INDIGOLOG: EXECUTION OF GUARDED ACTION THEORIES

by

Sebastian Sardiña

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

Copyright © 2000 by Sebastian Sardiña
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.
Abstract

IndiGolog: Execution of Guarded Action Theories

Sebastian Sardiña

Master of Science

Graduate Department of Computer Science

University of Toronto

2000

In AI, the problem of selecting (high-level) actions in dynamic and not completely predictable environments translates into the problem of designing controllers that can map sequence of observations into actions so that certain goals are achieved.

One approach to high-level controllers is *high-level programming*. Basically, we imagine a system executing a high-level program with respect to a background theory of action. Whereas the background theory describes the characteristics of the domain (action preconditions, action effects and non-effects, etc.), the high-level program provides strong, but usually incomplete, clues about what the desired sequence of actions should be like.

The work presented here combines two recent, but unexplored ideas: *Guarded Action Theories* and *Incremental Program Execution*. The result is a new high-level programming language, which we call *IndiGolog*, whose programs, compared with previous languages, are executed in a more practical way with respect to more open-world theories. We provide a theoretical exploration of both guarded theories and program execution, and develop a Prolog implementation for IndiGolog.
Acknowledgements

I am very grateful to Hector Levesque, my advisor, for so many helpful discussions and his continuous guidance at every stage of my work. Besides, I want to thank him for making the relation supervisor-student so comfortable and professional. My thanks go also to Raymond Reiter, my second reader, for taking the time to read the whole thesis and for his suggestions.

Thank you also to my first four friends in Canada, Carlos Hurtado, Jack Kwong, Flavio Rizzolo, and Alfredo Gabaldon, and to all my friends in Argentina whose support I could feel even from the distance.

Last, but certainly not least, I would like to thank Mario, Alicia and Rosana for their patience and love, as well as everything else.
# Contents

1 Introduction ................................................. 1

2 Literature Review .................................. 4
   2.1 Basic Action Theories ................................. 4
       2.1.1 Knowledge Producing Actions ..................... 10
   2.2 Towards An Alternative to Planning .................. 12
       2.2.1 Golog ........................................... 14
       2.2.2 ConGolog ......................................... 16

3 Guarded Action Theories .................. 21
   3.1 Motivations ........................................... 21
       3.1.1 Incomplete Causal Laws ........................... 22
       3.1.2 Gathering Knowledge .............................. 24
   3.2 Formal Definition of GATs ............................ 25
       3.2.1 Frame Problem and Online Sensing ................. 27
   3.3 The Projection Task in GAT ......................... 29
   3.4 Coherent Theories and Histories ..................... 31
   3.5 Sensors vs Sensing Actions ........................... 43
       3.5.1 Sensors and Knowledge ............................ 49
   3.6 Adding Functional Fluents ............................ 51
   3.7 Chapter Summary ...................................... 52
Chapter 1

Introduction

In the field of robotics, much research has been done concerning basic-level tasks like sensory processing, path planning, reactive agents, manipulator design and control, etc. The problem of selecting (high-level) actions in environments that are dynamic and not completely predictable or observable is a central problem in intelligent behavior. In AI, that problem translates into the problem of designing controllers that can map sequence of observations into actions so that certain goals are achieved. Alternatively put, we look for a high-level agent controller capable of generating actions in the world that are appropriate as a function of some current set of beliefs and desires.

Many attempts at high-level controllers are mainly based in automated planning, where the controller is derived automatically from a suitable description of the actions, sensors, and goals. However, as argued in [28], planning may not be the most appropriate tool for calculating an agent course of action in a dynamic changing world. Problems as computational intractability, lack of reactivity, lack of sensing information promptly arise with a planning strategy.

Because of that, the Cognitive Robotics group of the University of Toronto, has been working on an alternative to planning which is generally referred to as high-level programming. Basically, we imagine a system that takes as input a high-level program that
needs to be executed. and calls a program interpreter to generate primitive actions to execute it. By a high-level program. here, we mean a program whose primitive actions are domain dependent and which has non-deterministic choices that have to be filled by the interpreter. Because of these gaps that a program may contain, high-level programs resemble plans but are considerably more general. Besides, a high-level program typically provides strong clues about what the desired sequence of actions should be like while leaving blanks to be filled at the execution time.

Golog [25] and ConGolog [8] are examples of high-level programming languages where the programmer specifies, by using a rich set of programming constructs (including conditionals, loops, recursive procedures, concurrent actions, non-deterministic choice, etc.), a sketch of the controller he considers adequate. One of the important characteristics of these languages is that they derive zero, one or many possible courses of actions with respect to some model of the world. Such a model is a formal representation of the dynamics of the environment where the agent is situated and it includes, basically, the effects of primitive actions and their preconditions. To be more precise, Golog and ConGolog are - generally - assumed to work with a Basic Action Theory ([34],[33]) behind as a model of the world. A basic action theory provides a formal way of modeling a dynamic changing world where it is possible to express the frame problem solution given by Reiter [35]. It is very important to point out that both Golog and ConGolog have a formal semantic in the situation calculus as well, so that the whole approach to agent behavior turns out to be neat, principled and theoretical.

In this thesis we are concerned with the following points:

- Explore a recent extension to Basic Action Theories, namely Guarded Action Theories, proposed by Levesque and De Giacomo in [11]:

- Study in detail an alternative way of executing programs by mixing online execution with a controlled form of local lookahead. We suggest then a version of ConGolog, which we call IndiGolog (incremental deterministic Golog), and we study some
properties of a new construct: the search operator.

- Study an amalgamation of the above two ideas. Hence, we want to develop an IndiGolog interpreter, which has all the language features of ConGolog, but also works in (more) open-world settings with guarded action theories, allows for programmer control over online and offline execution, and in the online case, allows sensing information and exogenous events to affect subsequent computation.

The rest of the thesis is organized as follows. Chapter 2 gives the technical background needed for the following chapters. Chapter 3 introduces and studies guarded action theories as more general models than their predecessor basic action theories. Chapter 4 is devoted entirely to high-level programs, in particular, to their executions. There a compromise between online and offline executions is studied and some particular points related to concurrency and sensing are explored. Finally, Chapter 5 is concerned with developing an implementation of an interpreter for IndiGolog programs, that is for programs that execute as suggested in Chapter 4 with respect to theories of actions of the sort presented in Chapter 2. Surprisingly, the flexibility of those theories of action together with prioritized concurrency and search makes more difficult and subtle the task of obtaining a sound implementation for IndiGolog. Technically, Chapters 3 is purely situation calculus and theories of action: Chapter 4 is purely programs: and Chapter 5 is mainly Prolog and proofs of soundness and completeness.
Chapter 2

Literature Review

When it comes to modeling the behavior of agents at least two aspects should be addressed. First, it is mandatory to have a model of the world in which the agent is situated. Second, a mechanism for obtaining the agent behavior depending on goals, knowledge and abilities is needed as well.

This chapter is concerned with describing the two particular approaches to the above two issues that give the technical and historical background for the remaining chapters.

2.1 Basic Action Theories

As we said, the first problem we face whenever we want to model agent behavior is the need of a world-model. Many approaches in the literature made use of the situation calculus language to address this aspect. The situation calculus [30] is a second order language specifically designed for representing dynamically changing worlds where all changes to the world are the result of named actions.

In the language of the situation calculus, a possible world history, which is simply a sequence of actions, is represented by a first order term called a situation. The constant $S_0$ is used to denote the initial situation where no action has occurred yet. A sequence of action starting from a situation is expressed using the function symbol $do$
such that $do(A, s)$ denotes the successor situation to $s$ resulting from executing the action $A$. Actions themselves are functions with or without arguments so that they can be parameterized: action $\text{deliver}(x, y)$ might stand for the action of delivering package $x$ to customer $y$. A distinguished predicate $\text{Poss}(a, s)$ is used to state that it is possible to perform an action $a$ in situation $s$. Also, a binary predicate $\sqsubset$, whose arguments are situations, defines an ordering relation on situations. The intended interpretation of situations is as action histories, in which case $s \sqsubset s'$ means that $s$ is a proper subhistory of $s'$.

Finally, state attributes are modeled with relations and functions that may vary their values from situation to situation. They are called fluents, either relational or functional, and they take a situation term as their last argument. For example, $\text{light}(x, s)$ might stand for the light $x$ being on in situation $s$; and $\text{priority}(x, s)$ might stand for the priority of customer $x$ in situation $s$.

Although we have a language where we can naturally express things about dynamically changing worlds, it remains to be seen how we deal with two difficulties that arise in any attempt to formalize dynamic worlds. Moreover, it would be nice to deal with a computational attractive representation model.

A basic action theory is a world axiomatization which provides a way of modeling the dynamics of the world under some plausible assumptions. As explained in [34], a formula of the language is uniform in $s$ iff it does not mention the predicate $\text{Poss}$ or $\sqsubset$, it does not quantify over variables of sort situation, it does not mention equality on situations, and whenever it mentions a term of sort situation in the situation argument position of a fluent, then that term is $s$.

**Definition 2.1.1:** A basic action theory is any collection of axioms of the form

$$\mathcal{D} = \mathcal{D}_{FL} \cup \mathcal{D}_0 \cup \mathcal{D}_{ss} \cup \mathcal{D}_{Poss} \cup \mathcal{D}_{una}$$

where.
• $D_0$ is the set of axioms describing the initial situation $S_0$. In other words, it is a set of first order sentences that are uniform in $S_0$. $D_0$ need not to be a complete characterization of the world. i.e. there may be unknown things in the initial situation.

• $D_{pos}$ contains a precondition axiom for each action $A$ of the form:\(^{1}\)

$$\text{Poss}(A(x_n), s) \equiv \Pi_A(x_n, s)$$

where $A$ is an n-ary function symbol, and $\Pi_A(x_n, s)$ is a formula that is uniform in $s$. Precondition axioms state whether or not an action is executable in a situation. Dropping an object is only possible if one actually is holding it.

• $D_{ss}$ is the set of successor state axioms for relational and functional fluents.

For a relational fluent $F(\bar{x}, s)$ the successor state axioms is a sentence of the form:

$$F(\bar{x}, do(a, s)) \equiv \Phi_F(\bar{x}, a, s)$$

where $\Phi_F(\bar{x}, a, s)$ is a formula uniform in $s$.

For a functional fluent $f(\bar{x}, s)$ the successor state axioms is a sentence of the form:

$$f(\bar{x}, do(a, s)) = y \equiv \sigma_f(\bar{x}, y, a, s)$$

where $\sigma_f(\bar{x}, y, a, s)$ is a formula uniform in $s$ and such that

$$D_{una} \cup D_0 \models \forall \bar{x} \exists y. \sigma_f(\bar{x}, y, a, s) \land [\forall y. y' \sigma_f(\bar{x}, y, a, s) \land \sigma_f(\bar{x}, y', a, s) \supset y = y']$$

• $D_{una}$ is the set of unique names axioms for actions.

• $D_{fui}$ is a set of foundational, domain independent axioms.

\(^{1}\)From now on, free variables are assumed to be universally quantified from the outside.
As pointed out by Ginsberg and Smith ([17],[18]), “reasoning about action gives rise to three classical problems”. The first problem is the frame problem, recognized and so named by McCarthy and Hayes [30]: the difficulty is that of indicating all those things that do not change as actions are performed and time passes. For example, when the elevator goes up one floor, we need to determine that its door remains closed. The second problem is the ramification problem: the difficulty here is that it is unreasonable to explicitly record all those things that do change as actions are performed. For any given action there are essentially an infinite number of possible consequences that might occur. For example, when the elevator moves everything attached to it moves as well. The third problem is called the qualification problem, also named by McCarthy [30]: the difficulty is that the number of preconditions for each action is immense.

In a basic action theory, the qualification problem is quickly solved by ignoring all “minor” qualifications in favor of necessary and sufficient conditions defining when an action can be performed. Although it is an idealized approach, it will be realistic for our main objective, namely high-level reasoning for agents.

What is interesting about basic action theories is that, given that it is assumed that it is possible to characterize all the conditions under which an action \( A \) can lead to a fluent \( F \) becomes true or false, we can apply to them a particular attractive solution to the frame problem proposed by Reiter’s [35] work. The idea is to restrict the “syntactic” form of successor state axioms, more precisely, to restrict the right hand side of them. The solution is derived from Pednault [32] and Schubert [39] earlier proposals for solving the frame problem. The trick is to make \( \Phi_F(\bar{x},a,s) \) have the following form in each relational fluent successor state axiom:

\[
\Phi_F(\bar{x},a,s) \equiv \gamma_F^+(\bar{x},a,s) \lor F(\bar{x},do(a,s)) \land \gamma_F^-(\bar{x},a,s)
\]

where the background theory should entail

\[
\neg \exists \bar{x},a,s. \gamma_F^+(\bar{x},a,s) \land \gamma_F^-(\bar{x},a,s)
\]
Formula $\gamma F(x, a, s)$ characterize all the conditions under which action $a$ can cause $F$ to become true in the successor state axiom, and $\gamma \neg F(x, a, s)$ characterize all the conditions under which action $a$ can cause $F$ to become false in the successor state axiom.

For functional fluents it is even easier. For them, we require $\sigma_f(x, y, a, s)$ to have the following form in each functional fluent successor state axiom:

$$\sigma_f(x, do(a, s)) = v \equiv \gamma_f(x, y, a, s) \lor \neg \gamma_f(x, y, a, s)$$

This solution is attractive since it is only needed $F$ axioms to model the dynamics of all fluents, compared with the $2 \times A \times F$ explicit frame axioms that would otherwise be required. Here $F$ is the number of fluents and $A$ is the number of actions. Moreover, since the intuition leading to the frame problem is that most actions do not affect the fluent, each successor state axiom is expected to be short. One of the main disadvantages of this solution, apart from the causal laws completeness assumption, is that it is not favorable for modeling non-deterministic actions like flipping the coin which can be expressed in the Fluent Calculus ([40]) using disjunctive state update axioms.

For the sake of space, we are not going to explain all other formal aspects of basic action theories. A complete detailed explanation can be found in [34]. Intuitively we summarize some of them as follows:

- $D_{FL,N}$ are a set of foundational axioms for the situation calculus. They are domain independent and they provide the basic properties of situations (such as unique name axioms for situations) in any domain specific axiomatization for particular fluent and actions.

- Basic action theories assure us that provided the initial database together with the unique names axioms for actions are satisfiable, then unsatisfiability cannot be introduced by augmenting these with the foundational axioms for the situation calculus ($D_{FL,N}$), together with actions precondition ($D_{POSS}$) and successor state axioms. This is called the Relative Satisfiability property and it was shown in [33].
Regression arises as a sound and complete computational mechanism for an interesting set of formulas. The Regression Theorem proved in [33] shows that querying of a sentence \( W \) (a regressable sentence) wrt a basic action theory \( D \) can be reduced to a query over the initial database and unique names for actions, i.e. \( D_0 \cup D_{\text{ana}} \).

- Usually, we shall be interested only in executable situations in which actions are legal, that is their preconditions are satisfied. We say then that situation \( s \) is executable iff \( S_0 \leq s \) where \( s < s' \) means that \( s \) is an initial subhistory of \( s' \), and all the actions occurred between \( s \) and \( s' \) can be executed one after the other.

Having mentioned that there is a computational mechanism that can be applied to basic action theories, we have to state one of the most common task in AI: the projection task. Very common in planning, the projection task means to determine whether or not a formula \( \sigma \) is true in the situation resulting from performing a sequence of actions. Formally, suppose \( D \) is a basic action theory, \( a_1, \ldots, a_n \) is a sequence of ground action terms, and \( \sigma(s) \) is a formula with one free variable \( s \), whose only situation term is \( s \). The projection task is then to determine whether:

\[
D \models \sigma(\text{do}([a_1, \ldots, a_n], S_0))
\]

By \( \text{do}([a_1, \ldots, a_n], S_0) \) we denote the situation term starting from \( S_0 \), executing \( a_1 \), then \( a_2 \) and so on up to \( a_n \), i.e. the term \( \text{do}(a_n, \text{do}(a_{n-1}, \ldots \text{do}(a_1, S_0), \ldots)) \).

To motivate the following chapters we state two drawbacks of basic action theories:

1. They do not have an explicit account of sensing, a must for any autonomous agent. Information for the agent may come not only from her reasoning from the initial knowledge, but also from sensing capabilities the agent might have. For instance, the agent may obtain the state of the door which is in front of her by looking at it.

2. As already said, the causal laws for any fluents are required to be complete. Although we may have uncertainly about the initial state of the world, we need to
model its dynamics completely. We want then to have a more relaxed account where causal laws are not always known.

2.1.1 Knowledge Producing Actions

As said before, sensing is a major aspect in any realistic scenario. In [38], an approach to incorporating knowledge producing actions into the situation calculus is described. An example of a knowledge producing action is a sense operation performed by a robot to determine whether a certain door is open or closed. The idea behind the approach is to define an epistemic fluent to keep track of all the worlds an agent thinks it might possibly be in. This is called the "possible world analysis of knowledge" due to Moore [31], who was inspired by the work done in philosophy where "instead of talking about the individual propositions that an agent knows, we will talk about what states of affairs are compatible with what she knows". This approach to knowledge is very clearly described in Halpern’s survey of knowledge [20], where he calls Moore approach “the classical model” and where he also explains one of the major drawbacks of the classical model: logical omniscience.

To keep track of this possible worlds, Scherl and Levesque define an epistemic fluent \( K(s',s) \) meaning that in situation \( s \) as far as the agent can tell, it might be in the alternative situation \( s' \). That is, the agent's knowledge is such that \( s \) and \( s' \) are indistinguishable.

An agent \( A \) knows \( o \) true/false in situation \( s \) iff \( o(s') \) is true/false in every \( s' \) such that \( K(s',s) \) holds. Given that \( K \) is no more than a fluent, the whole approach rests on the specification of a successor state axiom for the \( K \) relation. The idea is that after performing an action, the agent should know that such an action was performed plus the truth value (or denotation) of the formula (or term) being sensed by the action, if any.

As an example, if \( Sense_P \) determines whether or not the fluent \( P \) is true and \( Sense_P \) is
the only sensing action, the successor state axiom for \( K \) would be:

\[
K(s'', \text{do}(a, s)) \equiv \exists s'. [(K(s', s') \land s'' = \text{do}(a, s')) \land \\
((a \neq \text{Sense}_p) \lor (a = \text{Sense}_p \land P(s) \equiv P(s')))]
\]

Probably the most interesting result of Sherl and Levesque work is that it is possible to come up with a single successor state axiom for the \( K' \) relation. This means that it is only necessary to specify \( K' \) for the initial situation. With the successor state axiom for \( K' \) it is possible to update the accessible relation when actions are performed. And, as a consequence, it is possible to reason about knowledge as actions are executed.

Assume that there are \( n \) knowledge producing actions \( a_1, ..., a_n \) and therefore \( n \) associated formulas \( \phi_1, ..., \phi_n \). The form of the successor state axiom for \( K' \) is then as follows:

\[
K(s'', \text{do}(a, s)) \equiv \exists s'. [K'(s', s) \land (s'' = \text{do}(a, s')) \land \\
((a \neq a_1 \land ... \land a \neq a_n) \\
\lor (a = a_1 \land \phi_1) \\
\lor (a = a_2 \land \phi_2) \\
\lor (a = a_n \land \phi_n)]
\]

This successor state axiom is restricting the situations accessible from \( \text{do}(a, s) \) so that the following two points are satisfied:

- The agent knows that the action \( a \) has been performed:

- The agent knows the formula or the term that \( a \) senses, if any.

As we can see, the \( K' \) relation at a particular situation \( \text{do}(a, s) \) is completely determined by the relation at \( s \) and the action \( a \). We will see that this is not the case in more flexible action theories. It is worth remarking that although the Sherl and Levesque work provides a clean and powerful account of knowledge for basic action theories, some problems arises associated with implementation. More precisely, specifying the initial
situation may involve having to list out an exponential number of possible worlds depending on how many things are unknown at the beginning. To overcome this problem, [16] proposed an alternative formalization of knowledge within the situation calculus based on arithmetic intervals which provides a finite way to represent uncertainty about a large, possible uncountable, set of alternatives. In such a proposal, we may say things like \( \text{temperature(s)} = [-10, 10] \) to denote that the temperature is known to be between -10 and 10. Sensing actions are then used to narrow the interval and knowledge can be restated to work with intervals as well.

Nonetheless, we just want to make clear that there has been some interesting work to accommodate sensing inside basic action theories. They all rest on the concept of knowledge producing actions, or simply sensing actions, which "sense" a formula or a term in a given situation. A different account of sensing will be given later for a different kind of action theory.

### 2.2 Towards An Alternative to Planning

Our last objective is to develop a system capable of generating actions in the world that are appropriate as a function of some current set of beliefs and desires. The classical way of accomplishing this is via *automated planning*: given a goal to achieve together with a description of some initial state of the worlds, and the prerequisites and effects of a set of primitive actions, find a sequence of actions that satisfy the goal, and then hand them over to the agent (robot) for execution. Bonet and Geffner [4] is a recent proposal in that way, and [1-4] is a comprehensive treatment on Planning and Control in AI.

However, planning becomes infeasible as soon as we are in a non-simple domain. What is worse, planning systems are generally expected to generate a sequence of actions without considering the result of sensing and they lack of reactivity to unexpected situations. Last, but not least, by using classical planning to model agent behavior we waste basic
- but very useful - rules of behavior for the agent. If we were able to state these basic rules, we may heavily reduce the amount of computational effort required.

An alternative to planning for high-level agent behavior was suggested in [28]:

| Instead of taking as input a goal that needs to be achieved and calling on a planner to generate primitive actions to achieve it, we imagine a system that takes as input a high-level program that needs to be executed, and calls a program interpreter to generate primitive actions to execute it. |

By a high-level program we mean one in which the primitive actions are domain dependent. For instance, \textit{pickup}(x) may be a primitive action denoting the action of picking up object \textit{x}. Moreover, tests in high-level programs pertain to conditions in the world where the agent is: if \textit{the door is open} then ... else ... Finally, high-level programs may be nondeterministic in the sense that they may contain choice points where the interpreter must make a reasoned selection that correctly satisfies some later constraints: \textit{choose any light that is on} and then ...

So high-level programs resemble plans in some way. However, they are not the solution itself but just a sketch of one or many solutions. To find the solution we should find a sequence of primitive actions that results in a legal execution of such program. But what is the advantage of high-level programs then? Mainly, they provide strong clues about what the desired sequence of primitive actions should be like. This has the effect of severely reducing the amount of search needed compared to traditional planning. The programmer is then asked to give a general sketch of the solution via a rich programming language and leave the details to be filled to the interpreter.

Clearly, a high-level program needs to have a model of the world such as a basic action theory of section 2.1 for instance. We now give two programming languages which give a formal account of the ideas just explained. As with basic action theories, maybe the most important attribute of these two languages is that they both have a principled, theoretical account by resting on a situation calculus specification. Following [24], we
will keep the following rule for all changes and extension: No implementation without a situation calculus specification.

### 2.2.1 Golog

Golog [25] was the first high-level programming language presented in the spirit of was described above. Golog programs are able to reason about the state of the world and consider the effects of various possible courses of action before committing to a particular behavior. For that, Golog's interpreter automatically maintains an explicit representation of the dynamic world being modeled, on the basis of user supplied axioms about the preconditions and effects of actions and the initial state of the world.

In particular, Golog's executions are relative to a basic action theory with the solution to the frame problem already presented which provides a situation calculus-based approach for representing and reasoning about simple actions. Thus, Golog provides more complex actions so as to be able to write rich high-level programs that will be later interpreted to get a legal execution corresponding to a desired course of action.

Golog execution is defined via a macro \( Do(\delta, s, s') \), where \( \delta \) is a program and \( s, s' \) are situations, with the intended meaning that \( s' \) is a legal execution of program \( \delta \) starting from situation \( s \). The programming language provides complex actions and procedures by expanding \( Do \) into a situation calculus formula as follows:

1. **Primitive actions**: \( Do(a, s, s') \overset{\text{def}}{=} Poss(a[s], s) \land s' = do(a[s], s) \).

   By the notation \( a[s] \) we mean the result of restoring the situation variable \( s \) to any functional fluent mentioned by the action term \( a \).

2. **Test actions**: \( Do(\sigma^?, s, s') \overset{\text{def}}{=} \sigma[s] \land s' = s \).

   Here, \( \sigma \) is a pseudo-fluent expression which stands for a formula in the language of the situation calculus, but with all situation arguments suppressed. By the notation
\( \sigma[s] \) we mean the result of restoring the situation variable \( s \) to any fluent (relational or functional) mentioned in \( \sigma \).

3. **Sequence**: \( \text{Do}(\{\delta_1; \delta_2\}. s, s') \overset{def}{=} \exists s''. \text{Do}(\delta_1, s, s'') \land \text{Do}(\delta_2, s'', s') \)

4. **Nondeterministic choice of two actions**: \( \text{Do}(\delta_1 | \delta_2, s, s') \overset{def}{=} \text{Do}(\delta_1, s, s') \lor \text{Do}(\delta_2, s, s') \)

5. **Nondeterministic choice of action arguments**: \( \text{Do}(\pi \cdot \delta(x), s, s') \overset{def}{=} \exists x. \text{Do}(\delta(x), s, s') \)

This means to nondeterministically pick an individual \( x \). and for that \( x \) perform \( \delta(x) \). This construct turns out to be very useful for robotics and similar applications.

6. **Nondeterministic iteration**: Execute \( \delta \) zero or more times.

\[
\text{Do}(\delta^*, s, s') \overset{def}{=} \forall P. [\forall s_1 P(s_1, s_1) \land \forall s_1, s_2, s_3 [P(s_1, s_2) \land \text{Do}(\delta, s_2, s_3) \supset P(s_1, s_3)]]
\]

7. **Conditional**: \( \text{Do}(\text{if } \sigma \text{ then } \delta_1 \text{ else } \delta_2, s, s') \overset{def}{=} \text{Do}(\left[ \sigma^?: \delta_1 \right] | \left[ \neg \sigma^?: \delta_2 \right], s, s') \)

8. **While-loops**: \( \text{Do}(\text{while } \sigma \text{ do } \delta, s, s') \overset{def}{=} \text{Do}(\left[ \sigma^?: \delta^* \right] | \left[ \neg \sigma^?: \delta_2 \right], s, s') \)

The only tricky definition arises for nondeterministic iterations where we need to appeal to second order logic to capture the transitive closure of \( \text{Do} \). For the sake of space, we do not explain here how recursive procedures are defined in Golog. We refer to [25] for how to do that. It is important to remark that \( \text{Do} \) is a macro that expands into a formula of the situation calculus. Therefore, \( \text{Do} \) is not in the language which means that we cannot quantify over complex actions since they are not objects of the situation calculus.

Once we have a domain description (a basic action theory together with the solution to the frame problem), a program \( \delta \), and an initial situation, what does account as a solution?. Actually, a solution is no more than an execution of program \( \delta \).

**Definition 2.2.1**: A successful offline execution of program \( \delta \) wrt a background theory \( \mathcal{D} \) starting from a ground situation \( s \) is a ground situation term \( s' \) such that \( \mathcal{D} \models \text{Do}(\delta, s, s'). \)
In particular, we just say that the ground situation term \( s' \) is a successful offline execution of program \( \delta \) wrt \( D \) iff \( D \models Do(\delta, S_0, s') \).

A successful offline execution can be obtained by looking for a *constructive proof* of \( D \models \exists s'. Do(\delta, S_0, s') \). Notice that for a realistic application it would be useless to prove that \( D \models \exists s'. Do(\delta, S_0, s') \) but not have any concrete ground situation \( s' \). What we are looking for is a situation term reflecting a course of action to actually execute in the world. We call the execution an *offline* one because the whole program is evaluated in order to obtain a complete sequence of action. After that, the sequence is given to the agent (robot) to actually perform them one by one in the world.

We finish by noting that it is possible to construct a Golog interpreter in Prolog that tries to find constructively a successful offline execution by looking for a constructive proof. Clearly, the projection task should be solved in order to do that. However, the projection task can be solved via a regression mechanism that can be implemented in Prolog as well.

### 2.2.2 ConGolog

ConGolog [9] is basically an extension of Golog that incorporates a rich account of *concurrency*. The execution task remains the same: what changes is that the programming language becomes considerably more expressive. One of the nice features of this new language is that it allows to conveniently formulate agent controllers that pursue goal-oriented tasks while concurrently monitoring and reacting to conditions in their environment.

ConGolog introduces four new constructs:

1. \((\delta_1||\delta_2)\) means the concurrent execution of programs \( \delta_1 \) and \( \delta_2 \):

2. \((\delta_1)\delta_2\) means the concurrent execution of programs \( \delta_1 \) and \( \delta_2 \) with \( \delta_1 \) having higher priority than \( \delta_2 \). Program \( \delta_2 \) executes only if \( \delta_1 \) is done or blocked:
3. \(\delta^\parallel\) means the concurrent iteration of program \(\delta\):

4. \(<\phi \rightarrow \delta>\) stands for an interrupt. Program \(\delta\) will execute some number of times. If \(\phi\) never becomes true, \(\delta\) will not execute at all. If the interrupt gets control from a higher priority process when \(\phi\) is true, then \(\delta\) will execute. Once it has completed its execution, the interrupt is ready to be triggered again.

Using concurrency it is possible to accommodate *exogenous actions*, something that cannot be done in Golog. Exogenous actions are primitive actions that may occur without being part of a user-specified program. Alternatively put, they are actions not performed by the agent itself, but still known and named ones. We assume that in the background theory, the user declares, using a predicate \(Exo\) which actions can occur exogenously. We define a special program for exogenous events \(\delta_{Exo} \overset{def}{=} (\pi a. Exo(a) : a)^*\). Executing this program involves performing zero, one, or more nondeterministically chosen exogenous events. Then we make the user-specified program \(\delta\) run concurrently with \(\delta_{Exo}\): \(\delta^\parallel \delta_{Exo}\). In this way we allow exogenous actions whose preconditions are satisfied to asynchronously occur (outside the control of \(\delta\)) during the execution of \(\delta\).

Not only did ConGolog introduce new powerful constructs, but it also gave a new semantics for programs. The kind of semantics \(Do\) associates to programs is sometimes called *evaluation semantics* since it is based on the complete evaluation of the program. Instead, ConGolog programs have a *computational semantics*, which is based on “single steps” of computation, or transitions. A step here is either a primitive action or testing whether a condition holds in the current state. Two predicates \(Trans\) and \(Final\) were introduced to give meaning to programs:

- \(Trans(\delta, s, s', s')\) is intended to say that program \(\delta\) in situation \(s\) may legally execute one step, ending in situation \(s'\) with program \(\delta'\) remaining.

- \(Final(\delta, s)\) is intended to say that program \(\delta\) may legally terminate in situation \(s\).
Final and Trans are characterized by a set of equivalence axioms. It will be necessary to quantify over programs and so, unlike with Do, we need to encode Golog programs as first-order terms, including introducing constants denoting variables. This laborious work is done in [8]. The notation $\delta |^x_v$ is $\delta$ where all free occurrences of $v$ are replaced by $x$. The set of axioms $C_F$ for Final is as follows (universally closing on $s$):

\[
\begin{align*}
\text{Final}(\text{nil}.s) & \equiv I \land E \\
\text{Final}(a.s) & \equiv \text{FALSE} \\
\text{Final}(\omega?.s) & \equiv \text{FALSE} \\
\text{Final}([\delta_1; \delta_2].s) & \equiv \text{Final}(\delta_1.s) \land \text{Final}(\delta_2.s) \\
\text{Final}([\delta_1 | \delta_2].s) & \equiv \text{Final}(\delta_1.s) \lor \text{Final}(\delta_2.s) \\
\text{Final}(\pi v.\delta.s) & \equiv \exists x. \text{Final}(\delta |^x_v.s) \\
\text{Final}(\delta^*.s) & \equiv \text{TRUE} \\
\text{Final}(\text{if } \phi \text{ then } \delta_1 \text{ else } \delta_2.s) & \equiv \phi[s] \land \text{Final}(\delta_1.s) \lor \neg \phi[s] \land \text{Final}(\delta_2.s) \\
\text{Final}(\text{while } \phi \text{ do } \delta.s) & \equiv \phi[s] \land \text{Final}(\delta_1.s) \lor \neg \phi[s]
\end{align*}
\]

Similarly, the axioms $C_T$ for Trans are as follows (universally closing on $s, \delta, s'$):

\[
\begin{align*}
\text{Trans}(\text{nil}.s, \delta, s') & \equiv \text{FALSE} \\
\text{Trans}(a.s, \delta, s') & \equiv \text{Poss}(a[s].s) \land \delta = \text{nil} \land s' = a(a.s') \\
\text{Trans}(\omega?.s, \delta, s') & \equiv \phi[s] \land \delta = \text{nil} \land s' = s \\
\text{Trans}([\delta_1; \delta_2].s, \delta, s') & \equiv \text{Final}(\delta_1.s) \land \text{Trans}(\delta_2.s, \delta, s') \lor \\
& \exists \delta'.\delta = (\delta'; \delta_2) \land \text{Trans}(\delta_1.s, \delta', s') \\
\text{Trans}([\delta_1 | \delta_2].s, \delta, s') & \equiv \text{Trans}(\delta_1.s, \delta, s') \lor \text{Trans}(\delta_2.s, \delta, s') \\
\text{Trans}(\pi v.\delta.s, \delta, s') & \equiv \exists x. \text{Trans}(\delta |^x_v.s, \delta, s') \\
\text{Trans}(\text{if } \phi \text{ then } \delta_1 \text{ else } \delta_2, s, s') & \equiv \phi[s] \land \text{Trans}(\delta_1.s, \delta, s') \lor \neg \phi[s] \land \text{Trans}(\delta_2.s, \delta, s') \\
\text{Trans}(\text{while } \phi \text{ do } \delta.s, s') & \equiv \phi[s] \land \exists \delta'.\delta = (\delta'; \text{while } \phi \text{ do } \delta_1) \land \text{Trans}(\delta.s, \delta', s')
\end{align*}
\]
We will call $\mathcal{C}$ to the set of axioms $\mathcal{C}_F \cup \mathcal{C}_T$ plus those axioms needed for the encoding of programs as first order terms. It is easy to verify that for each Golog program $\delta$:

$$\mathcal{C} \models \forall s, s'. \text{Do}(\delta, s, s') \equiv \exists \delta'. \text{Trans}^*(\delta, s, \delta', s') \land \text{Final}(\delta', s')$$

where $\text{Trans}^*$ stands for the (second-order) reflexive transitive closure of the transition relation and it is defined as follows:

$$\text{Trans}^*(\delta, s, \delta', s') \overset{\text{def}}{=} \forall T[\ldots \supset T(\delta, s, \delta', s')]$$

where $\ldots$ stands for the conjunction of the universal closure of the following

$$T(\delta, s, \delta, s) \land [\text{Trans}(\delta, s, \delta'', s'') \land T(\delta'', s'', \delta', s') \supset T(\delta, s, \delta', s')]$$

What remains is the definitions of the new constructs introduced by ConGolog. For that, we first expand the set $\mathcal{C}_F$ with the following axioms:

$$\text{Final}([\delta_1||\delta_2]. s) \equiv \text{Final}(\delta_1, s) \land \text{Final}(\delta_2, s)$$

$$\text{Final}([\delta_1])\delta_2]. s) \equiv \text{Final}(\delta_1, s) \land \text{Final}(\delta_2, s)$$

$$\text{Final}(\delta^\parallel, s) \equiv \text{TRUE}$$

Also, we extend $\mathcal{C}_T$ with the following axioms:

$$\text{Trans}([\delta_1||\delta_2]. s, \delta, s') \equiv \exists \delta'. \delta = (\delta'||\delta_2) \land \text{Trans}(\delta_1, s, \delta', s') \lor$$

$$\delta = (\delta_1||\delta') \land \text{Trans}(\delta_2, s, \delta', s')$$

$$\text{Trans}([\delta_1])\delta_2]. s, \delta, s') \equiv \exists \delta'. \delta = (\delta'||\delta_2) \land \text{Trans}(\delta_1, s, \delta', s') \lor \delta = (\delta_1||\delta') \land$$

$$\text{Trans}(\delta_2, s, \delta', s') \land \lnot \exists \delta''. s''. \text{Trans}(\delta_1, s, \delta'', s'')$$

$$\text{Trans}(\delta^\parallel, s, \delta, s') \equiv \exists \delta'. \delta = (\delta'||\delta^\parallel) \land \text{Trans}(s, \delta', s')$$

Finally, regarding interrupts, it turns out that these can be explained using other constructs of ConGolog:

$$< \phi \rightarrow \delta > \overset{\text{def}}{=} \text{while Interrupts.running do}$$

$$\text{if } \phi \text{ then } \delta \text{ else } \text{FALSE}$$
The special fluent \textit{Interrupts.running} is set to \textit{TRUE} at the beginning of the program, via a special action \textit{start.interrupts}, and set to \textit{FALSE} at the end, via a special action \textit{stop.interrupts}.

A detailed example on how ConGolog can be used to specify controllers for autonomous robots can be found in [23]. There, a controller for a mail delivery robot makes use of all Golog constructs plus concurrency (both normal and prioritized) and interrupts to react to exogenous events. Nonetheless there are still some aspects of any realistic application that are missing:

1. There is still no sensing behavior form the agent. The agent must reason about the world using only the initial knowledge and the actions performed in the world:

2. Offline execution does not seem to be neither realistic nor tractable for large programs. It makes no sense to compute a full program execution when what we are really interested in is what the agent has to do next. In the presence of exogenous events and sensing, the offline approach is even more unrealistic since plans would depend no both exogenous actions that may occur and sensing information that may be gathered in the future.
Chapter 3

Guarded Action Theories

As we saw in the previous chapter, basic action theories provide a convenient way - under certain assumptions - of formalizing dynamic changing worlds in the language of the situation calculus. More importantly, basic action theories may include a solution to the frame problem by relying on some plausible assumptions about the world being modeled. However, there are still two things these theories do not handle explicitly: (i) the ability to gather information via sensing; and (ii) the ability to express incomplete causal laws. In this chapter, we study a flexible extension of basic theories of actions, namely guarded action theories, that deal with these two problems. The objective is to have a better model of the world for constructing high-level programs for agent behavior.

3.1 Motivations

We start by explaining two reasons why basic theories may not be sufficient or practical to model the state and dynamics of the world. We hope then to motivate the need for a more flexible and powerful kind of theory. The first argument has to do with the need for having “more” open-world theories in the sense that incomplete knowledge can arise not only in the initial situation, but on any situation. The second argument has to do with the ability to gather information from the world in an intuitive manner.
3.1.1 Incomplete Causal Laws

When it comes to modeling incomplete knowledge of an agent, the ability to lose information plays an important role. There are mainly two reasons why an agent might lose part of her actual knowledge: (i) by not knowing the effects of an action; (ii) by having an incomplete knowledge of the causal laws. An example of the first case is an agent pulling a closed door. Unless she knows whether the door is locked or not, she cannot reason about the door state after the pull. However, assume that the agent knows the door is open and she goes outside. Now, unless she knows there is no one else but her, she cannot conclude that the door is still open.

This last example accounts for the second case where the agent is unaware of the actions performed by other potential agents.

While the first case can be correctly modeled with basic action theories, the second one cannot. As mentioned in Chapter 2, basic action theories require a complete description of the causal laws for each fluent. In particular, Reiter’s solution to the frame problem relies on a characterization of all the conditions under which a fluent $F$ becomes true and false. In other words, the successor state axiom of a fluent $F$ completely describes the causal laws for fluent $F$. Previous works like Pednault’s effects and frame axioms [32], and Davis/Hass/Shubert’s explanation closure axioms [39] made equivalent assumptions and, therefore, they suffer from the same limitation.

A consequence of this under Reiter’s solution to the frame problem is that whenever we execute an action totally unrelated to a fluent and that fluent was known before, the fluent remains known. For example, suppose the agent reasoning about a door state with the following successor state axioms:

$$ closed(do(a.s)) \equiv a = close \lor closed(s) \land a \neq open $$

If in certain situation, the agent knows that the door is closed, she goes outside the building, and stay there performing many actions totally unrelated to the door, she is
still able to infer that the door is closed! But, what if some unknown agent opened it while she was outside?

**Theorem 3.1.1:** Assume a basic action theory $D$ with Reiter's solution to the frame problem, a fluent $F(s)$ and a situation $S$. Let $A$ be an action term not mentioned in the successor state axiom of fluent $F$. If either $D \models F(S)$ or $D \models \neg F(S)$, then $D \models F(do(A.S))$ or $\square \models \neg F(do(A.S))$.

**Proof.** Assume $\{F(do(a.s)) \equiv \varphi_F^+(a.s) \vee F(s) \land \neg \varphi_F^-(a.s)\}$ is $F$'s successor state axiom. By hypothesis, either $D \models \forall \varphi_F^+(A.S)$ and $D \models \forall \varphi_F^-(A.S)$ hold. Hence, $D \models F(do(A.S)) \equiv F(S)$, and either $D \models F(do(A.S))$ or $D \models \neg F(do(A.S))$. $\blacksquare$

In other words, the "inertia law" is always applicable. This is a particular case of the Markov assumption, which always holds for basic theories of action, saying that truth value of a fluent can always be determined by considering the last performed action and what was true just before that action.

