.4)

\forall \textit{P}: \textit{sit} \rightarrow \textit{bool} \cdot (\textit{P} \textit{S}_0 \land (\forall \textit{a} \cdot \forall \textit{s} \cdot \textit{P} \textit{s} \Rightarrow \textit{P} \textit{(do a s)}) \Rightarrow \forall \textit{s} \cdot \textit{P} \textit{s}) \quad (3.5)

\neg \textit{s} < \textit{S}_0 \quad (3.6)

(\textit{s} < \textit{do a s'}) = (\textit{Poss a s'} \land (\textit{s} < \textit{s'} \lor \textit{s} = \textit{s'})). \quad (3.7)

Axioms (3.1) and (3.2) are not axioms in the situation calculus but meta-level assertions. Here they can be expressed as axioms. The symbol $\rightarrow$ denotes a left-associative infix binary function that take two bunches $A$ and $B$ and returns the bunch of functions whose domain includes $A$ and whose range is included in $B$. Axioms (3.1) and (3.2) are construction axioms for situations and (3.5) is the second-order induction axiom for situations. These axioms define the bunch \textit{sit}, in terms of the bunch of actions \textit{act}, which is domain dependent. Axioms (3.3) and (3.4) are unique names axioms for situations, which ensure that no two situations share the same representation in terms of \textit{S}_0 and \textit{do}. Finally axioms (3.6) and (3.7) provide the ordering relation $<$ on situations.

These translated foundational axioms, and the situation calculus theorems that follow from them, allow us to use all of the situation calculus techniques, but within the same logic and syntax used by Hehner's theory. This may not be of great interest in itself, but it will facilitate the analysis of the possible mutual contributions of the two formalisms. The programming language based on the situation calculus need not be translated since it is based on the predicate \textit{Do} which simply expands programs to situation calculus sentences. By applying the translation to the sentences generated by \textit{Do}, we obtain a definition of GOLOG based on this new version of the situation calculus. But at this point GOLOG remains a separate layer. Later we translate GOLOG itself to the new logic, when the representations of action of both formalisms have been compared.
Although the situations found in the situation calculus are different from the states found in Hehner’s theory, their abstract properties are very similar. By viewing states and situations as abstract data types, we can see that each provides a way to refer to the state of components of the dynamic system under consideration.

### 3.2.1 Referring to Parts of the State

Let us consider how each formalism allows us to access a part of the state. To refer to the value of a fluent or state variable \( \text{slope} \), in the situation or state \( s \), we would proceed as follows:

In the situation calculus we would apply the fluent \( \text{slope} \) to the situation \( s \).

\[ \text{slope } s \]

In slightly different variants of the situation calculus, such as the ones used by Elkan in [Elkan96] and Pinto in [Pinto94], a special function \( \text{Holds} \) is used. It takes a fluent and a situation as arguments. However, as the name suggests, \( \text{Holds} \) gives access only to propositional fluents, by telling us whether or not a proposition holds in a situation. If we want to allow access to functional fluents using a similar technique, we should name the special function \( \text{Val} \). It would return the value of the specified fluent in the specified situation. The returned value would be a boolean for propositional fluents.

\[ \text{Val } \text{slope } s \]

In Hehner’s theory, where states are lists, we would access the list element corresponding to the state variable \( \text{slope} \). If the index of variable \( \text{slope} \) were \( \text{index}_{\text{slope}} \), we would apply the state \( s \) to \( \text{index}_{\text{slope}} \). This is because in Hehner’s theory the lists are functions from a range of integers, the indexes of the list elements, to the corresponding list elements.

\[ s \ \text{index}_{\text{slope}} \]

However the use of the indexes of state variables is cumbersome and Hehner’s theory defines a notational convention such that \( \text{slope} \) and \( \text{slope}' \) may be used to mean \( s \ \text{index}_{\text{slope}} \) and \( s' \ \text{index}_{\text{slope}} \), respectively. This is useful only in contexts where \( s \) and \( s' \) have meaning
Hehner's theory which relate the initial state $s$ to the final state $s'$. Since this convention tends to hide the process by which state components are extracted, we will not use it in comparing how each formalism performs this function. Instead we will introduce a simpler convention saying that state variables names have their index as their value. For example, we would have $slopes = 24$. Then instead of using $s 24$ to refer to the value of $slopes$ in situation $s$, we would use the equivalent $s slopes$. But it not even necessary to establish such a correspondence between state variables and integers: if states are represented by general functions instead of functions of integers, it is sufficient to say that state variables are first class objects.

To summarize, three different approaches are used in the formalisms mentioned above. If $state$ is a state or a situation and $fluent$ is a fluent or a state variable, the three approaches refer to the corresponding state component as follows:

\[
\begin{align*}
  &fluent \text{ state} \\
  &Val \ fluent \ text{ state} \\
  &state \ fluent.
\end{align*}
\]

What distinguishes each approach is whether $fluent$ and $state$ designate functions or non functional objects. In the first case $fluent$ designates a function and $state$ designates an object. In the last case the roles are reversed and $state$ is a function while $fluent$ is an object. In the second case both $fluent$ and $state$ are objects, and a unique binary function $Val$ is used.

Within first order logic, there is a cost associated to letting states or fluents be functions. Functions are not first class objects in first order logic, therefore one cannot quantify over states or fluents if they are represented as functions. Quantification over states is required to specify the effect of actions, while quantification over fluents may or may not be required. Elkan, in [Elkan96], uses quantification over fluents to compress the successor state axioms for all fluents into only one axiom. Also since there are no operations to be performed on functions in first order logic, other than application, the only benefit
the shorter syntax. Wherever application is used, as fluent state or state fluent, the expression Val fluent state may be substituted. The use of Val is therefore the most general solution within first order logic.

However, within the higher order logic there are more operations defined on functions and, therefore, potential benefits of using functions for states or fluents. One such operation is selective union (|) and it could be used to express conveniently that all fluents in state s' have the same value as those in state s, except for one fluent\(^2\). However this could also be expressed, even in first order logic, with quantification over fluents. And therefore an operator could be defined to represent the same operation in a compact manner within first-order logic. So the additional operations available on functions could be recreated to operate on first-class states and situations that are not functions.

Since all the approaches can be replaced, with only minor modifications and no loss of generality, by the approach using Val, we adopt it in both formalisms.

\[3.2.2\] States and Situations

One significant difference is the use of states versus situations. We have introduced situations as histories, however it is immaterial whether situations are to be considered as the histories themselves, or merely as instantaneous states for which there happens to be a way of determining the previous situation and the primitive action that generated the last transition. Therefore, to facilitate the comparison of the two formalisms under consideration, we adopt the latter interpretation. We say that situations are states from which it is possible to refer to the previous situations and actions.

The foundational axioms of the situation calculus ensure that unless \(s' = S_0\), there are unique values for \(a\) and \(s\) such that \(do a s = s'\). This makes situations out of the states; it gives them their historical character. The foundational axioms are presented in section 3.1.2.

---

\(^2\)This convenient expression would have the form \(s' = fluent \rightarrow newValue | s\), where \(fluent \rightarrow newValue\) is a function from \(fluent\) to \(newValue\). The selective union operator \(|\) merges the two functions, giving priority to the one on the left when an element is in the domain of both functions.
3.3 Actions

A distinction is made in the situation calculus between primitive and complex actions, and their effects are specified differently. The effects of primitive actions are specified through successor state axioms. Complex actions are constructed from primitive actions using various operators that represent programming constructs such as sequencing or non-deterministic choice. To these operators are associated axioms that describe the effect of complex actions in terms of the effect of the primitive actions.

The term "action" is not used in Hehner's theory but the term "program" is used to describe the same notion, so we use both terms interchangeably. Usually, no distinction is made between primitive and complex actions in Hehner's theory, but the distinction can be made in that formalism by defining as complex any action that is built from one or more other actions. For example assignments would be primitive actions, and sequences and conditionals would be complex actions.

Before we turn our attention to comparing the ways in which the effects of actions are specified in both formalisms, we must clarify the logical nature of actions.

3.3.1 Logical Nature of Actions in the Situation Calculus

In the situation calculus primitive actions are first class objects but complex actions are not. Complex actions, from GOLOG, are formal objects but they are defined outside the main logic. A special predicate Do, also outside the main logic, is defined that expands complex actions to formulas that belong to the main logic. These formulas relate an initial and a final situation to describe the effect of the corresponding actions.

A more natural approach would consist in defining complex actions as normal functions of actions. This would work for some constructs such as sequencing and non-deterministic choice which are functions of actions only, but it poses a problem for the constructs test, if and while. These constructs are not composed of actions only, they also contain formulas, which are not first class objects and cannot be arguments of normal functions.
We discuss this approach in section 4.2.

But without reifying formulas, Do must be defined as a macro, which is a function that operates on the syntax of its arguments and yields a syntactic result (here a formula), instead of denoting an object by operating on the objects denoted by the arguments.

