Model-Based Execution Monitoring

Marcus Bjäreland

April 24, 2001

Abstract

The task of monitoring the execution of a software-based controller in order to detect, classify, and recover from discrepancies between the actual effects of control actions and the effects predicted by a model, is the topic of this thesis. Model-based execution monitoring is proposed as a technique for increasing the safety and optimality of operation of large and complex industrial process controllers, and of controllers operating in complex and unpredictable environments (such as unmanned aerial vehicles).

In this thesis we study various aspects of model-based execution monitoring, including the following:

The relation between previous approaches to execution monitoring in Control Theory, Artificial Intelligence and Computer Science is studied and a common conceptual framework for design and analysis is proposed.

An existing execution monitoring paradigm, *ontological control*, is generalized and extended. We also present a prototype implementation of ontological control with a first set of experimental results where the prototype is applied to an actual industrial process control system: The ABB STRESSOMETER cold mill flatness control system.

A second execution monitoring paradigm, *stability-based execution monitoring*, is introduced, inspired by the vast amount of work on the "stability" notion in Control Theory and Computer Science.

Finally, the two paradigms are applied in two different frameworks. First, in the "hybrid automata" framework, which is a state-of-the-art formal modeling framework for hybrid (that is, discrete+continuous) systems. Secondly, in the logical framework of GOLOG and the Situation Calculus.

Acknowledgments

There are numerous people that (more or less voluntarily) have contributed to this thesis with help, suggestions, criticism and support:

First and foremost, I would like to thank my main advisor, Prof. Patrick Doherty, for challenging and inspiring discussions and input, which have, without exception, led to significant improvements of my many drafts and versions of this thesis.

Next, I want to thank my secondary advisor, Dr. Dimiter Driankov, for his patience and for the many intense technical discussions we have had. Having two knowledgeable and strong (American and Bulgarian) advisors does not make thesis writing easier, but a lot more interesting.

Without the support of Dr. George Fodor, large parts of this thesis would not have been written. His help in making me understand his own work, in facilitating experiments on data from ABB, and in teaching me everything I know about practical control engineering, has had an immense impact on this thesis.

I have enjoyed cooperating with those co-authoring the papers on which parts of this thesis are based: Javier Pinto, Chitta Baral, Mutsumi Nakamura, and Patrik Haslum.

The experimental investigations of this thesis were done using a software system implemented by Per Lewau as part of his MSc thesis. I am greatful for his contributions and his patience with me as his advisor.

The working environment at IDA, and the AIICS division in particular, has been inspiring. I would like to thank all of you.

Finally, I want to thank my parents, Thorsten and Hillevi, for all the support, understanding, and encouragement they have given me throughout this period. This thesis is for you.

Marcus Bjäreland
Linköping, April 2001

Contents

Chapter 1

Introduction

The topic of this thesis is the task of monitoring the execution of software-based controllers in order to detect, classify, and recover from discrepancies between the actual (measured) effects of control actions and the effects predicted by a model. Model-based execution monitoring is proposed as a technique for increasing the safety and optimality of operation of large and complex industrial process controllers and for controllers operating in complex and upredictable environments (such as unmanned aerial vehicles).

We are interested in *discrete* or *hybrid software-based control systems.* Such systems can be found in applications ranging from microwave ovens to nuclear power plants and paper mills. However, we limit ourselves to two important classes of applications: *Autonomous Systems* and *Industrial Process Controllers.* By "Autonomous Systems" we, in this thesis, mean control systems that operate in highly unpredictable environments, such as control systems for mobile robots or unmanned aerial vehicles. An industrial process controller, on the other hand, may operate in very "well-engineered" environments, that is, environments where effects of actuator invocations may be predicted with high precision. One complicating factor for such systems is that they typically are very large. We will argue that these two types of systems exhibit similar kinds of problems, and we will address those problems within the same framework.

Another conceptual dimension of control systems that is important to this thesis is the *type of control* performed by the control systems in question. We are interested in two types of control: *Stabilizing* and *Sequential* control[1].