However, as [11] showed, we can imagine real settings where neither the Markov assumption nor the inertia law are suitable. In particular, settings where the agent cannot even reason about a fluent. These situations typically arise when a fluent might change as the result of actions or events that are exogenous to the system, i.e. not represented in the action theory: only if the agent is alone in the building, the state of the door is completely determined by the robot's open and close actions. But in a building with other occupants, doors will be opened and closed unpredictably. Another interesting example comes from the mail delivery robot [23], where the robot does not know, after starting to go somewhere, where she is until reaching the destination place. Since that example is built over a basic action theory, a distinguished place "unknown" is assumed to express that the robot's place is unknown:

$$\text{robotPlace}(p.do(a.s)) \equiv [\exists p'.a = \text{startGoTo}(p') \land p = \text{Unknown}] \land ... \quad (3.2)$$
To be able to express this example, we need a formalism allowing incomplete description about the causal law governing the fluent *robotPlace*. It is important to remark that incomplete description of the causal laws generally causes incomplete knowledge about the state of the world, and therefore, "more open theories".

We finish by noting that the possibility of expressing that a fluent $F$ is at some point exempt from the usual frame assumptions of persistence of inertia was identified in [37] as meta-property of any theory of action. In fact, this property is named there and in previous works as *occlusion*. Clearly, basic action theories leaves no space for such property since the assumption of persistence is always assumed.

### 3.1.2 Gathering Knowledge

Another limitation of basic theories of action has to do with the ability of gathering new information. To cope with incomplete knowledge, an agent usually has the ability to obtain - under certain circumstances - new information by "watching" the world. In fact, with exogenous events and the possibility of losing knowledge, sensing information becomes essential.

In contrast to the previous problem above, there has been an extensive work to accommodate sensing information, knowledge and ability into basic action theories. As in [3], [2] and [19], sensing information is taken from special (sensing) actions, called knowledge producing actions. These actions are assumed to have an outcome, which, together with sensing axioms, can contribute to the truth value of a fluent or an arbitrary formula [26]. Primitive actions are divided into two types: ordinary actions that change the world, and *binary sensing actions*, that is, sensing actions that tell the agent whether or not some condition $\phi_a$ holds in the current situation. To do that, it is assumed that the domain theory entails a *sensed fluent axiom* of the form $SF(a,s) \equiv \phi_a(s)$ for each sensing action $a$. Predicate $SF$ is a distinguished predicate like $\text{Poss}$, that relates actions to fluents. For example, the axiom $SF(\text{read.temp}.s) \equiv \exists x. \text{temperature}(x,s) \land x > 30$
models a sensing action \textit{read\_temp} that informs if it is too hot or not. Sensing of terms can be easily achieved by making $SF$ to be functional and write sensed fluent axioms like $SF(\text{read\_laser. } s) = n \equiv [\text{distance\_wall}(s) = n]$.

Besides, as explained in Chapter 2 (section 2.1.1), [38] proposed an account of knowledge by adapting the standard possible-world model of knowledge (Moore [31]) to the situation calculus. What is more, [1] presented a model in which it is possible to reason about an agent's probabilistic degrees of belief and the manner in which these beliefs change as various (noisy) actions are executed and where sensors are subject to error.

Although all this gives us a clear formal framework for sensing and knowledge for basic theories of action, we find it, sometimes, a bit counter-intuitive with the way we usually visualize sensing systems. Suppose that the only way of knowing the distance to the wall is by using a laser sensor which is constantly returning the actual distance to the wall. Every time we perform an action that may change the distance to the wall, the agent should perform a \textit{read\_laser} action to know the new distance. This idea of sensing information being associated with actions is, often, not the most natural. For instance, we do not think ourselves as performing a "looking" action every time we gather information from our eyes, or performing a "hearing" action every time we extract knowledge from our ears. Instead, we see our eyes and ears as "on-board" sensors that are always "running" and giving us information we may or may not use. Because of this informal argument, we wish to come up with a more intuitive way of treating knowledge producing sources. This change will have a big impact not only on the world model formalization, but also on the high-level programs as we will discuss later.

### 3.2 Formal Definition of GATs

Guarded action theories (GAT) [11] can be considered as an evolution of basic action theories in the sense that the first one completely captures the second one while providing
more flexibility, more power, and a more intuitive account of sensing.

A GAT is like a basic action theory where: (i) successor state axioms are replaced with more general versions: (ii) sensing relies on on-board sensors attached to the agent, which are represented as unary sensing functions, that provide online information from the outside.

To introduce GATs, we need to extend our original language. First, we drop the predicate $SF$ from the language, and introduce instead a finite number of sensing functions: unary functions whose only argument is a situation. Then, a laser sensor would be represented by the function $\text{laser}(s)$ and a thermometer can be represented with a function $\text{thermometer}(s)$.

Next, we classify formulas as follows: a sensor-fluent formula is a formula of the language that uses at most a single situation term, which is a variable, and this term only appears as the final argument of a fluent or sensor function: a sensor formula is a sensor-fluent formula containing no fluents; and a fluent formula is a sensor-fluent formula containing no sensors functions.\footnote{Fluent formulas are a particular case of Reiter’s [34] uniform formulas where the situation term is the variable $s$.}

A guarded successor state axiom (GSSA) for a fluent $F$ is an axiom of the form:

$$\alpha(\bar{x}.a.s) \supset [F(\bar{x}.do(a.s)) \equiv \nu(\bar{x}.a.s)]$$

A guarded sensed fluent axiom (GSFA) for a fluent $F$ is an axiom of the form:

$$\beta(\bar{x}.s) \supset [F(\bar{x}.s) \equiv \rho(\bar{x}.s)]$$

where $\beta$ is a sensor-fluent formula, $\alpha$ and $\nu$ are fluent formulas, and $\rho$ is a sensor formula.

While a GSSA provides a way of reasoning about a fluent by looking at the actions performed in the world, a GSFA provides a way of reasoning about a fluent by looking at
the on-board sensors. Notice that both reasoning mechanisms are conditioned by guard restrictions (α and 3), so that they may not always be applicable.

Having introduced our more general successor state axioms, a guarded action theory \( D \) is the union of the following sets:

- \( D_0 \) is the set of axioms describing the initial situation \( S_0 \).
- \( D_{P\text{ass}} \) contains a precondition axiom for each action.
- \( D_{GSSA} \) is the set of all GSSA. Each fluent may have zero, one or many GSSA.
- \( D_{GSFA} \) is the set of all GSFA. Each fluent may have zero, one or many GSFA.
- \( D_{una} \) is the set of unique names axioms for actions.
- \( D_{FU\text{N}} \) is the set of some foundational, domain independent axioms.

We denote with \( GSSA(F) \) and \( GSFA(F) \) the sets of GSSAs and GSFAs for fluent \( F \).

### 3.2.1 Frame Problem and Online Sensing

As it is easy to see, we can still apply Reiter's solution to the frame problem ([35]) to GATs by forcing

\[
\varphi^r(\vec{x}, s) \equiv \varphi^+(\vec{x}, a, s) \lor F(\vec{x}, s) \land \varphi^-F(\vec{x}, a, s)
\]

However, since the applicability of a GSSA is conditioned by the satisfaction of a guard, a solution to the frame problem may or may not be available. Coming to our door example, we would have the following GSSA

\[
alone(s) \supset [\text{closed}(do(a, s)) \equiv a = \text{close} \lor \text{closed}(s) \land a \neq \text{open}]
\]

where \( alone(s) \) is intended to mean that the agent is alone in the building in situation \( s \).

Similarly, the \( place(s) \) fluent of the mail delivery example [23] would be modeled by the
GSSA

\[ (-\exists p.a = startGoTo(p)) \supset \]

\[ [robot\ Place(p, do(a.s)) \equiv a = reached \land destination(p.s) \lor robot\ Place(p.s) \]

\[ \land \neg(a = reached \land destination(p.s))] \]

Because of the guard conditions, and without any additional information, when the robot starts going someplace, it does not know its position until reaching that place. Thus, the robot "loses" its knowledge about its location as soon as it executes a \textit{startGoTo}(p) action. However, this GSSA is not actually formalizing what we intuitively want. The reason is that the robot will be in the same place in all situations between action \textit{startGoTo} and action \textit{reached}. Since the robot is moving in all that situations, we expect its position not to be always the same. A GSSA that captures that would be the following

\[ (a = reached \lor robot\ State \neq \text{moving}) \supset \]

\[ [robot\ Place(p, do(a.s)) \equiv a = reached \land destination(p.s) \lor \]

\[ robot\ Place(p.s) \land \neg(a = reached \land destination(p.s))] \]

This shows that guards have a big impact on the knowledge of a fluent since they delimit the scope of a complete causal law.

In addition, in order to deal with unknown information the agent may have on-board sensors to watch the world. For instance, if being in front of the door requires a distance of less than 10 units, and the laser always returns the distance to the first object in front of us, then we may have a GSFA

\[ in\_front(s) \supset [closed(s) \equiv laser(s) < 10] \]

that says whether the door is closed or not.

As a result, we have two potential ways of reasoning about the door being closed: by reasoning via the GSSA about the actions already performed, or by watching the laser's
value with the GSFA. While the first way requires being alone in the building, the second one requires being in front of the door. It is worth pointing out that guard conditions not only serve as restrictions for the applicability of GSSA/GSFA, but can contribute to the knowledge as well: after closing the door, and later coming back in front of the door to find it open, a security guard robot would be able to infer that he is not alone by applying the contrapositive rule on the GSSA after using his laser sensor.

To show how guards make the theory more powerful, we show two more examples given in [11] that cannot be expressed in normal basic theories: (i) a warning light for an alarm can go on unpredictably and it will stay on until the robot turns it off; (ii) a robot can measure the distance to the wall by using a sonar which is only meaningful between two values. These cases would be expressed by the following GSSA and GSFA:

\[
\begin{align*}
\text{lightOn}(x, s) \lor a = \text{off}(x) \supset [\text{lightOn}(x, \text{do}(a, s)) & \equiv a \neq \text{off}(x) \land \text{lightOn}(x, s)] \\
\theta_1 \leq \text{sonar}(s) \leq \theta_2 \supset [\text{wallDist}(n, s) & \equiv \text{sonar}(s) = n]
\end{align*}
\]

Normal successor state axioms can still be expressed with guarded successor state axioms by making the guard to be a tautology. Successor state axioms 3.1 and 3.2 would become into

\[
\begin{align*}
\text{True} & \supset [\text{closed}(\text{do}(a, s))] \equiv a = \text{close} \lor \text{closed}(s) \land a \neq \text{open} \\
\text{True} & \supset [\text{robotPlace}(p, \text{do}(a, s))] \equiv (\exists p'.a = \text{startGoTo}(p') \land p = \text{unknown}) \land ...
\end{align*}
\]

3.3 The Projection Task in GAT

As stated in [31] and [25], the projection task is mandatory for agents trying to construct plans to achieve goals. As we recall, in the situation calculus, situations encode all the actions that have occurred in the world. Hence, the projection task can be viewed as asking the following question: "what is the truth value of formula \( \phi(s) \) if we were to execute a certain sequence of actions?" These is the typical case for database modeling
where the projection task reduces to checking the truth value of $o(s)$ wrt a background theory $D$ in some ground situation $S$ encoding the particular sequence of action, i.e. whether or not $D \models o(S)$.

With the incorporation of sensing information, the projection task can be restated as 

"what is the truth value of formula $o(s)$ given that we have executed a certain sequence of action, and we have obtained a particular sequence of sensors outcomes?" In this case, a situation term is not sufficient to encode everything that has happened.

**Definition 3.3.1:** A history $\sigma$ is a sequence $(\vec{\mu}_0) \cdot (A_1, \vec{\mu}_1) \cdots (A_n, \vec{\mu}_n)$ where $A_i$ is a ground action term ($0 \leq i \leq n$), and vector $\vec{\mu}_i = (\mu_{i1}, \ldots, \mu_{im})$, for $0 \leq i \leq n$, is a vector of (real) values, with $\mu_{ij}$ understood as the reading of the $j$-th sensor after the $i$-th action.

Therefore, a history contains both the sequence of actions performed in the world and all the sensing information up to that moment. Since we are not interested in the details on how sensing results are represented, we assume from now on that we have a logic with equality and some numerical account for sensing results. For instance, given that real sensor values will always be finite, we can assume that rational numbers are included in our logic.

We define then how to recover each part from a history.

**Definition 3.3.2:** Let $\sigma$ be a history. The ground situation term $\text{end}[\sigma]$ is defined inductively as: (i) $\text{end}[\vec{\mu}_0] = S_0$; and (ii) $\text{end}[\sigma \cdot (A, \vec{\mu})] = \text{do}(A, \text{end}[\sigma])$. Similarly, the ground sensor formula $\text{Sensed}[\sigma]$ is defined as $\text{Sensed}[\sigma] = \bigwedge_{i=0}^{A} \bigwedge_{j=1}^{m} h_j(\text{end}[\sigma_i]) = \mu_{ij}$, where $\sigma_i$ is the sub-history up to action $i$, i.e. $\sigma_i = (\vec{\mu}_0) \cdots (A_i, \vec{\mu}_i)$, and $h_j$ is the $j$-th sensor function.

Similar definitions were introduced in [10] for action theories where sensing is handled with sensing action. We can now define what we mean by the projection task in guarded action theories ([11]):
Given a GAT \( D \), a history \( \sigma \), and a formula \( o(s) \) with a single free variable \( s \), determine whether or not \( D \cup \text{Sensed}[\sigma] \models o(\text{end}[\sigma]) \).

We finish by noting that a kind of regression, similar to the one for basic action theories, can be defined to compute the projection task. This will be explained in Chapter 5 when talking about how to implement an evaluation procedure for GATs.

### 3.4 Coherent Theories and Histories

In [33], Pirri and Reiter proved an interesting and useful consistency result for basic action theories, namely, a Relative Satisfiability theorem that guarantees that a whole theory of action \( D \) is satisfiable, i.e. well behaved, iff \( D_{\text{ana}} \cup D_{S_0} \) is. However, this holds because the agent never learns any new information about the world apart from the initial knowledge and it always reasons in a single way, namely, the is one implied by successor state axioms. But, when introducing sensing outcomes into the framework and having many courses to reason, having a consistent initial view of the world is not sufficient to state that the agent will have a consistent view of the world at some future state: imagine the sensors saying that the door is open while we expect it to be closed?. We need stronger conditions to state when a guarded action theory is well-behaved.

Frequently, we shall be interested only in executable histories, namely, those ones where it is actually possible to perform the actions one after the other.

**Definition 3.4.1:** A history \( \sigma \) is executable wrt a GAT \( D \) iff either \( \sigma = \mu_0 \) or \( \sigma = \sigma' \cdot (A, \bar{\mu}) \). \( \sigma' \) is an executable history and \( D \cup \text{Sensed}[\sigma'] \models \text{Poss}(A, \text{end}[\sigma']) \).

The same concept can be expressed by requiring \( S_0 \leq \text{end}[\sigma] \) to be entailed by the theory \( D \cup \text{Sensed}[\sigma] \). Apart from being executable, a history should keep the theory consistent so that the reasoning about both the actions and the sensors makes sense.
Definition 3.4.2: A history $\sigma$ is coherent wrt a GAT $D$ iff it is an executable history and $D \cup \text{Sensed}[\sigma]$ is satisfiable.

Generally, we are really not interested in worlds that are impossible to reach. In such cases we require that the reasoning about both the actions and the sensors is confined only to executable histories. We can achieve that by imposing each guard condition $\alpha(\bar{f}. a. s)$ (of each GSSA), and each guard $\beta(\bar{f}. s)$ (of each GSFA) to have the following form:

$$\alpha(\bar{f}. a. s) \equiv S_0 < do(a. s) \land \alpha_1(\bar{f}. a. s) \quad (3.3)$$

$$\beta(\bar{f}. s) \equiv S_0 \leq s \land \beta_1(\bar{f}. s) \quad (3.4)$$

These two restrictions are not necessary in basic action theories given that each fluent has one and only one successor state axiom so that inconsistency cannot arise in any situation, whether executable or not. In contrast, when working with GATs it may be important to restrict the reasoning about fluents to executable histories because two contradicting conclusions in a non-legal situation would ruin the whole theory.\footnote{Observe that the new guards would not conform with the formal definition since they are not uniform in $s$. However, in practice, we can get rid of both restrictions because high-level programs will only consider executable actions at each step.}

Under assumptions 3.3 and 3.4, by knowing that the theory is satisfiable for executable situations, we know that the whole theory is satisfiable.

Theorem 3.4.1: Assume that $D$ is a GAT with restrictions 3.3 and 3.4 on all guards. $D \cup \text{Sensed}[\sigma]$ is satisfiable iff $D = \{D_{\text{GSSA}} \cup D_{\text{GSFA}}\} \cup \text{Sensed}[\sigma] \cup [\forall s. S_0 \leq s \supset D_{\text{GSSA}} \cup D_{\text{GSFA}}]$ is satisfiable.

Proof. It follows from the fact that all GSSA and all GSFA share the same formula $S_0 \leq s$ in their guards, namely, and the fact that

$$\models [(\alpha \land \beta_1 \supset v_1) \land (\alpha \land \beta_1 \supset v_1)] \equiv \alpha \supset [(\alpha \land \beta_1 \supset v_1) \land (\alpha \land \beta_1 \supset v_1)]$$

Note we can extract the formula $S_0 \leq s$ for the guards in each GSSA since $S_0 < do(a. s)$ is equivalent to $S_0 \leq s \land \text{Poss}(a. s)$.\hfill\llcorner
Coherent histories have the property that they do not rule out all possible sensor readings after performing an action whose preconditions are satisfied. After some legal sequence of action, there must be at least one reading of the sensors not contradicting the state of the world after the action.

**Theorem 3.4.2:** If \( \sigma \) is a coherent history wrt a GAT \( \mathcal{D} \), then for every action \( A \) such that \( \mathcal{D} \cup \text{Sensed}[\sigma] \models \text{Poss}(A.\text{end}[\sigma]) \) there exists a sensor-reading vector \( \vec{\mu} \) such that the history \( \sigma' = (A, \mu) \cdot \sigma \) is coherent wrt \( \mathcal{D} \).

**Proof.** Let \( M \) be a model such that \( M \models \mathcal{D} \cup \text{Sensed}[\sigma] \).

Clearly, \( M \) gives an interpretation to all sensors in situation \( \text{do}(A.\text{end}[\sigma]) \) and we use that interpretation to build the sensing outcomes \( \vec{\mu} \) after executing action \( A \). Because of our previous assumption about the logic we are dealing with, namely a logic with a number theory built-in for sensor results, there must be constants \( v_1, v_2, \ldots v_m \) such that \( M[v_i] = M[h_i(\text{do}(A.\text{end}[\sigma]))] \), where \( M[h_i(\text{do}(A.\text{end}[\sigma]))] \) denotes the interpretation of the \( i^{th} \) sensor function in \( \text{do}(A.\text{end}[\sigma]) \). Thus, if we take the vector of sensor readings \( \vec{\mu} = <v_1, v_2, \ldots v_m> \), it must be the case that \( M \models \mathcal{D} \cup \text{Sensed}[(A, \vec{\mu}) \cdot \sigma] \) due to the fact that \( \text{Sensed}[(A, \vec{\mu}) \cdot \sigma] = \text{Sensed}[\sigma] \land \bigwedge_{j=1}^{m} h_j(\text{do}(A.\text{end}[\sigma])) = v_j \) and \( M \models h_j(\text{do}(A.\text{end}[\sigma])) = v_j \) for each \( j = 1..m \). Therefore, from that and the fact that \( \sigma \) is executable and \( \text{Poss}(A.\text{end}[\sigma]) \) holds, the history \( \sigma' = \sigma \cdot (A, \vec{\mu}) \) is a coherent one.

We assume from now on that we always deal with coherent only, that is physical legal sequence of action together with non-contradicting readings of sensors.

**Definition 3.4.3:** A GAT \( \mathcal{D} \) is coherent iff it has an initial coherent history.

As one can easily guess, assuring that a GAT is coherent, i.e. that it is satisfiable, is not as easy as it used to be for basic action theories. This is due to the fact that

\(^3\models\) is used here to denote model satisfaction and not logically equivalence.
there may be multiple successor state axioms and sensed fluent axioms for a fluent, and, therefore, they should all be consistent with each other.

**Theorem 3.4.3:** Any GAT $\mathcal{D}$ entails the conjunction of the sentences obtained from the following schemas for each fluent $F$:

\[
\begin{align*}
\alpha_{i_1}(\bar{x}.a.s) \land \alpha_{i_2}(\bar{x}.a.s) & \supset \psi_{i_1}(\bar{x}.a.s) \equiv \psi_{i_2}(\bar{x}.a.s) \\
\beta_{j_1}(\bar{x}.s) \land \beta_{j_2}(\bar{x}.s) & \supset \rho_{j_1}(\bar{x}.s) \equiv \rho_{j_2}(\bar{x}.s) \\
\alpha_{i_1}(\bar{x}.a.s) \land \beta_{j_1}(\bar{x}.do(a.s)) & \supset \psi_{i_1}(\bar{x}.a.s) \equiv \rho_{j_1}(\bar{x}.do(a.s))
\end{align*}
\]

for every pair $(i_1,i_2), (j_1,j_2)$ and $(i_1,j_1)$ respectively, where $i_1,i_2 = 1..|\text{GSA}(F)|$ and $j_1,j_2 = 1..|\text{GSFA}(F)|$.

**Proof.** Suppose, by contradiction, that for some fluent $F$ there are two GSSA:

\[
\begin{align*}
\alpha_1(\bar{x}.a.s) \supset F(\bar{x}.do(a.s)) & \equiv \psi_1(\bar{x}.a.s) \\
\alpha_2(\bar{x}.a.s) \supset F(\bar{x}.do(a.s)) & \equiv \psi_2(\bar{x}.a.s)
\end{align*}
\]

such that $\mathcal{D} \not\models \forall \bar{x}.a.s.\alpha_2(\bar{x}.a.s) \land \alpha_1(\bar{x}.a.s) \supset \psi_1(\bar{x}.a.s) \equiv \psi_2(\bar{x}.a.s)$.

Thus, there is a model $M$ of $\mathcal{D}$ ($M \models \mathcal{D}$) and a variable assignment $\nu$ such that $M.\nu \models \alpha_1(\bar{x}.a.s) \land \alpha_2(\bar{x}.a.s)$, but $M.\nu \models \neg(\psi_1(\bar{x}.a.s) \equiv \psi_2(\bar{x}.a.s))$. However, since the above two GSSA are included in $\mathcal{D}$, the model $M$ should satisfy them. Therefore, $M.\nu \models F(\bar{x}.do(a.s)) \equiv \psi_1(\bar{x}.a.s)$ and $M.\nu \models F(\bar{x}.do(a.s)) \equiv \psi_2(\bar{x}.a.s)$. Next, and here comes the contradiction. $M.\nu \models \neg[F(\bar{x}.do(a.s)) \equiv F(\bar{x}.do(a.s))]$ holds. Since that is not possible, we conclude that such variable assignment $\nu$ cannot exist. Finally, it must be the case that $\mathcal{D} \models \forall \bar{x}.a.s.\alpha_2(\bar{x}.a.s) \land \alpha_2(\bar{x}.a.s) \supset \psi_1(\bar{x}.a.s) \equiv \psi_2(\bar{x}.a.s)$ because the above holds for any model $M$ of $\mathcal{D}$.

The cases for conditions 3.6 and 3.7 are similar.

Thus, any model of $\mathcal{D}$ should reason about the actions performed and about the sensors in a compatible way for each fluent. In other words, whenever there are two different ways of reasoning about a fluent, both ways should coincide.
Next, we define an interesting subset of guarded action theories where fluents can be ordered depending on how they are referenced in the guards of GSFA.

**Definition 3.4.4:** We say that \( F_1 \prec F_2 \) (\( F_2 \) depends on \( F_1 \)) if there exist a GSFA \( \{\alpha(\vec{x}, s) \supset [F_2(\vec{x}, s) \equiv \rho(\vec{x}, s)]\} \) in \( \mathcal{D} \) where \( F_1 \) is mentioned in \( \alpha(\vec{x}, s) \).

An GAT \( \mathcal{D} \) is said to be acyclic iff the relation \( \prec \) is well-founded. In such cases, we call the level of a fluent \( F \) the maximal distance in terms of \( \prec \)-chains from a bottom element of \( \prec \).

The following theorem resembles the Relative Satisfiability theorem proved in [33] for basic action theories. We restrict the theorem to acyclic theories since the proof is significantly simpler.

**Theorem 3.4.4:** Let \( \mathcal{D} \) be an acyclic GAT. Then, \( \mathcal{D} \) is coherent iff there is a model for \( \mathcal{D}_{una} \cup \mathcal{D}_0 \cup \mathcal{D}_{GSFA}[S_0] \)\(^4\) satisfying all sentences obtained from the following schemas:

\[
\begin{align*}
\alpha_{i_1}(\vec{x}, a, s) \land \alpha_{i_2}(\vec{x}, a, s) & \supset \psi_{i_1}(\vec{x}, a, s) \equiv \psi_{i_2}(\vec{x}, a, s) \\
\beta_{j_1}(\vec{x}, s) \land \beta_{j_2}(\vec{x}, s) & \supset \rho_{j_1}(\vec{x}, s) \equiv \rho_{j_2}(\vec{x}, s) \\
\alpha_{i_1}(\vec{x}, a, s) \land \beta_{j_1}(\vec{x}, do(a, s)) & \supset \psi_{i_1}(\vec{x}, a, s) \equiv \psi_{i_1}(\vec{x}, do(a, s))
\end{align*}
\]

for every fluent \( F \) and every pair \((i_1, i_2), (j_1, j_2)\) and \((i_1, j_1)\) respectively, where \( i_1, i_2 = 1..|GSFA(F)| \) and \( j_1, j_2 = 1..|GSFA(F)| \).

**Proof.**

(If) If \( \mathcal{D} \) is coherent, then it has an initial coherent history; that is, there is a history \( \sigma_0 = \mu_0 \) such that \( \mathcal{D} \cup Sensed[\sigma_0] \) has a model \( M \). Therefore, \( M \) is a model for the set \( \mathcal{D}_{una} \cup \mathcal{D}_0 \cup \mathcal{D}_{GSFA} \) as well since it is a subset of \( \mathcal{D} \cup Sensed[\sigma_0] \). What is more, given that \( \mathcal{D}_{GSFA}[S_0] \) is a particular instance of the universally quantified \( \mathcal{D}_{GSFA} \), it follows that \( \mathcal{D}_{una} \cup \mathcal{D}_0 \cup \mathcal{D}_{GSFA}[S_0] \) is satisfiable by \( M \). Finally, since \( M \) is a model of

\(^4\)\( \mathcal{D}_{GSFA}[S_0] \) means the set of formulas in \( \mathcal{D}_{GSFA} \) where the situation variable \( s \) is replaced by the constant \( S_0 \).
\[ \mathcal{D} \subseteq \mathcal{D} \cup \text{Sensed}[\sigma_0], \] by Theorem 3.4.3. \( M \) satisfies all the sentences derived from the above three schemas.

(Only If)

Assume \( M_0 \) is a model of \( \mathcal{D}_{\text{una}} \cup \mathcal{D}_0 \cup \mathcal{D}_{\text{GSF}}[S_0] \) satisfying all the sentences derived from the above three schemas. From it, we can obtain a model \( M \) of \( \mathcal{D} \) as follows:

1. \( M \)'s domains for sorts \textit{actions} and \textit{objects} are the same as \( M_0 \)'s ones.

2. First, we define how \( S_0 \) and \textit{do} are interpreted by \( M \) as done in [33]. With that it is immediate that \( M \), as specified so far, satisfies all the axioms of \( \mathcal{D}_{\text{FL}} \).

3. Since \( M \) and \( M_0 \) share the same domains for sorts \textit{object} and \textit{action}, we can let \( M \) interpret situation independent predicates and functions exactly as \( M_0 \) does. It follows that, given that \( M_0 \) satisfies \( \mathcal{D}_{\text{una}} \), whose sentences are conformed only from situation independent functions, so does \( M \). Furthermore, \( M \) interprets all sensing functions exactly as \( M_0 \) does which implies that all sensor-formulas will be interpreted equally by \( M \) and \( M_0 \).

4. Next, we specify how \( M \) interprets relational fluents and \textit{Poss} at \( S_0 \):

(a) Let \( F \) be a relational fluent, and \( v_0 \) a variable assignment for variables of sorts \textit{object} and \textit{action} for \( M_0 \). Since \( M \) and \( M_0 \) share the same domain of sorts \textit{object} and \textit{action}, it make sense to define: \( M, v_0 \models F(\bar{r}, S_0) \) iff \( M_0, v_0 \models F(\bar{r}, S_0) \). Given that \( M_0 \models \mathcal{D}_{S_0} \) and that \( \mathcal{D}_{S_0} \) is uniform in \( S_0 \), it follows that \( M \) is also a model of \( \mathcal{D}_{S_0} \) as well. For the same reason, \( M \) is also a model of the set \( \mathcal{D}_{\text{GSF}}[S_0] \).

(b) We give meaning to \textit{Poss} at \( S_0 \) exactly as [33] does (point 2b). Such construction guarantees that every action precondition axioms is satisfied by \( M \) at \( S_0 \).
5. At this stage in its construction, \( M \) interprets all situation independent predicates and all sensing functions. Moreover, \( M \) interprets \( \text{Poss} \) at \( S_0 \) and all relational fluents at \( S_0 \), and it does so in such a way that the action precondition axioms and the guarded sensed fluent axioms are all satisfied at \( S_0 \). We now inductively extend this interpretation for fluents to situation other than \( S_0 \). So assume that \( M \) interprets all fluents at a situation \( s \): we specify how \( M \) interprets these at situation \( do(a, s) \). Moreover, we do so in such a way that the guarded successor state axioms and guarded sensed fluent axioms will be satisfied.

Formally, suppose that \( S \in \text{Sit.} \) and that every variable assignment \( v \) that assigns \( S \) to the variable \( s \). \( M, v \) has interpreted every \( F(\vec{x}, s) \) for every \( n \)-ary relational fluent \( F \). We give meaning to all fluents in situation \( do(a, s) \) by induction on the level of fluents.

Assume fluent \( F \) is at level zero. For every variable assignment \( v \) assigning \( S \) to variable \( s \) define:

- If \( F \) has a GSFA \( \{ \beta(\vec{x}, s) \supset F(\vec{x}, s) \equiv \Phi_F(\vec{x}, s) \} \) such that \( M, v \models \beta(\vec{x}, s) \), then we define \( M, v \models F(\vec{x}, do(a, s)) \) iff \( M, v \models \Phi_F(\vec{x}, do(a, s)) \). It is very important to remark that \( \beta(\vec{x}, s) \) mentions no fluent because \( F \) is at level zero. Moreover, \( \Phi_F(\vec{x}, do(a, s)) \) is a sensor formula, and therefore, has already been assigned a truth value in \( do(a, s) \) by \( M \).

- If \( F \) has a GSSA \( \{ \alpha(\vec{x}, s) \supset F(\vec{x}, do(a, s)) \equiv \Pi_F(\vec{x}, a, s) \} \) and \( M, v \models \alpha(\vec{x}, a, s) \), then we define \( M, v \models F(\vec{x}, do(a, s)) \) iff \( M, v \models \Pi_F(\vec{x}, a, s) \). Note that both \( \alpha(\vec{x}, s) \) and \( \Pi_F(\vec{x}, a, s) \) are uniform in \( s \) and therefore, have already been assigned truth value in \( s \) by \( M \).

- If the above cases do not apply for any GSSA and any GSFA for fluent \( F \), then define \( M, v \models F(\vec{x}, do(a, s)) \) iff \( M_0, v \models F(\vec{x}, do(a, s)) \). Note that since the above two cases do not apply for any GSSA and any GSFA \( F(\vec{x}, do(a, s)) \).
is still unassigned so this step does not contradict any previous one.

Now assume that $F$ is at level $k + 1$ and that all fluents of level less or equal than $k$ have already been assigned a truth value by $M$. For every variable assignment $v$ assigning $\sigma$ to variable $s$, define:

- If $F$ has a GSFA $\{\beta(\bar{x}, s) \supset F(\bar{x}, s) \equiv \Phi_F(\bar{x}, s)\}$ such that $M. v \models \beta(\bar{x}, s)$, then we define $M. v \models F(\bar{x}, do(a, s))$ iff $M. v \models \Phi_F(\bar{x}, do(a, s))$. Notice here that $\beta(\bar{x}, s)$ can only mention fluents of level less than $k + 1$, and therefore, it has already been assigned a truth value in $do(a, s)$ by $M$. Moreover, $\Phi(\bar{x}, do(a, s))$ is a sensor formula, and therefore, has already been assigned a truth value in $do(a, s)$ by $M$ as well.

- If $F$ has a GSSA $\{\alpha(\bar{x}, s) \supset F(\bar{x}, do(a, s)) \equiv \Pi_F(\bar{x}, a, s)\}$ and $M. v \models \alpha(\bar{x}, a, s)$, then we define $M. v \models F(\bar{x}, do(a, s))$ iff $M. v \models \Pi_F(\bar{x}, a, s)$. Note that both $\alpha(\bar{x}, s)$ and $\Pi_F(\bar{x}, a, s)$ are uniform in $s$ and therefore, have already been assigned truth value in $s$ by $M$.

- If the above cases do not apply for any GSSA and any GSFA for fluent $F$, then define $M. v \models F(\bar{x}, do(a, s))$ iff $M_0. v \models F(\bar{x}, do(a, s))$.

These is well defined because of two reasons:

1. The above truth assignments to $F(\bar{x}, do(a, s))$ cannot conflict with any truth assignment made to $F$ at an early step of the construction of $M$ because those earlier truth assignments were made in situations different than $do(a, s)$.

2. Although there may be many GSSA and many GSFA in this step to apply, we know that they will not conflict because the model coincides in all applicable truth assignment. For instance, if some GSSA and some GSFA apply at the same step, that is $M. v \models \alpha(\bar{x}, a, s)$ and $M. v \models \beta(\bar{x}, do(a, s))$, then it is the case that $M. v \models \Pi_F(\bar{x}, a, s)$ iff $M. v \models \Phi_F(\bar{x}, do(a, s))$. The same happens if
two GSSA or two GSFA apply at the same step as well. Hence, all the possible ways of assigning a truth value to $F$ at $do(a, s)$ coincide.

6. Finally, we inductively extend the interpretation for $\text{Poss}$ at a situation $do(a, s)$.
Moreover, we do that in a way that action precondition axioms will be satisfied.
Thus, we need to specify how $M, v$ interprets $\text{Poss}(a', do(a, s))$ for every variable assignment $v$ that assigns $S$ to variable $s$. The construction for this is exactly parallel to that of case 4b above for $S_0$, using the fact that $M, v$ now interprets all fluents at $do(a, s)$. A consequence of this construction is that $M$ will satisfy all of the action preconditions axioms of $D_{\text{Poss}}$ at $do(a, s)$.

This complete the construction of $M$. By the nature of this construction, $M$ satisfies all the action precondition and guarded successor state axioms and guarded sensed fluent axioms.

Therefore, a GAT is coherent if and only if there is at least one reading of the sensors not contradicting the initial state and such that the reasoning about the actions and the reasoning about the sensors are compatible in any situation. As with histories, we are only interested in theories that are coherent.

In any GAT we can combine all the GSSAs of a fluent into a single GSSA without loss of expressivity.

**Lemma 3.4.1:** Let $T$ be a background theory. If $T \models \alpha_1 \land \alpha_2 \supset v_1 \equiv v_2$ then

$$T \models [(\alpha_1 \supset F \equiv v_1) \land (\alpha_2 \supset F \equiv v_2)] \equiv [(\alpha_1 \land \neg \alpha_2 \supset F \equiv v_1) \land (\alpha_2 \supset F \equiv v_2)]$$

**Proof.** $\Rightarrow$ Almost trivial given that $\models (\alpha \supset \beta) \supset (\alpha \land \gamma \supset \beta)$ for any $\alpha, \beta$ and $\gamma$.

$\Leftarrow$ Assume $M$ is a model for the set $T \cup \{\alpha_1 \land \neg \alpha_2 \supset F \equiv v_1, \alpha_2 \supset F \equiv v_2, v_1\}$. We need to prove that $M \models F \equiv v_1$. Next, we have two cases: (i) $M \models \neg \alpha_2$ or (ii) $M \models \alpha_2$. 
If (i) is the case, since \( M \models \alpha_1 \land \neg \alpha_2 \supset F \equiv \psi_1 \) and \( M \models \alpha_1 \land \neg \alpha_2 \) it is the case that \( M \models F \equiv \psi_1 \). If (ii) is the case and given that \( M \models T \) then \( M \models \alpha_1 \land \alpha_2 \supset \psi_1 \equiv \psi_2 \). From that and \( M \models \alpha_1 \land \alpha_2 \) it follows that \( M \models \psi_1 \equiv \psi_2 \). Finally, since \( M \models F \equiv \psi_2 \) we get that \( M \models F \equiv \psi_1 \) and we are done.

Furthermore, any model \( M \) of the set \( T \cup \{ \alpha_1 \land \neg \alpha_2 \supset F \equiv \psi_1, \alpha_2 \supset F \equiv \psi_2 \} \) is a model of \( \{ \alpha_2 \supset F \equiv \psi_2 \} \) and we are done.

**Lemma 3.4.2:** Let \( T \) be a background theory. If \( T \models \neg (\alpha_1 \land \alpha_2) \) then

\[
T \models [(\alpha_1 \supset F \equiv \psi_1) \land (\alpha_2 \supset F \equiv \psi_2)] \equiv [\alpha_1 \lor \alpha_2 \supset F \equiv (\alpha_1 \land \psi_1) \lor (\alpha_2 \land \psi_2)]
\]

**Proof.** \( \Rightarrow \) Assume that \( M \) is a model for \( T \cup \{ \alpha_1 \supset F \equiv \psi_1, \alpha_2 \supset F \equiv \psi_2, \alpha_1 \lor \alpha_2 \} \). Because of the assumption, there are two possible cases: (i) if \( M \models \alpha_1 \land \neg \alpha_2 \), then \( M \models F \equiv (\alpha_1 \land \psi_1) \) and \( M \models \neg (\alpha_2 \land \psi_2) \), so that \( M \models F \equiv (\alpha_1 \land \psi_1) \lor (\alpha_2 \land \psi_2) \) holds: (ii) if \( M \models \neg \alpha_1 \land \alpha_2 \), then \( M \models F \equiv (\alpha_2 \land \psi_2) \) and \( M \models \neg (\alpha_1 \land \psi_1) \), so that \( M \models F \equiv (\alpha_1 \land \psi_1) \lor (\alpha_2 \land \psi_2) \) holds.

Notice that for any model \( M \) such that \( M \models T \) it cannot be the case that \( M \models \alpha_1 \land \alpha_2 \).

\( \Leftarrow \) Assume \( M \) is a model for the set \( T \cup \{ \alpha_1 \lor \alpha_2 \supset F \equiv (\alpha_1 \land \psi_1) \lor (\alpha_2 \land \psi_2), \alpha_1 \} \). Then, \( M \models F \equiv (\alpha_1 \land \psi_1) \lor (\alpha_2 \land \psi_2) \). Since \( M \models \neg (\alpha_1 \land \alpha_2) \) it follows that \( M \models \neg \alpha_2 \) from which we know that \( M \models \neg (\alpha_2 \land \psi_2) \). Moreover, since \( M \models \alpha_1 \) then \( M \models \alpha_1 \land \psi_1 \equiv \psi_1 \) and \( M \models F \equiv \psi_1 \). If \( M \) is a model for the set \( T \cup \{ \alpha_1 \lor \alpha_2 \supset F \equiv (\alpha_1 \land \psi_1) \lor (\alpha_2 \land \psi_2), \alpha_1 \} \), the argument is symmetric.

Intuitively, two potential "overlapping", but consistent, GSSAs can be transformed into two disjoint GSSAs. From there, two disjoint GSSAs can be combined into a single one. All these suggest that, for the cases we are interested in, it is sufficient to have at most one GSSA per fluent. To state that, however, we are missing a concept of equivalence between two guarded action theories.

What is interesting about GATs is that, given that sensing is modeled separately from actions, it is possible to come up with a flexible notion of equivalence. We would like to
say that two theories of action are *equivalent* when the conclusions they draw about the world are the same, no matter how sensing is used. We are not really interested in the sensing capabilities of the agent, but only in the conclusions it derives from them.

**Definition 3.4.5:** Two GATs $D_1$ and $D_2$ are said to be *equivalent* iff

1. For any executable history $\sigma_1$ of $D_1$ there exists an executable history $\sigma_2$ of $D_2$
   where $end[\sigma_1] = end[\sigma_2]$ and for every fluent-formula $\phi(s)$. $\mathcal{D}_1 \cup \text{Sensed}[\sigma_1] \models \phi(s)$
   iff $\mathcal{D}_2 \cup \text{Sensed}[\sigma_2] \models \phi(s)$.