### 3.3.2 Logical Nature of Actions in Hehner's Theory

The logical nature of some of the notions introduced in Hehner's theory is not immediately obvious because of the algebraic approach that is used. In logics such as first order logic, the first class objects are simply the elements of the universe. Here it is more difficult to identify first class objects because no such universe is precisely defined. But to understand how actions may be manipulated, it is useful to know if they may be considered first class. First class actions and specifications would be highly flexible in allowing, for example, the creation of bunches of actions, or of functions that return actions. To help determine if programs and specifications are first class objects, we explore the different types of axioms found in Hehner's theory.

#### The types of axioms

Some axioms, such as \( A, A = A \) or \( L \text{ null} = \text{null} \) are standard axioms that could also be axioms in first or higher order logics. These often have free variables which are understood to be implicitly universally quantified. If the quantifications were made explicit the range of the quantified variables could be specified. This, along with the distinction among the types of axioms and their meaning, would be necessary to make the formalism amenable to implementation in a theorem prover.

A second category of axioms are axiom schemas such as \( \forall v \cdot T \) or \( \text{var } x: T \cdot P = \exists x, x': T \cdot P \). These are not standard axioms since some of their variables, like \( v \) and \( x \), should not be understood as being implicitly quantified over some range of first class objects. Instead they are meant to range over some class of syntactic objects. In this case \( v \) and \( x \) are meant to range over variables. Axiom schemas are short notations for the infinite set of axioms which would be generated if the syntactic variables were replaced by
the syntactic objects in their range, such as variables or formulas. Because of their syntactic features, axioms schemas may be used to give semantics that resemble that of macros to the operators they are used to define. For example the operator \texttt{var} defined above and the operator \texttt{while do} defined in \texttt{while b do P = if b then (P . while b do P) else ok} are made to operate on variables or formulas in spite of the fact that these are not first class objects. So this approach to the definition of programs, represented by \texttt{var}, \texttt{while do} and a few other operators, is similar to the macro approach used for GOLOG. The definition of GOLOG using macros could therefore be translated to the formalism of Hehner's theory using axiom schemas. We do not use this approach however, we opt for an alternative presented further in this section, where programs are reified, and which results in simpler standard axioms. In cases where reification and standard axioms are not of great value, the first approach could be used.

The last category of axioms are particularly complex axiom schemas. We will call members of this category proof procedures, in reference to the special manipulations that theorem provers must perform to apply these unusual axioms. While it is easy to verify that a sentence fits an axiom schema, by unifying its subexpressions to the syntactic variables of the schema, the process is more complex for proof procedures. Take the application axiom for example: $x : D \Rightarrow ((\lambda v : D \cdot b) x = \text{(substitute } x \text{ for } v \text{ in } b))$. The syntactic variable $b$ can be unified to a subexpression in its first occurrence, but we see that a transformation process must be performed on the syntactic object unified to $b$ before unification of the right-hand side of the equality is attempted. Proof procedures are the most complex type of axiom and, when it is possible, should be avoided to reduce the amount of special programming and processing involved in the use of a theorem prover, and to preserve portability to stricter logics. Likewise, standard axioms are preferable to axiom schemas when they can perform the same function.

The nature of actions

The use of axiom schemas to give macro-like semantics to the operators that construct programs suggests that programs are not first class objects. However, functions are first class and they may be constructed from the \texttt{\lambda} operator, which is axiomatized through the
not be considered first class in spite of the irregular axioms used to characterize them. This would surely be unusual semantics since programs generally contain free variables (s and s'). But we can show that programs cannot be first class objects since, for one thing, they cannot be grouped in bunches. Consider a program P different from the specifications T and \( \bot \). The bunch\(^3\) \((P, \neg P)\) is equal to the bunch \((T, \bot)\) by the completion rule\(^4\), which contradicts our hypothesis on the nature of P. Therefore programs, and consequently specifications, cannot be considered first class objects.

To simplify the axioms for programs and to facilitate our analysis of the relationship between the situation calculus and Hehner's theory, we modify the nature of specifications and programs in Hehner's theory. Instead of being boolean expressions with s and s' as variables, specifications become first class relations between initial and final states. This modification can be made by simply prepending \( \lambda s, s': \text{state}. \), where state is the bunch of all states, to each specification, and by applying these relations to initial and final states when necessary. This makes refinement proofs longer because of the additional abstractions and applications, but allows us to replace complex proof procedure axioms by simpler axioms. For example the proof procedure axiom for sequences

\[
(A.B) = \exists s': \text{state} \cdot (\text{replace } s' \text{ by } s'' \text{ in } A) \land (\text{replace } s \text{ by } s'' \text{ in } B)
\]

becomes the standard axiom

\[
(A.B) s s' = \exists s'': \text{state} \cdot A s s'' \land B s'' s'.
\]

The additional abstractions and applications may be cumbersome in manual proofs, so they could be removed as a notational convenience, with the understanding that they should be present to be perfectly precise. But manual proofs are of limited interest due to

\(^3\)The operator \( \cup \) is the bunch union operator which merges bunches, and consequently, creates bunches of many elements from individual elements (elementary bunches).

\(^4\)The completion rule, from [Hehner93], says that "if a boolean expression contains unclassified boolean subexpressions, and all ways of classifying them place it in the same class, then it is in that class." The classes to which this refers are the class of formulas that equal T and the class of formulas that equal \( \bot \), called theorems and anti-theorems in [Hehner93].
from the use of relations for specifications and programs because they then become first class objects characterized by standard axioms.

3.3.3 Specification of the Effect of Actions

The effect of actions may be specified by describing the final states, or poststates, that result from each initial state, or prestate. Since the number of possible poststates that correspond to each prestate may vary, the specifications are best represented by binary relations between prestate and poststate.

Actions as specifications

In the situation calculus and GOLOG, a distinction is made between an action and the specification of its effect. But in Hehner’s theory an action is identified with the specification of its effect. This affects the syntax used to specify the effect of an action. While the GOLOG description of the effect of sequence actions is

\[ \text{Do } [A; B] s s' = \exists s'': \text{sit} \cdot \text{Do } A s s'' \land \text{Do } B s'' s', \]

the corresponding and equivalent description in Hehner’s theory is

\[ (A.B) s s' = \exists s'': \text{state} \cdot A s s'' \land B s'' s'. \]

Except for the syntax of the sequence operator and the use of situations versus states, the only difference is the use of Do to convert programs into the specification of their effect, which is only required for GOLOG actions. Section 4.1 covers the usefulness of distinguishing or identifying actions and specifications.

Deterministic primitive actions in the situation calculus

In the situation calculus the effects of primitive actions are specified through a number of successor state axioms, one for each fluent. The successor state axiom of a fluent specifies
may depend on the executed action and on the values of all the fluents in the original situation. In most cases the value of a fluent will not change so successor state axioms rely on subexpressions that characterize the cases where the values change, and then specify that the values do not change in other cases. This allows the user of the formalism to specify the effect of actions without having to mention all the cases where no change occurs.

The successor state axiom for a fluent \( f \) has the form

\[
\text{Poss } a \ s \Rightarrow ((f \ (\text{do } a \ s) = y) = (\gamma_f(y, a, s) \vee (y = f s) \wedge \neg \exists y' \cdot \gamma_f(y', a, s)),
\]

where \( \gamma_f(y, a, s) \) is a formula with free variables among \( y, a \) and \( s \) that specifies from which situations \( s \) and for which actions \( a \) the fluent \( f \) changes to value \( y \). The \( \text{Val} \) convention is not used here because this axiom was designed in the context of fluents which were not reified. The axiom says that fluent \( f \) takes value \( y \) after the execution of action \( a \) if and only if the value of \( f \) changes to \( y \), or the value is already \( y \) and the value is not changed.

The above form for successor state axiom applies to functional fluents. In first order logic the relational fluents must be treated separately and the corresponding successor state axioms have the following form:

\[
\text{Poss } a \ s \Rightarrow (f \ (\text{do } a \ s) = t) = (\gamma_f^+(a, s) \vee f s \wedge \neg \gamma_f^-(a, s)),
\]

where \( \gamma_f^+(a, s) \) characterizes the cases where fluent \( f \) becomes true, and \( \gamma_f^-(a, s) \) characterizes the cases where it becomes false. However, since \( \top \) and \( \bot \) are values in our logic, it become possible to treat relational fluents as functional fluents and use the same form of successor state axioms. The formulas \( \gamma_f^+(a, s) \) and \( \gamma_f^-(a, s) \) will then be represented by the equivalent \( \gamma_f(\top, a, s) \) and \( \gamma_f(\bot, a, s) \), respectively.