A stabilizing controller is assigned to maintain a certain property in its operating environment. A trivial example is a thermostat that measures the temperature inside a room and then turns a heater on or off to maintain a predetermined temperature.

A sequential controller ensures that a sequence of actions are executed properly in an environment, until the system has reached some goal. An example of such a system is a chemical process plant where, for example, a tank must have a certain

[1]In control theory a third type, *tracking control*, is often studied. We will not consider this type.

temperature and pressure before a chemical is inserted into the tank. That is, first the temperature and pressure have to be set, then the chemical can be inserted. We will argue that, from the perspective of this thesis, that stabilizing and sequential control are fundamentally different, and that solutions to problems of control systems described in the next section, need to be different for the two types of control.

1.1 Problem

As more safety critical systems are automated, designers of control systems are experiencing an increasing demand for safe and optimal operation of their systems. This can ideally be handled by precise mathematical modeling of the control system, formal verification of safety and optimality criteria and extensive (exhaustive) testing of the system. However, for the types of applications that are in the focus of this thesis such methods can only be used with difficulty, if they can be used at all. The reason for this is *complexity.* For autonomous systems this means that the operating environments are too complex for mathematical modeling. It is also literally impossible to formally verify safe and optimal operation in every conceivable contingency, for example, for an unmanned aerial vehicle. For industrial process controllers a major problem is *size.* No formal verification techniques available can handle programs consisting of 1'000'000 lines of code, yet such controllers are considered to be small by control engineers in industry. An argument for the use of formal methods is that if the controller is *designed* in a modular way, every module could be verified by itself, and the integration of the modules verified as a last step. Unfortunately, as argued by Dr. Kevin Passino in the preface of [Fodor, 1998], there are no systematic approaches to design of discrete/hybrid systems available that would facilitate this kind of modeling and analysis, as there are (an abundance of) for continuous systems. Moreover, it is hard to economically justify the very time-consuming (re-) modeling process. This is, in particular, a problem for *legacy systems*, that is, large well-functioning systems that perhaps have been developed incrementally over decades. It is hard to motivate expensive re-design of systems that are working well (but perhaps not optimally or maximally safe), and that even are market leading (see Chapter 4 for an example of such a system). The typical solution to this in industry is to tweak subsystems, to make them perform better, and to extend the existing control program. "Verification" is then typically done by simulation.

The problems addressed in this thesis can be summarized with the following two questions:

- How can control engineers handle the increasing demands for safety and optimality of control systems, in settings where the systems themselves or their operating environments severely restricts the possibility of precise mathematical modeling?
- How can the problem above be solved with minimal introduction cost, that is, minimal cost for introducing new technology?

1.2 Execution monitoring

The topic of this thesis is *execution monitoring* which is a technique that addresses the two problems stated above. The basic idea is that a device (the execution monitor) has access to the environment in the same way as the controller, and that the monitor tracks the execution of the controller and its effect on the environment. This idea is not at all new, for example, the idea of feedback control in Control Theory (which predates the area itself) is a step in this direction. In Computer Science a similar first step was taken with the advent of debuggers in the early '60s. In Artificial Intelligence (AI) the idea was first applied to the control system of the mobile robot "Shakey" at SRI around 1970. Today, the idea has evolved into a sophisticated set of techniques which include "Fault-Tolerant Control" in Control Theory [Blanke *et al.*, 2000], "On-Line Steering" in Computer Science [Gu *et al.*, 1997], and "Mode Identification and Reconfiguration" in AI [Williams and Nayak, 1996] (these particular techniques will be discussed at more length in Chapter 3).