2. For any executable history $\sigma_2$ of $D_2$ there exists an executable history $\sigma_1$ of $D_1$
   where $end[\sigma_1] = end[\sigma_2]$ and for every fluent-formula $\phi(s)$. $\mathcal{D}_2 \cup \text{Sensed}[\sigma_2] \models \phi(s)$
   iff $\mathcal{D}_1 \cup \text{Sensed}[\sigma_1] \models \phi(s)$.

Also, two GATs $D_1$ and $D_2$ are said to be *strongly-equivalent* iff for any history $\sigma$ it is the case that $\mathcal{D}_1 \cup \text{Sensed}[\sigma] \equiv \mathcal{D}_2 \cup \text{Sensed}[\sigma]$. $\square$

Generally, we would like to know whether two theories are just equivalent instead of strongly-equivalent. This is because we are not actually interested in how the agent knows about the world, but rather in what she knows about it. Clearly, if two theories are strongly-equivalent then they are also equivalent. The converse, however, is not true. Between two equivalent theories, sensor capabilities and their results may vary, but conclusions from them should be the same.

**Lemma 3.4.3:** If $D_1$ and $D_2$ are strongly-equivalent GATs and $D_1$ is a coherent GAT, then $D_2$ is a coherent GAT as well.

**Proof.** Since $D_1$ is coherent, then there exists an initial coherent history $\sigma_0$ so that $\mathcal{D}_1 \cup \text{Sensed}[\sigma_0]$ is satisfiable. Given that $\mathcal{D}_1 \cup \text{Sensed}[\sigma_0] \equiv \mathcal{D}_2 \cup \text{Sensed}[\sigma_0]$, $\sigma_0$ is also an initial coherent history for $D_2$. $\blacksquare$
Lemma 3.4.4: Let $\mathcal{D}$ be a GAT and let

\[
\begin{align*}
GSSA_1 &= \alpha_1(\bar{x}.a.s) \supset F(\bar{x}.do(a.s)) \equiv \nu_1(\bar{x}.a.s) \\
GSSA_2 &= \alpha_2(\bar{x}.a.s) \supset F(\bar{x}.do(a.s)) \equiv \nu_2(\bar{x}.a.s)
\end{align*}
\]

be two GSSA for a fluent $F$. Then, $(\mathcal{D} - \{GSSA_1, GSSA_2\}) \cup \{GSSA_1 \setminus GSSA_2\}$ and $\mathcal{D}$ are strongly-equivalent theories, where $GSSA_1 \setminus GSSA_2$ stands for

\[
(\alpha_1(\bar{x}.a.s) \land \neg \alpha_2(\bar{x}.a.s)) \lor \alpha_2(\bar{x}.a.s) \supset [F(\bar{x}.do(a.s))] \equiv (\alpha_1(\bar{x}.a.s) \land \neg \alpha_2(\bar{x}.a.s) \land \nu_1(\bar{x}.a.s)) \lor (\alpha_2(\bar{x}.a.s) \land \nu_2(\bar{x}.a.s))
\]

PROOF. By Theorem 3.4.3, $\mathcal{D} \models \alpha_1(\bar{x}.a.s) \land \alpha_2(\bar{x}.a.s) \supset \nu_1(\bar{x}.a.s) \equiv \nu_2(\bar{x}.a.s)$. Next, by applying Lemma 3.4.1 and Lemma 3.4.2, $\mathcal{D} \models GSSA_1 \land GSSA_2 \equiv GSSA_1 \setminus GSSA_2$, so that $\models \mathcal{D} \equiv \mathcal{D} - \{GSSA_1, GSSA_2\} \cup \{GSSA_1 \setminus GSSA_2\}$ holds. From that, it follows immediately that $\mathcal{D}$ and $\mathcal{D} - \{GSSA_1, GSSA_2\} \cup \{GSSA_1 \setminus GSSA_2\}$ are strongly-equivalent.

Theorem 3.4.5: Let $\mathcal{D}$ be GAT. Then, there exist a GAT $\mathcal{D}'$ with at most one GSSA per fluent such that $\mathcal{D}$ and $\mathcal{D}'$ are equivalent.

PROOF. Straightforward by successive application of Lemma 3.4.4 for each fluent until we get only one GSSA for each fluent.

Therefore, we can always obtain theories with a single way of reasoning about the actions performed in the world. However, there may be zero, one or many ways of watching and interpreting the outcomes of sensors. This is because combining two GSFA might not lead to a GSFA since the RHS of the result may contain fluents and, hence, it would not be a sensor-formula. It is worth noting that if we require guards in GSFA to also be restricted to fluent-formulas, something that would not affect the expressivity of the system, we would also be able to combine all GSFA into a single one so that there would always be at most two ways of reasoning about a fluent in every situation.
Nonetheless - in practice - it may be better to have many GSSAs since after combining them into a single GSSA, we might need to reason by cases as the following example shows.

**Example 3.4.1:** Suppose an agent $A$ reasons about fluent $F$ using a combined GSSA $\{\alpha_1 \lor \alpha_2 \supset F \equiv (\alpha_1 \land \nu_1) \lor (\alpha_2 \land \nu_2)\}$. Assume also that $A$ knows that $\alpha_1$ is true, but knows nothing about $\alpha_2$. Unless $A$ is able to reason by cases or knows that such GSSA is a combination of two different GSSAs, she cannot conclude that $F \equiv \alpha_1$.

\[\square\]

### 3.5 Sensors vs Sensing Actions

Finally, we would like to compare the new way of modeling sensing with previous approaches ([10], [26]), where sensing information is retrieved via sensing actions.

In the presence of sensing actions, a history is a sequence of actions together with their respective outcomes. The formal definition of such histories and the definitions of $\text{Sensed}_A[\sigma]$ and $\text{end}_A[\sigma]$, which are the analogous constructs for sensing actions to the ones defined in Definition 3.3.2, can be found in [10].

As already said, sensing actions are related to fluents by sensed fluent axioms of the form $SF(A.s) \equiv \phi(s)$ for each sensing action $A$. In other words, executing $A$ in situation $s$ gives us the truth value of formula $\phi$ in situation $s$. This is one of the main differences between sensing actions and sensors given that the latter give information about the actual situation. We will explain this point below.

Given a theory of action with sensing via special actions, we can obtain a new theory based on sensors that - under some assumptions - behaves as the original one.

**Definition 3.5.1:** Let $D_{SA}$ be a theory of action based on sensing actions with only sense fluents (i.e. each sensing axiom has the form $SF(A.s) \equiv F_A(s)$), where $A$ is sensing action and $F_A$ is the fluent being sensed. We define its sensor transformation guarded action theory $D^{*}_{SA}$ as $D_{SA}$ with the following changes:
1. $D^*_s A$ has a distinguished fluent $sensing(a.s)$ where $a$ is an action variable.

2. $D^*_s A$ contains the following GSSA: \( \{ \text{true} \supset sensing(a'.do(a.s)) \equiv a = a' \} \)

3. $D^*_s A \models \forall a. \neg sensing(a.S_0)$.

4. $D^*_s A$ has a distinguished sensor function $sensor(s)$.

5. For each sensing action $A$ with a sensor fluent axiom $SF(A.s) \equiv F_A(s)$ in $D_s A$.
   $D^*_s A$ has the following GSFA:
   \[
   sensing(A.s) \supset F_A(s) \equiv (sensor(s) = 1)
   \]

6. All sensing fluent axioms are discarded.

Notice we still keep each sensing action because they may both sense and affect the world. Now, under two plausible assumptions, sensing actions can be reduced to sensors. The first assumption has to do with what is being sensed, either an arbitrary formula or a single fluent. The second assumption requires, intuitively, two things: (i) the sensing action cannot affect what is being sensed; (ii) the sensing action is fast enough to avoid the occurrence of any other action (modeled or not modeled in the theory). Assumptions (i) and (ii) guarantee that what is being sensed will never change with the execution of the sensing action in question.

**Theorem 3.5.1:** Let $D_s A$ be a theory based on sensing actions which only sense fluents and such that $D^*_s A \models [F_A(do(A.s)) \equiv F_A(s)]$ for each sensing action $A$. Let $D^*_s A$ be the corresponding sensor transformation. $\sigma$ be a legal history wrt $D_s A$, and $\sigma(s)$ be a formula mentioning neither $SF$, $sensing$, nor $sensor$. Then.

\[
D_s A \cup Sensed_A[\sigma] \models \sigma(\text{end}_A[\sigma]) \iff D^*_s A \cup Sensed[\sigma'] \models \sigma(\text{end}[\sigma'])
\]
where $\sigma'$ is constructed from $\sigma$ as follows: (i) $\sigma' = <1>$ if $\sigma = \epsilon$; (ii) $\sigma' = \sigma'_1 \cdot (A. <v>)$

if $\sigma = \sigma_1 \cdot (A. v)$

**Proof.** We show that a model in one system can be extended to be a model in the other system.

$\Rightarrow$) Assume that $D_{SA} \cup Sensed_A[\sigma] \models o(end_A[\sigma])$ and that $M^*$ is a model of the set $D^*_{SA} \cup Sensed[\sigma']$. We augment $M^*$ to obtain $M^*_{SA}$ by giving the following interpretation to $SF$ for each action $A$ and each situation $S$:

$$M^*_{SA}[SF(A.S)] = T \text{ if } M^*_{SA}[F_A(S)] = T \quad \text{where } A \text{ is a sensing action.}$$

$$M^*_{SA}[SF(A.S)] = T \quad \text{where } A \text{ is a non-sensing action.}$$

Clearly, $M^*_{SA} \models \forall_s SF(A.s) \equiv F_A(do(A.s))$ for each sensing action $A$, from which it follows that $M^*_{SA} \models D_{SA}$. Next, we prove that $M^*_{SA} \models Sensed_A[\sigma]$.

For that, remember that $Sensed_A[\sigma]$ is a conjunction of atoms of the form $SF(A.S)$ or $\neg SF(A.S)$ where $A$ is an action term and $S$ is a situation term. Take any of such atoms $\gamma$:

1. If $\gamma = SF(A.S)$ and $A$ is a sensing action, then $\sigma_1 \cdot (A.1)$ is a sub-history of $\sigma$, where $end_A[\sigma_1] = S$. Therefore, $\sigma'_1 \cdot (A. <1>)$ is a sub-history of $\sigma'$, where $end[\sigma'_1] = S$.
Thus, $sensor(do(A.S)) = 1$ is one of the terms of the conjunction $Sensed[\sigma']$ and $M^*[sensor(do(A.S))] = 1$. Moreover, $M^*[sensing(A.do(A.S))] = T$ by the GSSA for fluent sensing, and because of the GSFA for $A$, it follows that $M^*[F_A(do(A.S))] = T$. Since $M^*_{SA}$ is a model for $D_{SA}$ and the initial assumption, $M^*_{SA}[F_A(do(A.S))] = M^*_{SA}[F_A(S)] = T$ and - by the above extension - $M^*_{SA}[SF(A.S)] = T$. If $A$ is not a sensing action, then $M^*_{SA}[SF(A.S)] = T$ by the construction of $M^*_{SA}$.

2. If $\gamma = \neg SF(A.S)$ then $A$ has to be a sensing action, and $\sigma_1 \cdot (A.0)$ is a sub-history of $\sigma$, where $end_A[\sigma_1] = S$. Therefore, $\sigma'_1 \cdot (A. <0>)$ is a sub-history of $\sigma'$, where $end[\sigma'_1] = S$. Thus, $sensor(do(A.S)) = 0$ is one of the terms of the conjunction
Sensed[σ] and $M^*[\text{sensor}(do(A, S))]=0$. From the GSSA for fluent sensing, $M^*[\text{sensing}(A, do(A, S))]=\top$, and because of the GSFA of $A$ it follows that $M^*[F_A(do(A, S))]=\bot$. Again, $M^*_{\mathcal{A}}[F_A(do(A, S))]=M^*_{\mathcal{A}}[F_A(S)]=\bot$ because of $M^*_{\mathcal{A}} \models \mathcal{D}_{\mathcal{A}}$, and by the above extension - $M^*_{\mathcal{A}}[S \cdot F(A, do(A, S))]=\bot$. Note that $A$ has to be a sensing action since non-sensing actions are assumed to return 1 as sensing outcome.

To end, we note that because $M^*_{\mathcal{A}} \models \phi(\text{end}_A[\sigma])$. $\phi$ does not mention $SF$. and $\text{end}_A[\sigma]=\text{end}[\sigma']$ it has to be the case that $M^* \models \phi(\text{end}[\sigma'])$. Finally, since the above holds for any model $M^*$ of $\mathcal{D}_{\mathcal{A}} \cup \text{Sensed}[\sigma]$, it follows that $\mathcal{D}_{\mathcal{A}} \cup \text{Sensed}[\sigma'] \models \phi(\text{end}[\sigma'])$.

Assume that $\mathcal{D}_{\mathcal{A}} \cup \text{Sensed}_A[\sigma'] \models \phi(\text{end}[\sigma'])$ and assume that $M_{\mathcal{A}}$ is a model of $\mathcal{D}_{\mathcal{A}} \cup \text{Sensed}[\sigma']$. We augment $M_{\mathcal{A}}$ to obtain $M^*_{\mathcal{A}}$ by giving the following interpretation for fluent sensing and for sensing function sensor:

$$\begin{align*}
M^*_{\mathcal{A}}[\text{sensing}(A, do(A', S))]=\top & \quad \text{iff} \quad A=A' \\
M^*_{\mathcal{A}}[\text{sensing}(A, S_0)]=\bot & \quad \text{for any action } A \\
M^*_{\mathcal{A}}[\text{sensor}(S_0)]=1 & \\
M^*_{\mathcal{A}}[\text{sensor}(do(A, S))]=1 & \quad \text{if } A \text{ is a non-sensing action} \\
M^*_{\mathcal{A}}[\text{sensor}(do(A, S))]=1 & \quad \text{if } A \text{ is a sensing action} \\
M^*_{\mathcal{A}}[\text{sensor}(do(A, S))]=0 & \quad \text{if } A \text{ is a sensing action} \\
M^*_{\mathcal{A}}[F_A(do(A, S))]=\top & \quad \text{and } M^*_{\mathcal{A}}[F_A(do(A, S))]=\bot
\end{align*}$$

Clearly, $M^*_{\mathcal{A}} \models \forall s.\text{sensing}(a', do(a, s)) \equiv a=a'$ and $M^*_{\mathcal{A}} \models \forall a. \neg \text{sensing}(a, S_0)$. This, together with the fact that $M^*_{\mathcal{A}} \models \forall s.F_A(do(A, s)) \equiv (\text{sensor}(do(A, s))=1)$ for each sensing action $A$, implies that each GSFA is, in fact, satisfied by the model. or formally, $M^*_{\mathcal{A}} \models \forall s.\text{sensing}(A, s) \supset F_A(s) \equiv (\text{sensor}(s)=1)$.

It remains to show that $M^*_{\mathcal{A}} \models \text{Sensed}[\sigma]$. Again, $\text{Sensed}[\sigma]$ is a conjunction of
atoms of the form $sensor(S) = V$, where $S$ is a situation term and $V$ is a value. In this case either 1 or 0. Take any of such atoms $\gamma$:

1. If $\gamma = [sensor(do(A.S)) = 1]$ and $A$ is a sensing action, then $\sigma' \cdot \langle A, < 1 > \rangle$ is a sub-history of $\sigma'$. where $end[\sigma'_1] = S$. Therefore, $\sigma_1 \cdot \langle A, 1 \rangle$ is a sub-history of $\sigma$, where $end[\sigma'_1] = S$. Thus, $SF(A.S)$ is one of the terms of the conjunction $Sensed[\sigma]$, and $M_{SA}[SF(A.S)] = \top$. Since $A$ is a sensing action, $M_{SA} \models \forall s.SF(A.s) \equiv F_A(s)$ and $M_{SA}[F_A(S)] = M_{SA}[F_A(do(A.S))] = \top$ again because of the initial assumption and the fact that $M_{SA} \models D_{SA}$. Hence, $M^{*}_{SA}[F_A(do(A.S))] = \top$ as well and by the above extension $M^{*}_{SA}[sensor(do(A.S))] = 1$. If $A$ is not a sensing action, then $M^{*}_{SA}[sensor(do(A.S))] = \top$ by the construction of $M^{*}_{SA}$.

2. If $\gamma = [sensor(do(A.S)) = 0]$ and $A$ is a sensing action, then $\sigma' \cdot \langle A, < 0 > \rangle$ is a sub-history of $\sigma'$, where $end[\sigma'_1] = S$. Therefore, $\sigma_1 \cdot \langle A, 0 \rangle$ is a sub-history of $\sigma$, where $end[\sigma'_1] = S$. Thus, $\neg SF(A.S)$ is one of the terms of the conjunction $Sensed[\sigma]$ and $M_{SA}[SF(A.S)] = \bot$. Since $A$ is actually a sensing action, $M_{SA} \models \forall s.SF(A.s) \equiv F_A(s)$ and $M_{SA}[F_A(S)] = M_{SA}[F_A(do(A.S))] = \bot$ again due to $M_{SA} \models D_{SA}$ and the theorem assumption. Hence, $M^{*}_{SA}[F_A(do(A.S))] = \bot$ as well and - by the above extension - $M^{*}_{SA}[sensor(do(A.S))] = 0$. Note that $A$ cannot be a non-sensing action since non-sensing actions are assumed to have 1 as outcome.

To end, we note that because $M_{SA} \models \phi(end[\sigma'])$, $\phi$ mentions neither fluent sensing nor the sensing function $sensor$, and $end_A[\sigma] = end[\sigma']$. $M_{SA} \models \phi(end_A[\sigma])$ must hold. Due to the fact that the above holds for any model $M_{SA}$ of $D_{SA} \cup Sensed[\sigma]$, it follows that $D_{SA} \cup Sensed[\sigma] \models \phi(end_A[\sigma])$. 

In what follows, we state the most important differences between the two approaches to sensing:

(a) In general, we cannot reduce sensing action to sensors. While sensing actions can encapsulate the truth value of complex formulas, sensors can only contribute to
the truth value of one or more fluents. For instance, take the sensing axiom
\( SF(A,s) \equiv \exists n. temperature(n,s) \land n > 10 \). Action \( A \) does not get the value of the fluent \( temperature(s) \), but rather says "something" about it. How do we express such things within a sensor framework? We cannot unless we either introduce ramification constraints or change the ontology of our theory.

(b) Related to the previous point, while sensing of arbitrary formulas seems to be very difficult to implement, sensing of fluents are not. Therefore, in a practical point of view, sensing actions do not seem to provide extra expressiveness.

(c) A sensor framework seems to be a cleaner way of formalizing the sensing capabilities of an agent: once sensor sources have been correctly modeled in the theory, we do not have to worry about them anymore. In contrast, sensing actions have to be explicitly stated inside the high-level programs.

(d) While sensing actions gather knowledge about the "previous" situation, sensors gather knowledge about the "actual" situation. This difference disappears in most of the cases when we assume some restrictions on the sensing action as Theorem 3.5.1 does. However, in more general cases we may find some tricky aspects: For instance, assume an action that can both sense and affect what is being sensed: toggling a switch would not only cause the switch to change its state, but also would cause us to know the state of the switch (since we have to touch it). In such cases, sensing actions as defined above assume that the sensing is performed before the effects of the action take place. Therefore, toggling a switch that is off would make the switch to go on while returning that the switch was off. However, with the sensor approach the sensing is assumed to be performed after the effects of the last action take place.

Another case where sensing actions and sensors may differ arises when some not modeled action may be executed between the starting time and the ending time of
the action. Clearly, this cannot happen in basic action theories but it is the main motivation for guarded ones. In such cases, a sensing action may tell us something about the previous situation but nothing about the actual one. Similarly, sensors may tell us something about the current situation but nothing about the previous one. In many cases, it seems reasonable to think of sensing as providing information about how the world is now and not how it used to be.

(e) Related to implementation issues, it seems that using sensing actions is better when the sensing source is barely used, or when the sensing source changes very often (a timer). On the other hand, if the sensing source is frequently consulted or it hardly changes (temperature), we may prefer a sensor approach. The reason for this has to do with the space required to store the information gathered. With sensing actions, it is sufficient to store the sensing outcome each time a sensing action is performed. With sensors, however, it is necessary to keep track of the outcomes of each sensor in every past situation. This can be done by an efficient database storing only the changes of each sensor. In addition, when the initial state is progressed ([15]) after some time or number of actions the sensing database can be “progressed” as well.

We think that a practical implementation should offer the user both approaches to sensing, so that she decides whether a sensing source should be described with an on-board sensor or with an explicit action.

### 3.5.1 Sensors and Knowledge

As spelled out in Chapter 2 (section 2.1.1), a clean account of knowledge with sensing is suggested by Shetr and Levesque for the situation calculus. The work is achieved by extending Reiter’s solution to the frame problem [35] for ordinary actions and Moore’s work on knowledge and action [31]. Knowledge is expressed using a new (epistemic) fluent $K(s', s)$ - an accessible relation over situations - which is intended to mean that
from a situation $s$, as far as the agent knows in situation $s$, she might be in situation $s'$.

Our problem is that we do not have sensing actions anymore, but sensors instead. If we can come up with something like the above successor state axioms for $K$, but with sensors, then we still keep the clean solution to knowledge of Scherl and Levesque for situation calculus.

The main difference is that we cannot state the truth value of $K$ only in terms of the previous situation, since knowledge production is related to the actual situation (recall point (d) above). However, a correct axiom for $K$ can still be stated.

The intuitive idea is that, instead of explicitly performing sensing actions, the sensing is automatically performed whenever a GSFA's guard is satisfied. Remember that sensing is reduced to fluents and assume that there are $n \geq 0$ GSFAs of the form

$$J_i(\overline{x}, s) \supset F_i(\overline{x}, s) \equiv \rho(\overline{x}, s)$$

where $i = 1..n$. Then, the epistemic fluent $K$ can be characterized as follows:

$$Poss(a, s) \supset K(s'', do(a, s)) \equiv \exists s', [(K(s', s) \land s'' = do(a, s')) \land$$

$$((J_1(do(a, s)) \supset F_1(do(a, s)) \equiv F_1(s'')) \land$$

$$\vdots \) \land$$

$$(J_n(do(a, s)) \supset F_n(do(a, s)) \equiv F_n(s''))]$$

In moving from $s$ to $do(A, s)$, the agent not only knows that the action $A$ has been performed, but also the truth value of every predicate $F_i$ such that the guard $J_i$ is true in the situation $do(A, s)$. Observe that formulas $\rho_i(\overline{x}, s)$ are not mentioned in the axiom since they only serve to provide the truth value of the fluent being sensed. In other words, $\rho(\overline{x}, s)$ is analogous to the predicate $SF(\overline{x}, s)$ used with sensing actions to denote the actual sensing outcome. What we want to express with the $K$ relation is that, under the assumption that sensors are correctly modeled in the theory, whenever a GSFA is applicable, the truth value of the corresponding fluent will be known.
It is important to remark that the above $K$ characterization does not qualify as a guarded successor state axiom because the right hand side is not uniform on $s$. Sherl and Levesque's solution does qualify as a successor state axiom since sensing "talks" about the previous situation, which, because of the successor state axiom and the fact that the sensing action does not affect the world, persists equally after the action in question.

In the light of the above $K$ characterization, we conclude by remarking that we should be able to restate the properties of knowledge and the regression operator described in [38] for GAT. However, all this remains to be proved.

### 3.6 Adding Functional Fluents

Up to now we have restricted our attention to relational fluents only. However, in any realistic application many attributes of states are naturally functional. For instance, $\text{recipient}(p, s)$, $\text{robotState}(s)$, and $\text{robotDestination}(s)$ are typical functional fluents in a mail delivery domain [23].

We extend guarded action theories to accommodate functional fluents as follows:

A guarded successor state axiom (GSSA) for a functional fluent $F$ is an axiom of the form:

$$\alpha(\bar{F}.a,s) \supset [F(\bar{F}.do(a,s)) = v \equiv v'(\bar{F}.a.v,s)]$$

A guarded sensed fluent axiom (GSFA) for a functional fluent $F$ is an axiom of the form:

$$\beta(\bar{F}.s) \supset [F(\bar{F}.s) = v \equiv \rho(\bar{F}.v,s)]$$

where $\beta$ is a sensor-fluent formula, $\alpha$ and $v'$ are fluent formulas, and $\rho(\bar{F}.s)$ is a sensor formula.

As explained in [34], under Reiter's solution to the frame problem, $v'(\bar{F}.a.v.s)$ should satisfy a consistency property. Moreover, the concepts of coherent theories and histories
in section 3.4 should be extended for functional fluents as well, although we are not going to explain that here.

3.7 Chapter Summary

In this chapter, we studied an interesting improvement of basic theories of actions where a solution to the frame problem depends on the satisfaction of particular conditions, and where sensing information is modeled via online onboard sensors that may or may not be applicable in a situation.

We defined the notion of projection task for these theories and described how they can be solved via regression. After that, we stated the subset of these theories that are well-behaved, namely coherent theories, and we proved that such theories only need one GSSA per fluent.

Next, we talked about the differences between previous approaches to sensing information with sensing action and onboard sensors. In particular, we showed that for the most practical cases where sensing is limited to the sensing of fluents, the sensing method via sensing actions can be reduced to the sensors framework. Finally, we showed how Sherg and Levesque approach to knowledge in the situation calculus can be adapted when using sensors instead of sensing actions.
Chapter 4

Execution of High-Level Programs

Chapter 3 outlined the situation calculus-based approach for representing, and reasoning about, simple actions that we would use for modeling the world and its dynamics. However, as [28] and [24] suggested, our final aim is to have a high-level programming language for agents. In this chapter, we discuss a practical and convenient way of executing high-level programs in the presence of both sensing data and exogenous events.

4.1 Motivations

Two problems arise when high-level programs are executed as Golog [25] and ConGolog [9] do:

1. As argued in [21], why should we care about far away actions in order to execute the next action? It makes more sense to incrementally generate sub-plans with a certain horizon, which is extended once we are in it or near it. In addition, calculating a complete plan is not worth the effort when, generally, we will need to monitor its real execution for unexpected circumstances [12].

2. Off-line execution of programs are not suitable when there is online sensing data and where exogenous external events may occur.
To illustrate these two points, consider an agent delivering mail between people in a building. At a certain point, the agent realizes he must serve a particular customer. For that, he has to go to his office, deliver all packages that are for him, pickup all packages this customer has, and finally, deliver each of them to the corresponding recipients. Generally, it makes no sense to think about the specific actions he will have to perform once he is at the customer’s place. Instead, it would be faster and more effective to just start going to the customer’s office and, once there, think about what to do next. What makes things even more complicated is that, maybe, the agent does not know what he will need to do once in the office! That may depend on some future information - due to sensing - he does not have now. such us the number of packages to be picked up and the destination of each one.

Typically, Golog and ConGolog programs are intended to execute offline, and then the sequence of actions returned by this offline computation is executed online. To deal with examples like the above one, we need some other way of calculating legal executions where, basically, we are not so much interested in the whole plan but only in the next action to be performed.

### 4.2 Online Execution with Local Lookahead

Choosing a single next transition step among many legal ones, and execute it in the real world is defined as an online execution of a program. The brave online interpreter [12] does exactly that in order to couple with an execution monitor: select a first action from the program, commit to it (do it in the physical world), and finally repeat with the rest of the program.

Hence, the problem reduces to deciding how to select among the possible transitions the one to execute so that we are both efficient and secure. An online method with a random choice is very efficient but the choice made may lead to a dead-end, even if the
program has a terminating situation. This happens because when we execute a program online there may be no possibility of backtracking in the physical world. On the other hand, although an offline execution is totally impractical, it is completely secure. What is more, offline execution requires us to reason by cases when sensing is considered, which makes the problem even more intractable.

As [10] suggested, we choose a happy medium between these two approaches: programs are executed online but there is a mechanism to have a controlled form of lookahead. Even though the majority of the program is executed online, the programmer can explicitly point out where offline execution needs to be done in order to guarantee a successful execution. Since the scope of the offline execution is limited, although it can be the whole program, we obtain a trade-off between caution and practicality.

To provide a controlled form of lookahead, we make use of a new language construct ([10]), called the search operator ($\Sigma$). As with all the language constructs, we need to give the computational semantics it. In other words, we have to augment the sets of axioms $C_F$ and $C_T$ defined in Chapter 2 for ConGolog. For Final, we simply have that $(\Sigma \delta, s)$ is a final configuration if $(\delta, s)$ itself is. We therefore add to $C_F$ the following axiom:

$$Final(\Sigma \delta, s) \equiv Final(\delta, s)$$

Also, configuration $(\Sigma, \delta)$ can evolve to $(\Sigma \gamma', s')$ provided that $(\delta, s)$ can evolve to $(\gamma', s')$ and from $(\gamma', s')$ it is possible to reach a final configuration in a finite number of transitions:

$$Trans(\Sigma \delta, s, \delta', s') \equiv \exists \gamma, \gamma', s'', \delta' = \Sigma \gamma \land Trans(\delta, s, \gamma, s') \land Trans^*(\gamma', s', \gamma', s'') \land Final(\gamma', s'')$$

where $Trans^*$ is the reflexive transitive closure second-order definition of $Trans$ (Chapter 2 - section 2.2.2). From the definition, we see that $\Sigma \delta$ selects from all possible transitions of $(\delta, s)$ those for which there exists a sequence of further transitions leading to a final configuration (and hence a successful completion). Moreover, the $\Sigma$ operator is
propagated in the remaining program so that this restriction is enforced throughout the execution of $\delta$.

**Theorem 4.2.1:** The following property is a logical consequence of the above $\text{Trans}$ definition for $\Sigma$ (equation 4.1):

$$\text{Trans}(\Sigma \delta.s.\Sigma \delta'.s') \equiv \text{Trans}(\delta.s.\delta'.s') \land [\text{Final}(\delta'.s') \lor \delta''.\exists s''.\text{Trans}(\Sigma \delta'.s'.\delta''.s'')]$$

**Proof.**

$\Rightarrow$ Assume $\text{Trans}(\Sigma \delta.s.\Sigma \delta'.s')$ holds. Then, $\exists \gamma.\Sigma \delta' = \Sigma \gamma \land \text{Trans}(\delta.s.\gamma.s')$ by $\text{Trans}$ definition for $\Sigma$. Simply, take $\gamma = \delta'$ and we obtain $\text{Trans}(\delta.s.\delta'.s')$. Also, because of $\text{Trans}$ definition for $\Sigma \delta$. $\exists s''.\gamma'.\text{Trans}^*(\delta'.s'.\gamma'.s'') \land \text{Final}(\gamma'.s'')$ should be satisfiable by any model $M$ satisfying $\text{Trans}(\Sigma \delta.\Sigma \delta'.s')$. Let us prove now that $M$ satisfies the second conjunction as well. Assume $\Sigma \delta$ executes $n \geq 1$ transitions before terminating successfully wrt the model $M$:

- If $n = 1$ then it must be the case that $(\delta'.s')$ is final. Therefore $\gamma' = \delta'$ and $s'' = s'$.
  Hence, $\text{Final}(\delta'.s')$ holds in $M$.

- If $n > 1$ then $\exists s''.\gamma'.\delta_1.s_1.\text{Trans}(\delta'.s'.\delta_1.s_1) \land \text{Trans}^*(\delta_1.s_1.\gamma'.s'') \land \text{Final}(\gamma'.s'')$ holds in $M$. By $\text{Trans}$ definition for $\Sigma$. $\exists \delta_1.s_1.\text{Trans}(\Sigma \delta'.s'.\Sigma \delta_1.s_1)$ is true in $M$.

Therefore. $M$ satisfies the second conjunction as well and since this applies for any model $M$ such that $M \models \text{Trans}(\Sigma \delta.s.\Sigma \delta'.s')$ we are done.

$\Leftarrow$ Assume $\text{Trans}(\delta.s.\delta'.s') \land [\text{Final}(\delta'.s') \lor \delta''.\exists s''.\text{Trans}(\delta'.s'.\delta''.s'')]$ holds. It is trivial to see that $\exists \gamma.\Sigma \delta' = \Sigma \gamma \land \text{Trans}(\delta.s.\gamma.s')$ by making $\gamma = \delta'$.

It remains to prove that $\exists \gamma'.s''.\text{Trans}^*(\delta'.s'.\gamma'.s'') \land \text{Final}(\delta'.s'')$. Then, given a model $M$ of the RHS at least one of the following must apply:

- If $\text{Final}(\delta'.s')$ holds. then we get $\text{Trans}^*(\delta'.s'.\delta'.s') \land \text{Final}(\delta'.s')$ by taking $\gamma' = \delta'$ and $s'' = s'$. Putting this together with the first result. $\text{Trans}(\Sigma \delta.s.\Sigma \delta'.s')$ applies in $M$. 

If \( \exists \delta'', s'' . Trans(\delta'', s' . \delta'', s'') \) holds, then by equation 4.1 we get

\[
\exists \delta'', s'' \exists \gamma_1, \gamma_1', s_1'', \delta'' = \Sigma \gamma_1 \land Trans(\delta', s' . \gamma_1, s'') \land Trans^*(\gamma_1, s'', \gamma_1', s_1') \land Final(\gamma_1, s_1')
\]

It follows from the fact that \( Trans^* \) is the reflexive transitive closure of \( Trans \) that

\[
\exists \gamma_1, s'' . Trans^*(\delta', s' . \gamma_1, s'') \land Final(\gamma_1, s_1'') \text{ holds in } M. \text{ By just taking } \gamma' = \gamma_1' \text{ and } s'' = s_1'' \text{ together with the first result above we conclude } M \models Trans(\Sigma \delta . s, \Sigma \delta'. s').
\]

With the local offline search included as a construct of our programming language we now define what we mean by a program execution. From now on, by \textit{Axioms} we mean the background theory \( \mathcal{D} \) plus \( \mathcal{C} \).

\textbf{Definition 4.2.1}: An \textit{online execution} of a program \( \delta_0 \) starting from a history \( \sigma_0 \) is a sequence \( (\delta_0, \sigma_0), \ldots, (\delta_n, \sigma_n) \), such that for \( i = 0, \ldots, n - 1 \)

\[
\text{Axioms} \cup \text{Sensed}[\sigma_i] \models Trans(\delta_i . \text{end}[\sigma_i], \delta_{i+1} . \text{end}[\sigma_{i+1}])
\]

\[
\sigma_{i+1} = \begin{cases} 
\sigma_i, & \text{if } \text{end}[\sigma_{i+1}] = \text{end}[\sigma_i] \\
\sigma_i \cdot (a, \bar{\mu}), & \text{if } \text{end}[\sigma_{i+1}] = \text{do}(a . \text{end}[\sigma_i])
\end{cases}
\]

and \( \bar{\mu} \) is the sensor results after \( a \)

Furthermore, the online execution terminates successfully if:

\[
\text{Axioms} \cup \text{Sensed}[\sigma_n] \models Final(\delta_n . \text{end}[\sigma_n])
\]

A similar definition can be obtained when sensing is performed via sensing actions ([10]).
It is easy to see that an online execution can also be defined by the following equation

\[
\text{Axioms} \models Sensed[\sigma_0] \supset Trans(\delta_0, \text{end}[\sigma_0], \delta_1, \text{end}[\sigma_1]) \\
Sensed[\sigma_1] \supset Trans(\delta_1, \text{end}[\sigma_1], \delta_2, \text{end}[\sigma_2]) \\
\vdots \\
Sensed[\sigma_{n-1}] \supset Trans(\delta_{n-1}, \text{end}[\sigma_{n-1}], \delta_n, \text{end}[\sigma_n]) \\
Sensed[\sigma_n] \supset \text{Final}(\delta_n, \text{end}[\sigma_n])
\]

(4.2)

It can be shown that, in the absence of sensing (i.e. \( Sensed[\sigma] = TRUE \)), whenever there is an online execution, there is also an offline\(^1\) one ([10]). However, there may be an offline execution, but no online one.

**Example 4.2.1:** Consider the simple program \( \delta = (\phi?: \phi_1?: a) \mid (\neg \phi?: a) \), such that \( \text{Axioms} \not\models \phi[S_0], \text{Axioms} \not\models \neg \phi[S_0] \), but \( \text{Axioms} \models \text{Poss}(a.S_0) \land \phi_1(S_0) \). Although \( \delta \) can be executed offline (where \( \text{do}(a.S_0) \) is seen to be a legal execution), there is no online execution of \( \delta \) in \( S_0 \): there is no single transition logically implied by \( \text{Axioms} \). Indeed, neither \( \text{Axioms} \models \text{Trans}(\delta, S_0, (\phi_1?: a).S_0) \) nor \( \text{Axioms} \models \text{Trans}(\delta, S_0, a.S_0) \) applies.

\[\Box\]

Intuitively, while the offline definition does not worry about the evolution of the program, the online one does. In the latter, to perform a legal step, we should exactly know both the remaining program and the next situation. This is not the case with an offline "step" where just the next situation needs to be known. In other words, we can guarantee an offline step of execution without having any remaining concrete program.

Nonetheless, we can distinguish a subset of offline executions for which there is a corresponding online one.

**Definition 4.2.2:** Let \( \delta \) be a program. Let \( s \) and \( s' \) be two situations \( s \). We say that \( \delta \)

\(^1\)Offline execution was defined in Chapter 2, Definition 2.2.1.
executes (offline) from $s$ to $s'$ without reasoning by cases (on the program) iff it is the case that $D \models Do(\delta, s, s')$ and

- $\delta = (\delta_1|\delta_2)$ and either $\delta_1$ executes from $s$ to $s'$ without reasoning by cases: or $\delta_1$ executes from $s$ to $s'$ without reasoning by cases:

- $\delta = (A; \delta_1)$ and $\delta_1$ executes from $do(A; s)$ to $s'$ without reasoning by cases:

- $\delta = \pi x.\delta_1$ and there is a constant $c$ such that $\delta_1|_c$ executes from $s$ to $s'$ without reasoning by cases:

- $\delta = (?; \delta_1)$ and $\delta_1$ executes from $s$ to $s'$ without reasoning by cases:

- $\delta = (if \ \circ \ \text{and} \ \delta_1 \ \text{else} \ \delta_2; \delta_1)$ then either $D \models \circ[s]$ and $(\delta_1; \delta_2)$ executes from $s$ to $s'$ without reasoning by cases: or $D \models \neg\circ[s]$ and $(\delta_2; \delta_3)$ executes from $s$ to $s'$ without reasoning by cases:

- $\delta = (while \ \circ \ \text{do} \ \delta_1; \delta_2)$ and either $D \models \neg\circ[s]$ and $\delta_2$ executes from $s$ to $s'$ without reasoning by cases: or $D \models \neg\circ[s]$ and $(\delta_1; \ \text{while} \ \circ \ \text{do} \ \delta_2; \delta_1)$ executes from $s$ to $s'$ without reasoning by cases:

- $\delta = \text{nil}$.

**Theorem 4.2.2:** If $\delta$ executes without reasoning by cases from a situation $s$ to a situation $s'$, then there is a successful online execution of $\delta$ from $s$ to $s'$.

**Proof.** By induction on the number of $Do$ expansions needed. Although an online execution is a sequence of pairs program-history, we are going to simplify the proof by using situations instead of histories. Given that the execution from $s$ to $s'$ should work no matter how sensing will turn out to be, any sensing outcomes should be good for the sequence of action.
**Base Case:** Only one Do expansion is needed. If \( \delta = \text{nil} \), then \( \mathcal{D} \models Do(\delta, s, s) \) and \( s' = s \). Hence, \((\delta, s)\) is a successful online execution given that \( \mathcal{D} \cup \mathcal{C} \models \text{Final}(\text{nil}, s) \) and we are done. The cases where \( \delta = A \) or \( \delta = \sigma ? \) can be viewed as \( \delta = (A: \text{nil}) \) and \( \delta = (\sigma?: \text{nil}) \) respectively.

**Induction:** Suppose that \( \delta \) needs \( n + 1 \) expansions of Do:

- If \( \delta = \delta_1 | \delta_2 \), then by Definition 4.2.2, either (i) \( \delta_1 \) executes without reasoning by cases from \( s \) to \( s' \) with \( n \) Do expansions: or (ii) \( \delta_2 \) executes without reasoning by cases from \( s \) to \( s' \) by \( n \) Do expansions. Assume (i) is the case. By induction, \((\delta_1, s), (\delta'_1, s_1), \ldots, (\delta'_n, s')\) is a successful online execution where

\[
\mathcal{D} \cup \mathcal{C} \models \text{Trans}(\delta_1, s, \delta'_1, s_1)
\]

By definition of \text{Trans} for non-deterministic choice

\[
\mathcal{D} \cup \mathcal{C} \models \text{Trans}(\delta_1, s, \delta'_1, s_1)
\]

and therefore, \((\delta, s), (\delta'_1, s_1), \ldots, (\delta'_n, s')\) is a successful online execution of \( \delta \) from \( s \). The case where (ii) holds is analogous.