The relational successor state axiom may then be expressed as

\[
\text{Poss } a \ s \Rightarrow ((f \ (\text{do } a \ s) = \top) = (\gamma_f(\top, a, s) \vee (f s = \top) \wedge \neg \gamma_f(\bot, a, s)),
\]
Poss a s ⇒ \((f \ (do \ a \ s) = \bot) = (\neg \gamma_f(\top, a, s) \land (f s = \bot) \lor \gamma_f(\bot, a, s))\).

Since \(\gamma_f(y, a, s)\) is true only when \(f\)'s value changes, \(y = f s \land \neg \exists y' \cdot \gamma_f(y', a, s)\) may be replaced by \(y = f s \land \neg \exists y' \cdot y' \neq y \land \gamma_f(y', a, s)\) in the functional state successor axiom. The two are equivalent because \(\gamma_f(y, a, s)\) cannot hold when \(y = f s\) holds: fluent \(f\) cannot change to value \(y\) if it already has value \(y\). In the case of relational fluents, \(\neg \exists y' \cdot y' \neq y \land \gamma_f(y', a, s)\) becomes \(\gamma_f(-y, a, s)\), and we see that the conjunction of the two equivalent axioms above is equivalent to the successor state axiom for functional fluents. Therefore when relational fluents are considered as functional fluents whose range is \(\top\) and \(\bot\), the successor state axioms of the relational form are not necessary and may be replaced by successor state axioms of the functional form.

By accessing the value of fluents through \(Val\) and by considering fluents as first class objects over which we may quantify, the successor state axioms for each fluent may be combined into a single successor state axiom. This is done in [Elkan96] for relational fluents whose truth value is accessed by \(Holds\). Using our logic and syntax, Elkan's axiom would appear as

\[
\forall a, s, f \cdot (Holds f (do a s) = (causes a s f \lor Holds f s \land \neg cancels a s f)),
\]

where \(causes\) and \(cancels\) are predicates that are similar to \(\gamma^+\) and \(\gamma^-\) but differ in the status of the fluent, which is an argument in \(causes\) and \(cancels\), and a subscript in \(\gamma^+\) and \(\gamma^-\).

Similarly, we may unify the successor state axioms for functional fluents as the axiom

\[
Poss a s ⇒
\begin{align*}
((Val f (do a s) = y) = (\gamma f y a s \lor (Val f s = y) \land \neg \exists y' \cdot \gamma f y' a s)),
\end{align*}
\]

where \(\gamma\) is a predicate corresponding to the formulas \(\gamma_f\). The application \(\gamma f y a s\) yields \(\top\) if and only if fluent \(f\) changes to value \(y\) after the execution of action \(a\) from situation \(s\).
It also imposes an approach which may seem overly complex and restrictive to the user of the formalism. There are many ways of specifying the cases under which no change occurs in the value of the fluents, especially since fluents have been declared first class objects over which it is possible to quantify. The successor state axioms were introduced as a solution to the frame problem, the problem of knowing what does not change. But given the possibility of quantifying over states, actions and fluents, and given the restricted context in which we operate (there are no state constraints), the user of the formalism may not require any help with the specification of what does not change. Let us examine what approach is used in Hehner's theory.

**Primitive actions in Hehner's theory**

The effects of actions are specified differently in Hehner's theory. One axiom is introduced for each action, instead of one axiom for each fluent or one axiom for all actions and fluents. Although there are very few primitive actions in Hehner's theory, this method of specifying their effect is very intuitive and could be used to specify as many primitive actions as are usually found in situation calculus theories. Programmers used to imperative languages are used to considering all the variables affected by an action, since that is what is required to define such an action. It is more difficult to consider all the actions which may affect a specific variable. This favors effect specifications that are introduced action by action, instead of fluent by fluent. Successor state axioms, which are fluent specific, could be preferable in particular proofs that require rapid access to all the actions that affect a particular fluent, but it is likely that users of the formalism would still prefer to specify the effect of actions through action specific axioms.

An action open that attempts to open a door could be specified as

\[
open s s' = (Val \text{ opened } s' = \neg Val \text{ locked } s \land \\
\forall f: f \cdot f \neq \text{ opened} \Rightarrow Val f s' = Val f s),
\]

where \( fl \) is the bunch of fluents. This says that the door will be open if and only if it was not locked, and that all other fluents will be unaffected. Note that we assume here, and in the following uses of this example, that the door is always closed when it is locked.
compact. This can be achieved by defining the action using more adequate operations like assignments and conditionals. Since these are actions, this could in effect make the defined action a complex action. However this distinction is artificial in Hehner’s theory. And even in the situation calculus a primitive action could be defined in terms of a complex action, so that they both affect fluents in exactly the same manner yet the defined action remains primitive. Therefore the best way to specify primitive actions may be to treat them as programs. Let us introduce some actions that can be used in the definition of primitive actions such as open.

A trivial but useful action is ok, which causes absolutely no change:

\[ ok \, s \, s' = (s' = s). \]

Since all situation calculus primitive actions must change the situation to be compatible with the foundational axioms, a corresponding definition for the situation calculus would say that all fluents keep their value:

\[ (do \, ok \, s = s') = \forall f : fl \cdot Val \, f \, s' = Val \, f \, s. \]

The assignment of fluent \( f \) to value \( v \) may be defined as follows:

\[ (f := v) \, s \, s' = (Val \, f \, s' = v \land \forall f' : fl \cdot f' \neq f \Rightarrow Val \, f' \, s' = Val \, f' \, s). \]

The assignment action is not usually defined in the situation calculus but it may be axiomatized similarly. The only differences would be the application of the function do to the action (as \( do \, (f := v) \, s = s' \) instead of \( (f := v) \, s \, s' \)), and the presence of the Poss antecedent (Poss \( (f := v) \, s \Rightarrow \)), which ensures that do is defined on these arguments.

Other actions are useful in the definition of the effect of primitive actions, but ok and assignments are sufficient to illustrate the approach with the action open:

\[ open \, s \, s' = (\neg \, Val \, locked \, s \Rightarrow (opened := T) \, s \, s' \land Val \, locked \, s \Rightarrow ok \, s \, s'). \]
\begin{verbatim}
open = if \lambda s \cdot \text{Val locked } s \text{ then } \text{ok } \text{ else } \text{opened := T,}
\end{verbatim}

where the condition on the fluent locked has been expressed as a predicate of the state. We arrive at such a convention when we present the semantics of complex actions in the next few pages. So complex action constructors such as \texttt{if then else} will be analyzed later in this section, but even without this constructor, we see that it is possible and convenient to specify the effect of a primitive action in terms of assignments. This approach removes the need to add a conjunct that means “and all other fluents remain unaffected”. This default behavior is specified in, and inherited from, assignment actions and \texttt{ok}.

In addition to providing a convenient way of specifying the effect of primitive actions, the approach used here is more general because it allows the specification of actions with nondeterministic effects. Nondeterminism, in the form of an action that may affect some fluent or another, can be introduced through a disjunction in the axiom that specifies the effect of an action. For example the action of dropping some fragile object from a moderate height may break it or scratch it:

\begin{verbatim}
drop = (\text{broken := T} \lor \text{scratched := T}).
\end{verbatim}

If the effect of actions were axiomatized through a successor state axiom for each fluent, then such nondeterminism would require the axioms to depend on a common function that acts as an indeterminate value. The above example would be specified as:

\begin{verbatim}
Poss a s \Rightarrow (\text{broken (do a s) = ((a = drop } \land j a s) \lor \text{broken s)}) \text{ and}

Poss a s \Rightarrow (\text{scratched (do a s) = ((a = drop } \land \neg j a s) \lor \text{scratched s)}),}
\end{verbatim}

where \texttt{j} is a predicate taking two arguments that is left undefined at every point. When its value is \text{T}, the action \texttt{drop} results in the fluent \texttt{broken} being true in the final situation. When its value is \text{\bot}, the action results in the fluent \texttt{scratched} being true in the final situation. These two possible effects are the same as the ones we allowed in the previous
Here Hehner’s theory brings an approach to the specification of the effect of primitive actions that may be preferable to successor state axioms. It handles nondeterminism conveniently and it should have more appeal to programmers who are used to considering each action separately instead of considering each fluent separately.

However, in some cases such as automatic proofs using algorithms that require rapid access to the actions that affect a particular fluent, successor state axioms may still be preferable as a way of representing effects for computation. Then the formulas $\gamma_f$ would also need to have a regular structure, such as a series of conjuncts for each action affecting $f$. These axioms could be derived from whatever form of axiom was used initially by the users of the formalisms to specify the effect of actions. Of course, it would then be more accurate to call the derived sentences successor state theorems.

It should be noted that within Hehner’s theory the above approach to the specification of the effect of actions does not require the use of a predicate $Poss$. An action which cannot be performed from a state $s$ will simply have a prestate to poststate relation $R$ such that no poststate makes $R$ hold from $s$ ($\neg \exists s' : state \cdot R s s'$). Within the theory this is described as specification $R$ being unsatisfiable for prestate $s$.