We adopt the (abstract) view that an execution monitor is an entity in a system that observes the execution of the system (we will be more precise in Chapter 2). In Schroeder [1995] the following seven areas of functionality of execution monitors (or, *on-line monitors*) are identified:

- **Dependability** includes monitoring fault tolerance and safety.
- **Performance enhancement** includes dynamic system configuration, dynamic program tuning, and on-line steering.
- **Correctness checking** is the monitoring of an application to ensure consistency with a formal specification.
- **Security** monitoring to detect attempts of security violations such as illegal login or attempted file access.
- **Control** includes cases where the monitor system is a part of the environment, possibly to provide computational functionality.
- **Debugging and testing** employs monitoring techniques to extract data from an application being tested.
- **Performance evaluation** uses monitoring to extract data from a system that is later analyzed to assess the system.

This framework will be discussed at more length in Chapter 3.

1.3 Model-based execution monitoring

In the AI-sub-area of "Model-Based Reasoning" (see e.g. [Hamscher *et al.*, 1992]) the fundamental idea is to develop domain-independent devices for, e.g., diagnosis and simulation. Such devices are used in a particular domain by giving them a model of the domain. We adopt this methodology in this thesis and aim at the development and study of domain-independent execution monitoring engines constructed to work properly if a model with certain properties is given as input. The execution monitoring task is performed by comparing measurements from the environment with predictions made by the model. We are concerned with detecting discrepancies between the measurements and the predictions, in a first step. The next step is then to find a *cause* for a detected discrepancy. In model-based diagnosis this would typically involve identifying a physical component of the system that has failed and is a potential cause of the discrepancy. In execution monitoring, however, we are interested in discrepancies between the actual and predicted *execution* of the controller, and, thus, the causes are often more easily described on a more abstract level than on the component-level of the system. For example, in Chapter 5 we will discuss how various *stability criteria* of systems may be monitored. Finally, as a step towards increasing autonomy of a system, we study principles for automatic recovery from detected and classified discrepancies.

From the standpoint of Schroeder's framework we are interested in **dependability**, **performance enhancement** and **correctness checking**.

1.4 Overview of the thesis

One of the ambitions of this thesis is to structure and describe the problem of execution monitoring which is of interest to computer scientists, control theorists and engineers, and AI researchers and practitioners. In addition we propose methods for design and analysis of execution monitors. Thus, in Chapter 2 a detailed account of the concepts used in the rest of the thesis is presented. The concepts are put into a framework that could serve as a design methodology for execution monitors. There we also consider the research issues involved in this thesis. The framework is also used in Chapter 3 where the related work is presented.

As mentioned above we separate stabilizing and sequential control systems. The execution monitoring schema for sequential control, *ontological control*, is described in Chapter 4. An implementation and a first set of experiments on a real industrial process controller is also described there. In Chapter 5 the schema for stabilizing controllers, *stability-based execution monitoring*, is presented. As "stability" is a well-studied subject in both control theory and computer science, we review that work. However, we will argue that for autonomous systems, the existing approaches are not sufficient. We introduce a new notion "*maintainability*" that relaxes the previous notions, and develop algorithms for analysis, controller synthesis, and execution monitoring of maintainability.

Chapters 4 and 5 are central to this thesis, as the techniques introduced there are also applied in various settings in the subsequent chapters.

In Chapter 6 we study how a wide-spread modeling and verification formalism, *Hybrid Automata* can be used for execution monitoring. We construct an execution monitoring engine and show how a hybrid-automata domain model can be translated into a representation suitable for that engine. That is, how a model useful for execution monitoring can be *synthesized* from a specification. We also show how ontological control and stability-based execution monitoring can be applied in this setting.

In Chapter 7 we present a well-studied formal control framework, *the Situation Calculus/*GOLOG *framework,* which is a logical framework, and describe how execution monitoring can be applied there. Again, both ontological control and stability-based execution monitoring are used. Presentations of some technical details of this chapter, that are not directly relevant to the presentation are postponed to Appendices A and B.

In Chapter 8 we draw some conclusions and present some ideas for future work. In brief summary, we will go through the following steps in this thesis:

1. Construct a conceptual framework and a design methodology for execution monitoring, as well as discuss research issues involved (Chapter 2).