- If \( \delta = (A; \delta_1) \), then by Definition 4.2.2, \( \delta_1 \) executes without reasoning by cases from \( do(A, s) \) to \( s' \), and \( \mathcal{D} \models \text{Poss}(A, s) \) is true. By induction, \((\delta_1, do(A, s)), \ldots, (\delta'_n, s')\) is a successful online execution. Due to the definition of \text{Trans} for primitive actions

\[
\mathcal{D} \cup \mathcal{C} \models \text{Trans}(\delta, s, \delta_1, do(A, s))
\]

Thus, \((\delta, s), (\delta_1, do(A, s)), \ldots, (\delta'_n, s')\) is a successful online execution of \( \delta \) from \( s \).

- If \( \delta = \pi x. \delta_1 \), then there is a constant \( c \) such that \( \delta_1|_c^x \) executes from \( do(A, s) \) to \( s' \) without reasoning by cases. By induction, there is a successful online execution \((\delta_1|_c^x, s), (\delta'_1, s_1), \ldots, (\delta'_n, s')\) where

\[
\mathcal{D} \cup \mathcal{C} \models \text{Trans}(\delta_1|_c^x, s, \delta'_1, s_1)
\]
By definition of $\text{Trans}$ for non-deterministic choice

$$\mathcal{D} \cup \mathcal{C} \models \text{Trans}(\delta, s, \delta_1, s_1)$$

given that $\exists y. \text{Trans}(\delta_1|_y, s, \delta_1', s_1)$. In fact, for $y = c$ it holds!

Thus, $(\delta, s), (\delta_1, s), ..., (\delta', s')$ is a successful online execution of $\delta$ from $s$.

- If $\delta = (\varphi?; \delta_1)$, then by Definition 4.2.2, $\delta_1$ executes without reasoning by cases from $\text{do}(\varphi, s)$ to $s'$ and $\mathcal{D} \models \varphi[s]$. By induction, there is a successful online execution $(\delta_1, s), ..., (\delta', s')$. By definition of $\text{Trans}$ for conditions

$$\mathcal{D} \cup \mathcal{C} \models \text{Trans}(\delta, s, \delta_1, s)$$

Thus, $(\delta, s), (\delta_1, s), ..., (\delta', s')$ is a successful online execution of $\delta$ from $s$.

- If $\delta = (\text{if } \varphi \text{ then } \delta_1 \text{ else } \delta_2; \delta_3)$, then by Definition 4.2.2, either (i) $\delta_1$ executes without reasoning by cases from $s$ to $s'$ with $n \text{ Do}$ expansions and $\mathcal{D} \models \varphi[s]$; or (ii) $\delta_2$ executes without reasoning by cases from $s$ to $s'$ with $n \text{ Do}$ expansions and $\mathcal{D} \models \neg \varphi[s]$. If (i) is the case, by induction, there is a successful online execution $(\delta_1, s), (\delta_1', s_1), ..., (\delta', s')$ where $\mathcal{D} \cup \mathcal{C} \models \text{Trans}(\delta_1, s, \delta_1', s_1)$. By definition of $\text{Trans}$ for (if $\varphi$ then ... else ...) $\mathcal{D} \cup \mathcal{C} \models \text{Trans}(\delta, s, \delta_1, s_1)$ applies and therefore $(\delta, s), (\delta_1', s_1), ..., (\delta', s')$ is a successful online execution of $\delta$ from $s$. The case where (ii) holds is analogous.

- If $\delta = (\text{while } \varphi \text{ do } \delta_1) : \delta_2$ then either (i) $\delta' = \delta_1 \text{: while } \varphi \text{ do } \delta_1) : \delta_2$ executes without reasoning by cases from $s$ to $s'$ with $n \text{ Do}$ expansions and $\mathcal{D} \models \varphi[s]$; or (ii) $\delta_2$ executes without reasoning by cases from $s$ to $s'$ with $n \text{ Do}$ expansions and $\mathcal{D} \models \neg \varphi[s]$. If (i) is the case, by induction, there is a successful online execution $(\delta', s), (\delta_1', s_1), ..., (\delta', s')$. By definition of $\text{Trans}$ for while $\varphi$ do $\delta_1$

$$\mathcal{D} \cup \mathcal{C} \models \text{Trans}(\delta, s, \delta_1, s_1)$$
and therefore, \((\delta, s), (\delta_1', s_1), \ldots, (\delta_n', s')\) is a successful online execution of \(\delta\) from \(s\).

If (ii) is the case, by induction, there is an online execution \((\delta_2, s), (\delta_1', s_1), \ldots, (\delta_n', s')\).

By definition of \(\text{Trans}\) for while and the fact that \(\downarrow \models \neg \sigma[s]\) we have that
\[\mathcal{D} \cup \mathcal{C} \models \text{Trans}(\delta, s, \delta_1', s_1)\]

and therefore, \((\delta, s), (\delta_1', s_1), \ldots, (\delta_n', s')\) is a successful online execution of \(\delta\) from \(s\).

\[
\text{Trans}(\Sigma \delta, s, \Sigma \delta', s') \equiv \text{Trans}(\delta, s, \delta', s') \land \exists s'' \cdot \text{Do}(\delta', s', s'') \tag{4.3}
\]

On the other hand, in the presence of sensing online execution plus search can end up being more powerful than just offline: in particular when sensing information is crucial to determine the plan to execute as it will be showed in section 4.4).

We finish this discussion by noting a relaxed notion of offline execution. In general, it is useless to know that \(\text{Axioms} \models \exists s', \text{Do}(\delta, s, s')\) without actually having any concrete situation \(s'\). For example, consider the non-deterministic program \(\delta = \omega; a \mid \neg \omega; b\) such that \(\text{Axioms} \not\models \omega[S_0]\), \(\text{Axioms} \not\models \neg \omega[S_0]\), and \(\text{Axioms} \models \text{Poss}(a, S_0) \land \text{Poss}(b, S_0)\).

Although \(\text{Axioms} \models \exists s, \text{Do}(\delta, S_0, s)\), there is no ground situation term \(S\) such that \(\text{Axioms} \models \text{Do}(\delta, S_0, S)\), which means that an agent would be stuck without knowing which action to perform next. However, if the offline execution is not used to get the next action that must be performed, then knowing that there is an offline execution without knowing any particular one can be useful. In fact, that is how search works and the reason of way it - combined with an online execution - ends up in a powerful schema that can be compared to the execution of conditional plans as later explained in section 4.4.

In this new framework of high-level program execution, it is necessary to understand the behavior of \(\Sigma\) as part of online execution. Three intuitively plausible properties were proved in [10]. The most important may be the following
In performing a transition step we are verifying that such a transition eventually leads to a successful termination. Using this property, [10] explains what search means in the context of online executions and in the presence of sensing: “when making a transition on program $\Sigma \delta$ we do not take into account how the sensing values will turn out to be (we will know these values only when we actually execute the actions in the transitions)”. In other words, no matter how things will turn out to be in the future, there will be a plan still working.

Although we are usually interested in finding successful executions, it is also interesting whether there is an unsuccessful one, namely a sequence of (bad) transitions that derives to a dead-end.

**Definition 4.2.3:** An unsuccessful online execution of a program $\delta$ starting in history $\sigma$ is a sequence of configurations $(\delta, \sigma), (\delta_1, \sigma_1), \ldots, (\delta_n, \sigma_n)$ such that

$$Axioms \cup Sensed[\sigma] \models Trans(\delta, end[\sigma], \delta_1, end[\sigma_1])$$

$$Axioms \cup Sensed[\sigma_1] \models Trans(\delta_1, end[\sigma_1], \delta_2, end[\sigma_2])$$

$$\vdots$$

$$Axioms \cup Sensed[\sigma_{n-1}] \models Trans(\delta_{n-1}, end[\sigma_{n-1}], \delta_n, end[\sigma_n])$$

$$Axioms \cup Sensed[\sigma_n] \not\models Final(\delta_n, end[\sigma_n])$$

but there is no program $\delta_{n+1}$ and situation $s$ such that

$$Axioms \cup Sensed[\sigma_n] \models Trans(\delta_n, end[\sigma_n], \delta_{n+1}, s)$$

Clearly it is easily to fall into an unsuccessful execution when not using search by making bad choices. What is more, even though search guarantees the existence of a successful execution, it does assures us that we will actually find it! This is the reason why even when using search we might fall into an unsuccessful online execution. We will come to this later in this chapter.
There are at least three questions to address: (i) how does search mix with concurrency?: (ii) how does online execution plus search mix with sensing and compares with other executions types?: and (iii) how does search mix with exogenous events? In the remaining of this chapter, we will talk about point (i) and (ii). Also, motivated by the desire of having a practical implementation, we suggest an approximation to sensing and search.

4.3 Search and Concurrency

Imagine a program of the form \( \delta_1 \parallel \Sigma \delta_2 \). When looking for a step in \( \Sigma \delta_2 \) we have to guarantee that the step performed would, eventually, lead to a successful execution of \( \delta_2 \). However, we do not take into account that \( \delta_2 \) is executing together with \( \delta_1 \), i.e. all other concurrent processes are totally ignored. Because of this, we would like to know how \( \Sigma \) can be used reasonably in high-level programs in the presence of concurrency. While mixing search with normal concurrency (\( \parallel \)) can be done smoothly, mixing search with prioritized (\( \parallel_1 \)) concurrency has to be carefully designed.

We first reconsider the definitions of the constructs for concurrency, already introduced in Chapter 2 (section 2.2.2). After that, we define what we mean by a good use of search under concurrency and we give some trivial results (mainly for normal concurrency). Later, we study the case for prioritized concurrency. In particular we state one "declarative" and two syntactic conditions assuring a good use of search. Finally, we talk about the special case of prioritized concurrency together with interrupts.

The definitions for the concurrency constructs were the following:

1. **Normal Concurrency** \( (\delta_1 \parallel \delta_2) \) denotes the concurrent execution of the actions in \( \delta_1 \).
and $\delta_2$:

$$\text{Final}(\delta_1||\delta_2, s) \equiv \text{Final}(\delta_1, s) \land \text{Final}(\delta_2, s)$$

$$\text{Trans}(\delta_1||\delta_2, s, \delta', s') \equiv \exists \gamma. \delta' = (\gamma||\delta_2) \land \text{Trans}(\delta_1, s, \gamma, s') \lor \exists \gamma. \delta' = (\delta_1||\gamma) \land \text{Trans}(\delta_2, s, \gamma, s')$$

2. Prioritized Concurrency: $(\delta_1||\delta_2)$ denotes the concurrent execution of the actions in $\delta_1$ and $\delta_2$ with $\delta_1$ having higher priority than $\delta_2$. This means that $\delta_2$ executes only when $\delta_1$ is either done or blocked:

$$\text{Final}(\delta_1||\delta_2, s) \equiv \text{Final}(\delta_1, s) \land \text{Final}(\delta_2, s)$$

$$\text{Trans}(\delta_1||\delta_2, s, \delta', s') \equiv \exists \gamma. \delta' = (\gamma||\delta_2) \land \text{Trans}(\delta_1, s, \gamma, s') \lor \exists \gamma. \delta' = (\delta_1||\gamma) \land \text{Trans}(\delta_2, s, \gamma, s')$$

4.3.1 Definitions and Trivial Results

What do we mean by a "good use" of search in the presence of concurrent processes? Intuitively, we say that a local offline lookahead is safe in program $\delta_1||\Sigma \delta_2 (\delta_1||\Sigma \delta_2)$ if whenever we can completely execute $\delta_2$ alone, we are able to completely execute $\delta_2$ when run concurrently with $\delta_1$ as well. Otherwise, there would be no point in performing the lookahead.

In the case of prioritized concurrency, we want to consider a weaker restriction since it may be the case that $\delta_1$ takes control of the execution without terminating. The fact that a program $\delta_1$ runs forever in situation $s$ can be expressed formally as follows:

$$\text{cycle}(\delta, s) \overset{df}{=} \neg \exists \delta', s'. \text{Trans}^*(\delta, s, \delta', s') \land \left( \text{Final}(\delta', s') \lor \neg \exists \delta'', s''. \text{Trans}(\delta', s', \delta'', s'') \right)$$

Definition 4.3.1: We say that search is safe in program $\delta_1||\Sigma \delta_2$ iff

$$\text{Axioms } s, \exists \delta', s'. \text{Trans}(\Sigma \delta_2, s, \delta', s') \supset$$

$$\exists \delta_1', \delta_2', s'. \text{Trans}^*(\delta_1||\Sigma \delta_2, s, \delta_1'||\delta_2', s') \land \text{Final}(\delta_2, s')$$
Similarly, we say that search is safe in program $\delta_1)\Sigma\delta_2$ iff

$$\text{Axioms} \models [-\exists s'_1. s'_1. \text{Trans}(\delta_1, s, \delta'_1, s'_1) \land \exists s'_2. s'_2. \text{Trans}(\Sigma\delta_2, s, \delta'_2, s'_2)] \Rightarrow \
\exists s'_1. s'_2. s'. \text{Trans}^*(\delta_1)\Sigma\delta_2, s, \delta'_1, s') \land \text{Final}(\delta'_2, s')$$

Finally, we say that search is semi-safe in program $\delta_1)\Sigma\delta_2$ iff

$$\text{Axioms} \models [-\exists s'_1. s'_1. \text{Trans}(\delta_1, s, \delta'_1, s'_1) \land \exists s'_2. s'_2. \text{Trans}(\Sigma\delta_2, s, \delta'_2, s'_2)] \Rightarrow \
\exists s'_1. s'_2. s'. \text{Trans}^*(\delta_1)\Sigma\delta_2, s, \delta'_1, s') \land [\text{Final}(\delta'_2, s') \lor \text{cycle}(\delta'_1, s')]$$

Similar definitions can be obtained for programs $\Sigma\delta_1||\delta_2$ and $\Sigma\delta_1)\delta_2$. It is straightforward to prove that whenever search is safe it is also semi-safe, but not the converse.

It is worth noting that our definitions only require the existence of at least one "good path" when both processes run concurrently. This means that if bad choices are taken on $\delta_1$ execution, we may still be unable to execute $\delta_2$ completely.

Now imagine the program $(a: \delta_1)\Sigma\delta_2$ such that action $a$ is always possible. Although search may be safe in such a program, it may be not safe in the remaining program after executing $a$. i.e. in $\delta_1)\Sigma\delta_2$. Because of this, we need a stronger requirement: the higher-priority process may execute partially before getting blocked. A particular exception - discussed in section 4.3.4 - arises with interrupts.

**Definition 4.3.2:** We say that search is totally safe (totally semi-safe) in program $\delta_1)\Sigma\delta_2$ iff for every program $\delta'_1$ such that $\exists s'_1. \text{Trans}^*(\delta_1, s, \delta'_1, s'_1)$ holds, search is safe (semi-safe) in program $\delta'_1)\Sigma\delta_2$.

We expect, then, search to be totally safe, or at least totally semi-safe in our programs.

**Lemma 4.3.1:** If $\text{Trans}^*(\delta, s, \delta', s') \land \text{Final}(\delta', s')$ holds, then $\text{Trans}^*(\Sigma\delta, s, \Sigma\delta', s')$ holds as well.

**Proof.** We prove this for every model $M$ of the assumption by induction on the number $n \geq 0$ of transition steps needed:
1. If $n = 0$, then $\delta' = \delta$ and $s' = s$ from which, by $Trans^*$ definition, $Trans^*(\Sigma\delta \cdot s \cdot \Sigma\delta \cdot s)$ follows.

2. If $n \geq 1$, then $\exists \delta^*. s^* \cdot Trans(\delta \cdot s \cdot \delta^*. s^*) \land Trans^*(\delta^*. s^*. \delta' \cdot s') \land Final(\delta' \cdot s')$ is true in $M$. Next, $M \models \exists \delta^*. s^* \cdot Trans(\Sigma\delta \cdot s \cdot \Sigma\delta^*. s^*) \land Trans^*(\Sigma\delta^*. s^*. \Sigma\delta' \cdot s')$ follows form $Trans$ definition for $\Sigma$ and induction. Hence, $Trans^*(\Sigma\delta \cdot s \cdot \Sigma\delta' \cdot s')$ holds in $M$.

Since this applies for every model $M$ such that $M \models Trans^*(\delta \cdot s \cdot \delta' \cdot s') \land Final(\delta' \cdot s')$ we are done.

**Theorem 4.3.1:** Search is safe in programs $\delta_1 || \Sigma\delta_2 \cdot \Sigma\delta_1 || \delta_2$ and $\Sigma\delta_1 || \delta_2$.

**Proof.** Assume that $\exists \delta' \cdot s' \cdot Trans(\Sigma\delta_2 \cdot s \cdot \delta' \cdot s')$. By equation 4.1.

$$\exists \delta' \cdot s' \cdot \exists \gamma \cdot s'' \cdot \delta' = \Sigma\gamma \land Trans(\delta_2 \cdot s \cdot \gamma \cdot s') \land Trans^*(\gamma \cdot s' \cdot \gamma' \cdot s'') \land Final(\gamma' \cdot s'')$$

We can reformulate this to $\exists \gamma' \cdot s'' \cdot Trans^*(\delta_2 \cdot s \cdot \gamma' \cdot s'') \land Final(\gamma' \cdot s'')$ Using Lemma 4.3.1 we conclude that $\exists \gamma' \cdot s'' \cdot Trans^*(\Sigma\delta_2 \cdot s \cdot \gamma' \cdot s'') \land Final(\gamma' \cdot s'')$. By the definition of $Trans$ for $||$ it is easy to see that for any $\delta_1$, $\exists \gamma' \cdot s'' \cdot Trans^*(\delta_1 || \Sigma\delta_2 \cdot s \cdot \delta_1 || \Sigma\gamma' \cdot s'') \land Final(\gamma' \cdot s'')$ holds. By renaming, introducing existential quantification and $Final$ definition for search

$$\exists \delta'_1 \cdot \delta'_2 \cdot s' \cdot Trans^*(\delta_1 || \Sigma\delta_2 \cdot s \cdot \delta'_1 || \delta'_2 \cdot s') \land Final(\delta'_2 \cdot s')$$

Proofs for $\Sigma\delta_1 || \delta_2$ and $\Sigma\delta_1 || \delta_2$ are very similar.

Informally put, it is secure to do an offline lookahead even when the program is running concurrently with equal or lower priority processes.

### 4.3.2 Safe use in Prioritized Concurrency

We will now center our attention on the only non-trivial use of search which arises in programs of the form $\delta_1 || \Sigma\delta_2$. 
Claim 4.3.1: Assume a program $\delta = \delta_1()|\Sigma|\delta_2$ and a situation $s$. Assume also that $\text{Axioms} \models \neg \exists \delta'.s'.\text{Trans}(\delta_1,s,\delta',s')$, i.e. program $\delta_1$ is blocked in $s$. Then, it can consistently be the case that

$$\text{Axioms} \models \exists \delta'.s'.\text{Trans}(\Sigma \delta_2,s,\delta',s') \land \neg \exists \delta_1', \delta_2',s'.\text{Trans}*(\delta,s,\delta_1')\delta_2',s') \land \left[\text{Final}(\delta_2',s') \lor \text{cycle}(\delta_1,s')\right]$$

In other words, search may be neither semi-safe nor safe in program $\delta$. 

Hence, it is not always "safe" to do a local offline search on $\delta_2$ by totally ignoring process $\delta_1$. In contrast with normal concurrency, a successful execution of $\delta_2$ does not mean a successful execution of $\delta_2$ when executed concurrently with a higher-priority process.

Example 4.3.1: Let $\delta = (\phi_1?:a)|\Sigma|\phi_2?:c$ be a program and assume the following: (i) $\phi_1$ is false in $S_0$: (ii) actions $a$, $b$ and $c$ are always possible: (iii) action $b$ makes $\phi_1$ and $\phi_2$ to be true; and (iv) action $a$ makes $\phi_2$ to be false. Given that $\text{Axioms} \models \neg \phi_1[S_0]$, it follows that $\text{Axioms} \models \neg \exists \delta'.s'.\text{Trans}((\phi_1?:a),S_0,\delta',s')$ holds, i.e. process $\delta_1$ is blocked. Also, there is a successful execution for program $(b:\phi_2?:c)$ by first executing $b$ and then $c$. However - and here comes the problem - this offline search does not take into account that as soon as action $b$ is executed, process $\delta_1$ will not be blocked anymore, and since it has more priority, it must execute. Finally, the remaining program $(a)|(\phi_2?:c)$ will execute action $a$ and get stuck!: after $a$, condition $\phi_2$ will not hold anymore as assumed when the local search was done. Notice that $(\phi_1?:a)$ can never run forever. 

Due to this kind of example, we want to investigate how the programmer should avoid these situations by "safely" mixing concurrency and offline lookahead.

We start by presenting a declarative condition, which even though it may be difficult to check, it reflects many realistic cases. The idea is to know that $\delta_2$ cannot unblock $\delta_1$. 
Theorem 4.3.2: Let $\delta_1$ and $\delta_2$ be two programs. Assume that the background theory entails the following condition:

$$(\neg \exists \delta'_1, s'_1. \text{Trans}(\delta_1, s, \delta'_1, s'_1)) \land \text{Trans}^*(\delta_2, s, \delta'_2, s'_2) \supset \neg \exists \delta''. s''. \text{Trans}(\delta_1, s'_2, \delta'', s'')$$

Then, search is safe in program $\delta_1)\Sigma\delta_2$.

**Proof.** Assume that $\exists \delta'. s'. \text{Trans}(\Sigma\delta_2, s, \delta', s')$. By equation 4.1.

$$\exists \delta', s'. \exists \gamma', s''. \delta' = \Sigma \gamma \land \text{Trans}(\delta_2, s, \gamma, s') \land \text{Trans}^*(\gamma, s', \gamma', s'') \land \text{Final}(\gamma', s'')$$

We can reformulate this to $\exists \gamma', s''. \text{Trans}^*(\delta_2, s, \gamma', s'') \land \text{Final}(\gamma', s'')$. Using Lemma 4.3.1. $\exists \gamma', s''. \text{Trans}^*(\Sigma\delta_2, s, \gamma', s'') \land \text{Final}(\gamma', s'')$ applies (+). Given that it always holds that $\forall \delta, s, \delta', s'. \text{Trans}^*(\Sigma\delta, s, \Sigma \delta', s') \supset \text{Trans}^*(\delta, s, \delta', s')$ and the theorem assumption:

$$(\neg \exists \delta'_1, s'_1. \text{Trans}(\delta_1, s, \delta'_1, s'_1)) \land \text{Trans}^*(\Sigma\delta_2, s, \Sigma \delta'_2, s'_2) \supset \neg \exists \delta''. s''. \text{Trans}(\delta_1, s'_2, \delta'', s'')$$

In words, any partial execution of $\Sigma\delta_2$ from $s$ will not unblock the blocked process $\delta_1$. Then, by the definition of $\text{Trans}$ for prioritized concurrency $\Sigma\delta_2$ any partial execution of $\Sigma\delta_2$ would still be legal in program $\delta_1)\Sigma\delta_2$ since at each step there is no transition for $\delta_1$ and there is one for $\Sigma\delta_2$ (the same one).

Formally, $\forall \delta'_2, s'. \text{Trans}^*(\Sigma\delta_2, s, \Sigma \delta'_2, s') \supset \text{Trans}^*(\delta_1)\Sigma\delta_2, s, \delta_1))\Sigma \delta'_2, s'$) applies and by using it in (+): $\exists \gamma', s''. \text{Trans}^*(\delta_1)\Sigma \delta_2, s, \delta_1))\Sigma \gamma', s'') \land \text{Final}(\gamma', s'')$. Finally, by renaming, introducing existential quantification and $\text{Final}$ definition for $\Sigma$, we conclude that search is safe in $\delta_1)\Sigma\delta_2$ since $\exists \delta'_1, \delta'_2, s'. \text{Trans}^*(\delta_1)\Sigma \delta_2, s, \delta'_1))\delta'_2, s'') \land \text{Final}(\delta'_2, s')$ should hold. 

The theorem assumption is saying that whenever $\delta_1$ is blocked, i.e. there is no legal transition for $\delta_1$, it is the case that after executing any number of legal actions of $\delta_2$, program $\delta_1$ will still be blocked. Therefore, in such cases it is safe to perform the search in $\delta_2$: if a successful execution of $\delta_2$ exists, then there will also be a successful execution of $\delta_2$ when executed concurrently with $\delta_1$. Although the assumption may not be easy to
check, it is very realistic given that in many applications, such as the mail delivery robot [23], higher priority processes deal with uncommon situations caused only by exogenous actions and not by lower-priority processes.

As an example, suppose that $\delta_1$ is an interrupt triggered when the temperature is too high, and that $\delta_2$ is a program to serve a set of customers. Generally, serving a customer will not change the room’s temperature.

A problem arises when, in fact, $\delta_2$ can “activate” process $\delta_1$. For instance, suppose that a customer can order the robot to rise the temperature of the room. Then, how do we know that a successful offline search in $\delta_2$ will still be valid when considering also process $\delta_1$? Observe that when performing the search, any action potentially executed by $\delta_1$ is treated as an exogenous one, and therefore, not considered. However, this is really not the case given that the programmer knows the program! Which conditions should tell us that the mix between concurrency and search would have a correct behavior if $\delta_2$ might unblock $\delta_1$ seems not trivial. The first of the two sufficient conditions we present below applies for such cases.

### 4.3.3 Syntactic Conditions

We now give two syntactic conditions that may be applicable for some cases. The advantage is that they both rely on the syntactic form of the programs: the disadvantage is that they only apply for basic action theories and not for guarded ones. Nonetheless, it seems possible to extend them to fit the latter as well.

**Definition 4.3.3:** Given an action $A$, a program $\delta_1$, we define the following sets of fluents with respect to a basic action theory:

- $\text{Effects}(A) = \{ F : A \text{ is mentioned in the successor state axiom of fluent } F \}$.

- $\text{Effects}(\delta_1) = \bigcup_{i=1}^{n} \text{Effects}(A_i)$ where $A_1...A_n$ are all the primitive actions mentioned in program $\delta$: 
- \( \text{Fluents}(\delta) = \{ F : F \text{ is mentioned in } \delta \} \cup \{ F : F \text{ is mentioned in the precondition axiom of some action mentioned in } \delta \} \)

The sets \( \text{Effects}(A) \) and \( \text{Effects}(\delta) \) are the sets of fluents that can be potentially affected by the action \( A \) and the program \( \delta \) respectively. On the other hand, \( \text{Fluents}(\delta) \) is the set of all fluents that can be potentially "checked" when executing program \( \delta \).

Observe that if \( \delta_1 \) cannot affect any fluent in \( \delta_2 \) then it cannot preclude a successful execution of \( \delta_2 \), but it may run forever once it takes control.

**Theorem 4.3.3:** Assume a basic action theory and two programs \( \delta_1 \) and \( \delta_2 \). Search is **totally semi-safe** in program \( \delta_1 ) \Sigma \delta_2 \) whenever \( \text{Effects}(\delta_1) \cap \text{Fluents}(\delta_2) = \emptyset \).

**Proof.** We start by giving the intuitive reason behind: Suppose we do have a successful execution of \( \delta_2 \). Then start executing the whole program as if you were executing \( \delta_2 \) since \( \delta_1 \) starts blocked. If at some step, \( \delta_1 \) gets unblocked then execute \( \delta_1 \) as much as possible: (i) if \( \delta_1 \) never ends nor gets blocked then the search is semi-safe; (ii) if \( \delta_1 \) ends or gets blocked then resume the execution of \( \delta_2 \) which should be possible given that no fluent in \( \delta_2 \) was affected by the actions performed by \( \delta_1 \). Now let's put it in a formal argument.

Let \( M \) be a model of the background theory such that for some situation \( s \) it is the case that \( M \models \neg \exists \delta_1, s'_{11}.\text{Trans}(\delta_1, s, s'_{11}) \wedge \exists \delta_2, s'_{21}.\text{Trans}(\Sigma \delta_2, s, s'_{21}). \) Hence, for some program \( \delta_{M_2} \) and situation \( s'_{M_2} \):

\[
M \models \neg \exists \delta_1, s'_{11}.\text{Trans}(\delta_1, s, s'_{11}) \wedge \text{Trans}(\Sigma \delta_{M_2}, s, s'_{M_2}).
\]

We have to prove that for such model \( M \), the RHS of the semi-safe definition also holds.

We prove that by induction on the number \( n \geq 1 \) of transition steps required by \( \delta_2 \) to terminate successfully under the model \( M \):

\(^2\)We are assuming, without loss of generality, that each element of the domain in \( M \) can be named.
1. If \( n = 1 \), by Theorem 4.2.1, \( M \models Trans(\delta_2.s,\delta'_M.s'_M) \land Final(\delta'_M.s'_M) \) holds. From which it follows easily that \( M \models Trans(\Sigma \delta_2.s,\Sigma \delta'_M.s'_M) \land Final(\delta'_M.s'_M) \). Because of \( M \models \neg \exists \delta_1.s'.Trans(\delta_1.s,\delta'_1.s'_1) \) and the definition of \( Trans \) for \( ) \):

\[
M \models Trans(\delta_1) \Sigma \delta_2.s,\delta_1) \Sigma \delta'_M.s'_M) \land Final(\delta'_M.s'_M)
\]

Putting this in the same style as the RHS of the semi-safe definition.

\[
M \models \exists \delta'_1,\delta'_2.s'.Trans^*(\delta_1) \Sigma \delta_2.s,\delta'_1) \delta'_2.s'_1) \land Final(\delta'_2.s'_1)
\]

2. Assume that \( \delta_2 \) needs \( n + 1 \) steps in \( M \). The induction hypothesis says that it holds for any program needing less than \( n + 1 \) steps. Again, because of Theorem 4.2.1:

\[
M \models Trans(\delta_2.s,\delta'_M.s'_M) \land \exists \delta*.s*.Trans(\Sigma \delta'_M.s'_M,\delta*.s*)
\]

where \( \delta'_M \) now is supposed to need \( n \) transitions steps to finalize. After the first step of \( \delta_2 \), one of the following cases applies:

(a) \( \delta_1 \) cycles in \( s'_M \): Then \( M \models Trans^*(\delta_1) \delta_2.s,\delta_1) \delta'_M.s'_M) \land cycle(\delta_1.s'_M) \) holds and the RHS of the semi-safe definition applies in \( M \).

(b) \( \delta_1 \) executes \( m \geq 0 \) steps and gets blocked: This means that

\[
M \models \exists \delta*.s*.Trans^*(\delta_1.s'_M,\delta*.s*) \land \neg \exists \delta**.s**.Trans^*(\delta*.s*,\delta**.s**)
\]

The tricky part is that given \( M \models \exists \delta*.s*.Trans(\Sigma \delta'_M.s'_M,\delta*.s*) \) it also holds the following:

\[
M \models \exists \delta*.s*.Trans^*(\delta_1.s'_M,\delta*.s*) \land \neg \exists \delta**.s**.Trans^*(\delta*.s*,\delta**.s**) \land \exists \delta*.s*.Trans(\Sigma \delta'_M.s*.\delta*.s*)
\]

The reason is the following: all fluents mentioned in \( \delta'_1 \) (either in the program or in the precondition of some action there) have the same truth value in \( s^* \) than in \( s'_2 \) because all the actions performed between \( s'_2 \) and \( s^* \) do not change
any of those fluents because of $Effects(\delta_1) \cap Fluents(\delta_2') = \emptyset$. Observe that this argument is only valid for basic action theories and not for guarded ones. 

Next, we can apply the induction hypothesis over $\Sigma \delta'_{M2}, \delta^* \text{ and } s^*$ given that $Effects(\delta^*) \cap Fluents(\delta'_{M2}) = \emptyset$:

$$
M \models \exists \delta^*. s^*. Trans^*(\delta_1. s'_{M2}. \delta^*. s^*) \land \exists \delta^*. \delta'_{M2}. s'. Trans^*(\delta^*) \Sigma \delta'_{M2}. s'. \delta^*') \Sigma \delta'_{M2}. s' \land
$$

$$
[Final(\delta'_{M2}, s') \lor cycle(\delta^*. s')]
$$

Next, by the definition of $Trans$ for $\delta^*$

$$
M \models \exists \delta^*. s^*. Trans^*(\delta_1) \Sigma \delta'_{M2}. s'. \delta^*) \Sigma \delta'_{M2}. s^*) \land
$$

$$
\exists \delta^*. \delta'_{M2}. s'. Trans^*(\delta^*) \Sigma \delta'_{M2}. s^* \Sigma \delta^*') \Sigma \delta'_{M2}. s' \land
$$

$$
[Final(\delta''_{M2}, s') \lor cycle(\delta^*. s')]
$$

By property of the transitive relation $Trans^*$

$$
M \models \exists \delta^*. \delta''_{M2}. s'. Trans^*(\delta_1) \Sigma \delta'_{M2}. s'. \delta^*) \Sigma \delta''_{M2}. s' \land
$$

$$
[Final(\delta''_{M2}, s') \lor cycle(\delta^*. s')]
$$

Remember that $M \models \neg \exists \delta_1. s'. Trans(\delta_1, s, \delta'_1, s'_1) \land Trans(\Sigma \delta_2. s, \Sigma \delta'_{M2}. s_{M2})$

from which we know by using the $Trans$ definition for $\delta^*$ that

$$
M \models Trans(\delta_1) \Sigma \delta_2. s. \delta_1) \Sigma \delta'_{M2}. s_{M2})
$$

Joining the last two equations with the $Trans^*$ definition we obtain

$$
M \models \exists \delta^*. \delta''_{M2}. s'. Trans^*(\delta_1) \Sigma \delta_2. s. \delta^*) \Sigma \delta''_{M2}. s') \land
$$

$$
[Final(\delta''_{M2}, s') \lor cycle(\delta^*. s')]
$$

which means that the RHS of the semi-safe definition holds for the model $M$. 

Therefore, either if \( \delta_1 \) cycles in \( s'_M \) or makes some finite number of transitions (possibly zero), the following holds:

\[
M \models \exists \delta'_1, \delta'_2, s'. \text{Trans}^*(\delta_1) \cup \delta'_2, s'. \text{Trans}^*(\delta_2) \cup [\text{Final}(\delta'_3, s') \lor \text{cycle}(\delta'_1, s')]
\]

Finally, given that this holds for any model \( M \) of the background theory we conclude that search is semi-safe in \( \delta_1 \cup \delta_2 \). To prove that search is \textit{totally semi-safe} we only need to note that for any program \( \delta'_1 \) such that \( \exists s'_1, \text{Trans}(\delta_1, s, \delta'_1, s'_1) \), it is the case that \( \text{Effects}(\delta_1) \subseteq \text{Effects}(\delta'_1) \) and therefore \( \text{Effects}(\delta'_1) \cap \text{Fluents}(\delta_2) = \emptyset \).

The following theorem says that once \( \delta_1 \) gets blocked, \( \delta_2 \) cannot change any fluent mentioned in \( \delta_1 \) and, therefore, Theorem 4.3.2's assumption holds. In the next section, we will see a particular case where the theorem requirement can be relaxed.

**Theorem 4.3.4:** Assume a basic action theory and two programs \( \delta_1 \) and \( \delta_2 \). If \( \text{Fluents}(\delta_1) \cap \text{Effects}(\delta_2) = \emptyset \), then search is \textit{totally safe} in program \( \delta_1 \cup \delta_2 \).

**Proof.** We first prove that search is \textit{safe} in program \( \delta_1 \cup \delta_2 \) by showing that the condition \( \neg \exists \delta'_1, s'_1. \text{Trans}(\delta_1, s, \delta'_1, s'_1) \lor \text{Trans}^*(\delta_2, s, \delta'_2, s'_2) \lor \neg \exists \delta'' . s''. \text{Trans}(\delta_1, s', \delta'' . s'') \lor \neg \exists \delta'' . s''. \text{Trans}(\delta_2, s'', \delta'' . s'') \) of Theorem 4.3.2 applies. The argument is the following. Suppose that \( \delta_1 = \sigma \cup A \delta_1 \). Since there was no transition in situation \( s \), \( \text{Axioms} \models \neg \sigma[s] \) holds. Moreover, none of the actions performed by \( \delta_2 \) from \( s \) to \( s_2 \) can change any of the fluent mentioned in \( \sigma(s) \) because all fluents in \( \sigma(s) \) are not mentioned in the successor state axiom of any of such actions and fluents are changed only by the effects of axioms. Hence, \( \text{Axioms} \models \neg \sigma[s'_2] \) and \( \neg \exists \delta'' . s''. \text{Trans}(\sigma: \delta_1, s', \delta'' . s'') \) is true.

The argument is similar for all other programs except for primitive actions. Suppose then that \( \delta_1 = A; \delta_1 \) where \( A \) is primitive action. Since there was no transition in situation \( s \), \( \text{Axioms} \models \neg \text{Poss}(A.s) \) is true. Again, all actions performed by \( \delta_2 \) between \( s \) and \( s_2 \) cannot change any of the fluent mentioned in \( \text{Poss}(A.s) \) given that a fluent in \( \text{Poss}(A.s) \) is not mentioned in the successor state axiom of any of such actions. Hence, \( \text{Axioms} \models \neg \text{Poss}(A.s'_2) \) and \( \neg \exists \delta'' . s''. \text{Trans}(A; \delta_1, s', \delta'' . s'') \) applies again.
Thus, Theorem 4.3.2 applies and search is safe in $\delta_1 \cup S$. To prove that search is totally safe we only note that for any program $\delta_1'$ such that $\exists s'.\text{Trans}(\delta_1, s, \delta_1', s')$, it is the case that $\text{Fluents}(\delta_1) \subseteq \text{Fluents}(\delta_1')$, and so $\text{Fluents}(\delta_1') \cap \text{Effects}(\delta_2) = \emptyset$.

As we said, the advantage of these two conditions is that they can be checked syntactically by examining the programs, the precondition axioms, and the successor state axiom of fluents. The disadvantage is that they may be too conservative for many cases.

### 4.3.4 A Particular Case: Interrupts

Finally, we discuss a special case when prioritized concurrency is used in combination with high-level interrupts. As first introduced in [9] and explained in Chapter 2, $\langle \varnothing \rightarrow \delta \rangle$ means that the body of $\delta$ will execute when $\varnothing$ becomes true. The idea is that as long as the condition does not hold, the interrupt is blocked in the condition. Once the condition is satisfied, the interrupt is triggered and program $\delta$ starts executing. For the case where $\langle \varnothing \rightarrow \delta \rangle$ is blocked in its condition, we can come up with a similar, but more flexible, result than Theorem 4.3.4.

**Theorem 4.3.5:** Let $\delta_1 = \langle \varnothing \rightarrow \delta \rangle$ such that $\text{Axioms} \models \varnothing(s) \supset \exists \delta', s'.\text{Trans}(\delta, s, \delta', s')$. If $\delta_2$ is a program such that there in no fluent $F \in \text{Effects}(\delta_2)$ mentioned in formula $\varnothing(s)$, then search is safe in program $\delta_1 \cup S$.

**Proof.** Because none of the actions performed by $\delta_2$ can change any fluent mentioned in condition $\varnothing(s)$ it has to be that $\text{Axioms} \models \text{Trans}^*(\delta_2, s, \delta_2', s_2') \supset \varnothing(s) \equiv \varnothing(s_2')$. Next, given that whenever $\varnothing(s)$ is true, there must be at least one step in $\delta$, it must be the case that $\text{Axioms} \models \neg \exists \delta_1', s_1'.\text{Trans}(\delta_1, s, \delta_1', s_1') \supset \neg \varnothing(s)$.

Finally, given that $\text{Axioms} \models \neg \varnothing(s) \wedge \text{Trans}^*(\delta_2, s, \delta_2', s_2') \supset \neg \varnothing(s_2)$ and that it is also true that $\text{Axioms} \models \neg \varnothing(s) \wedge \neg \exists \delta_1', s_1'.\text{Trans}(\delta_1, s, \delta_1', s_1')$ we arrive to the conclusion $\text{Axioms} \models \neg \exists \delta_1', s_1'.\text{Trans}(\delta_1, s, \delta_1', s_1') \wedge \text{Trans}^*(\delta_2, s, \delta_2', s_2') \supset \neg \exists \delta_1', s_1'.\text{Trans}(\delta_1, \delta_2, \delta_1', \delta_2')$.
so that the hypothesis of Theorem 4.3.2 is satisfied. Hence, search is safe in program \(<\sigma \rightarrow \delta>\) and \(\delta_2\).

This result can be seen as a particular case of Theorem 4.3.4 where we only look at the interrupt condition \(\sigma\) instead of examining the entire program. This is a realistic condition because, usually, higher priority processes are only activated by exogenous actions, and not by lower priority processes. This is the case, for instance, of the reactive mail delivery system [23] where higher processes are used to react to temperature, fire and button's calling. Thus, if the lower priority process cannot activate the interrupt there is nothing else to worry about.

### 4.4 Sensing, Online Execution and Search

Sensing while planning offline is a complex issue because many sensing outcomes might be crucial to decide which plan to construct. However, these outcomes will be known in the future when sensing actions are actually executed or when readings are actually retrieved from the sensors. In this section, we restrict our attention to sensing via sensing actions.

Golog and ConGolog, when extended with sensing actions [26], overcome this problem by require their plans to work for *any* potential future outcome. In other words, the plan returned cannot depend on how sensing will turn out. Lakemeyer [22] remarked that “Golog’s applicability is severely limited because the derivation of a linear sequence of actions requires that the outcome of each action is known beforehand” and that “sensing actions do not meet this requirement”.