In the case of the situation calculus, where situations are used instead of states, the above approach still requires the use of the $Poss$ antecedent. This is because axiom (3.8) below in this section depends on $Poss$ to determine if there is a final situation that results from the performance of the action, and because the foundational axioms also refer to $Poss$.

**Complex actions in the situation calculus**

The specification of the effect of complex actions is a result of the definition of the $Do$ predicate in axioms such as

$$Do [A; B] s s' = \exists s'': sit \cdot Do A s s'' \land Do B s'' s'.$$

The predicate $Do$ was presented as a macro because some axioms that define complex
When dealing with non-instantiable formulas, which are not first-class objects, this means that *Do* could not be specified with standard axioms but would require axiom schemas or proof procedure axioms. This complexity may be avoided by reifying formulas, a possibility that we explore in 4.2, but also by using λ-expressions instead of formulas. By transforming formulas with a free variable of the sort situation into λ-expressions that take a situation argument, the obstacle to the use of standard axioms for the definition of *Do* is removed. We adopt this approach here and present the standard axioms that can define the GOLOG language. However this solution is not as versatile as the reification of the formulas which appear in programs, and this aspect will be studied in section 4.2.

By representing conditions using λ-expressions, programs may now be treated as first class objects. The predicate *Do* may then be interpreted as a normal predicate that takes an action and two situations to return a boolean. In that case applying *Do* to only an action returns a relation between two situations, which, as in Hehner's theory, specifies the effect of the action. The semantics of GOLOG programs may then be expressed by rewriting the usual axioms, which can be found in [Levesque94], as follows:

\[
\begin{align*}
\text{Do} \; a \; s \; s' & = (\text{Poss} \; a \; s \land s' = \text{do} \; a \; s) \\
\text{Do} \; (\text{test} \; p) \; s \; s' & = (p \; s \land s = s') \\
\text{Do} \; [a_1; a_2] \; s \; s' & = (\exists s''. \; \text{Do} \; a_1 \; s \; s'' \land \text{Do} \; a_2 \; s'' \; s') \\
\text{Do} \; (\text{or} \; a_1 \; a_2) \; s \; s' & = (\text{Do} \; a_1 \; s \; s' \lor \text{Do} \; a_2 \; s \; s') \\
\text{Do} \; (\text{choose} \; f) \; s \; s' & = (\exists x : \Delta f \cdot \text{Do} \; (f \; x) \; s \; s') \\
\text{if} \; p \; a_1 \; a_2 & = (\text{or} \; [(\text{test} \; p); a_1] \; [(\text{test} \; \lambda_s \cdot \neg p \; s); a_2]) \\
\text{while} \; p \; a & = [(\text{iter} \; [(\text{test} \; p); a]); (\text{test} \; \lambda_s \cdot \neg p \; s)] \\
\text{Do} \; (\text{iter} \; a) \; s \; s' & = (\forall P : \text{sit} \rightarrow \text{sit} \rightarrow \text{bool}. \\
& \quad (\forall s_1 \cdot P \; s_1 \; s_1) \land \\
& \quad (\forall s_1, s_2, s_3 \cdot P \; s_1 \; s_2 \land \text{Do} \; a \; s_2 \; s_3 \Rightarrow P \; s_1 \; s_3) \\
& \Rightarrow P \; s \; s')
\end{align*}
\]

These axioms implement respectively: primitive actions, tests, sequences, nondeter-
minimistic choices of two actions, nondeterministic choices of action parameters, conditional actions, while loops and nondeterministic iteration.

Here the syntax of GOLOG programs has been modified to eliminate conflicts with the symbols already in use in Hehner's theory. The symbols test, or and choose replace the symbols ?, | and π, respectively. Also the ternary function symbol if is used for conditionals instead of the series of symbols if then else endIf which could be confused with the conditionals if then else defined in Hehner's theory.

The condition p in axioms (3.9), (3.13) and (3.14) is a boolean function of the situation at that point in the program. It must therefore be expressed as a λ-expression that takes the situation as its argument. This can be witnessed in axioms (3.13) and (3.14) where test is applied to the condition λs . ¬ p s. Note that if a propositional fluent f is an appropriate condition, it is not necessary to express the condition as λs : sit . Val fs. The equivalent but simpler condition Val f is sufficient. Also Hehner's theory includes a functional composition convention which allows the replacement of λs . ¬ p s by the equivalent but nicer expression ¬p, where the function ¬ is composed with p.

The choose construct, which is called nondeterministic choice of action parameter, also relies on λ-expressions. The construct allows the execution of a program that has some parameter. The execution proceeds by performing the program with any value for the parameter. Here parameterized programs are represented by functions that take a parameter and return a program. Applying choose to such a function creates the nondeterministic program. Since only values within the domain of the function are admissible, the range of the existential quantifier in axiom (3.12) is specified as the domain Δf of the function f.

The iter construct executes its argument 0 or more times consecutively. To provide complete semantics at infinity, the least fixed point is used. The axiom says that Do (iter a) s s' holds if and only if P s s' holds for all fixed points P. This means that Do (iter a) s s' holds if and only if P s s' holds for the least fixed point P.

GOLOG also implements procedures. To fully specify their effect in recursive cases, a least fixed point definition is used in [Levesque94]. We have not axiomatized procedures here because, as in Hehner's theory, λ-expressions that return programs may be used as
symbol. For example a procedure $P$ could be defined that performed action $a$ or action $b$ depending on the value of the argument:

$$P = \lambda x: \text{int} \cdot \text{if } \lambda x \cdot x > 4 \text{ then } a \text{ else } b.$$

Note that procedures of no arguments may be created, by simply naming a complex action. These procedures may also be recursive. For example:

$$Q = \text{if } \lambda s \cdot \text{Val } f \cdot s > 0 \text{ then } S \text{ else } [R; Q],$$

where $f$ is a fluent and $Q$, $R$ and $S$ are procedures of no arguments, also called programs.

In the case of recursive procedures, there may be certain procedure arguments and initial and final situations for which the procedure’s expansion contains the same procedure application, preventing a finite expansion where the named procedure would not occur. In such cases the axioms defining $Do$ may not determine a unique solution for the value of $Do$ on that action and that pair of situations. A solution is the introduction of a least fixed point definition to force the indeterminate values to be false. This is done in [Levesque94] and could be reproduced in this variant of the formalism by adding, for each recursive procedure, an axiom that constrains its semantics to the least fixed point. A trivial but illustrative example is the procedure $Z$ recursively defined as $Z = Z$. Moving to the procedure semantics we get $Do Z = Do Z$. Here $Do Z$ could be anything since all relations are fixed points of this equation. An axiom which would constrain the solution to the least fixed point is

$$\forall Y : \text{sit} \rightarrow \text{sit} \rightarrow \text{bool} \cdot Y = Y \Rightarrow \forall s, s' : \text{sit} \cdot Do Z s s' \Rightarrow Y s s',$$

which says that $Do Z$ is as strong as all fixed points. Here this means that it as strong as all relations, so $Do Z s s' = \bot$. A more typical example would use a similar axiom with the $Y = Y$ replaced by a less trivial recursive definition.

Another approach, used in Hehner’s theory, consists in accepting the indeterminate
Recursively defined programs will only have indeterminate semantics on arguments (including the initial and final situations) that lead to infinite recursion. For carefully axiomatized programs this will only occur with arguments for which the programs do not terminate. For example, a recursive program that decrements an integer valued fluent until its value reaches 0 would only be indeterminate for initial situations in which the fluent had a negative value. For all other situations, the recursive definition would lead to a finite expansion, and therefore, to a precise definition. Such partially specified behavior may well be all that the author of the axiom wished to say. Therefore a least fixed point definition, although complete, may not always be what is intended. There is a risk of writing recursive definitions that are stronger than we intend them to be. Conversely, the approach of tolerating incompleteness requires us to be careful to specify recursive definitions which are as strong as we intend them to be, by ensuring that they expand finitely on arguments for which we want the definitions to be precise. An example which does not expand finitely is a procedure \( Z \) defined as \( Z = (ok \lor Z) \) which we might expect to be equivalent to \( ok \), but which is highly indeterminate.

Each approach to recursive procedure definitions has its advantages, but since the approaches may be exchanged at will between the two formalisms, we need not establish a preference here.

Later, in section 4.5, we address consistency concerns that can arise using recursive program definitions.