2. Review existing work on execution monitoring in the light of the conceptual framework (Chapter 3).

3. Develop and study two different "paradigms" of execution monitoring: Ontological control (Chapter 4) and Stability-based execution monitoring (Chapter 5).

4. Apply the two paradigms in two formal frameworks: In the Hybrid Automata framework (Chapter 6) and in the Situation Calculus/GOLOG framework (In Chapter 7 and the Appendices).

1.5 Contributions

To the best of our knowledge, there are no systematic accounts of execution monitoring spanning computer science, control theory, and AI. In fact, there are very few examples where researchers reference work in other areas than their own. The attempt to bridge the gaps between research disciplines studying execution monitoring is one of the main contribution of this thesis. The idea that execution monitoring can be regarded as a research area *per se*, is not very common in the literature. Execution monitors are more often entities that are added to a system as a link between a reactive control-program execution mechanism and more deliberative mechanisms such as planners, learning devices etc. However, the success of the model-based execution monitoring system "Livingstone" [Williams and Nayak, 1996] in AI provides an argument that this is a research issue that should be explored to a greater extent. Briefly, Livingstone is a model-based execution monitoring system used in NASA's Deep Space probes (see an account of this and other sophisticated approaches in Chapter 3).

Below, we discuss the contributions of each chapter of the thesis on a more detailed level:

Chapter 2: Although all the concepts presented in this chapter have been studied in detail by various research communities, there has been no attempt to find general constituting principles for execution monitors. We present a number of dimensions of execution monitoring and propose a design and analysis methodology.
Chapter 3: In this chapter, we attempt to discuss and compare approaches to execution monitoring in computer science, control theory, and AI.
Chapter 4: Ontological control was introduced by George Fodor in [Fodor, 1995, Fodor, 1998]. In this chapter we formalize, generalize and extend his work. We also present an implementation and a first set of experimental results on an actual industrial process system.
Chapter 5: The stability notion has received a large amount of attention both by the control theory community and the distributed-systems sub-area of computer science. However, it is not clear how their approaches should be applied to autonomous systems, where the disturbances of the operating environment are common phenomena. In this chapter we introduce a new stability-like notion, maintainability, suitable for these systems. We formally relate this notion to the existing ones, and provide algorithms for analysis, synthesis and execution monitoring.
Chapter 6: There is a problematic gap between the *specification* of a system, and the *implementation* and *execution* of the same system. In this chapter we attempt a novel approach of bridging this gap by taking a wide-spread specification formalism and transforming it into a formalism that can be used for both ontological control and stability-based execution monitoring.
Chapter 7: We believe that logic-based approaches to modeling systems provide many insights that can be used when applying other modeling techniques. We spend this chapter applying ideas and techniques introduced in earlier chapters to the Situation Calculus/GOLOG framework, a well-known logic-based modeling technique used in the area of "Cognitive Robotics". This framework was not designed to accommodate a number of techniques discussed in this thesis, so we extend the framework to be able to model stability, to give a logical account of detection of discrepancies, and to implement a general recovery mechanism.
Chapter 8: In this chapter we conclude the thesis and sketch a number of future paths that seem to be feasible, interesting and important.

1.6 Papers

The work presented in this thesis is based on a set of papers, consisting of published, refereed material, and unpublished manuscripts intended for publication in the future. We organize these by the Chapters in which they are used followed by a complete bibliography:

Chapter 2: The functional view of execution monitoring was introduced and analyzed in Bjäreland [1999a].
Chapter 4: The theory in this chapter is loosely based on the paper Bjäreland and Fodor [1998]. The implementation was done by Per Lewau for his Master's Thesis

[1999]. The application and the results are reported in [Bjäreland and Fodor, 2000].
Chapter 5: This work is an extension of the paper [Nakamura *et al.*, 2000].
Chapter 6: Preliminary work on execution monitor synthesis was reported in [Bjäreland, 1999a]. Closely related work on controller synthesis can be found in [Bjäreland and Driankov, 1999].
Chapter 7: The work on stability is to a large extent based on work by Bjäreland and Haslum [1999] and [Nakamura *et al.*, 2000] (the presentation in this thesis is an extension of that paper, with new examples, a clarified theory and new proofs). The extension of Pinto's framework to handle surprises was reported by Bjäreland and Pinto [2000]. The execution monitoring architecture was introduced by Pinto and Bjäreland [2001] and the recovery strategy (MT) in [Bjäreland, 1999b].