Take the following example due to [22]:

1. ...
proc catch_plane

  sense_gate:

  buy_paper:

  if gate ≤ 90 then goto_gate; buy_coffee else

  buy_coffee: goto_gate endif

endproc

Notice there is no ground situation term s' such that \( D \models Do(catch\_plane, S_0, s') \) whenever gate is unknown in \( S_0 \). Although there are two plausible candidates.\(^3\)

\[
\begin{align*}
\text{plan}_1 & = \{ \text{sense\_gate} \cdot \text{buy\_paper} \cdot \text{goto\_gate} \cdot \text{buy\_coffee} \} \\
\text{plan}_2 & = \{ \text{sense\_gate} \cdot \text{buy\_paper} \cdot \text{buy\_coffee} \cdot \text{goto\_gate} \}
\end{align*}
\]

we cannot decide between either of them, since that depends on the outcome of sense_gate.

Nevertheless, we know that as soon as the agent performs the action sense_gate, she will learn the value of fluent gate. She will be able to select the correct candidate, and she will finish the plan successfully.

To solve this problem, Lakemeyer suggested constructing conditional plans instead of linear plans. While the latter are expressed via plain situations terms, the former are expressed using new terms called conditional action trees (CATs). A branching in a CAT has the form \([\phi, c_1, c_2]\) where \(\phi\) is the condition to branch. \(c_1\) is the CAT that must be followed if \(\phi\) is true, and \(c_2\) is the CAT that must be followed if \(\phi\) is false. It follows, then, that \(c = \text{sense\_gate} \cdot \text{buy\_paper} \cdot [\text{gate} > 90, \text{goto\_gate} \cdot \text{buy\_coffee} \cdot \text{buy\_coffee} \cdot \text{goto\_gate}]\) is a correct conditional plan for the above example.

Another (practical) way of dealing with sensing inside offline execution is the one suggested by Golex, an extension of Golog designed to control the behavior of a museum

\(^3\)We abuse the notation here for better readability.
tour-guide robot [6]. In Golex, some Golog's actions are translated into pre-specified conditional plans, i.e., plans that are conditioned on the outcome of sensing actions. Such conditional plans must always succeed; however, the specific sequence of sub-actions may vary. The main difference with Golex is that the conditional plan is already given in advance by the programmer and it is known to work well no matter how sensing turn out to be. sGolog, in contrast, calculates the conditional plans, if any, from individual primitive actions, and therefore, it results in a more general account for sensing inside offline execution. Nonetheless, Golex's idea may be useful in many real applications where a single Golog action can encapsulate the behavior of a full conditional behavior.

In IndiGolog, the problem of sensing only arises when performed inside a search. Even though search does not suffer from Golog's above problem, it does not construct conditional plans. The reason has to do with the kind of offline execution implied by search together with the fact that it is only the next action to be performed that we are interested in.

To make this more clear, let us recall the search's property 4.3. We observe that a concrete next situation $s'$ and a concrete remaining program $\delta'$ are needed. Moreover, the property says that there must be a successful execution of program $\delta'$ from situation $s'$. However, as already discussed at the end of section 4.2, this last requirement is not more than a "relaxed" offline execution of $\delta'$ in $s'$. and, so, it is not required to actually have any concrete offline execution of it. In the above program, it is legal to execute the action $\text{sense.gate}$ because

$$\exists s'. \text{Do}([\text{buy.pap} \text{er.} \text{if gate} \leq 90 \text{ then ... else ...}]. \text{do}(\text{sense.gate}.S_0).s')$$

holds, even though we cannot state any term $s''$. This example shows that the online execution is only concerned with the next action to be performed by leaving potential future decisions to be made, actually, in the future and not now. Coming back to the example, suppose that the sensing outcome of the action $\text{sense.gate}$ is 101. Then, it is
legal to perform the action \textit{buy\_paper} and we have the following two transitions:

\[
\text{Axioms} \cup \text{Sensed}[\sigma_0] \models \text{Trans}(\Sigma\text{catch\_plane. } S_0, \Sigma[\text{buy\_paper: } \ldots]. \text{do}(\text{sense\_gate. } \text{end}[\sigma_0]))
\]
\[
\text{Axioms} \cup \text{Sensed}[\sigma_1] \models \text{Trans}(\Sigma[\text{buy\_paper: } \ldots]. \text{end}[\sigma_1]).
\]

\[
\Sigma \text{if gate} \leq 90 \text{ then } \ldots \text{do}(\text{buy\_paper. } \text{end}[\sigma_1]))
\]

where \(\sigma_0 = \epsilon, \sigma_1 = \epsilon \cdot (\text{sense\_gate. } 101)\). We see now that the following also would hold

\[
\text{Axioms} \cup \text{Sensed}[\sigma_2] \models \text{Trans}(\Sigma \text{if gate} \leq 90 \text{ then } \ldots \text{else } [\text{buy\_coffee. } \text{goto\_gate}. \text{end}[\sigma_2]).
\]

\[
\Sigma \text{goto\_gate. } \text{do}(\text{buy\_coffee. } \text{end}[\sigma_2]))
\]

\[
\text{Axioms} \cup \text{Sensed}[\sigma_3] \models \text{Trans}(\Sigma\text{goto\_gate. } \text{end}[\sigma_3]. \Sigma\text{nil. } \text{do}(\text{goto\_gate. } \text{end}[\sigma_3]))
\]

\[
\text{Axioms} \cup \text{Sensed}[\sigma_4] \models \text{Final}(\Sigma\text{nil. } \text{end}[\sigma_4])
\]

where \(\sigma_2 = \epsilon \cdot (\text{sense\_gate. } 101) \cdot (\text{buy\_paper. } 1), \sigma_3 = \epsilon \cdot (\text{sense\_gate. } 101) \cdot (\text{buy\_paper. } 1) \cdot (\text{buy\_coffee. } 1), \) \(\text{and } \sigma_4 = \epsilon \cdot (\text{sense\_gate. } 101) \cdot (\text{buy\_paper. } 1) \cdot (\text{buy\_coffee. } 1) \cdot (\text{goto\_gate. } 1).
\]

Therefore, the sequence of histories \(\sigma_0, \sigma_1, \sigma_2, \sigma_3, \sigma_4\) together with their corresponding programs provides a successful execution of program \(\Sigma\text{catch\_plane}\). Note that if action \textit{sense\_gate} would have returned the outcome 30 the execution chosen would be \(\sigma_0, \sigma_1, \sigma_2, \sigma_3, \sigma_4\) where \(\sigma_2 = \epsilon \cdot (\text{sense\_gate. } 101) \cdot (\text{buy\_paper. } 1), \sigma_3 = \epsilon \cdot (\text{sense\_gate. } 101) \cdot (\text{buy\_paper. } 1) \cdot (\text{goto\_gate. } 1),\) \(\text{and } \sigma_4 = \epsilon \cdot (\text{sense\_gate. } 101) \cdot (\text{buy\_paper. } 1) \cdot (\text{goto\_gate. } 1) \cdot (\text{buy\_coffee. } 1)\) and the following holds

\[
\text{Axioms} \cup \text{Sensed}[\sigma_2] \models \text{Trans}(\Sigma \text{if gate} \leq 90 \text{ then } [\text{goto\_gate. } \text{buy\_coffee}] \text{else } \ldots \text{end}[\sigma_2]).
\]

\[
\Sigma \text{goto\_gate. } \text{do}(\text{goto\_gate. } \text{end}[\sigma_2]))
\]

\[
\text{Axioms} \cup \text{Sensed}[\sigma_3] \models \text{Trans}(\Sigma\text{buy\_coffee. } \text{end}[\sigma_3]. \Sigma\text{nil. } \text{do}(\text{buy\_coffee. } \text{end}[\sigma_3]))
\]

\[
\text{Axioms} \cup \text{Sensed}[\sigma_4] \models \text{Final}(\Sigma\text{nil. } \text{end}[\sigma_4])
\]

The online execution plus search is constructing and following an implicit complete conditional plan at each step by requiring the existence of a future successful execution after the next action to be performed. It is worth noting that in this particular example, search is really not needed since the program execution is totally deterministic. It is very
important to remark as well that search does not guarantee that all the information will be available in order to construct a good plan. It may think that there is an execution without actually having anyone! As a matter of fact, search just “hopes” that future sensing information would guide the execution to a successful one.

A clear consequence of this is that search does not always guarantee that unsuccessful online executions are ruled out. While program \( \text{catch} \cdot \text{plane}(\!\langle A: \text{FALSE}? \rangle) \) has an unsuccessful execution (by executing \( A \) first and then stopping), the alternative program \( \Sigma(\text{catch} \cdot \text{plane}(\!\langle A: \text{FALSE}? \rangle)) \) does not. This is because \( \Sigma \) guarantees that the second program will not be chosen to execute since it has no successful execution. Nevertheless, knowing that there is a successful execution does not mean that we can find it.

**Example 4.4.1:** Assume the program

\[
\delta = [a_1; \text{if } p \text{ then } a_2 \text{ else } a_3][a_4; \text{if } q \text{ then } a_4 \text{ else } a_5]
\]

where actions \( a_1 \ldots a_5 \) are always possible, condition \( q \) is always true but condition \( p \) is unknown in \( S_0 \) (and \( a_1 \) does not affect it).

First, \( \text{Trans}(\Sigma \delta. S_0, \text{if } p \text{ then } a_2 \text{ else } a_3. \text{do}(a_1, S_0)) \) holds given that no matter the truth value of \( p \) we could terminate the program.

Second, \( \text{Trans}(\Sigma \delta. S_0, \text{if } q \text{ then } a_4 \text{ else } a_5. \text{do}(a_1, S_0)) \) holds given that \( a_1 \) followed by \( a_4 \) is a successful execution of \( \delta \).

However, if we choose the first step we will be stuck at the next transition because none of the following is entailed

\[
\text{Trans}(\Sigma \text{if } p \text{ then } a_2 \text{ else } a_3. \text{do}(a_1, S_0). \text{nil. do}(a_2, \text{do}(a_1, S_0)))
\]

\[
\text{Trans}(\Sigma \text{if } p \text{ then } a_2 \text{ else } a_3. \text{do}(a_1, S_0). \text{nil. do}(a_3, \text{do}(a_1, S_0)))
\]

\[
\text{Final}(\Sigma \text{if } p \text{ then } a_2 \text{ else } a_3. \text{do}(a_1, S_0))
\]

given that they all depend on the truth value of \( p \) which is not known. Hence, we have an unsuccessful execution.
If we chose the second transition at the beginning we would have a successful execution by executing $a_4$ followed by $a_1$. 

The reason of $\Sigma$ resulting in an unsuccessful execution is that search is not intelligent enough to know when an execution will actually be found. It checks that after a known step, there would eventually be a good execution: future sensing will provide the missing information in order to know the course of action at each step. We remark that the CATs approach without a theory of knowledge\footnote{Without a theory of knowledge a branch action should always be considered legal.} suffers from the same drawback if the branching is not always related to some sensing action. In the above example, if the programmer (wrongly) states a branch statement with respect to condition $p$, i.e. he writes the program $\delta' = [a_1; \text{branch}(p); \text{if } p \text{ then } a_2 \text{ else } a_3][a_1; \text{if } q \text{ then } a_4 \text{ else } a_2]$, then $Do(\delta', S_0, a_1 \cdot [p, a_2, a_4])$ would be a legal conditional plan, but once we execute action $a_1$ which action should we execute: $a_2$ or $a_4$? The intuitive idea is that since $p$ is not sensed at all, we should not branch because we want a path that is good for any truth value of $p$. In that way, $Do(\delta, S_0, c)$ holds only for $c = a_1 \cdot a_4$. In other words, while search always branches implicitly on every unknown condition, the CATs approach leaves the decision on where to branch to the programmer.

What is more, the offline conditional plan approach can still reason by cases over the remaining program in each step as the linear offline execution does. However, we can extend Theorem 4.2.2 for conditional action trees.

**Theorem 4.4.1:** Let $D$ be an action theory and let $Do_{s\text{Golog}}$ stands for the corresponding definition of $Do$ for sGolog. Let $\delta$ be a program, $s$ be a ground situation term, and $c$ be a CAT such that $D \models Do_{s\text{Golog}}(\delta, s, c)$.

Then, for any history $\sigma$ ($s \subseteq \text{end}[\sigma]$) reflecting a path in the CAT $c$ together with sensing outcomes and such that $\delta$ executes without reasoning by cases from $s$ to $\text{end}[\sigma]$, there exists a successful online execution of $\delta$ resembling $\sigma$. 

PROOF. The fact that $\sigma$ reflects a path in $c$ means, formally, that $D \cup Sensed[\sigma] \models cdo(s.c) = end[\sigma]$, where $cdo(s.c)$ is defined as the situation resulting from executing the CAT $c$ starting from situation $s$ ([22]). Thus, if we use Golog Do macro it is clear that $D \cup Sensed[\sigma] \models Do(\delta.s.end[\sigma])$.\footnote{In order to execute a sGolog program in Golog we should remove all branch statements in it.} In other words, with the sensing information available, $\delta$ has a successful offline execution from $s$ to $end[\sigma]$. Notice that it is very important to add the sensing information gathered through the history since there may be no such offline execution without it.

Moreover, such offline execution is done without reasoning by cases which means, by Theorem 4.2.2, that there must be a successful online execution of $\delta$ from $s$ to $end[\sigma]$. To fit the definition and such theorem, we consider $Sensed[\sigma]$ to be included in the set of formulas $D$.

This last theorem together with the previous one for classical offline execution assures us that if there is an offline way of executing the program, then there is an online way as well. However, an online execution may make wrong choices and therefore obtain unsuccessful executions. We also saw that search does not rule out unsuccessful executions, but it provides a convenient reasoned way of finding the next action to perform.

The advantages of the online approach with search can be summarize as follows. First, it seems the only practical way of executing large programs since offline execution is limited to local places in a program. Offline execution, whether with linear plans or conditional plans, seems intractable for large programs. Second, online execution provides a natural way of allowing both sensing information and exogenous actions. After each action execution in the world, generated by a transition, we check for sensing information and examine for exogenous events. Third, search provides a mechanism inside the whole online framework to do lookahead. Although, we know it fails in reasoning about which information is gathered via sensing and which one is not, if the program always contains sufficient information, executing an offline plan by successive steps is
like following a branch in the conditional plan of sGolog. Search together with a general online execution framework has recently been tested successfully in some delivery tasks [27] with a system that uses IndiGolog to control a MINDSTORMS\textsuperscript{TM} robot.

Finally, in the presence of sensing, an online execution can obtain executions that even the sGolog’s offline one cannot obtain by being “optimistic”. This may be reasonable in cases where we want to fail as late as possible by trying to do as much as possible.

**Example 4.4.2**: Assume the program \( \delta = (\text{sense}_p : \text{if } P \text{ then } a \text{ else } (\text{false})?) \). where action \( a \) is always possible and \( P \) is unknown. The sGolog corresponding program is \( \delta' = (\text{sense}_p : \text{branch}(P) ; \text{if } P \text{ then } a \text{ else } \text{false}?) \). First, there is no sGolog CAT such that \( \text{Axioms} \models \text{Do}(\delta, S_0, c). \) (actually, \( \text{Axioms} \not\models \exists c. \text{Do}(\delta, S_0, c) \)). Similarly, there is no online execution of program \( \Sigma \delta \). However, the online execution

\[
(\delta,(<0>))
\]

\[
(\text{if } P \text{ then } a \text{ else } (\text{false})?, (<0>)) \cdot (\text{sense}_p, <1>))
\]

\[
(\text{nil}, (<0>) \cdot (\text{sense}_p, <1>)) \cdot (a, <0>)
\]

is a valid one for program \( \delta \).

Intuitively, we do not have a CAT execution because we cannot guarantee a successful execution for *every possible* outcome of action \( \text{sense}_p \). Nevertheless, the online execution “waits” until the last moment to “fail” (in this case, up to the *if* statement). We remark that, in many cases, we prefer an agent “reasoning optimistically” by waiting until having no more choice but to stop. We say optimistically since future new information might result in good executions. For the cases where the agent needs to be sure that a solution exists, the search operator should be applied.

The following table summarize the three kind of executions we have explored in this chapter:
The abilities can be described as follows:

- **Reasoning by cases**: the ability to ignore how the program evolves through its execution. See example 4.2.1.

- **Reasoning about sensing**: the ability to reason on which information will be known by sensing and which one will not. When some unknown knowledge will be known in the future due to sensing we want to reason by cases. Otherwise, we should behave as classical offline execution by requiring a successful plan that works no matter the truth value of such unknown information. This is the example 4.4.1. We assume that sGolog programs are constructed with "correct"\(^6\) branch decision.

- **Reasoning with sensing**: the ability to construct executions depending on how future sensing will turn out to be.

- **Large programs**: the ability to "compute" the execution of large programs in a reasonable amount of time.

- **Exogenous actions**: the ability to accommodate exogenous events in a natural way.

- **Optimistic reasoning**: the ability to abort an execution as late as possible. See example 4.4.2

\(^6\)We do not have a notion of when branching should be used and when not yet. It remains as a programmer decision.
We finally remark that we consider the ability to reason about sensing a very interesting one. It would be desirable to have such property for search, even if we need to rely on the programmer intervention as it is done in sGolog.

4.4.1 Implementation Issues

In order to fully capture the definition of Σ, we generally need to reason by cases on the outcome of sensing. Actually, that is exactly what the Σ's implementation does in [10]. Thus, it is clear that allowing sensing inside offline search is explosive. It is precisely to deal with this issue in a flexible way that we have taken an online approach instead of a totally offline one.

In IndiGolog, given that offline execution is narrowed to local places (and not to the whole program), we rely on the following assumption: before performing a search, the agent has all the information needed. In other words, all sensing inside Σ should not be necessary for finding the plan.

Next, we formalize the intuitive idea of a system not being able to gather information via sensors further than situation s, where s' is supposed to be the current situation.

Definition 4.4.1: Given a GAT D and a ground situation term s', the GAT D' is obtained by replacing each GSFA \{α(\bar{x},s) ⊃ F(\bar{x},s) ≡ ρ(\bar{x},s)\} in D with the new version \{α(\bar{x},s) ∧ s ⊆ s' ⊃ F(\bar{x},s) ≡ ρ(\bar{x},s)\}.

Intuitively, D' is an action theory where the reasoning about sensors is restricted to situations preceding s'. After that, the sensors say nothing about the fluents.

Hence, the idea is to build an implementation that behaves as if all relevant sensing information were available at the outset of the search. A clear drawback is that the implementation will not be able to determine whether some currently unknown fluent will

\textsuperscript{7}Although D' does not conform with the GSFA definition (because of the form of the guards), it makes no theoretical difference here.
be known later. For example, suppose that fluent $f$ is unknown, but that after performing action $a$, a sensor will provide its correct value. Then the program $\Sigma[a.\, if(f, \, p_1, \, p_2)]$ fails since $a$ is being considered offline. However, the alternative program $[a.\, \Sigma[i\, f(f, \, p_1, \, p_2)]$ works fine because the sensor is consulted before the search.

### 4.4.2 Towards a stronger search operator

As we have seen, search has the disadvantage of not being able to reason about which information will be available in the future. This causes search not to be exempt from deriving unsuccessful executions. Nonetheless, it is not hard to imagine a new search operator that behaves as sGolog does. that is a local lookahead that instead of returning just the next step it tries to find a conditional plan.

Yet, it would be needed to change the concept of online execution given that $Trans$ would now return a whole conditional action tree ready to be executed, which would imply more than just one action. Luckily, instead of doing that, which would mean to change all previous framework, we can think of search returning a remaining program that is ready to execute. and such that it has no search and resembles the conditional action tree that sGolog would construct.

If we restrict to sGolog programs inside search this can be easily done by redefining $\Sigma$ as follows:

$$Trans(\Sigma \delta. \, s. \, \delta'. \, s') \equiv \exists c.\, Do_{sGolog}(\delta. \, s. \, c) \wedge catToprog(c) = \delta' \wedge s = s' \quad (4.4)$$

Function $catToprog(c)$ (which stands for conditional action tree to program) calculates the corresponding program $\delta'$ which resembles the conditional action tree $c$. The good news is that $\delta'$ would have no search and therefore it would be ready to execute in an online fashion. Moreover, it would work for any possible future sensing outcome. Notice that while the original search operator actually make one step, this alternative definition makes no step at all. The step is actually reflected in the program that remains
It is easy to see how a program can resemble a CAT since a branch in a CAT can be translated to a conditional. Then, the axioms for \( catToprog(c) \) would are the following:

\[
\begin{align*}
\text{catToprog}(\varepsilon) &= \text{nil} \\
\text{catToprog}(a) &= a \\
\text{catToprog}( [\varepsilon, c_1, c_2] ) &= \text{if } \varepsilon \text{ then } \text{catToprog}(c_1) \text{ else } \text{catToprog}(c_2)
\end{align*}
\]

This new definition of search would be as powerful as sGolog, which means that if the programmer makes appropriate decisions on where to branch we would never obtain unsuccessful executions. Furthermore, if sGolog returns a CAT which can be followed as new information is gathered via sensing, there must also be a successful online execution of the remaining program calculated by the search construct. In addition, since the offline execution is restricted to limited areas in IndiGolog, we may expect to have little sensing so that conditional plans have more chances to remain tractable. We also note that it seems possible, although we have not obtained yet a complete sets of axioms, to allow all ConGolog constructs inside search. For that, we should redefine \( Trans \) and \( Final \) and add new axioms as sGolog does.

The main drawback of conditional action trees is that they are far away from the expressivity power of the robot programming language proposed in [29] and [26]. While an augmented version of this programming language is proved in [29] to be universal for goal achievability, conditional action trees lack of complex structures like loops which causes some goals not to be achievable by any CAT. Since the IndiGolog programming language already has all the complex constructs, it would be very interesting to come up with a search definition that instead of returning a program resembling a CAT, returns more complicate programs that are actually plans in the way [26] suggests. It seems, however, that there is a compromise between the expressivity we can achieve in the theory and what we can actually implement. In that way, while the search implementation that
we will propose in the next Chapter is quite near to its original definition. It seems very difficult, although very interesting as well, to implement a more general account of plans like the robot programming language of [29].

4.5 Chapter Summary

In this chapter, we discussed different ways of executing high-level programs by mainly giving examples that show the differences among them. In particular, we committed to a combination between online and offline execution where programs are mostly executed online except from local places in it where offline lookahead is required via a new construct $\Sigma$. In such a framework, the search operator arose as the most interesting aspect, which motivates us to study its relation with both concurrency and sensing. We also proposed a realistic treatment of sensing in high-level programs such that an efficient implementation can be developed.
Chapter 5

An Incremental Interpreter over Guarded Theories

We have developed a simple incremental interpreter in Prolog to implement the online execution of high-level programs based on guarded theories of action including sensing, exogenous actions, and an approximation of the search operator. The task is fundamentally a theorem-proving one: does a certain $Trans$ or $Final$ formula follow logically from the axioms of the action theory and the sensing information obtained so far. The interpreter can be divided into three parts: the main loop, the implementation of $Trans$ and $Final$, and the evaluation of test conditions.

5.1 Two Problems

When it comes to implementing the online execution of IndiGolog programs two problems arise:

(a) The need for a "complete for free variables" procedure for evaluating tests:

(b) The need for implementing lookahead in an efficient manner.
By a complete for free variables procedure, we mean one that determines the truth value of a formula in every point of the domain of the free variables. For instance consider the formula $\text{open}(x, s)$ saying whether door $x$ is open or not in situation $s$. If we were to give such a formula to our procedure, we would like to obtain its truth value for all possible $x$ in the domain, including the cases where the formula is "unknown". In [11], an evaluation procedure for test conditions was given for guarded theories. However, when such procedure is used to execute high-level programs, unsound results can be obtained in the presence of prioritized concurrency. To see why we give the following example.

**Example 5.1.1:** Consider the program $\delta = (\pi x. \delta(x))\delta_2$, where $\text{Axioms} \nvdash \delta(3, S_0)$. $\text{Axioms} \nvdash \neg \delta(3, S_0)$ but $\text{Axioms} \models \forall x. x \neq 3 \supset \neg \delta(x, S_0)$. It is easy to see that there should be no execution of this program in $S_0$. First, we cannot prove any transition for the subprogram $\delta_1 = (\pi x. \delta(x))$ in $S_0$ since $\text{Axioms} \nvdash \exists x. \delta(x, S_0)$. What is more, we cannot prove that $\delta_1$ has no transition either, and so $\delta_2$ cannot be allowed to execute. In other words, we have no transition implied for $\delta$ in situation $S_0$.

But, if the interpreter simply fails to get a transition for $\{\pi x. \delta(x)\}$, then it would try to execute $\delta_2$, potentially deriving an unsound transition.  

To solve this first problem we decided to restrict ourselves to a subset of programs, called **bounded programs**, for which we always derive sound transitions.

The second problem has to do with the implementation of the search construct described in the previous chapter. As we recall from one of the properties of search (equation 4.3), computing the search of a program $\delta$ in a situation $s$ requires us to compute whether $\delta$ would eventually end up in a successful execution. Since there may be sensing information that is still unknown at the time of doing the search, we should, in general, reason by cases on all possible sensing outcomes. For instance, in the *catch plane* example of the previous chapter, search should reason by cases over the sensing obtained after executing action *sense gate*. The lookahead implementation suggested in [10] does that at each
sensing action by computing a path for each possible sensing outcome (1 or 0). However, this results in an explosive problem and it is even more intractable when sensing is not binary, as in the case of sensors. Here, we propose an approximation of search where sensors are assumed to be off once the search is initiated. This approximation needs less computation effort while it remains realistic for many cases.

5.2 Generalized Regression

A regression mechanism for solving the projection task for basic action theories is suggested by Pirri and Reiter in [33]. Here, we present a similar form of regression to the one suggested in [11], which can be used to solve the projection task for guarded action theories. Besides, we will give similar soundness and completeness results to those ones given in [11] and, later, we will show a Prolog interpreter based on such regression.

This form of regression can be described as a "generalized form of regression that is a sensible compromise between syntactic transformations and logical reasoning" ([11]). In the following we assume $D$ is a guarded action theory, $\sigma$ is a history, and $\phi(\bar{x}.s)$ and $\psi(\bar{x}.s)$ are sensor-fluent formulas. $\rho(\bar{x}.s)$ is a sensor formula and $\gamma(\bar{x}.s)$ is a fluent formula. We use the notation $\phi \sigma$ to mean the formula that results from replacing every sensor function $h_j(s)$ in $\phi$ by the $j$-th component of the final sensor reading in $\sigma$.

Lemma 5.2.1: Let $\rho(\bar{x}.s)$ be a sensor formula. Then for every history $\sigma$ the following holds:

$$\models Sensed[\sigma] \supset \forall \bar{x}.\rho(\bar{x}.end[\sigma]) \equiv \rho(\bar{x}.s) \sigma$$

Proof. Straightforward from the fact that $\rho(\bar{x}.s) \sigma$ is exactly $\rho(\bar{x}.end[\sigma])$ with each sensing function term $h_j(end[\sigma])$ replaced by the value assigned by formula $Sensed[\sigma]$. Because $\rho(\bar{x}.s)$ mentions no fluents, but only sensing function, $\rho(\bar{x}.s) \sigma$ mentions no situation term.
We can simplify a formula by making use of sensor information for a particular fluent.

**Definition 5.2.1:** \( \phi(\vec{x}, s) \) simplifies to \( \psi(\vec{x}, s) \) at \( \sigma \) wrt a relational fluent \( F(\vec{t}, s) \) mentioned in \( \phi(\vec{x}, s) \) if there is a GSFA in

\[
\exists(\vec{z}, s) \supset [ F(\vec{z}, s) \equiv \rho(\vec{z}, s) ]
\]

where \( D \cup \text{Sensed}[\sigma] \models \forall \exists(\vec{t}.\text{end}[\sigma]) \) and \( \psi(\vec{x}, s) \) is the result of replacing \( F(\vec{t}, s) \) in \( \phi(\vec{x}, s) \) by \( \rho(\vec{t}, s) \setminus \sigma \).

**Lemma 5.2.2:** If \( \phi(\vec{x}, s) \) simplifies to \( \psi(\vec{x}, s) \) wrt to some fluent at \( \sigma \), then

\[
D \cup \text{Sensed}[\sigma] \models \forall \vec{x}.\phi(\vec{x}.\text{end}[\sigma]) \equiv \psi(\vec{x}.\text{end}[\sigma])
\]

**Proof.** By induction on the structure of \( \phi(\vec{x}, s) \) and Lemma 5.2.1.

**Base Case:** If \( \phi(\vec{x}, s) = F(\vec{t}, s) \) then \( \psi(\vec{x}, s) = \rho(\vec{t}, s) \setminus \sigma \). First,

\[
D \cup \text{Sensed}[\sigma] \models F(\vec{t}.\text{end}[\sigma]) \equiv \rho(\vec{t}.\text{end}[\sigma])
\]

given that there is a GSFA such that \( D \models \exists(\vec{t}.\text{end}[\sigma]) \supset \forall \exists(\vec{t}.F(\vec{t}.\text{end}[\sigma]) \equiv \rho(\vec{t}.\text{end}[\sigma]) \) and \( D \cup \text{Sensed}[\sigma] \models \forall \exists(\vec{t}.\phi(\vec{t}.\text{end}[\sigma]). \)

By Lemma 5.2.1, \( D \cup \text{Sensed}[\sigma] \models \forall \exists F(\vec{t}.\text{end}[\sigma]) = \rho(\vec{t}, s) \setminus \sigma \) from which we conclude that \( D \cup \text{Sensed}[\sigma] \models \forall \exists F(\vec{t}.\text{end}[\sigma]) \equiv \rho(\vec{t}, s) \setminus \sigma \). Since \( \rho(\vec{t}, s) \setminus \sigma \) mentions no situation term, \( \psi(\vec{t}.\text{end}[\sigma]) = \psi(\vec{x}, s) \) and \( D \cup \text{Sensed}[\sigma] \models \forall \exists F(\vec{t}.\text{end}[\sigma]) \equiv \psi(\vec{t}.\text{end}[\sigma]). \)

**Induction Step:** We prove the case \( \phi(\vec{x}, s) = \phi_1(\vec{x}, s) \land \phi_2(\vec{x}, s) \). All the others are similar.

Since \( \phi(\vec{x}, s) \) simplifies wrt some fluent \( F(\vec{t}, s) \), then such a fluent must be mentioned either in \( \phi_1 \) in \( \phi_2 \), or in both. Suppose it is only mentioned in \( \phi_1 \). Because simplification is basically a syntactic transformation it must be the case that \( \psi(\vec{x}, s) = \psi_1(\vec{x}, s) \land \psi_2(\vec{x}, s) \) where \( \phi_1 \) simplifies to \( \psi_1 \) wrt the relational fluent \( F(\vec{t}, s) \) and \( \psi_2 = \phi_2 \). By induction hypothesis on \( \phi_1 \). \( D \cup \text{Sensed}[\sigma] \models \forall \exists F(\vec{t}.\phi_1(\vec{x}.\text{end}[\sigma]) \equiv \psi_1(\vec{t}.\text{end}[\sigma]) \).

It follows then that

\[
D \cup \text{Sensed}[\sigma] \models \forall \exists F(\vec{t}.\phi_1(\vec{x}.\text{end}[\sigma]) \land \phi_2(\vec{x}.\text{end}[\sigma]) \equiv \psi_1(\vec{t}.\text{end}[\sigma]) \land \psi_2(\vec{t}.\text{end}[\sigma])
\]
which is the same as $D \cup Sensed[\sigma] \models \forall \vec{F}. o(\vec{F}. end[\sigma]) \equiv \psi(\vec{F}. end[\sigma])$.

When $F(\vec{F}. s)$ is mentioned in $\phi_2$ but not in $\phi_1$ is symmetric. However, if $F(\vec{F}. s)$ is mentioned in both $\phi_1$ and $\phi_2$, then $\psi(\vec{F}. s) = \psi_1(\vec{F}. s) \land \psi_2(\vec{F}. s)$ where $\phi_1$ simplifies to $\psi_1$ wrt the fluent $F(\vec{F}. s)$, and $\phi_2$ simplifies to $\psi_2$ wrt the fluent $F(\vec{F}. s)$.

By induction hypothesis on both simplification steps.

$$D \cup Sensed[\sigma] \models \forall \vec{F}. o_1(\vec{F}. end[\sigma]) \equiv \psi_1(\vec{F}. end[\sigma])$$

$$D \cup Sensed[\sigma] \models \forall \vec{F}. o_2(\vec{F}. end[\sigma]) \equiv \psi_2(\vec{F}. end[\sigma])$$

It follows then that

$$D \cup Sensed[\sigma] \models \forall \vec{F}. o_1(\vec{F}. end[\sigma]) \land o_2(\vec{F}. end[\sigma]) \equiv \psi_1(\vec{F}. end[\sigma]) \land \psi_2(\vec{F}. end[\sigma])$$

which is the same as $D \cup Sensed[\sigma] \models \forall \vec{F}. o(\vec{F}. end[\sigma]) \equiv \psi(\vec{F}. end[\sigma])$. 

Several simplifications can be applied to a formula so that many online sensor information is used at the same time.

**Definition 5.2.2**: $o(\vec{F}. s)$ simplifies to $\psi(\vec{F}. s)$ at $\sigma$ if there exists a sequence of formulas $o_1(\vec{F}. s), \ldots, o_n(\vec{F}. s)$ and a sequence of fluents $F_1, \ldots, F_{n-1}$, where $n \geq 1$, such that $o_1(\vec{F}. s) = o(\vec{F}. s)$, $\psi(\vec{F}. s) = o_n(\vec{F}. s) \setminus \sigma$ and $\psi_i(\vec{F}. s)$ simplifies to $\psi_{i+1}(\vec{F}. s)$ wrt fluent $F_i$ for $i = 1 \ldots n - 1$.

Observe that a formula may simplify to another formula because of zero, one or many individual simplifications. Hence, a formula may simplify to many different formulas. Because the number of fluents in $o(\vec{F}. s)$ is finite and simplification is no more than simplifying wrt one fluent at a time, the definition is well-defined. Next, we consider simplifications involving reasoning backwards using the GSSA.

**Definition 5.2.3**: $o(\vec{F}. s)$ rolls back to $\psi(\vec{F}. s)$ from a non-initial history $\sigma = \sigma' \cdot (A. \vec{\mu})$ iff $o(\vec{F}. s)$ simplifies to $o'(\vec{F}. s)$ at $\sigma$, and for every fluent $F_1(\vec{F}_1. s), \ldots, F_k(\vec{F}_k. s)$ mentioned
in $\alpha'(\overline{t}.s)$ with $k \geq 0$, there is a GSSA in $D$

$$\alpha_i(\overline{t}.a.s) \supset [F_i(\overline{t}.do(a.s)) \equiv \gamma_i(\overline{t}.a.s)]$$

where $D \cup Sensed[\sigma'] = \forall \alpha_i(\overline{t}.a.end[\sigma'])$. and $\nu'(\overline{t}.s)$ is the result of replacing each $F_i(\overline{t}_i.s)$ in $\alpha'(\overline{t}.s)$ by $\gamma_i(\overline{t}_i.a.s)$

**Lemma 5.2.3:** If $\alpha(\overline{t}.s)$ rolls back to $\nu'(\overline{t}.s)$ from a non-initial history $\sigma = \sigma' \cdot (A, \overline{\mu})$. then

$$D \cup Sensed[\sigma] \models \forall \overline{t}.\alpha(\overline{t}.end[\sigma]) \equiv \nu(\overline{t}.end[\sigma'])$$

**Proof.** By induction on the structure of $\alpha(\overline{t}.s)$ and Lemma 5.2.2.

**Base Case:** if $\alpha(\overline{t}.s) = F(\overline{t}.s)$ then either it simplifies to itself or to some formula $\rho(\overline{t}.s) \\setminus \sigma$ that mentions no situation term. In the latter case, it is easy to see that the Lemma holds given that $\nu'(\overline{t}.s) = \rho(\overline{t}.s) \setminus \sigma$ is a simplification step of $F(\overline{t}.s)$ wrt itself. Thus, by Lemma 5.2.2, it follows that $D \cup Sensed[\sigma] \models \forall \overline{t}.\alpha(\overline{t}.end[\sigma]) \equiv \nu(\overline{t}.end[\sigma'])$.

If, however, $F(\overline{t}.s)$ simplifies to itself, i.e. $\alpha'(\overline{t}.s) = F(\overline{t}.s)$, then $\nu'(\overline{t}.s) = \gamma(\overline{t}.A.s)$ and there is a GSSA in $D$ such that

$$\alpha(\overline{t}.a.s) \supset [F(\overline{t}.do(A.s)) \equiv \gamma(\overline{t}.A.s)]$$

and $D \cup Sensed[\sigma] \models \forall \overline{t}.\alpha(\overline{t}.A.end[\sigma'])$.

Therefore, $D \cup Sensed[\sigma] \models F(\overline{t}.do(A.end[\sigma'])) \equiv \gamma(\overline{t}.A.end[\sigma'])$. which, because of $end[\sigma] = do(A.end[\sigma'])$, it is the same as $D \cup Sensed[\sigma] \models \forall \overline{t}.\alpha(\overline{t}.end[\sigma]) \equiv \nu'(\overline{t}.end[\sigma'])$.

**Induction Step:** We only prove the case $\alpha(\overline{t}.s) = \alpha_1(\overline{t}.s) \land \alpha_2(\overline{t}.s)$.

Since $\alpha(\overline{t}.s)$ rolls back to $\nu'(\overline{t}.s)$, then $\nu'(\overline{t}.s) = \nu_1(\overline{t}.s) \land \nu_2(\overline{t}.s)$ where $\alpha_1$ rolls back to $\nu_1$, and $\alpha_2$ rolls back to $\nu_2$. By applying the induction hypothesis twice

$$D \cup Sensed[\sigma] \models \forall \overline{t}.\alpha_1(\overline{t}.end[\sigma]) \equiv \nu_1(\overline{t}.end[\sigma])$$

and

$$D \cup Sensed[\sigma] \models \forall \overline{t}.\alpha_2(\overline{t}.end[\sigma]) \equiv \nu_2(\overline{t}.end[\sigma])$$
From that, it follows immediately that

$$\mathcal{D} \cup \text{Sensed}[\sigma] \models \forall \bar{x}. o_1(\bar{x}. end[\sigma]) \land o_2(\bar{x}. end[\sigma]) \equiv v_1(\bar{x}. end[\sigma]) \land v_2(\bar{x}. end[\sigma])$$

or, what is the same, $$\mathcal{D} \cup \text{Sensed}[\sigma] \models \forall \bar{x}. o(\bar{x}. end[\sigma]) \equiv v(\bar{x}. end[\sigma])$$

Now we define what we mean by regression as successive roll backs.

**Definition 5.2.4:** $\circ(\bar{x}. s)$ regresses to $\nu(\bar{x}. s)$ from $\sigma$ iff either

- $\sigma = (\bar{\mu}_0)$ and $\circ(\bar{x}. s)$ simplifies to $\nu(\bar{x}. s)$ at $\sigma$:
- $\sigma = \sigma' \cdot \langle A, \bar{\mu} \rangle$ and $\circ(\bar{x}. s)$ rolls back to $\nu'(\bar{x}. s)$ from $\sigma$, and $\nu'(\bar{x}. s)$ regresses to $\nu(\bar{x}. s)$ from $\sigma'$.

**Theorem 5.2.1:** If $\circ(\bar{x}. s)$ regresses to $\nu(\bar{x}. s)$ from $\sigma$ then

$$\mathcal{D} \cup \text{Sensed}[\sigma] \models \forall \bar{x}. o(\bar{x}. end[\sigma]) \equiv v(\bar{x}. S_0)$$

**Proof.** By induction on the length of $\sigma$.

**Base Case:** If $\sigma = (\bar{\mu}_0)$, $\circ(\bar{x}. s)$ simplifies to $\nu(\bar{x}. s)$ at $\sigma$ and using Lemma 5.2.2

$$\mathcal{D} \cup \text{Sensed}[\sigma] \models \forall \bar{x}. o(\bar{x}. end[\sigma]) \equiv v(\bar{x}. end[\sigma])$$

Given that $end[\sigma] = S_0$ we conclude $\mathcal{D} \cup \text{Sensed}[\sigma] \models \forall \bar{x}. o(\bar{x}. end[\sigma]) \equiv v(\bar{x}. S_0)$.

**Induction Step:** If $\sigma = \sigma' \cdot (\bar{\mu})$, then $\circ(\bar{x}. s)$ rolls back to $\nu'(\bar{x}. s)$ and by Lemma 5.2.3

$$\mathcal{D} \cup \text{Sensed}[\sigma] \models \forall \bar{x}. o(\bar{x}. end[\sigma]) \equiv v'(\bar{x}. end[\sigma'])$$

Moreover, $\nu'(\bar{x}. s)$ regresses to $\nu(\bar{x}. s)$ from $\sigma'$. Because of the induction hypothesis,

$$\mathcal{D} \cup \text{Sensed}[\sigma'] \models \forall \bar{x}. \nu'(\bar{x}. end[\sigma']) \equiv v(\bar{x}. S_0)$$
Given that \( \models Sensed[\sigma] \supset Sensed[\sigma'] \) (\( \sigma' \) is a sub-history of \( \sigma \)) it follows that

\[
\mathcal{D} \cup Sensed[\sigma] \models \forall \vec{x}. \phi(\vec{x}, \text{end}[\sigma]) \equiv \psi(\vec{x}, S_0)
\]

We then have a sound mechanism to evaluate a formula at a history by regressing it and checking the result against the initial database. In contrast with the regression stated in [33], this generalized regression has two drawbacks: (i) it is not complete; (ii) it is not specified how guards are evaluated.