**Complex actions in Hehner's theory**

The specification of the effect of a sequence of actions was originally specified as

\[
(P \cdot Q) = \exists x'' , y'' , ... (\text{substitute } x'' \text{ for } x', y'' \text{ for } y', \ldots \text{ in } P) \\
\quad \land (\text{substitute } x'' \text{ for } x, y'' \text{ for } y, \ldots \text{ in } Q).
\]

By using our convention for referring to the value of fluents in situations, the axiom becomes

\[
(P \cdot Q) = \exists s'' : \text{state} (\text{substitute } s'' \text{ for } s' \text{ in } P)
\]
The axiom is further simplified, and becomes a standard axiom, when it is expressed
according to our convention of representing actions as prestate to poststate relations
instead of formulas:

\[(P \cdot Q) s s' = \exists s'': \text{state} \cdot P s s'' \wedge Q s'' s'.\]

However, it is not as easy to give standard axioms for the complex actions such as
\textbf{if then else} and \textbf{while do}, which contain conditions in addition to programs. Just as
we did for the conditions of GOLOG programs, we must transform conditions from formu-
las of one free variable to predicates of one argument represented by \(\lambda\)-expressions. The
axioms that specify the effect of complex actions may then be expressed as the following
standard axioms:

\[(P \cdot Q) s s' = \exists s'': \text{state} \cdot P s s'' \wedge Q s'' s'.\]

\[(P \parallel Q) s s' = \exists s_P, s_Q: \text{state} .
\]
\[P s s_P \wedge Q s s_Q \]
\[\wedge \forall f: fl \cdot f \neq t \Rightarrow (\text{Val } f s_P = \text{Val } f s \Rightarrow \text{Val } f s' = \text{Val } f s_Q) \]
\[\wedge (\text{Val } f s_Q = \text{Val } f s \Rightarrow \text{Val } f s' = \text{Val } f s_P) \]
\[\wedge \text{Val } t s' = \max (\text{Val } t s_P) (\text{Val } t s_Q) \]

\[(\text{if } b \text{ then } P \text{ else } Q) s s' = (b s \wedge P s s' \lor \neg b s \wedge Q s s')\]

\[(\text{while } b \text{ do } P) = \text{if } b \text{ then } (P \cdot \text{while } b \text{ do } P) \text{ else ok}\]

\[(\text{repeat } P \text{ until } b) = (P \cdot \text{if } b \text{ then } \text{ok else repeat } P \text{ until } b),\]

where \(P\) and \(Q\) are programs, \(fl\) is the bunch of fluents, \(t\) is a fluent that represents time,
and \(b\) is a boolean function of the state.

The \textbf{if then else} construct has the same semantics as the \textbf{if} construct from GOLOG
defined in axiom (3.13). This can be proved by expanding the semantics of the \textbf{or}, \textbf{test}
The original definition of the if then else construct was

\[(\text{if } b \text{ then } P \text{ else } Q) = (b \land P \lor \neg b \land Q).\]

Since it was defined in terms of boolean operators only, it could be used on boolean expressions. Originally actions were boolean expressions, but we adopted the convention of representing actions with relations instead of boolean expressions. This means that by modifying the construct to operate on relations, we prevented its use on boolean expressions. This can be avoided by using both definitions. Since the arguments to if then else are of different types in each axiom, there can be no ambiguity or inconsistency. It is simply a matter of extending the domain over which the ternary function is defined.

The while do construct differs slightly from the while of axiom (3.14). The GOLOG version relies on the iter construct which is defined as a least fixed point, while this version is specified only as a fixed point. We could mirror the GOLOG technique of relying on iter, independently of the use of the least fixed point, by defining

\[
\text{while } b \text{ do } P = \left(\left(\text{iter } (\text{test } b \cdot P)\right) \cdot \text{test } \lambda s \cdot \neg b \cdot s,\right)
\]

where \((\text{iter } P) = (\text{ok } \lor (P \cdot \text{iter } P)),\)

and \((\text{test } b) s s' = (b s \land \text{ok } s s').\) In that case iter is only a fixed point. This definition of iter could be used as the definition of iter if we did not want to use the least fixed point in GOLOG.

Alternatively we could use the least fixed point in both formalisms. This could be done by using axioms of the same form as iter from the GOLOG definition. A slightly different form of axiom could also be used to specify least fixed points. Hehner presents an axiom which may be added to a recursive definition to constrain its solution to the weakest fixed point. The same can be done to obtain the strongest fixed point, which is
\[(W = \text{if } b \text{ then } P.W \text{ else } \text{ok}) \Rightarrow (\forall s, s' : \text{sit}. (\text{while } b \text{ do } P) \ s \ s' \Rightarrow W \ s \ s'),\]

where \(W\) is a universally quantified specification. This says that \(\text{while } b \text{ do } P\) refines all fixed points \(W\), so that it is as strong or stronger than the strongest fixed point. Conjoined with the other axiom which says that \(\text{while } b \text{ do } P\) is a fixed point, this means that it is the strongest fixed point, or the least fixed point. Similar axioms can be added to other fixed point definitions, such as the definition of \text{iter} or the definition of recursive procedures (as we already discussed on page 29).

The sequencing construct \((\cdot)\), called dependent composition in Hehner's theory, has the same semantics as the GOLOG version.

The construct \(\|\) is called independent composition and does not exist in GOLOG. It implements a type of concurrency such that its two arguments are executed independently and at the same time. The two processes do not share any state variable. The result of the concurrent execution is defined as a state that incorporates the changes made to the fluents by each process. For this construct to be useful, only one process must change each fluent. The fluent \(t\), which represents time, is an exception since it must become the latest of the final times of the two processes. While the axiom specifying this construct is more complex than the others, it is still simpler than the original version of the axiom which viewed actions as formulas and did not have one symbol to represent a state but one for each fluent in each state. The original version was a proof procedure while this version is a standard axiom.

While all other constructs defined above can be used in the situation calculus, where states are replaced by situations, the independent composition cannot be used in this way. That is because for each initial situation \(s\), the final situation \(s'\) would not be specified in terms of the execution of a series of primitive actions from \(s\), but as a collection of fluent values, possibly\(^5\) distinct from the fluent values in all situations accessible from \(s\). This is

---

\(^5\)This possibility cannot occur when the assignment action is available, but it may occur with some other restricted sets of actions. For example consider two fluents initially \(T\) in \(S_0\), and two actions which affect them as follows: the first action makes the first fluent \(\bot\) if and only if the two fluents were \(T\), and
Now that we have introduced the convention of using λ-expressions to represent conditions, we should reconsider the assignment axiom:

\[(f := v) s s' = (Val f s' = v \land \forall f': fl \cdot f' \neq f \Rightarrow Val f' s' = Val f' s).\]

While there are no conditions in this axiom, there is an expression, the value \(v\), that in general should depend on a situation. This version of the axiom requires a constant value \(v\), and therefore cannot refer to the value of fluents in the current situation. An improved version of the axiom may now be introduced that expects not a value, but a function \(e\) that generates the appropriate value from the situation:

\[(f := e) s s' = (Val f s' = e s \land \forall f': fl \cdot f' \neq f \Rightarrow Val f' s' = Val f' s).\]

An assignment action placing the sum of the values of numeric fluents \(g\) and \(h\) into fluent \(f\), would therefore have the following syntax:

\[f := \lambda s \cdot Val g s + Val h s.\]

In general, treating the expressions contained in programs as functions of the current state is elegant semantically but not syntactically. The ideal solution, syntactically, would be to express the above assignment as

\[f := g + h,\]

but this is not yet possible according to the semantics we have defined.

---

the second action makes the second fluent \(\perp\) if and only if the two fluents were \(\top\). The fluents could represent the vitality of two enemies A and B, and the two actions could represent B kills A and A kills B. In that case executing the two actions in parallel leads to a situation were both are dead (the two fluents are \(\perp\)), even though this situation cannot be reached by any sequence of these actions.
very similar to this will become possible with the reification of state dependent expressions performed in section 4.2. Of course, a special syntactic convention could also be introduced to allow the simpler syntax.

3.4 Time and Concurrency

The treatments of time and concurrency of both formalisms are not easily unified. One reason is that the situation calculus has many variants but, for the moment, no approach to time and concurrency that offers a completely satisfactory treatment of GOLOG. These approaches also differ from Hehner's approach in [Hehner93] on many fundamental aspects, which makes unification difficult. We therefore limit ourselves to a brief comparison of the approaches, which should make the main difficulties apparent. The modifications we have made previously to the formalisms should not prevent the implementation of any of these approaches.

A variant of the situation calculus has been introduced by Pinto in [Pinto94] which includes a treatment of time and concurrency. Pinto also covers the integration of the complex actions of GOLOG in this variant, but only for the time aspect. An actual path of situations is introduced, and predicates \textit{Occurs} and \textit{Occurs}_τ are defined to express statements about the occurrence of an action or program execution in an actual situation or at a point in time. Primitive actions occur at a particular time point and have no duration. Situations remain unchanged between the occurrence of actions and therefore have duration.