1.6.1 Bibliography

[Bjäreland and Driankov, 1999] M. Bjäreland and D. Driankov. Synthesizing discrete controllers from hybrid automata - preliminary report. In *Working Papers of the AAAI Spring Symposium on Hybrid Systems and AI*, Stanford, CA, USA, March 1999.

[Bjäreland and Fodor, 1998] M. Bjäreland and G. Fodor. Ontological control. In *Working Papers of the Ninth International Workshop on Principles of Diagnosis (Dx'98)*, Sea Crest Resort, N. Falmouth, MA, USA, May 1998.

[Bjäreland and Fodor, 2000] M. Bjäreland and G. Fodor. Execution monitoring of industrial process controllers: An application of ontological control. In SAFEPROCESS 2000 [2000].

[Bjäreland and Haslum, 1999] M. Bjäreland and P. Haslum. Stability, stabilizability, and GOLOG. Unpublished, August 1999.

[Bjäreland and Pinto, 2000] M. Bjäreland and J. Pinto. Handling surprises in logics of action and change. Unpublished manuscript, 2000.

[Bjäreland, 1999a] M. Bjäreland. Execution monitor synthesis for hybrid systems – preliminary report. In *Proceedings of the Fourteenth IEEE International Symposium on Intelligent Control (ISIC'99)*, Boston, USA, September 1999.

[Bjäreland, 1999b] M. Bjäreland. Recovering from modeling faults in GOLOG. In *Proceedings of the IJCAI'99 Workshop: Scheduling and Planning meet Real-time Monitoring in a Dynamic and Uncertain World*, Stockholm, Sweden, August 1999.

[Lewau, 1999] P. Lewau. A prototype of an ontological controller. Master's thesis, Linköping Studies in Science and Technology, Linköpings universitet, April 1999. No. LiTH–IDA–Ex–9949.

[Pinto and Bjäreland, 2001] J. Pinto and M. Bjäreland. An architecture for execution monitoring. Submitted to the *Seventeenth International Joint Conference on Artificial Intelligence (IJCAI '01)*, Seattle, US, August 2001.

[Nakamura *et al.*, 2000] M. Nakamura, C. Baral, and M. Bjäreland. Maintainability: a weaker stabilizability-like notion for high-level control agents. In AAAI '00 [2000].

Chapter 2

Conceptual framework

2.1 Introduction

In this Chapter a conceptual framework for execution monitoring is described. This framework will be used both as a guideline for design of execution monitors used in subsequent Chapters, and as a framework for facilitating comparisons between various approaches to execution monitoring. The core of the framework is an extraction of five functions that constitute an execution monitor: **Situation Assessment**, where the execution monitor computes the current state of the system, given the measured inputs, **Expectation Assessment** which concerns computing the predicted current state, **Discrepancy Detection** where the actual and predicted states are compared and discrepancies between the two are identified, **Discrepancy Classification** where detected discrepancies are classified by what *caused* them, and **Recovery** where the execution monitor try to force the controlled system back to normal operation.

2.1.1 Overview

In Section 2.2, a glossary for concepts used in this thesis is provided. Different systems, types of control actions, and properties of model representation formalisms that influence the possibilities of performing execution monitoring are discussed. In Section 2.3 we explore various definitions of execution monitoring proposed in the literature. A definition inspired by these definitions is proposed and examined. We also distinguish between meta- and object-level execution monitoring. In Section 2.4 we discuss interesting research issues involving execution monitoring, primarily to place the contributions in Chapters 4, 5, 6, and 7 into a proper research context. Finally, in Section 2.5 we present a systems-theory view of execution monitoring, which is used as a common formal framework for the results in Chapters 4, 6, and 7.