As an example of the first problem just take the formula \((F(s) \lor \neg F(s))\) where nothing is known about fluent \( F \). It will not regress even though it will be entailed at any history. This problem will be solved by restricting to special histories called just-in-time histories. We call a history just-in-time for a formula, if the actions and sensing readings it contains are enough to guarantee that suitable formulas (including guards) can be evaluated at appropriate points to determine the truth value of all the fluents in the formula.

**Definition 5.2.5:** A history \( \sigma \) is a just-in-time-history (JIT-history) for a sensor-fluent formula \( \phi(\vec{x}, s) \) wrt a background GAT \( \mathcal{D} \) iff

- \( \phi(\vec{x}, s) \) is a sensor formula:\(^1\)
- \( \phi(\vec{x}, s) = \neg \phi_1(\vec{x}, s) \lor \phi_1(\vec{x}, s) \land \phi_2(\vec{x}, s) \) and \( \sigma \) is a JIT-history for \( \phi_1(\vec{x}, s) \) and \( \phi_2(\vec{x}, s) \):
- \( \phi(\vec{x}, s) = \exists y. \phi_1(\vec{x}, y, s) \) and \( \sigma \) is a JIT-history for \( \phi_1(\vec{x}, y, s) \):
- \( \phi(\vec{x}, s) = F(\vec{x}, s) \) where \( F \) is a fluent and \( \sigma \) is an initial history \((\mu_0)\). and either \( \mathcal{D}_0 \models \forall F(\vec{x}, S_0) \) or \( \mathcal{D}_0 \models \forall \neg F(\vec{x}, S_0) \):

---

\(^1\)This includes formulas with no sensor functions and no fluents like \( \exists y. a = \text{pickUp}(y) \).
• $\sigma(\vec{x}, s) = F(\vec{t}, s)$, where $F$ is a (relational) fluent. and for some GSFA of the form
  \[ \{ \beta(\vec{z}, s) \supset [F(\vec{z}, s) \equiv \rho(\vec{z}, s)] \}. \]  
  $\sigma$ is a JIT-history for formula $\beta(\vec{t}, s)$, and such that
  \[ \mathcal{D} \cup \text{Sensed}[\sigma] \models \forall \beta(\vec{t}, \text{end}[\sigma]). \]

• $\sigma(\vec{x}, s) = F(\vec{t}, s)$, where $F$ is a fluent. $\sigma = \sigma' \cdot (A, \vec{\mu})$. and there is a GSSA
  \[ \{ \alpha(\vec{z}, a, s) \supset [F(a, do(a, s)) \equiv \gamma(\vec{z}, a, s)] \}. \]  
  $\sigma'$ is a JIT-history for both $\alpha(\vec{t}, A, s)$ and $\gamma(\vec{t}, A, s)$, and $\mathcal{D} \cup \text{Sensed}[\sigma'] \models \forall \alpha(\vec{t}, A, \text{end}[\sigma])$.

\[ \square \]

When a history has sufficient information for a formula, such formula should regress.

**Theorem 5.2.2:** If $\sigma$ is a JIT-history for $\phi(\vec{x}, s)$, then $\phi(\vec{x}, s)$ regresses to some formula  
$\psi(\vec{x}, s)$. Moreover, if every formula mentioning no fluent, no sensing function, and no free variable is always known to be true or false wrt $D_0$, then for any ground term $t_j$ of 
the sort of $\vec{x}$, either $D_0 \models \psi(t_j, S_0)$ or $D_0 \models \neg \psi(t_j, S_0)$.

**Proof.** We prove it by induction on the length of $\sigma$ and on the structure of $\phi(\vec{x}, s)$.

For the second part, observe that if $\phi(\vec{x}, s)$ is a sensor formula and $\psi(\vec{x}, s)$ is its simplification at some history, then $D_0 \models \psi(t_j, S_0)$ or $D_0 \models \neg \psi(t_j, S_0)$ for any ground vector of 
terms $t_j$. This is because after simplifying all sensing functions, the result is a formula 
not mentioning any fluent or any sensing function.

**Base Case:** Suppose the initial history $\sigma_0 = \vec{\mu}_0$. We now perform induction on the 
structure of $\phi(\vec{x}, s)$.

- **Base Case:** Suppose $\phi(\vec{x}, s) = F(\vec{t}, s)$. Either $\phi(\vec{x}, s)$ regresses to itself or to some 
  $\psi(\vec{x}, s)$ because of a single simplification step wrt $F$. In the first case, either $D_0 \models \forall F(\vec{t}, S_0)$ or $D_0 \models \forall \neg F(\vec{t}, S_0)$ so that the second part of the theorem holds trivially.
  If, however, $F(\vec{t}, s)$ simplifies wrt $F$, then $\psi(\vec{x}, s)$ contains no fluent, and no sensing function 
  so that the second part of the theorem should hold as well.

If $\phi(\vec{x}, s)$ is a sensor formula then it fully simplifies to $\phi(\vec{x}, s) \setminus \sigma$ given that there 
is no fluent mentioned in it, and therefore no possible simplification step. Thus,
\( \sigma(\bar{x}.s) \) regresses to \( \sigma(\bar{x}.s)\backslash \sigma \). What is more, \( \sigma(\bar{x}.s)\backslash \sigma \) is a formula not mentioning any sensing function and any fluent so that the second part of the theorem holds easily.

- **Induction Step:** On the structure of \( \sigma(\bar{x}.s) \): (i) If \( \sigma(\bar{x}.s) = \neg \psi_1(\bar{x}.s) \), then by induction \( \psi_1(\bar{x}.s) \) regresses to some \( \psi_1(\bar{x}.s) \) from \( \sigma_0 \). Hence, \( \sigma(\bar{x}.s) \) has to regress to \( \neg \psi_1(\bar{x}.s) \) from \( \sigma_0 \). (ii) If \( \sigma(\bar{x}.s) = \psi_1(\bar{x}.s) \land \psi_2(\bar{x}.s) \), then by induction \( \psi_1(\bar{x}.s) \) regresses to some \( \psi_1(\bar{x}.s) \) from \( \sigma_0 \); and \( \psi_2(\bar{x}.s) \) regresses to some \( \psi_2(\bar{x}.s) \) from \( \sigma_0 \). Hence, \( \sigma(\bar{x}.s) \) has to regress to \( \psi_1(\bar{x}.s) \land \psi_1(\bar{x}.s) \) from \( \sigma_0 \). (iii) Finally, if \( \sigma(\bar{x}.s) = \exists y.\psi_1(\bar{x}.y.s) \), then by induction \( \psi_1(\bar{x}.y.s) \) regresses to some \( \psi_1(\bar{x}.y.s) \) from \( \sigma_0 \). Hence, \( \sigma(\bar{x}.s) \) has to regress to \( \psi(\bar{x}.s) = \exists y.\psi_1(\bar{x}.y.s) \) from \( \sigma_0 \).

**Induction Step:** Suppose the history \( \sigma' = (A.\bar{\mu}) \cdot \sigma \). We now perform induction on the structure of \( \sigma(\bar{x}.s) \).

- **Base Case:** Suppose \( \sigma(\bar{x}.s) \) is a sensor formula. Then it fully simplifies to \( \sigma(\bar{x}.s)\backslash \sigma \) given that there is no fluent mentioned. What is more, \( \sigma(\bar{x}.s) \) also rolls back to \( \sigma(\bar{x}.s)\backslash \sigma \) from \( \sigma' \). Next, by induction hypothesis on \( \sigma \), \( \sigma(\bar{x}.s)\backslash \sigma \) regresses to some \( \psi(\bar{x}.s) \) from \( \sigma \). Therefore, \( \sigma(\bar{x}.s) \) regresses to \( \psi(\bar{x}.s) \) from \( \sigma' \). The second part of the theorem holds as well from the induction hypothesis on \( \psi(\bar{x}.s) \).

Suppose \( \sigma(\bar{x}.s) = F(\bar{t}.s) \). One of the following cases should happen:

- There is a GSFA \( \{ \beta(\bar{x}.s) \supset [F(\bar{x}.s) \equiv \rho(\bar{x}.s)] \} \), such that \( \sigma \) is a JIT-history for \( \beta(\bar{t}.s) \), and \( D \cup Sensed[\sigma'] \models \forall \beta(\bar{t}.end[\sigma]) \) holds. Thus, \( \sigma(\bar{x}.s) \) simplifies (wrt fluent \( F \)), fully simplifies, and rolls back to \( \psi_1(\bar{x}.s) = \rho(\bar{t}.s)\backslash \sigma' \) at \( \sigma' \). Next, by induction hypothesis on \( \sigma \), \( \psi_1(\bar{x}.s) \) regresses to some \( \psi(\bar{x}.s) \) from \( \sigma \). Therefore, \( \sigma(\bar{x}.s) \) regresses to \( \psi(\bar{x}.s) \) from \( \sigma' \). The second part of the theorem also is concluded from the induction hypothesis.

- There is a GSSA \( \{ \alpha(\bar{x}.a.s) \supset [F(\bar{x}.do(a.s)) \equiv \gamma(\bar{x}.a.s)] \} \), \( \sigma' \) is a JIT-history for both \( \alpha(\bar{t}.A.s) \) and \( \gamma(\bar{t}.A.s) \), and \( D \cup Sensed[\sigma'] \models \forall \alpha(\bar{t}.A.end[\sigma]) \).
Thus, $\sigma(\bar{x}.s)$ rolls back to $\nu_1(\bar{x}.s) = \gamma(\bar{r}.A.s)$ at $\sigma'$. Next, since $\sigma$ is a JIT-history for $\gamma(\bar{r}.A.s)$, by induction hypothesis on $\sigma$. $\nu_1(\bar{x}.s)$ regresses to some $\nu(\bar{x}.s)$ from $\sigma$. Therefore, by definition of regression, $\sigma(\bar{x}.s)$ regresses to $\nu(\bar{x}.s)$ from $\sigma'$. The second part of the theorem also is concluded from the induction hypothesis.

- **induction Step:** On the structure of $\sigma(\bar{x}.s)$: (i) If $\sigma(\bar{x}.s) = \neg \nu_1(\bar{x}.s)$, then by induction $\sigma_1(\bar{x}.s)$ regresses to some $\nu_1(\bar{x}.s)$ from $\sigma'$. Hence, $\sigma(\bar{x}.s)$ has to regress to $\neg \nu_1(\bar{x}.s)$ from $\sigma'$. (ii) If $\sigma(\bar{x}.s) = \nu_1(\bar{x}.s) \land \nu_2(\bar{x}.s)$, then by induction $\sigma_1(\bar{x}.s)$ regresses to some $\nu_1(\bar{x}.s)$ from $\sigma_0$: and $\sigma_2(\bar{x}.s)$ regresses to some $\nu_2(\bar{x}.s)$ from $\sigma'$. Hence, $\sigma(\bar{x}.s)$ has to regress to $\nu_1(\bar{x}.s) \land \nu_2(\bar{x}.s)$ from $\sigma'$. (iii) Finally, if $\sigma(\bar{x}.s) = \exists y. \sigma_1(\bar{x}.y.s)$, then by induction $\sigma_1(\bar{x}.y.s)$ regresses to some $\nu_1(\bar{x}.y.s)$ from $\sigma'$. Hence, $\sigma(\bar{x}.s)$ has to regress to $\nu(\bar{x}.s) = \exists y. \nu_1(\bar{x}.y.s)$ from $\sigma'$.

As stated above, the regression mechanism says nothing about the evaluation of guards in the process of regressing a formula. However, one may think of using the same regression mechanism to achieve that. It is not hard then to imagine problematic cases due to the fact that guards in GSFA are formulas in the same situation as the fluent we are willing to evaluate. Assume then a GAT containing the following two GSFA:

$$
F_1(s) \supset F_2(s) \equiv \rho_2(s)
$$

$$
F_2(s) \land \alpha(s) \supset F_1(s) \equiv \rho_1(s)
$$

To know the truth value of fluent $F_2$ at a history $\sigma$, we should know that $F_1(s)$ is true at $\sigma$. Besides, to obtain $F_1(s)$ truth value we should know that $F_2(s)$ is true at $\sigma$! This cyclic dependency relation would have a big impact in any algorithm based on generalized regression. As a matter of fact, we may fall into infinite loop by first trying to obtain the truth value of $F_2$ at $\sigma$, for which we try to obtain the truth value of $F_1$ at
\( \sigma \) for which we try to compute the truth value of \( F_2 \) in \( \sigma \) and so on. We expect then to have theories of action avoiding this kind of circular relations between fluents.

**Definition 5.2.6:** The sensor-formula \( \phi(\bar{x},\bar{s}) \) \text{g-regresses to} \( \psi(\bar{x},\bar{s}) \) from \( \sigma \) iff \( \phi(\bar{x},\bar{s}) \) regresses to \( \psi(\bar{x},\bar{s}) \) from \( \sigma \) and

1. for every simplification step where a GSF\( A \) in \( D \{J(\bar{x},\bar{s}) \supset [F(\bar{x},\bar{s}) \equiv \rho(\bar{x},\bar{s})]\}\) is used to simplify \( F(\bar{x},\bar{s}) \) at some subhistory \( \sigma' \). It is the case that: (i) if \( \sigma' = \sigma'' \cdot (A,\bar{\mu}) \) is not an initial history, then \( J(\bar{x},\bar{s}) \) rolls back to some \( J'(\bar{x},\bar{s}) \) without using any GSF\( A \) in \( GSF\!(F) \) and \( J'(\bar{x},\bar{s}) \) g-regresses to some \( J''(\bar{x},\bar{s}) \): (ii) if \( \sigma' \) is an initial history then either \( D_0 \models \forall F(\bar{x},S_0) \) or \( D_0 \models \forall \neg F(\bar{x},S_0) \) applies: or \( J(\bar{x},\bar{s}) \) g-regresses to some \( J'(\bar{x},\bar{s}) \) wrt the background theory \( D = \text{GSF\!(F)} \).

2. for every roll back step, where a GSS\( A \) in \( D \{\phi(\bar{x},\bar{s}) \supset [F(\bar{x},\bar{s}) \equiv \gamma(\bar{x},\bar{a},\bar{s})]\}\) is used to simplify \( F(\bar{x},\bar{s}) \) at some subhistory \( \sigma' \cdot (A,\bar{\mu}) \). \( \phi(\bar{x},\bar{a},\bar{s}) \) g-regresses to some \( \phi'(\bar{x},\bar{a},\bar{s}) \) at \( \sigma' \).

\( \square \)

Intuitively, a \text{g-regression} is a regression where guards are safely evaluated using the regression procedure as well. As a result, we only ever need to evaluate formulas at \( S_0 \).

**Theorem 5.2.3:** If \( D \) is an acyclic GAT and \( \sigma \) is a JIT-history for \( \phi(\bar{x},\bar{s}) \), then \( \phi(\bar{x},\bar{s}) \) g-regresses to some \( \psi(\bar{x},\bar{s}) \) at \( \sigma \).

**Proof.** By simultaneous induction on the length of \( \sigma \) and the \text{(max) level of the fluents in the} \( \phi(\bar{x},\bar{s}) \) and induction on the structure of \( \phi(\bar{x},\bar{s}) \).

To do that, we define the induction parameter as the function \text{step}(\phi(\bar{x},\bar{s}),\sigma) form a pair formula-history to the natural numbers:

\[
\text{step}(\phi(\bar{x},\bar{s}),\sigma) = \text{level}(\phi(\bar{x},\bar{s})) + \text{length}(\text{end}[\sigma]) \times \mathcal{F}
\]
where \( \text{level}(\phi) \) is the level of the highest fluent mentioned in \( \phi(\vec{x}.s) \), \( \text{length}(s) \) is the number of actions performed in situation \( s \), and \( F \) is the number of fluent names in the theory.

**Base Case:** The base case arises when \( \text{step}(\phi(\vec{x}.s), \sigma) = 0 \). In this case it should be the case that \( \text{end}[] = S_0 \) and all fluents mentioned in \( \phi(\vec{x}.s) \) are at level zero. Now we perform induction on the structure of the formula.

- **Base Case:** \( \phi(\vec{x}.s) = F(\vec{i}.s) \), where \( F \) is a relational fluent of level zero. Two things can now happen: \( F(\vec{i}.s) \) does not simplifies with any GSFA or it does. In the first case, then \( \nu(\vec{x}.s) = F(\vec{i}.s) \) is a g-regression. In the latter case, given that \( F \) is at level zero then the GSFA's guard should mention no fluent and therefore it must g-regress without using any GSFA. Then, point 1-(ii) of the g-regression definition is guaranteed.

- **Induction Step:** If \( \phi(\vec{x}.s) = \phi_1(\vec{x}.s) \land \phi_2(\vec{x}.s) \), then, by induction, \( \phi_1(\vec{x}.s) \) must g-regress to some \( \nu_1(\vec{x}.s) \) and \( \phi_2(\vec{x}.s) \) must g-regress to some \( \nu_2(\vec{x}.s) \). From that it follows that \( \phi(\vec{x}.s) \) g-regresses to \( \nu(\vec{x}.s) = \nu_1(\vec{x}.s) \land \nu_2(\vec{x}.s) \). Notice that \( \text{step}(\phi_1(\vec{x}.s), \sigma) = \text{step}(\phi_2(\vec{x}.s), \sigma) = 0 \) and that \( \sigma \) is a JIT-history for both \( \phi_1(\vec{x}.s) \) and \( \phi_2(\vec{x}.s) \).

**Induction Step:** Assume \( \text{step}(\phi(\vec{x}.s), \sigma) = k + 1 \). The induction hypothesis is that for any formula \( \phi'(\vec{x}.s) \) and any history \( \sigma' \), which is JIT for that formula, such that \( \text{step}(\phi(\vec{x}.s), \sigma) \leq k \), \( \phi'(\vec{x}.s) \) g-regresses to some \( \nu'(\vec{x}.s) \) at \( \sigma' \).

Again, we perform again induction on the structure of the formula.

- **Base Case:** \( \phi(\vec{x}.s) = F(\vec{i}.s) \), where \( F \) is a fluent of level \( k + 1 \). If \( \sigma \) is an initial history, then given that it is a JIT-history for \( F(\vec{i}.s) \) one of the following must apply:

1. \( D_0 \models \forall F(\vec{i}.S_0) \) or \( D_0 \models \forall \neg F(\vec{i}.S_0) \) holds. In such case, since a formula always regresses to some formula at an initial history (either if it fully simplifies to
itself or to another formula), then \( F(\overline{t}, s) \) g-regresses to that formula as well. Point 1-(ii) of the g-regression is the satisfied.

2. there exists a GSFA \( \{ J(\overline{z}, s) \supset F(\overline{z}, s) \equiv \rho(\overline{z}, s) \} \) such that \( \sigma \) is JIT for \( J(\overline{t}, s) \) and \( D \cup Sensed[\sigma] \models \forall J(\overline{t}, s) \). Next, \( \text{level}(J(\overline{t}, s)) < \text{level}(F(\overline{z}, s)) = k + 1 \) because \( D \) is acyclic. from which we know that \( \text{step}(J(\overline{t}, s), \sigma) \leq k \) must hold. Then, by induction, \( J(\overline{t}, s) \) g-regresses to some formula \( J'(\overline{t}, s) \). Moreover, any fluent in \( J(\overline{t}, s) \) has lower level than \( F \), which implies that every guard of any GSFA for them should not mention \( F \) at all. Then, in any simplification of those fluents, no GSFA of \( F \) can be used. Besides, the only way to regress those fluents is via simplifications (roll back is not used in the initial history).

Then, we conclude that \( J(\overline{t}, s) \) should g-regress at \( \sigma \) without using any GSFA present in the set \( GSFA(F) \). Finally, \( F(\overline{t}, s) \) g-regresses to \( \rho(\overline{t}, s) \setminus \sigma \).

Next, we consider the cases where \( \sigma = \sigma' - (A, \overline{\mu}) \) is not an initial history. Given that \( \sigma \) is JIT for \( F(\overline{t}, s) \) then either:

1. there exists a GSFA \( \{ J(\overline{z}, s) \supset F(\overline{z}, s) \equiv \rho(\overline{z}, s) \} \) such that \( \sigma \) is JIT for \( J(\overline{t}, s) \) and \( D \cup Sensed[\sigma] \models \forall J(\overline{t}, s) \). Again, \( \text{level}(J(\overline{t}, s)) < \text{level}(F(\overline{z}, s)) = k + 1 \) because \( D \) is acyclic. from which we derive that \( \text{step}(J(\overline{t}, s), \sigma) \leq k \) must hold. Then, by induction, \( J(\overline{t}, s) \) g-regresses to some formula \( J'(\overline{t}, s) \). For the same reason as point (2) above, \( J(\overline{t}, s) \) must, in fact, g-regress at \( \sigma \) without using any GSFA present in the set \( GSFA(F) \). Moreover, \( F(\overline{t}, s) \) simplifies. fully simplifies and rolls back to \( \nu'(\overline{z}, s) = \rho(\overline{t}, s) \setminus \sigma \) at \( \sigma \). It is clear to see that \( \text{step}(\nu'(\overline{t}, s), \sigma') < \text{step}(F(\overline{t}, s), \sigma) \) since \( \text{length}(\sigma') = \text{length}(\sigma) - 1 \) and \( \nu'(\overline{z}, s) \) contains no fluent. By induction, given that \( \sigma' \) is JIT for the sensor formula \( \nu'(\overline{z}, s) \), the latter g-regresses to some \( \nu'(\overline{t}, s) \) at \( \sigma' \). Hence, \( F(\overline{t}, s) \) g-regresses to \( \nu'(\overline{t}, s) \) at \( \sigma \).

2. there exists a GSFA \( \{ A(\overline{z}, a, s) \supset F(\overline{z}, d(a, s)) \equiv \gamma(\overline{z}, a, s) \} \) such that \( \sigma \) is JIT
for both $\alpha(\vec{I} \cdot A \cdot s)$ and $\gamma(\vec{Z} \cdot a \cdot s)$. Besides, $D \cup Sensed[\sigma] \models \forall \alpha(\vec{I} \cdot A \cdot s)$. In order to apply the induction hypothesis, we have to show that $step(\alpha(\vec{I} \cdot A \cdot s), \sigma') \leq k$ and $step(\gamma(\vec{I} \cdot A \cdot s), \sigma') \leq k$.

First, given that the highest level in the ordering of fluents can be $F - 1$.

$$level(\alpha(\vec{I} \cdot A \cdot s)) + length(end[\sigma']) \times F \leq F - 1 + length(end[\sigma']) \times F$$

Next, because $length(end[\sigma]) = length(end[\sigma']) + 1$ it follows easily that

$$F - 1 + length(end[\sigma']) \times F = F - 1 + length(end[\sigma]) \times F$$

The same argument for $\gamma(\vec{I} \cdot A \cdot s)$ applies so that it must g-regress to some $\gamma'(\vec{I} \cdot A \cdot s)$ at $\sigma'$. What is more, $F(\vec{I} \cdot s)$ rolls back to $\gamma(\vec{I} \cdot A \cdot s)$ from $\sigma$ so that it has to g-regress to $\nu'(\vec{Z} \cdot s) = \gamma'(\vec{I} \cdot A \cdot s)$ from $\sigma$.

**Induction Step:** If $\phi(\vec{Z} \cdot s) = \phi_1(\vec{Z} \cdot s) \land \phi_2(\vec{Z} \cdot s)$, then by induction, $\phi_1(\vec{Z} \cdot s)$ must g-regress to some $\nu_1'(\vec{Z} \cdot s)$ and $\phi_2(\vec{Z} \cdot s)$ must g-regress to some $\nu_2'(\vec{Z} \cdot s)$. From that it follows that $\phi(\vec{Z} \cdot s)$ g-regresses to $\nu(\vec{Z} \cdot s) = \nu_1(\vec{Z} \cdot s) \land \nu_2(\vec{Z} \cdot s)$. Notice that $step(\phi(\vec{Z} \cdot s), \sigma) = max\{step(\phi_1(\vec{Z} \cdot s), \sigma), step(\phi_2(\vec{Z} \cdot s), \sigma)\}$ and that $\sigma$ is a JIT-history for both $\phi_1(\vec{Z} \cdot s)$ and $\phi_2(\vec{Z} \cdot s)$. For all the other logical constructs the arguments are similar.
A severe drawback of generalized regression comes from the fact that each fluent is only considered "locally". For instance, suppose \( \phi(s) = \exists x. x = john \land busy(x, s) \). If there is a GSSA \( \{ \alpha(x, a, s) \supset busy(x, do(a, s)) \equiv \gamma(x, a, s) \} \), then for regressing \( \phi(s) \) at any history it is required \( \alpha(\bar{a}, a, s) \) to be universally entailed on \( x \). Nevertheless, the formula \( \phi'(s) = busy(john, s) \) would just require \( \alpha(john, a, s) \) to hold!

Although this might not be a big problem now, it will be as soon as we wish to adapt the generalized regression to work with functional fluents. To see why, suppose \( \phi(s) = busy(roommate(john, s), s) \). If we use the regression suggested in [33] as inspiration, we would first regress the functional fluent \( roommate(john, s) \) to obtain a formula like \( \phi'(s) = \exists v. v \land busy(v, s) \) where we can now regress the fluent \( busy(v, s) \). Unfortunately, the argument of \( busy \) is now a variable, which means that we will have the problem mentioned above. It seems, then, that accommodating functional fluents in a suitable way is not trivial.

5.3 An Incremental Interpreter for Bounded Programs

As we said, we will restrict our attention to programs that are bounded in the sense that they never evaluate open formulas. To do so, we limit the variables of the nondeterministic choice construct \( \pi x. \delta(x) \) and the existential quantification \( \exists x. \alpha(x) \) to range over a finite domain or sort.

Thus, an IndiGolog bounded program is a ConGolog program except for the following three points:

1. A closure domain axiom is associated to each variable \( v \) used in a non-deterministic
choice of argument. Therefore.

\[
Trans(\pi v. \delta, \delta, s, s') \equiv \exists x. (x = d_1 \lor \ldots \lor x = d_n) \land Trans(\delta^v_x, s, s')
\]

\[
Final(\pi v. \delta, s) \equiv \exists x. (x = d_1 \lor \ldots \lor x = d_n) \land Final(\delta^v_x, s)
\]

Here \(\delta^v_x\) is the program resulting from substituting \(v\) with the variable \(x\) which has a closed domain given by the constants \(d_1, d_2, \ldots, d_n\). DTGolog [5], a recent extension of Golog to integrate it with decision-theoretic planning, uses a similar construct \textit{pickBest} which specifies the elements that a variable can take.

2. Each variable \(x\) existentially quantified in any formula \(\phi\) is closed over some domain \(d_1, d_2, \ldots, d_n\) so that \(\exists x. \phi(x, s) \equiv (x = d_1 \lor \ldots \lor x = d_n) \land \phi(x, s)\)

3. \(\delta = \Sigma \delta_1\), where \(\delta_1\) does not mention \(\Sigma\) is a legal program for which \(Trans\) and \(Final\) definitions have already been defined in Chapter 4. Observe that nesting search operators is equivalent to applying the search operator only once ([10]).

### 5.3.1 Main loop

The top part of the interpreter deals with the execution of actions in the world and exogenous events. In addition, it makes use of \(Trans\) and \(Final\) to determine the next action to perform and to end the execution.

\[
\begin{align*}
\text{indigo}(E, H) & : = \text{exog\_occurs}(\text{Act}), !, \text{indigo}(E, [\text{Act} | H]). \\
\text{indigo}(E, H) & : = \text{final}(E, H). \\
\text{indigo}(E, H) & : = \text{trans}(E, H, E_1, H), !, \text{indigo}(E_1, H). \\
\text{indigo}(E, H) & : = \text{trans}(E, H, E_1, [A | H]), \text{execute}(A), !, \text{indigo}(E_1, [A | H]).
\end{align*}
\]

The first clause allows for the occurrence of an exogenous action at each step. If there is no exogenous action, then either we find a legal transition, commit to it, and continue with the new history, or we terminate successfully.

Committing means to execute a new action in the world if the transition requires it. Recall condition checking causes a transition without adding any new action to the
history. The call to \texttt{execute(Act)} should execute the action \texttt{Act} in the real world and inspect all sensors. How sensors can be treated will be explained below.

### 5.3.2 Implementation of \texttt{Trans} and \texttt{Final}

Clauses for \texttt{Trans} and \texttt{Final} are needed for each of the program constructs. For simplicity, we do not handle procedures here, but they can be accommodated in the same way as [9] does. Since many language constructs involve conditions, we also need a procedure to evaluate them. In previous implementations, a procedure \texttt{holds(C,S)} was used to test if condition \texttt{C} held in situation \texttt{S}. Those implementations relied on a CWA over the formulas being evaluated. Here, we do not want to rely on a strict CWA but still have sound execution of programs. To achieve this, we use a stronger evaluation procedure \texttt{evalsec/3} explained in section 5.3.3 for both \texttt{Trans} and \texttt{Final}.

Apart from describing the underlying theory of action behind, which we will explain in section 5.3.3, the programmer should specify the following:

- \texttt{prim.action(a)} for each primitive action \texttt{a}.
- \texttt{poss(a,c)} for each precondition axiom.
- \texttt{var.sort(var.sort)} to assert that variable \texttt{var} ranges over the sets of elements \texttt{sort}.
- \texttt{sort(d)} to assert that \texttt{d} is an element of \texttt{sort}.

As discussed below, to prevent unsound transitions, \texttt{evalsec/3} aborts to the top level whenever a test required from \texttt{Trans} and \texttt{Final} cannot be evaluated.

Prolog terms representing IndiGolog programs are as follows:

- \texttt{nil}. empty program.
- \texttt{a}. atomic action. where \texttt{a} is an action term.
- \texttt{(c)}?. test. where \texttt{c} is a condition of the form explained below.
- \texttt{[d1,d2]}. sequence.
- \texttt{nndet(d1,d2)}. nondeterministic branch.
• \texttt{star}(\delta). nondeterministic iteration.
• \texttt{pi}(v, \delta). nondeterministic choice of argument. where \(v\) is a Prolog constant and \(\delta\) is a program that uses \(v\).
• \texttt{if}(\phi, \delta_1, \delta_2). if-then-else. with \(\delta_1\) the then-branch and \(\delta_2\) the else-branch.
• \texttt{while}(\phi, \delta). while-do.
• \texttt{conc}(\delta_1, \delta_2). concurrency.
• \texttt{pconc}(\delta_1, \delta_2). prioritized concurrency.
• \texttt{iconc}(\delta_1, \delta_2). iterated concurrency.
• \texttt{search}(\delta). search over \(\delta\).

The clauses for the typical ConGolog and Golog constructs are the following:

/* (a) - CONGOLOG */
final(conc(El,E2),H) :- final(El,H), final(E2,H).
final(pconc(El,E2),H) :- final(El,H), final(E2,H).
final(iconc(E),H).

trans(conc(El,E2),H,conc(EE,E2),H1) :- trans(E1,H,EE,H1).
trans(conc(El,E2),H,conc(EE,E2),H1) :- trans(E2,H,EE,H1).

trans(pconc(El,E2),H,pconc(EE,E2),H1) :- trans(E1,H,EE,H1).
trans(pconc(El,E2),H,pconc(EE,E2),H1) :- not trans(E1,H,_,_),
trans(E2,H,EE,H1).

trans(iconc(E),H,conc(E,iconc(E)),H1) :- trans(E,H,E1,H1).

/* (b) - GOLOG */
final([],_).
final([E|L],H) :- final(E,H), final(L,H).

final(ndet(El,E2),H) :- final(El,H) ; final(E2,H).
final(if(P,E1,E2),H) :- evalsec(P,H,Bool_P),
                    ( ( Bool_P=true, final(E1,H)) ;
                    ( Bool_P=false, final(E2,H)) ).
final(star(E),H).
final(while(P,E),H) :- evalsec(P,H,Bool_P), (Bool_P=false ; final(E,H)).

final(pi(V,E),H) :- varsor(V,D), call(D(0)), subv(V,0,E,E2), final(E2,H).
trans([E|L],H,[E|L],H1) :- trans(E,H,E1,H1).
trans([E|L],H,E1,H1) :- \(+ L=[]\), final(E,H), trans(L,H,E1,H1).
CHAPTER 5. AN INCREMENTAL INTERPRETER OVER GUARDED THEORIES

\text{trans}(\text{if}(P,E1,E2),H,E,H1) :- \text{evalsec}(P,H,\text{true}), \\
(\text{Bool}=\text{true}, \text{trans}(E1,H,E,H1)) ; (\text{Bool}=\text{false}, \text{trans}(E2,H,E,H1))).

\text{trans}(\text{while}(P,E),H,[E1,\text{while}(P,E)],H1) :- \text{evalsec}(P,H,\text{true}), \\
\text{trans}(E,H,E1,H1).

\text{trans}(\text{pi}(V,E),H,E1,H1) :- \text{varsort}(V,D), \text{call}(D(O)), \\
\text{subv}(V,O,E,E2), \text{trans}(E2,H,E1,H1).

\text{trans}(E,H,[\emptyset],[E|H]) :- \text{prim}\_\text{action}(E), \text{poss}(E,P), \text{evalsec}(P,H,\text{true}).

Notice that the implementation of \text{if}) relies on the finite failure of \(\delta_1\) to try to execute \(\delta_2\). This means that \text{the finite failure of } \delta_1 \text{ has to imply the non-existence of a transition for } \delta_1.

To handle \text{pi}(V,E) the interpreter first retrieves \(V\)'s sort by calling \text{varsort}(V,D) and then it retrieves each element of the domain of \(V\) by calling \text{call}(D(O)) where the predicate \(D\) is also specified by the programmer as one or more clauses \(D/1\).

For instance, if we have the program \text{pi}(p,\text{neg}(\text{busy}(p))) meaning "pick up a person who is not busy". we may have the following two clauses to state the \(p\)'s range:

\text{varsort}(p,person) \\
person(X):= X=\text{john} ; X=\text{peter} ; X=\text{mark}.

Search Implementation

The only tricky part of \text{Trans} and \text{Final} has to do with the search operator. Because of the potential presence of sensing information inside a local lookahead, it would be impractical to have an implementation that coincides exactly with the formal definition. taking into account any possible future outcome of the sensors. What is more, it would also be impractical to deal with all possible external actions (exogenous or resulting from other concurrent programs) when doing a local search. Nevertheless, we can obtain a simple, but we think useful version of the search operator:
5.3.3 Evaluating test conditions

In [11], an evaluation procedure for test conditions was given for guarded theories. However, it needs to be modified here in the presence of prioritized concurrency to abort when
a formula cannot be evaluated. If that is the case, whenever a goal $Trans$ finitely fails we can safely conclude that there is no logical transition implied.

A big issue now is how to deal with sensor outcomes. For that, we assume we have a database of all sensor results up to the actual history. In Prolog, this database consists of clauses $\text{has.sense}(h_i,v_i,s)$ where $v_i$ is the value of the sensor function $h_i$ in history $\sigma$ ($s = \text{end}[$$\sigma$]$)$, i.e. $D \cup \text{Sensed}[$$\sigma$]$ \models h_i(s) = v_i$. Given that we are simulating a real execution, our implementation adds these clauses on demand. In practice, we would use an efficient data structure storing only changes to sensor values. In addition, both the sensor database and the initial database should be progressed after some number of actions as suggested in [15].

In order to specify the guarded action theory that will model the dynamics of the world, we assume the user provides the following clauses:

- $\text{fluent}(F,\text{rel})$. for each relational fluent $F$.
- $\text{sensor}(h_i)$. for each sensor function $h_i$.
- $\text{init}(F)$. for each fluent $f$ such that $D_0 \models F(S_0)$.
- $\text{closed}(F)$, whenever $D_0 \models F(S_0)$ or $D_0 \models \neg F(S_0)$.
- $\text{gsfa}(F(\vec{X}),\mathcal{A}(\vec{X}),\rho(\vec{X}))$. for each GSFA.
- $\text{gsfa}(A,F(\vec{X}),\alpha(\vec{X},A),\nu(\vec{X},A))$. for each GSSA.$^2$

A condition $\circ$ is either a Prolog-term representing an atomic formula/fluent/sensor or an expression of the form $\text{and}(\circ_1,\circ_2)$, $\text{or}(\circ_1,\circ_2)$, $\text{neg}(\circ_1)$, $\text{some}(\nu,\circ_1)$ with the obvious intended meaning. In a condition, all situation arguments in fluents and sensors are deleted and they will be restored with the actual situation through the program execution. Thus, we define $\text{evalsec}/3$ as follows:

$^2$In the implementation, we restrict $\alpha(\vec{X},A)$ to be a fluent-formula.
/* Secure Evaluation of P in situation H */
evalsec(P, H, B) :- eva1(P, H, B1) -> B1 = B ; (write('Formula Unknown!'), abort).

eval(and(P1, P2), H, Bool) :- eval(P1, H, Bool_P1),
   (Bool_P1=false -> Bool=false ; eval(P2, H, Bool)).
eval(or(P1, P2), H, Bool) :- eval(P1, H, Bool_P1),
   (Bool_P1=true -> Bool=true ; eval(P2, H, Bool)).

eval(some(V, P), H, False) :- varsorp(V, D), eval(some(V, D, P), H, Bool).
eval(some(V, D, P), H, true) :- D(O), subv(V, O, P, Pl), eval(Pl, H, true).
eval(some(V, D, P), H, false) :- (not D(O), subv(V, O, P, Pl),
   not eval(Pl, H, false)).

eval(neg(P), H, Bool) :- eval(P, H, Bool_P),
   (Bool_P=true, Bool=false) ; (Bool_P=false, Bool=true)).

% P is a structure with some construct (=,>,<,is)
eval(P, H, Bool) :- simplec(P), subf(Pl, P1, H),
   (Bool=true, call(P1)) ; (Bool=false, not call(P1)).

simplec(P) :- \+ (P = and(P1, P2) ; P = or(P1, P2) ; P = neg(P1) ;
   P = some(V, P1) ; P = some(V, D, P1)).

/* T2 is T1 with X1 replaced by X2 */
subv(X1, X2, T1, T2) :- (var(T1); integer(T1)), !, T2 = T1.
subv(X1, X2, T1, T2) :- T1 =..[F|L1], subv1(X1, X2, L1, L2), T2 =..[F|L2].
subv1(X1, X2, [T1|L1], [T2|L2]) :- subv(X1, X2, T1, T2), subv1(X1, X2, L1, L2).

/* P2 is P1 with fluents replaced by their values */
subf(P1, P2, H) :- has_sensor(P1), !,
   (has_sensor(P1, P2, H) -> true ; ask_sensor(P1, P2, H)).
subf(P1, P2, H) :- fluent(P1, rel), !, bool(P1, P2, H).
subf(P1, P2, H) :- atomic(P1), !, P2 = P1.
subf(P1, P2, H) :- P1=..[F|L1], subf1(L1, L2, H), P2=..[F|L2].
subf1([], [], H).
subf1([T1|L1], [T2|L2], H) :- subf(T1, T2, H), subf1(L1, L2, H).

/* Relational fluent F has value V in H */
bool(F, true, []) :- init(F).
bool(F, false, []) :- closed(F), not init(F).
bool(F, V, H) :- gsfa(F, G, P), eval(G, H, true), eval(P, H, V).

% Recovers the Value of Sensor in situation H
ask_sensor(Sensor, Value, H) :-
write('Sensor: '), write(Sensor), nl,
write('At: '), write(H), nl,
write('Value: '), read(Value),
(Value=nil -> fail ; assert(has_sensor(Sensor,Value,H))

To clarify this code we point out the following:

- Predicate `evalsec/3` calls `eval/3` to determine the truth value of a formula, aborting if it cannot do so. `Trans` and `Final` implementations make use of `evalsec/3` and not `eval/3`.

- Predicate `eval/3` first recursively reduces `and`, `neg`, and `some` formulas to the atomic cases, and then, it uses `subf/3` to replace any fluent or sensor function in an atomic formula by its current value and leaves the rest unchanged. Recall that we are assuming that existential quantification is restricted to finite domains. We discuss this limitation below.

- `bool/3` determines the truth value of a relational fluent in any history. For the initial situation, `bool/3` tries to use the user-provided value; otherwise, it attempts to find a GSFA or GSSA whose guard recursively evaluates to true.

- `subf/3` replaces each sensing function or fluent for its value in a situation. An atomic formula may be a single relational fluent such as `busy(john)`, or an atomic expression with one or more sensor functions such as `temperature > 30` or `sonar1 = sonar2`. This clause provides a flexible way of extending the evaluation to accommodate functional fluents by adding the following code:

```
subf(P1,P2,H):- prim_fluent(P1,fun), val(P1,P2,H).
```

where `val(P1,P2,H)` denotes that `P2` is the value of the functional fluent `P1` in situation `H`. 
Observe that when a sensor outcome in a situation is not in the database, i.e. has_sensor/3 finitely fails, we ask the value to the user. However, if the user enters nil it means that the sensor outcome is not available yet as it happens inside a search. This treatment on demand of sensor outcomes is due to the fact that at each step, most of the sensor readings are not meaningful given that the related GSFA are not applicable.