This differs considerably from the treatment of time in Hehner's theory, which is mainly concerned with the determination of the time taken by the execution of programs, to ensure that they meet their specification regarding time. States are extended to include a time variable and programs are modified to include increments in the time variable to reflect the passage of time. Since states include a time variable, they have no duration, and since primitive actions may modify the time variable, they may have duration. No actual path of states is defined, and there is no global initial state from which such a
The approaches on concurrency also vary considerably. Hehner presents a programming construct called independent composition, where processes are executed independently, and their effects are merged once both processes have terminated. The fact that parts of these processes may be executed at the same time has no impact on their effect. The processes are truly independent until they terminate. However dependencies may be inserted deliberately using communication channels, which are structures shared by the processes.

In [Hehner94] there is also a different presentation of time and concurrency for Hehner’s theory. Its concurrency partitions fluents between processes. Each process is responsible for specifying the values taken by some fluents, the other fluents are described by other processes. This allows a concurrency operator to be defined as mostly a conjunction of the processes. Time is also treated differently so that time points replace states.

Variants of the situation calculus extended to include concurrency treat it at the level of primitive actions. Some consider the interaction of sets of primitive actions occurring at the same time, in order to derive the conditions under which such an execution is possible and the effect of that execution on the state. Pinto’s treatment follows this approach, and accordingly does not present a concurrent programming construct that would create GOLOG processes. In other variants [Lespérance96], concurrency is introduced through an interleaving of the actions. It extends the programming language to CONGOLOG (CONcurrent GOLOG). While this enables the concurrent execution of complex programs, it is not easily axiomatized, and in contrast to the independent composition of Hehner’s theory, the rate of progress of a process depends on the presence of other concurrent processes. The final effect of a process is also dependent on the intermediate states of the other processes.

Considering the numerous alternatives for time and concurrency which are under consideration, in both the situation calculus and Hehner’s theory, we do not attempt the considerable task of determining which options are preferable in which cases. However, given the harmonization which we have performed regarding the other aspects of the formalisms, it becomes possible to consider the use in one formalism of approaches to time.
in the future as researchers in both fields continue to explore and compare alternatives for the treatment of these issues.
Chapter 4

Increasing the expressiveness

4.1 Reifying Actions

Complex actions from GOLOG and Hehner’s theory are syntactic objects and this prevents the use of standard axioms to characterize their semantics. We have solved this problem by representing actions by first class objects. In the case of GOLOG we have truly reified actions but for Hehner’s theory we have given first class status to actions without reifying them: actions were identified with their effect. The effect of an action is represented by a prestate to poststate relation, which is a first class object. However the true concept of a complex action, as a program, is distinct from the the effect of the action; there can be many different programs which have the same effect. Therefore some information is lost by treating actions as the relations that specify their effect.

Distinguishing between actions and their effect specification has the advantage of allowing, when actions are first class objects, the formulation of programming problems within the logic. Finding the bunch of programs that refine a specification S would involve the evaluation of $\exists a : act \cdot \forall s, s' : state \cdot S s s' \iff Do a$, where $\exists$ is a quantifier that returns the bunch of bindings that satisfy its body, $Do$ is the function from actions to the specification of their effect, and the specification $S$ is in the form of a relation between prestate and poststate.

Even with our modification of the status of programs from formulas to relations, the first class status of programs is not sufficient to express a programming problem within
for a program matching some criteria would return only a relation between states, like
\( \lambda s, s' : state \cdot s = s' \), and not an executable structure, like \( ok \). There would also be no
guarantee of the existence of an action that has the returned specification as its effect. If
the bunch of executable specifications were characterized and used to bound the search,
a returned specification would correspond to an existing program, but the problem of
finding the program, as an executable structure, would not be solved.

Programs must therefore be reified. No two programming constructs should lead to
the same program. The construction process must be reversible so that it is possible
to analyze unambiguously the structure of a program. Treating programs as prestate
to poststate relations does not satisfy this requirement. In fact even our reification of
GOLOG programs was incomplete in that the functions \( if, or, \) etc. were not specified
to yield distinct results. Unique names axioms for actions are required. This may be
done with two axioms for each construct: one that specifies that the constructor function
is injective, and another which names the range of the function. Then one more axiom
is needed that says that all the following bunches are disjoint: the bunch of primitive
actions, the range of \( if \), the range of \( or \), etc. If some primitive actions take arguments,
then these should be treated as constructors, so that unique names axioms are provided
for them too.

The same technique may be used to reify the programs of Hehner’s theory. However
most of the work in a refinement proof is done at the level of prestate to poststate
relations. The programming constructs in the form of functions over relations are useful in
representing the effect of programs. If we were to eliminate these functions to replace them
by functions between actions, it would make effect relations very difficult to manipulate.
For example, even a trivial program

\[
P = (b := \lambda s - Val a s) \cdot (c := \lambda s - Val b s)
\]

loses its simplicity after the sequencing operator’s semantics is expanded to yield

\[
Do P = \lambda s, s' - \exists s'': state. Do (b := \lambda s - Val a s) s s''
\]
and would become much larger if the assignment semantics were also expanded. These steps are necessary to reason about the effect of programs, but we would like to limit the expansion of the semantics to the part of the program where reasoning will occur. However, this would not be possible if the functions between relations were eliminated. It would be necessary to expand the semantics of the constructs at all nodes on the path between the root and the region of interest in the syntax tree of program. A solution is to preserve the functions that operate on relations. The functions could be used to expand the nodes mentioned above without creating unnecessary complexity. By representing the functions between relations and the functions between actions with the same symbols, we can express the semantics of a program the same way we express the program itself. This allows us to move from a program to its semantics without increasing syntactic complexity.

For the program $P$, the first step in the expression of its semantics would become

$$\text{Do } P = \text{Do } (b := \lambda s \cdot \text{Val } a \ s) \ . \ \text{Do } (c := \lambda s \cdot \text{Val } b \ s),$$

where the period here denotes a function between relations, while it denoted a function between actions in the definition of $P$ above. If no reasoning is to occur at the level of the sequencing operator, this representation is preferable to the previous one, which revealed too much detail. Therefore we adopt this convention of overloading the functions of program constructs so that they may apply to both programs and prestate to poststate relations. The axiomatization of this convention is straightforward: we preserve the axioms introduced on page 31 and supplement them with axioms that define the functions that operate on actions. These axioms are very simple since they rely on the first set of axioms to provide the program semantics:

$$\text{Do } (P \cdot Q) = (\text{Do } P \cdot \text{Do } Q)$$

$$\text{Do } (P \parallel Q) = (\text{Do } P \parallel \text{Do } Q)$$

$$\text{Do } (\text{if } b \text{ then } P \text{ else } Q) = (\text{if } b \text{ then } \text{Do } P \text{ else } \text{Do } Q)$$
Then we must modify the axioms that define primitive actions to reflect the fact that actions are now reified, and that consequently Do must be applied to them before their effect may be specified:

\[
\text{Do} (f := e') s s' = (\text{Val} f s' = e s \land \forall f': f f' \neq f \Rightarrow \text{Val} f' s' = \text{Val} f' s)
\]

\[
\text{Do ok} s s' = (s' = s).
\]

In addition to allowing the formulation of the programming problem within the logic, another benefit of reified programs is that we may now place constraints on the form of a program intended to solve a programming problem. While the theory could define a vast number of programming constructs, a particular programming problem may call for the use of only a restricted subset of constructs. Different subsets could correspond to different programming languages. Restrictions could also be placed on the size of a program. These restrictions cannot be imposed through a specification of the allowed transitions from initial to final states. However, a bunch allowed of allowable programs could be characterized. It would then be used in the expression of the programming problem as follows:

\[
\forall\text{ allowed} \cdot \forall s, s': \text{state} \cdot S s s' \Leftarrow \text{Do} a,
\]

where \( S \) is the specification of allowed prestate to poststate transitions.

### 4.2 Reifying Expressions

#### 4.2.1 Benefits

The reification of programs presented in the previous section did not reify all aspects of programs. The state dependent expressions found in programs are part of the programs
as prestate to poststate relations are not a valid reification of programs because many distinct programs correspond to the same relation, the functions of states used to represent state dependent expressions are not a valid reification because many distinct expressions correspond to the same function.

However, reifying expressions is not strictly necessary if the evaluation of expressions has no effect on the state, as is the case in GOLOG and Hehner's theory, where the evaluation of a condition for a test, or of an expression for an assignment, cannot affect the state in any way. Under these conditions, different expressions which always agree on their value could be considered equivalent and therefore could be represented adequately as λ-expressions. But there are also many reasons to reify state dependent expressions. One of them is to facilitate the embedding of the new formalisms within first-order logic, where λ-expressions are not available. Other reasons follow.

Eliminating a loophole

Depending on the domain of application, it may not be satisfactory to treat the potentially complex expressions found in programs as being instantly evaluated without any effect on the state. In some cases the evaluation of expressions could take an amount of time which cannot be neglected. We would not want to obtain a solution to the programming problem that shifts most of the computational work to the expressions of the program. This tactic would avoid the restrictions placed on execution time in the specification of allowed prestate to poststate transitions.