2.2 Closed-loop control systems

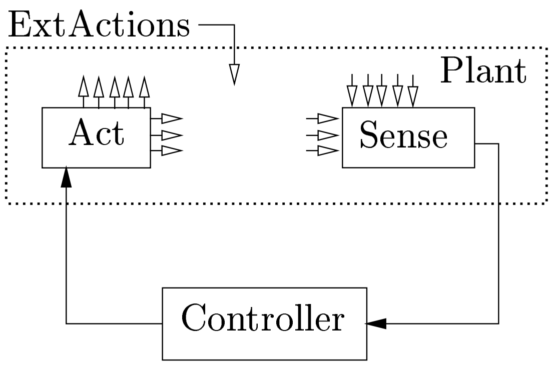

Figure 2.1: A closed-loop control system.

The systems of interest in this thesis are *discrete closed-loop control systems* depicted in Figure 2.1. Such systems consist of a mechanism called a *controller* which, given some input, generates an *actuator invocation signal* or *control action.* The control action is executed by *actuators* that are in direct contact with the plant. There is also a possibility of observing the behavior of the plant via *sensors*, which send signals back to the controller.

Typically in control theory, there is a distinction between two different types of control: *stabilization* and *tracking.* Stabilization concerns maintaining the outputs of the plant within a given set of states (the reference "point"). For tracking, the given set of states may vary (the reference point is time variant). Tracking control is not considered in this thesis.

In this thesis we focus on stabilizing control as well as *sequential control*, which could be viewed as a third type of control. In sequential control the control goal is to force the plant into a *goal state* with the highest priority possible. This can correspond to plan execution systems, where the execution is successful if the given goal is reached. However, in industrial process control it is common that the goal state is trivially satisfied during the entire execution of the sequence of actions and that the important issue is to successfully execute the actions in the given order. For example, in a chemical process system there may be a sequence where a tank must have a certain temperature before a chemical is introduced and then the pressure in the tank has to have a certain level before a second chemical is added. The goal is that the system should be ready for "normal" execution. This is also the case in the application described in Chapter 4.

In Section 2.4 we will discuss differences between stabilizing and sequential control and argue that they require different execution monitoring strategies.

We assume that control is governed by *control programs*, which are instructions that are executed by the controller. Formally, we view a control program as a relation between sensor inputs and control actions. A system where there is no feedback to the controller is called an *open-loop* control system.

This generic view of a control system in its abstract form is common in both control theory and AI. In AI, a control system often contains a *planner* which is a system that *synthesizes* a control program (or, plan). Conventional AI planners, such as Fikes and Nilsson's [1971] STRIPS , Chapman's [1987] TWEAK, and Blum and Furst's [1997] GRAPHPLAN, can be viewed as control-program synthesis programs. For execution, it is possible that the plans may be used for open-loop control, if they are executed without any sensor feedback. If the intended use of the plans are closed-loop control, it is necessary to add some monitoring mechanism to handle feedback to the controller. For such systems, the controller in Figure 2.1 can correspond to a plan execution mechanism. There are other AI systems where the synthesized control program or the planner itself considers sensor information while controlling, and for those cases the controller in Figure 2.1 may depict the entire planning system. This issue will be discussed in Chapter 3.

In control theory the well-studied *feedback control* systems, for example PID controllers [Faurre and Depeyrot, 1977], provide an instance of the class of closed-loop systems.

Plant states are assumed to be sampled with a certain time interval. There are a number of problems associated with sampling intervals of too long a duration, but we bypass such problems in this thesis by assuming that the sampling intervals can be set to be sufficiently short. This may appear as an unrealistic assumption, but in our experience (from the application described in Section 4.5), discrete state changes can be captured within sampling frequencies of 50-100 Hz.