5.4 Correctness of the interpreter

To finish, we show soundness and completeness results for the interpreter. Our proofs rely on the following assumptions:

- The domain theory $D$ enforces the unique name assumption (UNA) on both actions and objects.
- The predicate subv/4 correctly implements the substitution for both programs and formulas.
- We work with a proper Prolog interpreter. In particular, it flounders on goals of the form not trans($\delta, h, \ldots$) with non-ground $\delta$ and $h$.

Some results in this section make use of the definition of a GAT $D^s$ defined in Chapter 4 (section 4.1.1). Recall that - informally - $D^s$ is the same theory as $D$ except that sensors are not used further than situation $s$.

5.4.1 Correctness of the evaluation procedure

Observe that evalsec(o.s.b) can either succeed returning true, succeed returning false, fail with error, or not terminate. See also that even though eval/3 can finitely

\[^3\text{This form of floundering arises for example when we expand } \pi \text{ in programs of the form } \pi z.((\delta_1(z)))\delta_2(z)\text{. Notably this cannot happen in bounded programs because } z \text{ is grounded to elements of its domain.} \]
fail. eval/sec/3 cannot. Indeed, a finitely failure of the former implies an error failure of the latter. First, let us prove that eval/sec/3 is always sound. From now on we keep the translation between the formal logic and Prolog implicit.

Theorem 5.4.1: (Soundness of eval/sec/3) Let \( o(s) \) be a sensor-fluent formula with no free variable except the situation argument \( s \). Let \( \sigma \) be a history and assume that eval/sec\((o, \text{end}[] \sigma], B) \) succeeds with computed answer \( B = b \). Hence, if \( b = \text{true} \), then \( D \cup \text{Sensed}[\sigma] \models o(\text{end}[\sigma]) \); if \( b = \text{false} \), then \( \bullet \cup \text{Sensed}[\sigma] \models \neg o(\text{end}[\sigma]) \).

Proof. First of all notice that eval\((o, \text{end}[] \sigma], B) \) must succeed with \( B1 = b \). So let us prove that eval/3 is in fact sound. To do that we perform induction on the number of calls to eval/3 in the finite SLD-tree to the goal. We refer to the ground term \( \text{end}[] \sigma \) as \( h \).

Base Case: Suppose there is just one call to eval/3. This can happen only in two cases:

1. \( o(s) \) is an atomic sensor formula. First, subf\((o(s), P2, h) \) is called which returns with \( P2 = o' \) such that \( o'(s) \) is \( o(s) \) where all sensing functions are replaced by their values at history \( \sigma \). Then, either call\((o') \) or not call\((o') \) must succeed. In the first case, \( B1 = b = \text{true} \). and by Clark's Theorem ([7]) the procedure behaves in a sound way. When \( B1 = b = \text{false} \) the argument is the same.

2. \( o(s) \) is a relational fluent \( F(l, s) \) and \( \sigma \) is an initial history. i.e. \( h = [ ] \). Therefore, if \( b = \text{true} \) then init\((F[l]) \) must succeed; if \( b = \text{false} \) then init\((\neg F[l]) \) must succeed. In both cases, eval/3 is sound.

Induction Step: Suppose eval/3 is sound for every goal needing \( n \) calls to eval/3. Assume now that the goal eval\((o(s), h, B) \) succeeds needing \( n + 1 \) calls to eval/3.

If \( o(s) = o_1(s) \land o_2(s) \), then eval\((o_1(s), h, B1) \) and eval\((o_2(s), h, B2) \) need \( n \) or less calls to eval/3. Therefore, they are both sound and by the clause for conjunction
eval(o(s), h, B) should be sound as well. The cases for disjunction, negation and existential are similar. It remains to prove the atomic cases where o(s) is a fluent or an atomic expression with sensor functions.

If o(s) = F(ł, s), then either the third or the fourth clause for bool/3 succeeds. In the first case, \{ gsfa(F(ł, G, P), eval(G, h, true), eval(P, h, V) \} succeeds with answers \( V = b, G = \beta(ł, s), P = \rho(ł, s) \). This means there is a GSF \( \{ \beta(ł, s) \cup F(ł, s) \equiv \rho(ł, s) \} \) in \( D \). Moreover, both goals eval(3(ł, h, true) and eval(ρ(ł, h, V) succeed \( V = b \) with \( n \) or less calls to eval/3. Therefore, by induction \( D \cup Sens[\sigma] \models 3(ł, end[\sigma]) \). Also if \( V = b = true \) then \( D \cup Sens[\sigma] \models \rho(ł, end[\sigma]) \) and \( D \cup Sens[\sigma] \models F(ł, end[\sigma]) \); if \( V = b = false \) then \( D \cup Sens[\sigma] \models \neg \rho(ł, end[\sigma]) \) and \( D \cup Sens[\sigma] \models \neg F(ł, end[\sigma]) \). In both cases, eval/3 behaves in a sound way.

If the fourth clause of bool/3 is the one that succeeds, then \( h = do(\text{act}. h1) \) for some ground action term act and ground subhistory h1, i.e. \( \sigma = \sigma' \cdot (A, \hat{\mu}) \) and \( h1 = end[\sigma'] \).

Next, \{ gsfa(ł, act, G, P), eval(G, h, true), eval(P, h, V) \} succeeds with computed answers \( V = b, G = \alpha(ł, act, h1, s), P = \gamma(ł, act, h1, s) \). This means that there is a GSSA \( \{ \alpha(ł, a, s) \cup F(ł, do(a, s)) \equiv \gamma(ł, a, s) \} \) in \( D \). Both goals eval(α(ł, act, h1, true) and eval(γ(ł, act, h1, V) succeed \( V = b \) with \( n \) or less calls to eval/3. By induction, \( D \cup Sens[\sigma] \models \alpha(ł, act, end[\sigma']) \). Also, if \( V = b = true \) then it is the case that \( D \cup Sens[\sigma] \models \gamma(ł, act, end[\sigma']) \) and \( D \cup Sens[\sigma] \models F(ł, do(act, end[\sigma'])) \); otherwise if \( V = b = false \) then it is the case that \( D \cup Sens[\sigma] \models \neg \gamma(ł, act, end[\sigma']) \) so that \( D \cup Sens[\sigma] \models \neg F(ł, do(act, end[\sigma'])) \).

In both cases, given that \( end[\sigma] = do(act, end[\sigma']) \), eval/3 behaves in a sound way.

If o(s) is an atomic sensor formula (such as sonar(s) > 10) then \( b = true \) iff call(o') succeeds; or \( b = false \) iff not call(o') succeeds. o'(s) is o(s) where sensors functions are replaced by their values at \( \sigma \). Since \( o'(s) \) is a ground Prolog term, then eval/3 is sound by Clark's Theorem ([7]).
If a formula is unknown, the evaluation has to neither succeed nor finitely fail. This is one of the requirements needed for obtaining a sound implementation of Trans and Final.

Corollary 5.4.1: Let $\sigma$ be a history and $\phi(s)$ a sensor-fluent formula with no free variables except the situation argument $s$. If neither $D \cup Sensed[\sigma] \not\models \phi(\text{end}[\sigma])$ nor $D \cup Sensed[\sigma] \not\models \neg \phi(\text{end}[\sigma])$, then $\text{evalsec}(\phi.\text{end}[\sigma], B)$ neither succeeds nor it finitely fails.

PROOF. By contradiction, assume that it succeeds with $B = b$. If $b = true$, then since $\text{evalsec}/3$ is sound, $D \cup Sensed[\sigma] \models \phi(\text{end}[\sigma])$ and that is an absurd. If $b = false$, then since $\text{evalsec}/3$ is sound, $D \cup Sensed[\sigma] \models \neg \phi(\text{end}[\sigma])$ and we get a contradiction as well. Therefore, $\text{evalsec}(\phi.\text{end}[\sigma], B)$ cannot succeed. As a result, $\text{evalsec}(\phi.\text{end}[\sigma], B)$ either does not terminate (whenever $\text{eval}(\phi.\text{end}[\sigma], B1)$ does not terminate), or aborts with error (whenever $\text{eval}(\phi.\text{end}[\sigma], B1)$ finitely fails).

For JIT-histories, either the procedure returns with some truth value or does not terminate. This resembles in some way Theorem 5.2.2 above.

Theorem 5.4.2: Let $\sigma$ be a JIT-history for $\phi(s)$. Then $\text{evalsec}(\phi.\text{end}[\sigma], B)$ neither fails with error nor finitely fails.

PROOF. To show that $\text{evalsec}/3$ never fail with error when the history is JIT for the formula is equivalent to show that $\text{eval}/3$ cannot finitely fail in such cases. Thus, we need to prove that $\text{eval}(\phi.\text{end}[\sigma], B1)$ either succeeds or does not terminate.

Assume then that, in fact, $\text{eval}(\phi.\text{end}[\sigma], B1)$ does terminate so that we want to prove that it actually succeeds. We do that by induction on the number of $\text{eval}/3$ calls performed in the proof tree constructed by Prolog. We refer to $\text{end}[\sigma]$ as $h$.

Base Case: Suppose there is just one call to $\text{eval}/3$. This can happen only in two cases:
1. If $o(s)$ is an atomic sensor formula then $\text{eval}(o.h.B1)$ has to succeed with some $B1 = \text{true}/\text{false}$. This is because either $\text{call}(\sigma')$ or not $\text{call}(\sigma')$ has to succeed, where $\sigma'(s)$ is $o(s)$ with sensors functions replaced by their values at $\sigma$. This replacement is performed by $\text{subv}/4$.

2. $o(s)$ is a relational fluent $F(\vec{t}.s)$ and $\sigma$ is an initial history. i.e. $h = \emptyset$. Since $\sigma$ is a JIT-history for $F(\vec{t}.s)$, then either $D \models F(\vec{t}.s)$ or $D \models \neg F(\vec{t}.s)$. In the first case, $\text{init}(F(\vec{t}))$ succeeds. In the second case, $\text{init}(\neg(F(\vec{t})))$ succeeds. In either case, $\text{eval}/3$ succeeds.

**Induction Step:** Suppose that the theorem holds whenever the tree has $n$ or less calls to $\text{eval}/3$. Assume that the goal $\text{eval}(o(s).h.B1)$ needs $n + 1$ calls to $\text{eval}/3$.

If $o(s) = o_1(s) \land o_2(s)$, then $\text{eval}(o_1(s).h.B1)$ and $\text{eval}(o_2(s).h.B2)$ need $n$ or less calls to $\text{eval}/3$. Moreover, both calls terminate because $\text{eval}(o(s).h.B1)$ terminates. i.e. either they succeed or they finitely fail. Because $\sigma$ is a JIT-history for $o(s)$, it is also a JIT-history for $o_1(s)$ and $o_2(s)$. By induction, both goals must succeed and, because of that and the clause for conjunction, $\text{eval}(o(s).h.B1)$ should succeed too.

If $o(s) = F(\vec{t}.s)$, then because $\sigma$ is a JIT-history for $F(\vec{t}.s)$, one of these two cases must apply:

1. There exists a GSFA $\{J(\vec{z}.s) \supset F(\vec{z}.s) \equiv \rho(\vec{z}.s)\}$ such that $\sigma$ is a JIT-history for $J(\vec{t}.s)$. Given that $\text{eval}/3$ does terminate, the third clause of $\text{bool}/3$ should succeed with some boolean answer for $V$. This is because $\text{gsfa}(F(\vec{t}).G.P)$ succeeds with $G = J(\vec{t})$ and $P = \rho(\vec{t})$ at some point. Next, $\text{eval}(J(\vec{t}).h.true)$ performs $n$ or less calls to $\text{eval}/3$ and since $\sigma$ is a JIT-history for $J(\vec{t}.s)$ and $\text{eval}/3$ is always sound (Theorem 5.4.1), such goal should succeed. Also, $\text{eval}(\rho(\vec{z}).h.V)$ must succeed (by induction) given that it does terminate and it needs less than $n$ calls to $\text{eval}/3$. Because of all this, $\text{bool}(F(\vec{t}).V.h)$ succeeds and so does $\text{eval}(F(\vec{t}).h.B1)$. 


2. \( \sigma \) is not an initial one. i.e. \( \sigma = \sigma' \cdot (A, \tilde{\mu}) \), and there exists a GSSA of the form 
\[ \{ \alpha(a.s) \supset F(\tilde{a}, do(a.s)) \equiv \gamma(\tilde{a}.a.s) \} \] such that \( \sigma' \) is a JIT-history for \( \alpha(\tilde{\beta}.A.s) \) and \( \gamma(\tilde{\beta}.A.s) \). Given that \texttt{eval/3} does terminate, the fourth clause of \texttt{bool/3} should succeed with some boolean answer for \( V \). This is because at some point, \texttt{gssa}(A,F(\tilde{\alpha}).G,P) should succeed with \( G = \alpha(\tilde{\beta}.A) \) and \( P = \gamma(\tilde{\beta}.A) \). Next, \texttt{eval}(\alpha(\tilde{\beta}.A),h1.true), where \( h1 = \text{end}[\sigma'] \) performs \( n \) or less calls to \texttt{eval/3} and since \( \sigma' \) is a JIT-history for \( \alpha(\tilde{\beta}.A.s) \) and that \texttt{eval/3} is always sound (Theorem 5.4.1), such goal should succeed. Also, \texttt{eval}(\gamma(\tilde{\beta}.A),h1.V) must succeed as well by the induction hypothesis. Note that such a goal has to terminate since \texttt{eval}(F(\tilde{\alpha}).h,B1) does.

All these proves that \texttt{eval/3} either succeeds or does not terminate whenever the history is a JIT one for the formula being evaluated. This implies immediately that \texttt{eval/sec/3} cannot fail with error.

For acyclic theories (see Chapter 3), the procedure is guarantee to terminate. Recall that non-termination can arise when evaluating guards in GSFAs as we have seen in section 5.2. Notice that in the example shown at the end of such section, it is the case that \( F_1 < F_2 \) and \( F_2 < F_1 \) from which we know that \( < \) is not well-founded. and, hence, the theory is not acyclic. Intuitively, \( F_1 < F_2 \) means that for obtaining the truth value of \( F_2 \) at a history \( \sigma \), we might need the truth value of \( F_1 \) at \( \sigma \). The advantage of acyclic theories is that it is always possible to use the regression mechanism described in section 5.2 to evaluate guards without falling into cycles. As a matter of fact, the regression performed is a g-regression, and hence, the following result resembles the one given in Theorem 5.2.3.

**Theorem 5.4.3:** Let \( \mathcal{D} \) be acyclic. let \( \phi(s) \) be a sensor-fluent formula with no free variable except the situation argument \( s \), and let \( \sigma \) be a history. Then, \texttt{eval}(\phi.end[\sigma],B) does terminate.
Proof. We start by noting that the evaluation of any formula mentioning no fluent (a sensor formula) always terminates. With that, we prove the theorem by induction on the levels of the fluents and the length of the history. To do that, we use the same induction parameter \( \text{step}(\phi(s), \sigma) \) defined in Theorem 5.2.3.

Base Case: The base case arises when \( \text{step}(\phi(s), \sigma) = 0 \). In this case it should be the case that \( \text{end}[\sigma] = S_0 \) and all fluents mentioned in \( \phi(s) \) are at level zero. Now we perform induction on the structure of the formula.

- **Base Case**: \( \phi(s) = F(\tilde{r}.s) \), where \( F \) is a relational fluent of level zero. In this case, only the three first clauses of \( \text{bool/3} \) apply. The first two ones always terminate since they just check for \( \text{init/1} \) in the initial database. The third clause must always terminate as well because there are a finite number of GSFA for fluent \( F \), and the guards of all of them must mention no fluent at all given that \( F \) is at level zero. Thus, for any GSFA applicable to \( F \) the evaluation of their guard and their sensor formula must terminate (they do not mention any fluent). Clearly, the fourth clause of \( \text{bool/3} \) is not applicable because its last argument never unifies with [].

- **Induction Step**: If \( \phi(s) = \phi_1(s) \wedge \phi_2(s) \) then the goal \( \text{eval}(\phi, \text{end}[\sigma]).B \) must terminate given that, by induction, both \( \text{eval}(\phi_1, \text{end}[\sigma]).B1 \) and \( \text{eval}(\phi_2, \text{end}[\sigma]).B2 \) terminate. For all the other logical constructs the arguments are similar.

Induction Step: Assume \( \text{step}(\phi(s), \sigma) = k + 1 \). The induction hypothesis is that for any formula \( \phi'(s) \) and any history \( \sigma' \) such that \( \text{step}(\phi(s), \sigma) \leq k \), \( \text{eval}(\phi', \text{end}[\sigma']).B \) terminates.

Again, we perform again induction on the structure of the formula.

- **Base Case**: This is when \( \phi(s) = F(\tilde{r}.s) \), where \( F \) is a relational fluent of level \( k + 1 \). In this case, any of the four clauses of \( \text{bool/3} \) may apply. The first two ones

\footnote{We are assuming that all sensor outcomes are available up to \( \sigma \).}
always terminate since they just check for \texttt{init/1} in the initial database. The third clause must always terminate as well because there are a finite number of GSFA for fluent \( F \) and the guards of all of them must have a level less or equal than \( k \). In other words, if \( \beta(\bar{x}, s) \) is the guard of some GSFA for \( F \) it must be the case that \( \text{level}(\beta(\bar{x}, s)) < \text{level}(F(\bar{x}, s)) = k + 1 \) (because \( \mathcal{D} \) is acyclic). Since the history remains the same, \( \text{step}(\beta(\bar{t}, s), \sigma) \leq k \) and by induction, \( \text{eval}(\beta(\bar{t}, \text{end}[\sigma], \text{true}) \) should terminate. Moreover, given that the evaluation of a sensor formula always terminate, it follows that the third clause of \texttt{bool/3} can never cycle forever.

Next, we need to prove that the fourth clause of \texttt{bool/3} should terminate as well. This is true because of the following three reasons: (i) the number of GSSA is finite; (ii) if \( \{ \alpha(\bar{x}, a.s) \supset \gamma(\bar{x}, a.s) \} \) is a GSSA for fluent \( F \), and \( \sigma = \sigma' \cdot (A, \bar{\mu}) \) for some action \( A \) and reading of sensors \( \bar{\mu} \), then it must be the case that \( \text{step}(\alpha(\bar{t}, A.s), \sigma') \leq k \) and \( \text{step}(\gamma(\bar{t}, A.s), \sigma') \leq k \). If this is actually true, then the evaluation of both \( \alpha(\bar{t}, A.s) \) and \( \gamma(\bar{t}, A.s) \) at history \( \sigma' \) should terminate because of the induction hypothesis, which - in turn - implies immediately that the fourth clause of \texttt{bool/3} should finish for such GSSA.

So, let us prove the above inequation. First, given that the highest level in the ordering of fluents can be \( \mathcal{F} - 1 \), we conclude that

\[
\text{level}(\alpha(\bar{t}, A.s)) + \text{length}(\text{end}[\sigma']) \times \mathcal{F} \leq \mathcal{F} - 1 + \text{length}(\text{end}[\sigma']) \times \mathcal{F}
\]

Next, because \( \text{length}(\text{end}[\sigma]) = \text{length}(\text{end}[\sigma']) + 1 \) it follows easily that

\[
\mathcal{F} - 1 + \text{length}(\text{end}[\sigma']) \times \mathcal{F} = \mathcal{F} - 1 + \text{length}(\text{end}[\sigma]) \times \mathcal{F} - \mathcal{F}
\]

\[
\mathcal{F} - 1 + \text{length}(\text{end}[\sigma]) \times \mathcal{F} - \mathcal{F} = \text{length}(\text{end}[\sigma]) \times \mathcal{F} - 1
\]

Next, given that \( \text{level}(F) \geq 0 \)

\[
\text{length}(\text{end}[\sigma]) \times \mathcal{F} - 1 \leq \text{level}(F) + \text{length}(\text{end}[\sigma]) \times \mathcal{F} - 1
\]
and because \( \text{level}(F) + \text{length}(\text{end}[\sigma]) \times F = \text{step}(F, \sigma) = k + 1 \) we finally arrive to the fact that 

\[
\text{level}(\alpha(\tilde{I}, A, s)) + \text{length}(\text{end}[\sigma']) \times F \leq k + 1 - 1 = k
\]

By induction hypothesis \( \text{eval}(\alpha(\tilde{I}, A, \sigma'), B) \) should terminate. The case for \( \gamma(\tilde{I}, A, s) \) is identical.

- **Induction Step:** If \( \phi(s) = \phi_1(s) \land \phi_2(s) \) then the goal \( \text{eval}(\phi, \text{end}[\sigma], B) \) must terminate given that, by induction, both \( \text{eval}(\phi_1, \text{end}[\sigma], B_1) \) and \( \text{eval}(\phi_2, \text{end}[\sigma], B_2) \) terminate. Observe \( \text{step}(\phi_1(s), \sigma) \leq \text{step}(\phi(s), \sigma) \) and \( \text{step}(\phi_2(s), \sigma) \leq \text{step}(\phi(s), \sigma) \) are true given the fact that \( \text{step}(\phi(s), \sigma) = \max\{\text{step}(\phi_1(s), \sigma), \text{step}(\phi(s), \sigma)\} \).

For all the other logical constructs the arguments are similar.

\[\square\]

If the history contains sufficient information to evaluate a formula and the background theory avoids cycles in GSFA's then our evaluation procedure is complete.

**Corollary 5.4.2:** (Completeness of eval/3) Let \( \mathcal{D} \) be acyclic, let \( \phi(s) \) be a sensor-fluent formula with no free variable except the situation argument \( s \), and let \( \sigma \) be a JIT-history for \( \phi(s) \). Then, the goal \( \text{eval}(\phi, \text{end}[\sigma], B) \) succeeds with \( B = \text{true} \) or \( B = \text{false} \).

**Proof.** First, since \( \mathcal{D} \) is acyclic, the goal \( \text{eval}(\phi, \text{end}[\sigma], B) \) has to terminate. Moreover, since \( \sigma \) is a JIT-history for \( \phi(s) \) the same goal is guarantee not to finally fail. Therefore, \( \text{eval}(\phi, \text{end}[\sigma], B) \) should succeed with \( B = \text{true} \) or \( B = \text{false} \) because of the way \( \text{eval}/3 \) is constructed for each construct. From that, it is easy to see that \( \text{eval}(\phi, \text{end}[\sigma], B) \) succeeds with \( B = \text{true} \) or \( B = \text{false} \).

\[\square\]

Up to now, we have assumed that when computing the projection task of a formula \( \phi(s) \) at a history \( \sigma \), all the information of \( \sigma \) is available, including all sensing information
gathered up to the history in question. However, what would happen if we try to evaluate a formula in a history \( \sigma \) but sensor's results are available only up to some subhistory \( \sigma' \) of \( \sigma \)?

**Observation 5.4.1:** If a goal \( \text{evalsec}(o, \text{end}[\sigma], B) \) succeeds and sensor results are available only up to history \( \sigma' \) where \( \sigma' \) is a subhistory of \( \sigma \), then \( \text{evalsec}/3 \) is sound wrt the theory \( D^{\text{end}[\sigma']} \cup \text{Sensed}[\sigma'] \).

**Proof.** The fact that the database only has the sensing results up to history \( \sigma' \) causes that any attempt to evaluate a fluent using a GSFA further than \( \sigma' \) would fail. In other words, if there is a goal \( \text{bool}(F, V, H) \) where \( H \) represents a situation further than \( \text{end}[\sigma'] \), then the third clause of \( \text{bool}/3 \) can never succeed because the evaluation of the sensor-formula would fail.

Intuitively, this means that GSFA are never considered in situations after \( \text{end}[\sigma'] \), which means that we are actually working with the background theory \( D^{\text{end}[\sigma']} \) instead of \( D \) (see Chapter 3 - Definition 4.4.1).

### 5.4.2 Correctness of trans/4 and final/2

In this subsection we prove that the implementation for \( \text{Trans} \) and \( \text{Final} \) we have proposed for bounded programs are always sound and under suitable circumstances “complete”.

**Soundness Results**

We start by proving the soundness of programs that do not use the search operator. Observe that we assume that there may be only a partial availability of sensors readings: we may be at history \( \sigma \), but we only have the readings of sensors up to some subhistory of \( \sigma \). This is relevant for proving later the meaning of the search implementation.
Theorem 5.4.4: (Soundness without search) Let $\sigma$ be a bounded program not mentioning search and without free variables. Let $\sigma$ be a history, and let $P$ and $H$ be Prolog variables. Assume sensors are available up to some subhistory $\sigma'$ of $\sigma$.

1. If $\text{trans}(\delta, \text{end}[^{\sigma}], P, H)$ succeeds with $P = \delta', H = s'$, then

$$D^{\text{end}[\sigma']} \cup C \cup Sensed[^{\sigma'}] \models \text{Trans}(\delta, \text{end}[^{\sigma}], \delta', s')$$

Furthermore, $\delta'$ and $s'$ do not contain any free variable.

2. If $\text{trans}(\delta, \text{end}[^{\sigma}], P, H)$ finitely fails, then

$$D^{\text{end}[\sigma']} \cup C \cup Sensed[^{\sigma']} \models \forall \delta', s'. \neg \text{Trans}(\delta, \text{end}[^{\sigma}], \delta', s')$$

3. If $\text{final}(\delta, \text{end}[^{\sigma}])$ succeeds, then $D^{\text{end}[\sigma']} \cup C \cup Sensed[^{\sigma'}] \models \text{Final}(\delta, \text{end}[^{\sigma}])$

4. If $\text{final}(\delta, \text{end}[^{\sigma}])$ finitely fails, then $D^{\text{end}[\sigma']} \cup C \cup Sensed[^{\sigma'}] \models \neg \text{Final}(\delta, \text{end}[^{\sigma}])$

**Proof.** We prove here the cases of primitive actions, condition checking, prioritized concurrency, and non-deterministic choice of argument for $\text{trans}/4$.

(A) First, we show the case of $\delta = A$, where $A$ is a primitive action ground term.

**Success.** If $\text{trans}(A, \text{end}[^{\sigma}], P, H)$ succeeds with $P = \delta', H = s'$ then the goal $\text{eval}(\Pi_A(s), \text{end}[^{\sigma}], B)$, where $\Pi_A(s)$ is the precondition of action $A$ succeeds with $B = \text{true}$ and with sensor available up to history $\sigma'$. Also $\delta' = \text{nil}, s' = \text{do}(\text{end}[^{\sigma}])$. Note that $\delta'$ has no free variable and neither has $s'$ given that $A$ is a ground action term.

By Observation 5.4.1, $D^{\text{end}[\sigma']} \cup Sensed[^{\sigma'}] \models \text{Poss}(A, \text{end}[^{\sigma}])$ and by definition of $\text{Trans}$ for condition checking: $D^{\text{end}[\sigma']} \cup Sensed[^{\sigma'}] \models \text{Trans}(A, \text{end}[^{\sigma}], \text{nil}, \text{do}(A, \text{end}[^{\sigma}]))$

**Failure.** If $\text{trans}(A, \text{end}[^{\sigma}], P, H)$ finitely fails then $\text{eval}(\Pi_A, \text{end}[^{\sigma}], B)$, where $\Pi_A(s)$ is the precondition of action $A$ succeeds with $B = \text{false}$ and with sensor available up to history $\sigma'$. By Observation 5.4.1, $D^{\text{end}[\sigma']} \cup Sensed[^{\sigma'}] \models \neg \text{Poss}(A, \text{end}[^{\sigma}])$ and by definition of $\text{Trans}$: $D^{\text{end}[\sigma']} \cup Sensed[^{\sigma'}] \models \forall \delta', s'. \neg \text{Trans}(A, \text{end}[^{\sigma}], \delta', s')$. It is important to notice that $\text{eval}(\Pi_A, \text{end}[^{\sigma}], B)$ cannot finitely fail.
(B) Suppose the case of \( \delta = \sigma(s) \).

**Success.** If \( \text{trans}(\sigma(s) . \text{end}[\sigma], P, H) \) succeeds with \( P = \delta', H = s' \) then the goal \( \text{evalsec}(\sigma(s), \text{end}[\sigma], B) \) succeeds with \( B = \text{true} \) and with sensor available up to history \( \sigma' \). Also \( \delta' = \text{nil}. s' = \text{end}[\sigma] \).

By Observation 5.4.1, \( D^{\text{end}[\sigma']} \cup Sensed[\sigma'] \models \sigma(\text{end}[\sigma]) \) and by definition of Trans for condition checking: \( D^{\text{end}[\sigma']} \cup Sensed[\sigma'] \models \text{trans}(\sigma(s), \text{end}[\sigma], \text{nil}. \text{end}[\sigma]) \). It is trivial that both \( \delta' \) and \( s' \) do not contain any free variable.

**Failure.** If \( \text{trans}(\sigma(s) . \text{end}[\sigma], P, H) \) finitely fails then \( \text{evalsec}(\sigma(s), \text{end}[\sigma], B) \) succeeds with \( B = \text{false} \) and with sensor available up to history \( \sigma' \). By Observation 5.4.1, \( D^{\text{end}[\sigma']} \cup Sensed[\sigma'] \models \neg \sigma(\text{end}[\sigma]) \land \forall \delta', s'. \neg \text{trans}(\sigma(s), \text{end}[\sigma], \delta', s') \) by definition of Trans. It is important to notice that \( \text{evalsec}(\sigma(s), \text{end}[\sigma], B) \) cannot finitely fail.

(C) Next, we show the case of \( \delta = \delta_1 \rangle \delta_2 \) for trans/4.

**Success.** If \( \text{trans}(\delta_1 \rangle \delta_2, \text{end}[\sigma], P, H) \) succeeds with \( P = \delta', H = s' \) then either of the following two things must happen: (i) \( \text{trans}(\delta_1, \text{end}[\sigma], P_1, H) \) succeeds with \( P_1 = \delta', H = s' \) and \( \delta' = \delta_1 \rangle \delta_2 \) with sensors available up to \( \sigma' \); (ii) \( \text{trans}(\delta_1, \text{end}[\sigma], \_\_\_) \) finitely fails and \( \text{trans}(\delta_2, \text{end}[\sigma], P_2, H) \) succeeds with \( P_2 = \delta_2'. H = s' \) and \( \delta' = \delta_1 \rangle \delta_2 \) with sensors available up to \( \sigma' \).

In case (i) by the induction hypothesis \( D^{\text{end}[\sigma']} \cup C \cup Sensed[\sigma'] \models \text{trans}(\delta_1, \text{end}[\sigma], \delta_1', s') \) and \( D^{\text{end}[\sigma']} \cup C \cup Sensed[\sigma'] \models \text{trans}(\delta_1 \rangle \delta_2, \text{end}[\sigma], \delta_1', s') \) by definition of Trans for prioritized concurrency. Also \( \delta' \) does not contain any free variable because \( \delta_1' \) has no free variables (by induction).

In case (ii), \( D^{\text{end}[\sigma']} \cup C \cup Sensed[\sigma'] \models \forall \delta_1', s'. \neg \text{trans}(\delta_1, \text{end}[\sigma], \delta_1', s') \) because of the induction hypothesis, and \( D^{\text{end}[\sigma']} \cup C \cup Sensed[\sigma'] \models \text{trans}(\delta_2, \text{end}[\sigma], \delta_2', s') \). Therefore, \( D^{\text{end}[\sigma']} \cup C \cup Sensed[\sigma'] \models \text{trans}(\delta_1 \rangle \delta_2, \text{end}[\sigma], \delta_1', \delta_2', s') \) follows from the Trans definition for prioritized concurrency. Also \( \delta' \) does not contain any free variable because \( \delta_2' \) has no free variables (by induction).
Failure. If \( \text{trans}(\delta_1)\delta_2.\text{end}[\sigma], P.H) \)    
finely fails then: (i) \( \text{trans}(\delta_1.\text{end}[\sigma], P1.H) \)   
should finitely fail with sensors available up to \( \sigma' \) and (ii) \( \text{trans}(\delta_2.\text{end}[\sigma], P2.H) \)  
should finitely fail with sensors available up to \( \sigma' \). By applying the induction hypothesis on (i)  
and (ii), \( \mathcal{D}^{\text{end}[\sigma]} \cup C \cup \text{Sensed}[\sigma'] \models \forall \delta_1'.s'.\text{trans}(\delta_1.\text{end}[\sigma], \delta_1', s') \) holds and so does \( \mathcal{D}^{\text{end}[\sigma]} \cup C \cup \text{Sensed}[\sigma'] \models \forall \delta_2'.s'.\text{trans}(\delta_2.\text{end}[\sigma], \delta_2', s') \). By definition of \( \text{Trans} \) for  
prioritized concurrency: \( \mathcal{D}^{\text{end}[\sigma]} \cup C \cup \text{Sensed}[\sigma'] \models \forall \delta'.s'.\neg \text{trans}(\delta_1)\delta_2.\text{end}[\sigma], \delta'.s'). \)  

(D) Finally, suppose \( \delta = \pi.v.\delta_1). \)

Success. If \( \text{trans}(\pi.x.\delta_1.\text{end}[\sigma], P.H) \) succeeds with answers \( P = \delta', H = s' \) then \( \text{trans}(\delta_1|_d.\text{end}[\sigma], P1.H1) \) succeed for some constant \( d \) with \( P1 = \delta', H = s' \). By induction, \( \mathcal{D}^{\text{end}[\sigma]} \cup \text{Sensed}[\sigma'] \models \text{trans}(\delta_1|_d.\text{end}[\sigma], \delta', s') \) and both \( \delta' \) and \( s' \) do not have any free variable. It follows then by the definition of \( \text{Trans} \) for non-deterministic choice of argument that \( \mathcal{D}^{\text{end}[\sigma]} \cup \text{Sensed}[\sigma'] \models \text{trans}(\pi.x.\delta_1.\text{end}[\sigma], \delta', s'). \)

Failure. If \( \text{trans}(\pi.x.\delta_1.\text{end}[\sigma], P.H) \) finitely fails then \( \text{trans}(\delta_1|_d.\text{end}[\sigma], P1.H1) \)  
must fail for every constant \( d \) in the domain of variable \( x \). Since \( x \) has a domain closure and by induction hypothesis: \( \mathcal{D}^{\text{end}[\sigma]} \cup \text{Sensed}[\sigma'] \models \forall x.\delta'.s'.\neg \text{trans}(\delta_1(x).\text{end}[\sigma], \delta', s'). \)

Hence, \( \mathcal{D}^{\text{end}[\sigma]} \cup \text{Sensed}[\sigma'] \models \forall \delta'.s'.\neg \text{trans}(\pi.x.\delta_1.\text{end}[\sigma], \delta', s') \) due to the definition of \( \text{Trans} \) for non-deterministic choice of argument. \( \blacksquare \)

Observation 5.4.2: If \( \sigma' = \sigma \) in the above theorem, then we can replace \( \mathcal{D}^{\text{end}[\sigma]} \) by \( \mathcal{D} \).

This is because for evaluating \( \text{Trans} \) and \( \text{Final} \) at history \( \sigma \) we never have to evaluate a condition in a history further than \( \sigma \). In other words, if \( \mathcal{D}^{\text{end}[\sigma]} \cup C \cup \text{Sensed}[\sigma] \models \text{trans}(\delta.\text{end}[\sigma], \delta', s') \) iff \( \mathcal{D} \cup C \cup \text{Sensed}[\sigma] \models \text{trans}(\delta.\text{end}[\sigma], \delta', s'). \)

To state the correctness of \( \text{Trans} \) and \( \text{Final} \) it is important to first understand the behavior of search. As we said, our implementation of search is just an “approximation” of its definition.

Lemma 5.4.1: (Soundness of search) Let \( \Sigma \delta \) be a bounded program with no free variables and assume that sensors are available up to history \( \sigma \). If a goal \( \text{trans}(\Sigma \delta.\text{end}[\sigma], P.S) \)
succeeds with answer $P = \Sigma \delta'. S = s'$. then

$$D \cup C \cup Sensed[\sigma] \models Trans(\delta. end[\sigma], \delta'. s')$$

Furthermore, $\delta'$ and $s'$ do not contain any free variables.

**Proof.**

First, the goal $\text{trans}(\delta. end[\sigma], E1. H1)$ should succeed with $E1 = \delta'$ and $H1 = s'$. By Theorem 5.4.4 and Observation 5.4.2, it follows $D \cup C \cup Sensed[\sigma] \models Trans(\delta. end[\sigma], \delta'. s')$ where $\delta'$ and $s'$ do not contain any free variables.

Also the goal $\text{findpath}(\delta'. s')$ must succeed with sensors available up to $\sigma$. We show now that this implies $D^{end[\sigma]} \cup C \cup Sensed[\sigma] \models \exists s''. \text{Do}(\delta'. s'. s'')$.

We prove this by induction on the length of the sequence of $\text{trans}$/4's and $\text{final}$/2's.

- If the length of the sequence is one, then it is the case that $\text{final}(\delta'. s')$ succeeds with sensors available up to $\sigma$. Thus, $D^{end[\sigma]} \cup C \cup Sensed[\sigma] \models \text{Final}(\delta'. s')$ because of Theorem 5.4.4, which implies that $D^{end[\sigma]} \cup C \cup Sensed[\sigma] \models \exists s''. \text{Do}(\delta'. s'. s'')$.

- Assume the length of the sequence is $n + 1$. Then, $\text{trans}(\delta'. s'. E1. H1)$ succeeds with $E1 = \delta''. H1 = s''$ with sensors available up to history $\sigma$ which means that $D^{end[\sigma]} \cup C \cup Sensed[\sigma] \models Trans(\delta'. s'. \delta''. s'')$ because of Theorem 5.4.4. Moreover, since $\text{findpath}(\delta''. s'')$ succeeds as well in $n$ steps with sensors available up to $\sigma$, by induction, $D^{end[\sigma]} \cup C \cup Sensed[\sigma] \models \exists s''. \text{Do}(\delta''. s''. s''')$.

Theorem 5.4.5: **(Soundness of $Trans$ and $Final$)** Let $\delta$ be a bounded program without free variables. Let $\sigma$ be a history, and let $P$ and $H$ be Prolog variables. Finally, assume sensors are available up to history $\sigma$.

1. If $\text{trans}(\delta. end[\sigma], P. H)$ succeeds with $P = \delta'. H = s'$, then

$$D^{end[\sigma]} \cup C \cup Sensed[\sigma] \models Trans(\delta. end[\sigma], \delta'. s')$$
Furthermore, $\delta'$ and $s'$ do not contain any free variable.

2. If $\text{trans}(\delta.\text{end}[\sigma], P, H)$ finitely fails, then

$$\mathcal{D}^{\text{end}[\sigma]} \cup C \cup \text{Sensed}[\sigma] \models \forall \delta', s'. \neg \text{Trans}(\delta.\text{end}[\sigma], \delta', s')$$

3. If $\text{final}(\delta.\text{end}[\sigma])$, then $\mathcal{D}^{\text{end}[\sigma]} \cup C \cup \text{Sensed}[\sigma] \models \text{Final}(\delta.\text{end}[\sigma])$

4. If $\text{final}(\delta.\text{end}[\sigma])$ finitely fails, then $\mathcal{D}^{\text{end}[\sigma]} \cup C \cup \text{Sensed}[\sigma] \models \neg \text{Final}(\delta.\text{end}[\sigma])$

**Proof.** If $\delta = A|\sigma'|\text{nil}$, then the proof is identical to Theorem 5.4.4 where $\sigma' = \sigma$. If $\delta = \Sigma \delta_1$, then $P = \Sigma \delta'$ for some $\delta'$ and $S = s'$. By Lemma 5.4.1 and the fact that $\mathcal{D} \models \mathcal{D}'$ for any $s$ ($\mathcal{D}'$ is a restriction of $\mathcal{D}$), $\mathcal{D}^{\text{end}[\sigma]} \cup C \cup \text{Sensed}[\sigma] \models \text{Trans}(\delta_1.\text{end}[\sigma], \delta'. s')$. And moreover, $\mathcal{D}^{\text{end}[\sigma]} \cup C \cup \text{Sensed}[\sigma] \models \exists s''. \text{Do}(\delta', s'')$. Thus, by the property of Chapter 4 for $\Sigma$ (equation 4.3), $\mathcal{D}^{\text{end}[\sigma]} \cup C \cup \text{Sensed}[\sigma] \models \text{Trans}(\Sigma \delta_1.\text{end}[\sigma], \delta'. s')$.

All the other cases are proved by induction on the structure of the program $\delta$ as done in Theorem 5.4.4.

**Completeness Results**

We have already gave completeness results for the evaluation procedure under certain circumstances. Mainly, those assumptions had to do with JIT-histories, that is, with histories providing sufficient information to the evaluation mechanism. We would like a similar assumption for programs, and therefore, we will extend the notion of just-in-time to them.