A similar consideration is the set of allowed functions within expressions. The programming theory includes all functions, therefore it includes powerful functions which could solve a programming problem trivially. If state dependent expressions are not reified and functions are used instead, then the search for a program that solves a programming problem could return programs that solve the problem trivially by using functions that solve the whole problem, or even uncomputable functions. For example, a programming problem requesting a program that determines if an integer is a prime, could be solved by the trivial program \( \text{if } p \text{ then } \text{Yes else No} \), where \( p \) is a function that tests if the integer
problem could succeed because an uncomputable function was present in the solution. What we really want are analyzable expressions that refer only to a limited set of functions which can be easily computed. If those expressions are reified as part of programs, the allowed bunch may be characterized to allow only programs using adequate forms of expressions where only the allowed functions occur.

Functional programming

Another option which is made possible by the reification of expressions is the development of a functional programming language. This can be done by extending the range of admissible functions in expressions until this language of functions is sufficiently rich that a program can easily meet its specification by evaluating expressions, and rarely or never relying on imperative constructs such as assignments. This requires the consideration and axiomatization of the effects on the state of the evaluation of expressions. Some functional programming constructs could be axiomatized by establishing their equivalence to imperative alternatives.

Syntax of expressions

Another benefit of reified expressions is that their syntax may be made more convenient. Without reified expressions, assigning the sum of two fluents to another fluent would be done this way

\[ f := \lambda s \cdot \text{Val} \ g \ s + \text{Val} \ h \ s, \]

while with reified expressions this could be expressed as

\[ f := g + h, \]

if \(+\) were defined on expressions. To achieve the simpler syntax, fluents and constants must be declared as simple expressions, and functions such as \(+\) must be extended to apply on expressions to produce more complex expressions. However overloading functions in this manner is ambiguous in cases such as \(3 + 4\), which should sometimes yield 7, and
sometimes yield an expression evaluating to 1 by adding 3 to 4. Using a different symbol is therefore preferable. The convention suggested in [Levesque94] to denote such functions is to place a hat (\(\wedge\)) over the normal function’s symbol. The above assignment should therefore be rewritten as

\[ f := g\wedge h. \]

This inconvenience could be dismissed by adopting a syntactic convention saying that the \(\wedge\) is implicit where expressions are expected. However, for the theorem provers which are likely to be used on large expressions, the elegance of the syntax is not an important consideration.

**Rational communication between agents**

Some work has been done to derive variants of the situation calculus which allow interactions between rational agents. In this context, rational agents are processes which obtain, maintain, and act on rational knowledge about the world. A formalization of this is presented in [Shapiro95]. The communication of knowledge, treated in [Marcu95] and [Lespérance96], may involve the transmission of complex logical sentences about the world. Modal operators are used to operate on these sentences. However with reified expressions it becomes possible to transmit logical sentences in the form of expressions, and operate on them using standard functions. Although Hehner’s theory and the situation calculus may provide different concurrency and communication features, it should be possible to implement this approach in both formalisms.

**4.2.2 Axiomatization of reified expressions**

The function \(Val\) is the natural candidate for the evaluation of expressions. Its domain of application simply needs to be extended beyond fluents to cover any state dependent expression. The extension of \(Val\) may be axiomatized with axioms such as

\[ Val (a\wedge b) s = Val a s + Val b s \]

for each function which may be used in expressions.
like the ones described earlier for programs. We do not present them here since their details are uninteresting.

**Side effects of expressions**

In cases where the evaluation of expressions must affect the state, a predicate like Final should be defined that takes an expression and an initial situation and returns the state resulting from the evaluation. If an addition takes 1 unit of time, and time is simply represented as a fluent \( t \), Final would be defined as

\[
(Final (a \oplus b) s = s') = \exists s'': state \cdot s'' = Final b (Final a s) \\
\quad \land Val t s' = 1 + Val t s'' \\
\quad \land \forall f: fl \cdot f \neq t \Rightarrow Val f s' = Val f s'').
\]

Note that this allows expressions which would have side effects other than the passage of time. If there are such expressions, or if expressions may refer to the time, and one expects the side effects of a subexpression to affect the rest of the evaluation of an expression, then the axioms for Val must be modified to evaluate subexpressions sequentially:

\[
Val (a \oplus b) s = Val a s + Val b (Final a s)
\]

Some axioms which define program semantics, those where the evaluation of expressions takes place, must be slightly modified to refer to the state resulting from the evaluation. For assignments we get:

\[
Do (f := v) s s' = (Val f s' = Val v s) \\
\quad \land \forall f': fl \cdot f' \neq f \Rightarrow Val f' s' = Val f' (Final v s)),
\]

which means that the effect of the evaluation occurs before the effect of the assignment. And for conditionals the axiom is modified to:

\[
(if b \textbf{ then } P \textbf{ else } Q) s s' = ((Val b s) \land P (Final b s) s' \\
\quad \lor \neg(Val b s) \land Q (Final b s) s').
\]

45
This section describes reasoning with multiple representations, a technique which is made possible by the reification of fluents, and by the use of one axiom to specify the effect of each action.

An example of multiple representations can be found in software operating on binary hardware, where abstract data, such as a matrix, are represented as series of bits. Sometimes the abstract view must be used, and sometimes the concrete view must be used. Some actions, such as the stepping of a microprocessor through a cycle, must be specified in terms of low level fluents such as words of bits, while others should be specified at a higher level in terms of their effects on abstract structures, such as transposing a matrix.

This presents a problem, if there are abstract and concrete fluents which represent the same object in different forms, then there must be a state constraint which ensures that the fluents always change in a compatible manner. However state constraints [Lin94] introduce difficulties such as the ramification problem, which would require significant modifications to the specification of the effect of actions.

Here however the state constraint is very simple, it maintains a bijection between two fluents or sets of fluents. This suggests a solution that handles only this particular case to enable reasoning using multiple representations. Consider a fluent \( n \) which may take any value among 0, 1, 2 and 3, and two fluents \( b_1 \) and \( b_0 \) which may take only the values 0 and 1. The fluents \( b_1 \) and \( b_0 \) will be the most and least significant bits, respectively, of \( n \)'s binary representation. To make it more realistic and interesting, this example could be extended to a number of bits more typical of the word sizes in current computers, but using only two bits will suffice to illustrate the principle. We could say that \( b_1 \) and \( b_0 \) are the only real fluents, and that \( n \) is a defined fluent, which cannot be modified directly but only as a side effect of modifying \( b_1 \) and \( b_0 \). Alternatively, we could say that \( n \) is the only real fluent and that \( b_1 \) and \( b_0 \) are defined fluents. These approaches solve the problem but only by restricting the use of defined fluents to a read-only policy. Another problem with these approaches is that there are two of them, among which we must choose the most appropriate, and hope that we do not come to regret our choice and favor the other, since this may imply the reformulation of the effect of many actions.
A better approach is made possible by the way we axiomatize the effect of actions. Since each action has its axiom, we may use a different set of "real" fluents, which we call primitive fluents from now on, for the axiomatization of the effect of each action. The significance of primitive fluents is that they are the ones in terms of which the effect of an action is specified. Together, they fully specify the state of the system. Assuming that the bunch *other* contains fluents which, together with *n* or *b*₁ and *b*₀, fully specify the state, then the two bunches (*other*, *n*) and (*other*, *b*₁, *b*₀) are possible choices for primitive fluents. Instead of always using the same bunch of primitive fluents *fl* that was used in the previous axioms, we can choose which of (*other*, *n*) and (*other*, *b*₁, *b*₀) is the most appropriate for the action under consideration. To increment *n* by 1, the best bunch would be (*other*, *n*), but to rotate the bits the best bunch would be (*other*, *b*₁, *b*₀). We could axiomatize the fluents and the actions as:

\[
Val\ n\ s = 2 \times Val\ b_1\ s + Val\ b_0\ s
\]

\[
Do\ (incr\ n)\ s\ s' = (Val\ n\ s' = (1 + Val\ n\ s) \mod 4
\]
\[
\quad \land \forall f:\ other \cdot Val\ f\ s' = Val\ f\ s)
\]

\[
Do\ (rotate\ n)\ s\ s' = (Val\ b_1\ s' = Val\ b_0\ s \land Val\ b_0\ s' = Val\ b_1\ s
\]
\[
\quad \land \forall f:\ other \cdot Val\ f\ s' = Val\ f\ s).
\]

Here the bunch *other* is quite convenient, but was defined specifically for the axioms that modify *n*, *b*₁ or *b*₀. If other fluents have multiple representations it would not be practical to define a bunch for each. A solution could be to define a default bunch *fl* to be used in the axioms of all actions. It would include either *n* or *b*₁ and *b*₀, the particular choice being irrelevant, and similar arbitrary choices for other groups of fluents. Then each axiom would exclude the fluents it modifies from the quantification that specifies the fluents that do not change. For example the axiom for the increment action would appear as:

\[
Do\ (incr\ n)\ s\ s' = (Val\ n\ s' = (1 + Val\ n\ s) \mod 4
\]
\[
\quad \land \forall f:\ fl \cdot (f : n, b_1, b_0) \Rightarrow Val\ f\ s' = Val\ f\ s).
\]

47
Other examples where many representations can designate the same object are Cartesian versus polar coordinates, the position and orientation of a solid object versus the position and orientation of a part of that object, the list of edges in a graph versus a matrix representation of the same graph, etc.