**Definition 5.4.1:** A history $\sigma$ is just-in-time (JIT) history for a program $\delta$ wrt to a GAT $\mathcal{D}$ iff

- $\delta = A$, where $A$ is an action and $\sigma$ is a JIT-history for formula $\text{Poss}(A, s)$ wrt $\mathcal{D}$;
- $\delta = \delta_1; \delta_2$. $\sigma$ is a JIT-history for $\delta_1$, and if $\mathcal{D} \cup C \cup \text{Sensed}[\sigma] \models \text{Final}(\delta_1.\text{end}[\sigma])$ it also a JIT-history for $\delta_2$: 
\[ \delta = (\delta_1 \| \delta_2) \ | \ (\delta_1 \parallel \delta_2) \ | \ (\delta_1) \delta_2 \ | \ \delta_1^\parallel \ | \ \pi.x.\delta_1 \text{ and } \sigma \text{ is a JIT-history for programs } \delta_1 \text{ and } \delta_2; \]

\[ \delta = \sigma \text{ and } \sigma \text{ is a JIT-history for formula } \sigma(s) \text{ wrt } D; \]

\[ \delta = \text{if } \sigma \text{ then } \delta_1 \text{ else } \delta_2 \mid \text{ while } \sigma \text{ do } \delta_1 \mid < \sigma \rightarrow \delta_1 >. \sigma \text{ is a JIT-history for } \sigma(s) \text{ and } \sigma \text{ is a JIT-history for } \delta_1 \text{ and } \delta_2; \]

\[ \delta = \Sigma \delta_1. \sigma \text{ is a JIT-history for program } \delta_1. \text{ and for every pair } \delta'.s' \text{ such that } \mathcal{D}^{\text{end}[\sigma]} \cup \mathcal{C} \cup \text{Sensed}[\sigma] \models \text{Trans}^*(\delta_1.\text{end}[\sigma], \delta'.s'). \sigma' \text{ is a JIT-history for } \delta' \text{ wrt the GAT } \mathcal{D}^{\text{end}[\sigma]}. \text{ where } \sigma' \text{ is an extension of } \sigma \text{ such that } \text{end}[\sigma'] = s' \text{ where all the sensor readings after } \sigma \text{ are assumed to be } \bar{0}. \]

In other words, a history is JIT for a program if it is a JIT-history for every formula that needs to be evaluated. This guarantees that we can determine the truth-value of each test in the program. as needed using regression and avoiding general theorem proving. In particular, when performing a lookahead we require all the information needed to do the search on the program. However, notice that the program in the search might not terminate, but this is another problem we are not going to address. See [13] for a discussion on non-termination of programs.

**Lemma 5.4.2:** If \( \sigma \) is a JIT-history for a bounded program \( \delta \) not containing search. then: (A) either \( \text{Axioms} \models \text{Final}(\delta.\text{end}[\sigma]) \) or \( \text{Axioms} \models \neg\text{Final}(\delta.\text{end}[\sigma]) \); (B) for every ground program \( \delta' \) and ground situation \( s' \), either \( \text{Axioms} \models \text{Trans}(\delta.\text{end}[\sigma], \delta'.s') \) or \( \text{Axioms} \models \neg\text{Trans}(\delta.\text{end}[\sigma], \delta'.s') \) where \( \text{Axioms} \) stands for \( \mathcal{D}^{\text{end}[\sigma]} \cup \mathcal{C} \cup \text{Sensed}[\sigma] \).

**Proof.** We prove this for \( \text{Final} \) and \( \text{Trans} \) at the same time by performing induction on the structure of the program. We show here only some cases for \( \text{Trans} \).

**Base Case:** If \( \delta = A \). \( A \) is a primitive action ground term. Assume the hypothesis does not holds. That is possible only if \( \delta' = \text{nil}. s' = \text{do}(A.s) \) but \( \text{Axioms} \not\models \text{Poss}(A.\text{end}[\sigma]) \)
and Axioms \( \not\models \) \(-\operatorname{Poss}(\lambda. \operatorname{end}[\sigma]) \). However, \( \sigma \) is a JIT-history for \( \operatorname{Poss}(\lambda. s) \) so that either Axioms \( \models \) \( \operatorname{Poss}(\lambda. \operatorname{end}[\sigma]) \) or Axioms \( \models \) \(-\operatorname{Poss}(\lambda. \operatorname{end}[\sigma]) \) (Theorems 5.2.2 and 5.2.1).

If \( \delta = \varphi(s) \). Assume the hypothesis does not hold. That can only happen if Axioms \( \not\models \varphi(\operatorname{end}[\sigma]) \) and Axioms \( \not\models \) \(-\varphi(\operatorname{end}[\sigma]) \), \( \delta' = \text{nil} \) and \( s' = s \). However, \( \sigma \) is a JIT-history for \( \varphi(s) \) so that either Axioms \( \models \varphi(\operatorname{end}[\sigma]) \) or Axioms \( \models \) \(-\varphi(\operatorname{end}[\sigma]) \) (Theorems 5.2.2 and 5.2.1).

**Induction Step:**

If \( \delta = \delta_1 \| \delta_2 \), then by induction, either Axioms \( \models \operatorname{Trans}(\delta_1. \operatorname{end}[\sigma], \delta', s') \) holds or Axioms \( \models \) \(-\operatorname{Trans}(\delta_1. \operatorname{end}[\sigma], \delta', s') \) holds and either Axioms \( \models \operatorname{Trans}(\delta_2. \operatorname{end}[\sigma], \delta', s') \) or Axioms \( \models \) \(-\operatorname{Trans}(\delta_2. \operatorname{end}[\sigma], \delta', s') \). Thus, it is easy to see that in any of the four possible combinations Axioms \( \models \operatorname{Trans}(\delta. \operatorname{end}[\sigma], \delta', s') \) or Axioms \( \models \) \(-\operatorname{Trans}(\delta. \operatorname{end}[\sigma], \delta', s') \) must hold.

If \( \delta = \delta_1 \cdot \delta_2 \), then by induction, either Axioms \( \models \operatorname{Trans}(\delta_1. \operatorname{end}[\sigma], \delta', s') \) holds or Axioms \( \models \) \(-\operatorname{Trans}(\delta_1. \operatorname{end}[\sigma], \delta', s') \) holds. In the first case, Axioms \( \models \operatorname{Trans}(\delta. \operatorname{end}[\sigma], \delta', s') \) holds as well. In the second case, either Axioms \( \models \operatorname{Final}(\delta. \operatorname{end}[\sigma]) \) holds or Axioms \( \models \) \(-\operatorname{Final}(\delta. \operatorname{end}[\sigma]) \) holds (by induction again). If the latter case applies, then Axioms \( \models \) \(-\operatorname{Trans}(\delta. \operatorname{end}[\sigma], \delta', s') \) holds. Otherwise, if Axioms \( \models \operatorname{Final}(\delta. \operatorname{end}[\sigma]) \), then \( \sigma \) is a JIT-history for \( \delta_2 \) as well, so that we again apply the induction hypothesis: given that either Axioms \( \models \operatorname{Trans}(\delta_2. \operatorname{end}[\sigma], \delta', s') \) or Axioms \( \models \) \(-\operatorname{Trans}(\delta_2. \operatorname{end}[\sigma], \delta', s') \) must hold, either Axioms \( \models \operatorname{Trans}(\delta. \operatorname{end}[\sigma], \delta', s') \) or Axioms \( \models \) \(-\operatorname{Trans}(\delta. \operatorname{end}[\sigma], \delta', s') \).

If \( \delta = \pi x. \delta_1 \), then \( \sigma \) is a JIT-history for \( \delta_1[\zeta] \) where \( \zeta \) is a constant in the domain of \( x \).

Then, Axioms \( \models \operatorname{Trans}(\delta_1[\zeta]. \operatorname{end}[\sigma], \delta', s') \) or Axioms \( \models \) \(-\operatorname{Trans}(\delta_1[\zeta]. \operatorname{end}[\sigma], \delta', s') \) by induction. Hence, if for all constants \( \zeta \) in the domain of variable \( x \) it is the case that

\(^{3}\)It is easy to see that if \( \sigma \) is a JIT-history for a formula \( \varphi(\text{vec}x. s) \) it is also a JIT-history for the formula \( \varphi(\overline{t}. s) \) where \( \overline{t} \). The equivalent thing happens for programs.
Axioms $\models \neg Trans(\delta_1^{\varphi}.\text{end}[\sigma], \delta'.s')$. then it is also the case, given that $x$ domain is closed on all that constants. that Axioms $\models \neg Trans(\pi.x.\text{end}[\sigma], \delta'.s')$. On the other hand, if there is just one constant $c$ such that Axioms $\models Trans(\delta_1^{\varphi}.\text{end}[\sigma], \delta'.s')$. then Axioms $\models Trans(\pi.x.\text{end}[\sigma], \delta'.s')$ is true.

Lemma 5.4.3: If $\sigma$ is a JIT-history for a bounded program $\Sigma \delta_1$ then:

(A) Axioms $\models Final(\Sigma \delta_1.s)$ or Axioms $\models \neg Final(\Sigma \delta_1.s)$; (B) for every program $\delta'$ and situation $s'$ Axioms $\models Trans(\Sigma \delta_1.s, \delta'.s')$ or Axioms $\models \neg Trans(\Sigma \delta_1.s, \delta'.s')$. Axioms stands for $\mathcal{D}^{\text{end}[\sigma]} \cup \mathcal{C} \cup \mathcal{S}^{\text{ensed}[\sigma]}$.

Proof.

(A) is easy from Lemma 5.4.2 since $\delta_1$ has no search and $\sigma$ is a JIT-history for program $\delta_1$. So let us prove (B).

Since $\sigma$ is JIT-history for $\delta_1$ then - by Lemma 5.4.2 - one of the following cases applies: (i) Axioms $\models Trans(\delta_1.s, \delta'.s')$ or (ii) Axioms $\models \neg Trans(\delta_1.s, \delta'.s')$. If (ii) is the case, then we also know that either (a) Final($\delta'.s'$) or (b) $\neg Final(\delta'.s')$ must hold. If (a) is the case, then Axioms $\models Trans(\Sigma \delta_1.s, \delta'.s')$; if (b) is the case, then Axioms $\models \neg Trans(\Sigma \delta_1.s, \delta'.s')$ and we are done.

If (i) is the case, then we need to prove that one of the following two cases applies:

(a) Axioms $\models \exists \delta''.s''. Trans^*(\delta'.s'. \delta''.s'') \land Final(\delta''.s'')$

(b) Axioms $\models \neg \exists \delta''.s''. Trans^*(\delta'.s'. \delta''.s'') \land Final(\delta''.s'')$.

Assume - by contradiction - that neither (a) nor (b) holds.

Then, take any pair $(\delta''.s'')$ such that Axioms $\models Trans^*(\delta'.s'. \delta''.s'')$ holds. First, any such $(\delta''.s'')$ cannot be final, that is, it cannot happen that Axioms $\models Final(\delta''.s'')$ or (a) would hold immediately. Second, if for all possible pairs $(\delta''.s'')$ either it holds Axioms $\models Trans(\delta'.s'. \delta''.s'')$ or Axioms $\models \neg Trans(\delta'.s'. \delta''.s'')$ then, because all possible programs can be named as terms, (b) would hold immediately: we can always
reach a dead-end (where all transitions are false) or cycle forever (with an infinite sequence of \(\text{Trans}\) but no \(\text{Final}\)).

Hence, there must be a program \(\delta^*\) and a situation \(s^*\) (probably \(\delta'\) and \(s'\)) such that \(\text{Axioms} \not\models \text{Final}(\delta^*, s^*)\) and \(\text{Axioms} \not\models \text{\neg Final}(\delta^*, s^*)\): or for some \(\delta^{**}, s^{**}\) neither \(\text{Axioms} \models \text{Trans}(\delta^*, s^*, \delta^{**}, s^{**})\) nor \(\text{Axioms} \not\models \text{\neg Trans}(\delta^*, s^*, \delta^{**}, s^{**})\) holds. But, and here comes the important part, since \(\sigma^*\) is a JIT-history for \(\delta^*\) and \(\delta^*\) has no search, none of the above possibilities can happen (Lemma 5.4.2). Therefore, we have arrived at a contradiction and either (a) or (b) must hold.

**Corollary 5.4.3:** If \(\sigma\) is a JIT-history for a bounded program \(\delta\):

(A) either \(\text{Axioms} \models \text{Final}(\delta, \text{end}\{\sigma\})\) or \(\text{Axioms} \not\models \text{\neg Final}(\delta, \text{end}\{\sigma\})\); (B) for every program \(\delta'\) and situation term \(s'\). either it is the case that \(\text{Axioms} \models \text{Trans}(\delta, \text{end}\{\sigma\}, \delta', s')\) or \(\text{Axioms} \not\models \text{\neg Trans}(\delta, \text{end}\{\sigma\}, \delta', s')\) where \(\text{Axioms}\) stands for \(D^{\text{end}\{\sigma\}} \cup C \cup \text{Sensed}\{\sigma\}\).

**Proof.** Immediately from Lemma 5.4.2 and Lemma 5.4.3 for search.

If the actual history is JIT for a program, then a call to \text{trans}/4 and \text{final}/2 cannot abort (with error) the execution.

**Lemma 5.4.4:** Let \(\sigma\) be a JIT-history for a bounded program \(\delta\) and let \(D\) be the underlying acyclic GAT. Then: (A) the goal \(\text{\text{final}}(\delta, \text{end}\{\sigma\})\) eventually succeeds \(n \geq 0\) times and then it finitely fails: (B) the goal \(\text{\text{trans}}(\delta, \text{end}\{\sigma\}, P, S)\) eventually succeeds \(n \geq 0\) times and then finitely fails if \(\delta\) contains no search: (C) the goal \(\text{\text{trans}}(\delta, \text{end}\{\sigma\}, P, S)\) eventually succeeds \(n \geq 0\) times and then finitely fails or does not terminate.

**Proof.** By induction on the structure of the program. (A) and (B) proved at the same time (We just show the case for (B)). To prove (C) we just have to add to (B) the case for search.

**Base Case:** If \(\delta = .A|\sigma\) then it may succeed one time and then finitely fail or just finitely fail. The first case arises whenever both goals \(\text{\text{evalsec}}(\text{Poss}(A), \text{end}\{\sigma\}, B)\) and \(\text{\text{evalsec}}(\sigma, \text{end}\{\sigma\}, B)\) respectively succeed with \(B = \text{true}\). The second case arises when
they succeed with \( B = false \). Notice that one of the two cases should apply because of Corollary 5.4.2. If \( \delta = nil \) then the goal finitely fails.

**Induction Step:** If \( \delta = \text{while } \sigma \text{ do } \delta_1 \), then by Corollary 5.4.2, \( \text{evalsec}(\sigma. \text{end}[\sigma],B) \) succeeds with \( B = true \) or \( B = false \). In the first case, the goal behaves depending on \( \text{trans}(\delta_1. \text{end}[\sigma],P,H) \) and by induction we easily get the hypothesis. In the second case, the goal just finitely fails. If \( \delta = \delta_1 | \delta_2 \), then by induction hypothesis we trivially get the hypothesis given that \( \sigma \) is a JIT-history for both \( \delta_1 \) and \( \delta_2 \).

Finally, to get (C) we have to extend (B) for the search operator. Then, assume that \( \delta = \Sigma \delta_1 \). Because of (B) and the fact that \( \sigma \) is a JIT-history for \( \delta_1 \), the goal \( \text{trans}(\delta_1. \text{end}[\sigma],P,H) \) has to succeed some finite number of times and then finitely fails. However, after each success with \( P = \delta' \) and \( H = s' \), the goal \( \text{findpath}(\delta', s') \) is called so we have to prove that, in fact, such goal eventually succeeds some finite number of times before finitely failing or never terminate. We argue that this is in fact true because of the following: after any sequence of \( n \geq 0 \) transitions from \( \delta', s' \) to \( \delta'', s'' \) constructed by \( \text{findpath}/2 \), it is the case that (i) \( \text{final}(\delta'', s'') \) either succeeds or finitely fails, and that (ii) \( \text{trans}(\delta'', s'', E1, H1) \) succeeds \( m \geq 0 \) times after finitely failing. Take \( \sigma'' \) as an extension of \( \sigma \) such that \( \text{end}[\sigma''] = s'' \) and all sensor readings after \( \sigma \) are characterized by \( \bar{0} \). Then, \( \sigma'' \) has to be a JIT-history for \( \delta'' \) wrt the theory \( D_{\text{end}[\sigma]} \) because \( D_{\text{end}[\sigma]} \cup C \cup \text{Sensed}[\sigma] \models \text{Trans}^*(\delta_1. \text{end}[\sigma], \delta'', s'') \). We can now apply (A) and (B) to get that both (i) and (ii) must hold. Notice that in the implementation, the fact that the sensors from \( \sigma \) to \( \sigma'' \) are not available in the database makes the interpreter behaves as if the underlying theory were \( D_{\text{end}[\sigma]} \).

With this last theorem we are ready to prove the completeness of \( \text{Trans} \) and \( \text{Final} \) implementation for just-in-time histories.

**Theorem 5.4.6: (Completeness of \( \text{Trans} \) and \( \text{Final} \) without search)** Let \( \sigma \) be a JIT-history for a bounded program \( \delta \) not containing search, and let \( D \) be the underlying acyclic GAT.
1. The goal $\text{trans}(\delta, \text{end}[\sigma], P, S)$ succeeds with $P = \delta'$ and $S = s'$ whenever

$$D^\text{end}[\sigma] \cup C \cup \text{Sensed}[\sigma] \models \text{Trans}(\delta, \text{end}[\sigma], \delta'. s')$$

2. The goal $\text{final}(\delta, \text{end}[\sigma])$ succeeds whenever

$$D^\text{end}[\sigma] \cup C \cup \text{Sensed}[\sigma] \models \text{Final}(\delta, \text{end}[\sigma])$$

**Proof.** We just prove the case where $\text{Trans}(\delta, \text{end}[\sigma], \delta'. s')$ is entailed. For clarity, we refer to the set $D^\text{end}[\sigma] \cup C \cup \text{Sensed}[\sigma]$ as Axioms.

**Base Case:** The only case arises when $\delta = A$. Then, given that $D$ is acyclic and $\sigma$ is a JIT-history for $\text{Poss}(A, s)$, the goal $\text{evalsec}(\text{Poss}(A, \sigma), B)$ should succeed with either $B = \text{true}$ or $B = \text{false}$ (Corollary 5.4.2). However, since $D \models \text{Poss}(A, \text{end}[\sigma])$ it has to be the case (because $\text{evalsec}/3$ is sound) that such goal succeeds with $B = \text{true}$. Then, $\text{trans}(A, \text{end}[\sigma], P, S)$ must succeed with $P = \text{nil}$ and $S = \text{end}[\sigma]$ and we are done.

**Induction Step:**

- If $\delta = \delta_1 | \delta_2$ then it is sufficient to prove that one of the following cases must hold:
  (i) $\text{Axioms} \models \text{Trans}(\delta_1, \text{end}[\sigma], \delta'. s')$; or (ii) $\text{Axioms} \models \text{Trans}(\delta_2, \text{end}[\sigma], \delta'. s')$.
  Assume - by the contrary - that neither (i) nor (ii) holds. Then, by Lemma 5.4.3, $\text{Axioms} \models \neg \text{Trans}(\delta_1, \text{end}[\sigma], \delta'. s')$ and $\text{Axioms} \models \neg \text{Trans}(\delta_2, \text{end}[\sigma], \delta'. s')$. This implies that $\text{Axioms} \models \neg \text{Trans}(\delta, \text{end}[\sigma], \delta'. s')$ which is an absurd with the premise. Therefore either (i) or (ii) holds. If (i) holds, then by induction $\text{trans}(\delta_1, \text{end}[\sigma], P, S)$ eventually succeeds with $P = \delta, S = s'$. which implies by the $\text{trans}/4$ clause for nondeterministic choice, that $\text{trans}(\delta_1, \text{end}[\sigma], P, S)$ eventually succeeds with answers $P = \delta, S = s'$. The case for (ii) is analogous but we have to use Lemma 5.4.4 to state that the goal $\text{trans}(\delta_1, \text{end}[\sigma], P, S)$ would eventually finitely fail.

- If $\delta = \text{if } o \text{ then } \delta_1 \text{ else } \delta_2$, then given that $\sigma$ is a JIT-history for $o(s)$ the goal $\text{evalsec}(o, \text{end}[\sigma], B)$ succeeds with $B = \text{true}$ or $B = \text{false}$ (Corollary 5.4.2). If
\[ B = true, \text{ then by soundness of evaluate/3, } D \models \sigma(\text{end}[\sigma]) \text{ and it must be the} \]
\[ \text{case that } Trans(\delta_1.\text{end}[\sigma], \delta', s') \text{ is entailed. By induction, } \sigma \text{ is a JIT-history for} \]
\[ \text{program } \delta_1. \text{ it follows that } \text{trans}(\delta_1.\text{end}[\sigma], P, S) \text{ succeeds with } P = \delta'. S = s'. \]
\[ \text{which implies that } \text{trans}(\delta.\text{end}[\sigma], P, S) \text{ succeeds with } P = \delta'. S = s'. \]
\[ \text{If } B = false, \text{ the proof is almost equal by using } \delta_2 \text{ instead of } \delta_1. \]

- If \( \delta = \pi.x.\delta_1 \) then \( \sigma \) is a JIT-history for any \( \delta_1 | \xi \), where \( c \) is a constant in the domain of \( x \). Remember that since \( \delta \) is a bounded program all variables used have a domain closure axiom associated which implies that \( \text{Axioms } \models Trans(\delta_1 | \xi.\text{end}[\sigma], \delta, s') \) for some constant \( c \). By induction, \( \text{trans}(\delta_1 | \xi.\text{end}[\sigma], P, S) \) eventually succeeds with \( P = \delta'. S = s' \) and so does \( \text{trans}(\pi.x.\delta_1.\text{end}[\sigma], P, S) \) since \( x \) is eventually replaced by \( c \) in \( \delta_1 \) by \( \text{subv}/4 \). Keep in mind that . by Theorem 5.4.1. every goal \( \text{trans}(\delta_1 | \xi.\text{end}[\sigma], P, S) \) must eventually finitely fail for every \( c' \) chosen before \( c \) by the \( \text{trans}/4 \) clause for nondeterministic choice of argument.

So for bounded programs that do not make use of the search operator, we have a sound and compete implementation of \( Trans \) and \( Final \). When using search we are not guaranteed to have a finite computation of \( Trans \).

**Theorem 5.4.7: (Weak Completeness of Trans and Final for search)** Let \( \sigma \) be a JIT-history for a bounded program \( \delta = \Sigma \delta_1 \) and let \( D \) be the underlying acyclic GAT.

1. The goal \( \text{trans}(\delta.\text{end}[\sigma], P, S) \) eventually succeeds with \( P = \delta' \) and \( S = s' \) or does not terminate whenever
\[ D^{\text{end}[\sigma]} \cup C \cup \text{Sensed}[\sigma] \models Trans(\delta.\text{end}[\sigma], \delta', s') \]

2. The goal \( \text{final}(\delta.\text{end}[\sigma]) \) succeeds whenever
\[ D^{\text{end}[\sigma]} \cup C \cup \text{Sensed}[\sigma] \models Final(\delta.\text{end}[\sigma]) \]
If $\delta = \Sigma \delta_1$, then because of $Trans$ definition for $\Sigma$:

$$Axioms \models Trans(\delta_1, end[\sigma], \delta', s') \wedge \exists \delta'' . s'' . Trans^*(\delta', s', \delta'', s'') \wedge Final(\delta'', s'')$$

where $Axioms$ stands for $D^{end[\sigma]} \cup _{\Sigma} Sensed[\sigma]$. The following two things should happen:

1. Because $\sigma$ is a JIT-history for $\delta_1$, by Theorem 5.4.6, the goal $trans(\delta_1, end[\sigma], P, S)$ should eventually succeed with $P = \delta'$ and $S = s'$.

2. For any pair $(\delta^*, s^*)$ (including $(\delta', s')$) such that $Axioms \models Trans^*(\delta_1, end[\sigma], \delta^*, s^*)$, it has to be the case (by Lemma 5.4.3) that for any program $\delta''$ and situation $s''$ either $Axioms \models Trans(\delta^*, s^*, \delta'', s'')$ or $Axioms \models \neg Trans(\delta^*, s^*, \delta'', s'')$. Similarly, $Axioms \models Final(\delta^*, s^*)$ or $Axioms \models \neg Final(\delta^*, s^*)$ should hold as well. Because of this, and the fact that every possible program has a name, it must be the case that there exists a finite sequence of program-situation term pairs $(\delta'_0 = \delta', s'_0 = s'), (\delta'_1, s'_1), ..., (\delta'_n, s'_n)$ with $n \geq 0$ such that $Axioms \models Trans(\delta'_i, s'_i, \delta'_{i+1}, s'_{i+1})$ and $Axioms \models Final(\delta'_n, s'_n)$ for $i = 0..n$.

Now, if $\text{findpath}(\delta', s')$ does not terminate, neither does $\text{trans}(\delta, end[\sigma], P, S)$. However, if it does terminate, then using Theorem 5.4.6 and the above argument the goals $\text{trans}(\delta'_i, s'_i, P, \pi H)$ should eventually succeed with $P = \delta'_{i+1}, S = s'_{i+1}$ for $i = 0..n - 1$ and the goal $\text{final}(\delta'_n, s'_n)$ should eventually succeed as well. Given the two clauses for $\text{findpath}/2$ and the assumption that it does not run forever, the goal $\text{findpath}(\delta', s')$ must succeed.

Putting the above two results together we have a weak completeness result for any bounded program.

**Corollary 5.4.4: (Weak Completeness of Trans and Final)** Let $\sigma$ be a JIT-history for a bounded program $\delta$ and let $D$ be the underlying acyclic GAT.
1. The goal $\text{trans}(\delta. \text{end}[\sigma], P, S)$ succeeds with $P = \delta'$ and $S = s'$ or does not terminate whenever

$$D^{\text{end}[\sigma]} \cup C \cup \text{Sensed}[\sigma] \not\models \text{Trans}(\delta. \text{end}[\sigma], \delta', s')$$

2. The goal $\text{final}(\delta. \text{end}[\sigma])$ succeeds whenever

$$D^{\text{end}[\sigma]} \cup C \cup \text{Sensed}[\sigma] \not\models \text{Final}(\delta. \text{end}[\sigma])$$

**Proof.** Immediately from Theorem 5.4.6 and Theorem 5.4.7. Notice that termination cannot be guaranteed for $\text{trans}/4$ given the potential presence of search.

### 5.5 Unbounded Programs

The IndiGolog interpreter described here is provably correct over guarded action theories in open-world settings, but relies on a number of assumptions, which we have presented. One serious limitation concerns the use of bounded programs, which would be inconvenient when dealing with large sets of elements that are added and deleted dynamically and cannot be enumerated. For instance, we may have a fluent $\text{Student}(x, s)$ denoting that $x$ is a student in situation $s$ and such that students enroll and unenroll dynamically. In such cases, we clearly need an evaluation procedure that works with free variables. However, this is not sufficient to guarantee soundness of $\text{Trans}$ and $\text{Final}$ as already explained. To do so, either we need to drop the prioritized concurrency construct, or to use an evaluation that is complete over free variables.

The problem with free variables is that Prolog tells us the bindings for which the goal holds, but not the which ones for which it does not. The good news is that this would be required, generally, only when comparing actions terms. For instance, suppose we have an open formula $\text{student}(x, s)$ such that $D_0 \models \forall x. \neg \text{student}(x, S_0)$ so that students should enroll to become a student. The GSSA would be:

$$\text{true} \supset \text{student}(x, \text{do}(a, s)) \equiv a = \text{enroll}(x) \lor \text{student}(x, s)$$
Therefore, if the sort of \( x \) has no domain closure we are assuming that after any sequence of actions there will always be someone who is not a student!. Formally, \( D \models \forall s \exists x . \neg student(x, s) \). The goal \( \text{eval}(\text{student}(X), \sigma, B) \) for a history \( \sigma = (\mu_0) \cdot (\text{enroll}(\text{peter}), \mu_1) \) would succeed with \( X = \text{peter}, B = true \) and then finitely fail in the evaluation procedure presented in [11]. This is due to the fact that when evaluating \( \text{enroll}(\text{peter}) = \text{enroll}(X) \) we only get the positive answer that it holds for \( X = \text{peter} \), but not the negative answers for which it does not hold, for instance \( X = \text{john} \). This may lead to unsound results when \( \text{eval}/3 \) is used in \( \text{Trans} \) and \( \text{Final} \).

One way to solve the problem is to add \( \text{constraints} \) as a new argument to \( \text{eval}, \text{trans} \) and \( \text{final} \). These procedures would say things like “everything but John, Mark and Peter is not a student” or “there is a transition whenever \( X \) is not Mark”. With that, we would have a sound, and sometimes complete implementation of \( \text{Trans} \) and \( \text{Final} \), while keeping the ability to express open formulas.

Since we can restrict our theories so that this problem only arises in the evaluation of GSSA, in particular when comparing actions terms, it is not hard to think of an evaluation that behaves like this:

```prolog
1 ?- eval(student(X), [enroll(peter)], B, C).
X = _,
C = (X \= peter),
B = false ;

X = peter ,
C = (true) ,
B = true ;

no
```

The argument \( C \) is new and it expresses the constraints over the free variable for
which $B$ applies. For instance, the first argument says that everyone except Peter is not a student. The second answer says that Peter is in fact a student. As we can see the constraint argument $C$ is just a Prolog goal and it means all possible instantiations of that goal that will make it succeed. Note that eval/4 is now complete over the free variable $X$ and therefore it would be possible to check whether we have obtained an answer for every possible $X$. If not we would abort the execution so as to not get unsound transitions. However, we may now deal with open and dynamic fluents and also make use of Prolog instantiation mechanism which may result in faster evaluations.

We can implement this evaluation by making use of `getfalse(Act1 = Act2, C)` which builds the constraints under which $Act1$ and $Act2$ are in fact different. While $Act1$ is a ground action term representing the actual action in the history. $Act2$ may contain free Prolog variables coming from the GSSA definitions. All actions term should contain simple arguments so that `paint(table.color(wall))` is not valid but `paint(table.blue)` is.

```prolog
getfalse((P1=P2),C):- P1 =.. [Act1|Arg1], P2 =.. [Act2|Arg2],
    ( Act1 \= Act2 ; not equal_length(Arg1,Arg2) -> C=true ; pair(Arg1,Arg2,C) ).
```

% From a list of arguments construct a constrains goal
```prolog
pair([X1],[X2],X1 \= X2).
pair([X1|L1],[X2|L2],((X1 \= X2) ; C2)):- pair(L1,L2,C2).
```

If the names of the actions or the number of the arguments are different then $Act1$ and $Act2$ are always different. Otherwise, they are different whenever one of the arguments are different.

```prolog
| ?- getfalse(enroll(mark,math)=enroll(X,Y),C).
```

```prolog
C = (X\=mark ; Y\=math)
```

```
yes
```

Clearly, these constraints should be combined when evaluating complex formulas and
also translated to \textit{Trans} and \textit{Final}.

5.6 Chapter Summary

This chapter has addressed the problem of implementing an interpreter for IndiGolog, that is, an interpreter to execute online a high-level program with local lookahead capability and with respect to a background guarded action theory. Since the projection task is crucial for our objective, we have first given an abstract mechanism to solve it. This mechanism is a combination between regression and evaluation. However, under some plausible restrictions we showed that the evaluation needed can also be performed via regression so that the only evaluation needed is the one done in the initial database. Although this regression is not complete, we distinguished a subset of cases, namely just-in-time history, for which it actually is.

Inspired in that form of regression, we have developed a Prolog interpreter for bounded programs. Bounded programs are the ones where variables are associated to closure domain axioms. We have showed both sound and completeness results for both the implementation of an evaluation procedure to perform the projection task, and the implementation of \textit{Trans} and \textit{Final}. Given the potential presence of search, and the fact that the search implementation is an approximation of its definition, all those results are relative to an approximation to the background theory of action as well. Finally, we have proposed some clues on how to drop the assumption of bounded programs so that we can actually deal with open formulas.
Chapter 6

Conclusions

The development of a neat and principled formalism for intelligent agent controllers is of main interest in both the AI and the robotics communities. For that, mainly two problems should be addressed and combined. First, a formalism is needed to represent the world where the agent is actually situated. Apart from modeling the effects and preconditions of actions, a realistic formalism should accommodate sensing data, exogenous events, partial information and knowledge among the most important things. The second problem has to do with how appropriate actions for the agent are generated as a function of some current set of beliefs and desires. Classical planning has frequently been used for this propose.

In the present work, we have addressed the above two problems by exploring a practical, and still formal, system that combines a flexible kind of theory for modeling open-world settings with sensing, and a high-level programming framework for generating actions to be executed by the agent.

In Chapter 3 we have studied Guarded Actions Theories, extensions of Basic Action Theories, where sensing information is modeled in a very intuitive and clean manner and where causal laws can be partially expressed, something impossible in Basic Action Theories. In that Chapter, we stated the theories that are considered well-behaved.
we showed that all ways of reasoning about the actions performed in the world can be combined, and we finally proved that, in interesting and practical settings, our treatment of sensing is at least as powerful as in most approaches where sensing actions are used.

In Chapter 4 we focused on how actions can be generated automatically and practically. We argued that changing from a totally offline execution of high-level program to a combination between offline and online can result in many advantages. More precisely, by executing programs online while providing a local lookahead mechanism that the programmer can use, it is possible to obtain a system for large programs while allowing sensing information and exogenous events to be treated easily and efficiently. It is of main interest then to study the behavior of this mix under certain circumstances. In particular we explore how concurrency and sensing mixes with this type of execution.

IndiGolog then is an extension of ConGolog where programs are executed online and where the action theory behind them is a guarded one. We concluded by showing a simple interpreter that implements the execution of IndiGolog high-level programs. To do that, we suggested a form of regression, namely general regression, to implement the projection task, something mandatory for the execution of programs. Finally, we provided soundness and completeness results for both the general regression mechanism and the interpreter.

Many aspects remain open and are interesting to pursue in the near future. Among them, we consider that the following three should be addressed promptly if we where to obtain more expressive and practical systems:

1. As we have seen, our search operator is not totally adequate when sensing plays an important role inside offline lookahead since it does not reason with the information that is gathered via sensing. Then, it would be very interesting to strengthen the local lookahead construct to calculate more general plans. In particular, sGolog’s conditional action trees arise as a more expressive formalism while having good chances to be efficiently implemented. However, it seems possible to come up with
an even more powerful search definition that calculates sophisticated robot plans in the flavor of [29] and [26].

2. Once serious limitation concerns the use of bounded programs, which would be inconvenient when dealing with large sets of elements that are added and deleted dynamically or cannot be enumerated. We need then an implementation that can deal with free variables in a sound and efficient manner. Although we have provided some suggestions on how to achieve this, by using an evaluation with constraints, we have no provably correct implementation yet.

3. In the present work, we have only dealt with relational fluents. However, functional functions are necessary for any real-world domain. The only part that seems not trivial to extend is the generalized regression discussed in Chapter 5. It would be worth trying to come up with a full regression schema in the sense that the truth value of a formula in some situation is completed determined by the truth value of another formula, which is in fact its regression, in the initial database.
Bibliography


A model in which it is possible to reason about an agent's probabilistic degrees of belief is presented. The model also includes the manner in which these beliefs change as various actions are executed and sensors are used to gain further information, where both actions and sensors are subject to error.


Proposes a high level language similar to A that allows sensing actions and such that states are three valued interpretations.


Presents a high level action description language with sensing actions. A semantic is given using knowledge states and transition functions over them. The work is highly related to other similar and important approaches for formalizing knowledge producing actions.

Presents a unified approach for modeling and solving planning problems that is based on state models that handle various types of dynamics (deterministic, non-deterministic, and probabilistic), and sensor (null, partial, and complete) feedback. The approach makes use of logical representation languages, mathematical models of sequential decisions and heuristic search algorithms.


An combination between logic based high-level programming languages and decision-theory is suggested. A high-level program is viewed as an advisor to the decision-planner so that the actual planning needed is heavily reduced.


Describes the software architecture of an autonomous tour-guide/tutor robot. The robot’s control software integrates low-level probabilistic reasoning with high-level problem solving embedded in first order logic.


Presents the theoretical foundations for the ConGolog programming language.

Shows how it is possible to obtain a definition in the situation calculus of a programming language that supports a rich account of concurrency.


Suggests a practical way of dealing with large agent programs by executing them in an online fashion with local offline lookahead. A new constructor is presented for performing local search and binary sensing actions are accommodated as well. An incremental interpreter is showed using these ideas.


A theory of action where a solution to the frame problem may or may not be available and where online information from sensors may or may not be applicable is suggested. The projection task is then re-defined for such theories and a general form of regression is developed to solve it.


Describes an approach to proving and expressing properties of non-terminating Golog programs such as liveness, fairness, etc. Given the characteristics of Golog, properties can be expressed and proved in the very same logic.

An extensive coverage of planning and control where the two areas are reformulated in a common control framework and where opportunities for integrating their ideas and methods are identified.

Considers the general problem of progressing an initial database in response to a given sequence of actions. The work is done on the situation calculus and an axiomatization of actions which addresses the frame problem and it is proved that we can compute efficiently the progressed database in first order logic for many useful special cases.

A syntactic approach to representing knowledge within the situation calculus using interval arithmetic. The proposal is viewed as an efficient alternative to the well-known possible-worlds approach.

Describes an approach to reasoning about change for rich domains where it is not possible to anticipate all situations that might occur. It involves keeping a single model of the world that is updated when actions are performed by constructing the nearest world to the current one in which the consequences of the actions under consideration hold.

Presents a way of representing and reasoning about actions with many qualifications. Actions are qualified by general constraints describing the domain being investigated and not by specific facts that may or may not hold when they are executed.


Proposes a planning system called SADL which is a combination between UWL and ADL such that knowledge is inherently temporal and knowledge preconditions are unnecessary for an important class of domains. SADL is specially aimed to encode the rich domain theory of the Internet Softbots.


A survey of reasoning about knowledge with particular emphasis on work in computer science.


Suggests an incremental way of performing planning by calculating prefixes instead of whole plans. For that, it presents a correct and polynomial incremental planning algorithm for the 3S class.


Extends Golog to deal with sensing actions. A legal execution is now a conditional
plan instead of linear sequences of action, where decision points depend on the sensing outcome that are only known at execution time.


Describes how high-level controllers for robots that are reactive can conveniently be implemented in ConGolog. Reactivity is achieved by exploiting ConGolog's prioritized concurrent processes and interrupts facilities.


A survey of topics related to a novel approach to high-level robot programming based on a logical theory of action.


Describes a new logic programming language well suited for applications in high level control of robots and industrial processes, intelligent software agents, etc. The language is based on a formal theory of action specified in an extended version of the situation calculus.


Develop a specification within the situation calculus of conditional and iterative plans
over domains that include binary sensing actions. The idea is to understand how plans should be when there is sensing information available.

Describes an implementation of a system that uses a controller from the Golog family of planners to control a MINDSTORMSTM robot. The idea is to demonstrate that it is actually possible now to do practical cognitive robotics.

An explanation on which is the position of the Cognitive Robotics Group at University of Toronto about robotic research. The work proposes to shift from classical automated planning to high-level programming.

[29] Fangzhen Lin and Hector J. Levesque. What robots can do: Robot programs and effective achievability. Artificial Intelligence. 101:201–226. 1998. A deep study on robot achievability. The question “What goals can be achieved by a robot?” if formalized and a simple robot programming language is shown to be universal, in that any effectively achievable goal can be achieved by getting the robot to execute one of the robot programs.

An extensive work on the foundations of intelligent agent modeling. It explores many philosophical point of view that arise when making intelligent agents and it propose a rich formalism, namely the situation calculus, for modeling dynamical changing worlds within which it can be proved that a strategy will achieve a goal.

Describes a formal theory of knowledge and action based on a possible-world analysis of the former.


A theoretical contribution to the metatheory of situation calculus axiomatization of dynamical systems with the objective of simplifying the entailment problem.

[34] Reiter R. Knowledge in action: Logical foundations for describing and implementing dynamical system.

An extensive exploration of a logical approach to modeling dynamical systems based on the situation calculus.


Provides a solution to the frame problem derived from two previous proposals such that it provides a systematic treatment of goal regression that is provably sound and complete.

[36] Raymond Reiter. The projection problem in the situation calculus: A soundness and completeness result. with and application to database updates. In *Proceedings of

Describes how a database and its evolution under update transactions can be formalized in the situation calculus. Also, the relation between query evaluation and the projection task, and other similarities between databases and situation calculus, are explained.


Provides a metatheory of action and change that relates logics of action and change to their intended meaning.


A solution to knowledge in the situation calculus is developed. The work adapts the possible world approach to knowledge to the situation calculus by introducing an accessibility relation between situations and its corresponding successor state axiom for it.


Provides a solution to the frame problem by appealing to explanation closure axioms for worlds with fully specified actions. The inference of change and nonchange is monotonic and it is argue that explanation axioms are preferable to frame axioms.

Provides the basic idea and the formal specification of the Fluent Calculus. The Fluent Calculus addresses the inferential frame problem by specifying the effect of actions in terms of how an action modifies a state.