### 4.4 Using situations in Hehner’s theory

The property of a situation to determine completely the path of situations that precede it, can be put to use in Hehner’s theory to make specifications more expressive. Normally the specifications relate the initial and final states of a computation, and they have no way of imposing condition on the intermediate states of the computation. However if states are replaced by situations it becomes possible to refer to any intermediate situation by working back from the final situation.

Using situations, we may add to the body of a specification, a conjunct that characterizes the requirements on the intermediate situations of the computation. If the initial and final situations are also considered as intermediate situations, such a conjunct could have the form

\[ \forall s'' : \text{sit} \cdot s \leq s'' \land s'' \leq s' \Rightarrow P s'' , \]

where $P$ is a predicate on situations and $\leq$ is simply defined as $(s_1 \leq s_2) = (s_1 = s_2 \lor s_1 < s_2)$, which relies on the $<$ operator defined in the foundation axioms of section 3.1.2. Here the predicate $P$ imposes state constraints on all situations. Such constraints can be used to impose a strict domain for fluents, ensuring that their value is always within their domain. This is particularly important if the specification is used within a programming problem to derive programs automatically: programs that “abuse” the holding capacity of fluents must be disqualified. The difficulty in satisfying a specification may well be the strict requirements on memory usage. For example a specification could request a program that can sort a list without taking much more space than the list itself takes, imposing a solution such as the bubble-sort algorithm. If there is no way of specifying that fluents can only contain a limited range of values, then there is no way of imposing space requirements.
it demonstrates the procedure by which intermediate situations may be accessed, and because in most cases this form should be sufficient to express the desired constraints.

Unfortunately there is an important limitation to this approach: since it uses situations, it is not compatible with the concurrency found in Hehner’s theory. Perhaps some treatment of concurrency from the situation calculus could be used to generalize this approach to concurrent programs.

In some cases the communication channels of Hehner’s theory may be used to impose restrictions on the intermediate states. They are somewhat superior to our approach in that they are more compatible with concurrency, but they cannot be used to impose all conceivable constraints on intermediate states. They work to the degree that it is admissible to have programs record in channels the events which the constraints seek to restrict. Channels do not change from one state to the next, they are immutable lists. A program may only write on a channel by saying that an element of the list has some value, and by moving up a cursor that will enable it to “write” the next value in the following list element. The program does not change the contents of channel elements; it only specifies their value. A specification may then refer to values stored in the channel as a log of the activity of the program. But to restrict the domain of fluents, all changes to fluent values would have to be logged in channels. This approach would be quite cumbersome. Also, in the presence of concurrency, the logging of fluent updates would have to be performed by all processes. But this is impossible because only one process may write to a channel. Using many channels is not an interesting option since channels would then have to be assigned to processes as they are created, and there would always be a program that has more processes than the chosen number of channels, which must be fixed first. Therefore neither our approach nor that of channels can accommodate all cases.

Yet another approach is introduced in [Hehner94]. There it is possible to refer to intermediate states because the logical nature of specifications is modified to allow this. A specification is seen as a relation between time points. By using arithmetic on these time points it is possible to talk about the intermediate time points. The state of the various fluents may be accessed at the intermediate time points because the fluents are
expressing fluents as functions of the state, but this convention is independent of using time points as states: our convention of using *Val fluent state* (as *Val fluent time*) could also be used. This approach allows specifications to constrain the behavior of programs at intermediate steps, while handling both continuous or discreet time.

However there is one advantage to using situations instead of time points. The operations defined on time allow us to refer to intermediate times in a computation. This is also possible with situations. But the operations defined on time do not allow us to refer to the actions that effect the transitions between the time points. The operations defined on situations do allow us to talk about these actions. However the importance of this distinction is reduced by the possibility of inferring which actions occurred by studying the changes in the value of fluents.

While it may be preferable in some cases to use situations instead of states, it may also be preferable to use time points instead of situations in some of these cases.

4.5 Recursion and consistency

Logics with reified sets or predicates are often approached with great caution because of the difficult issues that can arise, such as Russell’s paradox. We may then wonder if considering reified programs is not also risky.

However the original cause of difficulties is not reification but recursion. Although this may not be immediately obvious, Russell’s paradox consists in defining a set recursively. When we say that *A* is the set of all sets that do not contain themselves, we refer to all sets, including an eventual solution for *A*. This recursion is hidden since a simple syntactic search for occurrences of the symbol *A* in the definition does not expose the recursion.

The inconsistencies that can result from assuming that recursive definitions have solutions are not limited to complex logics. Even in propositional logic there are recursive definitions without a solution, such as the simple *P = ~P*. What complex logics with reified constructs can do that cannot be done so easily in simpler logics, is to hide recursion.
Therefore concerns about consistency must be addressed by studying recursive definitions. Reification is relevant only to the degree that it may enable the creation of additional recursive definitions.

We already discussed recursively defined programs (on pages 28–30) which can lead to multiple solutions. If negation was a programming construct, it would be easy to specify contradictory recursive definitions such as \( P s s' = \neg P s s' \). There is no such programming construct, but if conditions are allowed to refer to programs, it becomes possible to specify contradictory recursive definitions. For example let \( a \) be any action that changes the state, then

\[
Do P = \text{if } Do P s s \text{ then } a \text{ else } \text{ok}
\]

has no solution for \( P \), since we can easily derive from it that \( Do P s s = \neg Do P s s \). Actually in Hehner's theory \( Q = \neg Q \) is not a contradiction since the bunches \textit{null} and \((T, \bot)\) are solutions, but these solutions are not useful here and we may exclude them by adding an axiom \((\not\in (Do P s s')) = 1\).

The possibility of specifying contradictory recursive definitions is certainly a problem, but it is not a result of reifying programs: it also occurs with programs as relations. For example the definition

\[
P = \text{if } P s s \text{ then } a \text{ else } \text{ok}
\]

is also contradictory (assuming we again exclude the non-elementary solutions mapping to \textit{null} or \((T, \bot)\)).

Because of such contradictions, we cannot expect to simply pass any recursive definition of a program to an interpreter and have it execute the program. Some recursive definitions have no solution. In such cases, the interpreter should ideally complain that there is no program satisfying the given axioms, and that therefore the recursive definition is inadequate. Although this is uncomputable, an interpreter could guarantee correct behavior for some types of recursion that are known to be safe, and it could reject definitions for which it cannot prove in a reasonable time that they have a solution.
Chapter 5

Summary and Future Work

We bridged a large part of the gap between the situation calculus and Hehner’s theory. We did this by first expressing the former in the logic and syntax of the latter, and then by reifying and unifying the concepts of the two formalisms.

By porting the situation calculus to the logic of Hehner’s theory, we reified booleans, functions, and sorts. We then reified fluents, which allowed us to refer to components of the state of a system in the same manner in both formalisms. Next, the reification of programs provided a single representation for them. This facilitated the identification of similar or identical axioms in the axiomatization of programming constructs. At that point the formalisms were identical except for the use of situations versus states, differences in some of the programming constructs, and different views of time and concurrency. By studying these areas where the formalisms differ, we observed that many of those differences could be migrated from one formalism to the other.

We also proceeded with the reifications in order to increase the expressiveness of the formalisms. The reification of programs was not absolutely necessary to give programs the same form in both formalisms: we mentioned that the GOLOG axioms could be translated directly to axiom schemas in Hehner’s theory. But the reification of programs has advantages which we discussed in chapter 4. We also extended the reification of programs to include the state dependent expressions found in programs. We presented some of the advantages of the reification of expressions such as the ability to represent functional programming with side effects, and the ability to express within the logic the
We conclude that reification is a powerful technique for increasing the expressiveness of the formalisms. The added expressiveness enables new applications of the formalisms, so more work is necessary to explore this potential.

We also conclude that the formalisms that result from the reifications are very similar, and that many of the remaining differences between the formalisms could be migrated between them. This suggests that future work on these differing issues, in either formalism, could be facilitated, improved or supported in its conclusions by taking into account the similar work done in the other formalism.
Bibliography


Raymond Reiter, “Natural actions, concurrency and continuous time in the situation calculus”, *Principles of Knowledge Representation and Rea-