From the perspective of the controller, the plant can, statically, be defined as the space of possible sensor value vectors. That is, the world to the controller is exactly defined in terms of what it can sense. It is, however, possible that control programs depending on the sensor value vectors can be constructed only with difficulty, that is, it is easier to construct the control program from more intuitively appealing aspects of the plant than from the input signals. Thus, for many sensory systems, for example systems with *vision sensors* (e.g. cameras), the control programs use some *abstraction* of the sensor inputs. For example, instead of using the large amount of data a vision system produces every sampling instance, the control may only be based on *changes* between frames. It is also possible that the control program is defined in terms of particular *objects* such as blocks or cars on roads. It is then necessary to map sensor inputs to such objects before the controller can be invoked. In control theory, such mappings may be trivial, if the control is defined directly in terms of the sensor signals. If the system is defined in terms of its "state-space" the mappings may not be as straight-forward. The current trend in process control is to use an object-oriented software architecture (see for example [Carpanzano *et al.*, 1999]. For such systems the sensor-to-representation mapping may be non-trivial. In AI there may be a considerable leap between sensor signals and the internal representation used by the controller. We henceforth use the term *state variable* to denote an aspect used by the controller. Examples of such aspects are temperature, pressure, and position or color of an object. Thus, the value of a state variable may directly correspond to a sensor value (as for temperature and pressure), or can be the result of a non-trivial computation on sensor value vectors (as for the position of an object on a map computed from sonar readings).

2.2.1 Control actions

In this thesis, we will be concerned with actual and expected effects of control actions to the extent that it is necessary to discuss the concept and to expose some of the intricate issues that it involves. From our perspective a control action is an invocation signal from a controller to an actuator. Thus, we use the phrases "executing the control action A" and "sending the action invocation A to the actuators" interchangeably. It is possible that in the control program a control action is guarded by a *condition*, that is, that the control action is only executed if the corresponding condition is satisfied.

A control action may be an invocation of a toggle action that can be assumed to be instantaneous (for example, flicking a switch), or invocation signals that are temporally extended (for example, invocation of a PID controller or throwing a rock up in the air where the effect of the control action, that is, that the rock hits the ground, can be sensed only after some time period). For the latter we distinguish between *energized* and *ballistic* control actions (this distinction is due to Nilsson [1994b]). By an energized control action we mean an action whose guarding condition must be true for the action to be executed, that is, an energized action is executed as long, and only as long as its condition is satisfied. A ballistic action invocation is instantaneous while the action itself is temporally extended. For example, the action of throwing a rock into the air may be seen as an instantaneous action invocation, but the action itself continues after the invocation. It has been argued that energized actions are special cases of ballistic actions [Lee and Durfee, 1994], and even if this is the case, the distinction is of importance for *prediction* of effects of control actions, which is central to execution monitoring: If every controller action is energized, then state transitions occur exactly when some actions stop being executed and some new ones are started. This does not hold in general for ballistic actions.

2.2.2 Representation

The concept of *models* has rendered a fair amount of attention in many different scientific disciplines, and a number of different meanings are associated with the concept. We equate models with prediction mechanisms, and thus, will adopt the following generic definition:

Definition 2.2.1 (Model)
A model is an entity that enables predictions of effects of a control system.□

In the context of Figure 2.1, this definition implies that a model is a predictive mechanism of the whole closed-loop system. Examples of modeling "formalisms" satisfying the definition include

- differential/difference equations,
- state transition diagrams,
- finite automata of various kinds,
- temporal logics,
- planning operators, and
- stochastic processes.

This definition may be criticized for various reasons, but for the purpose of execution monitoring we argue (in Section 2.3) that it is sufficient.

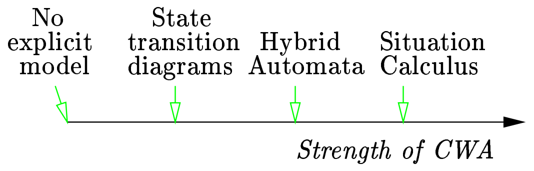

Figure 2.2: The Closed World Assumption dimension of modeling formalisms.

Clearly, there is a model underlying the design of every control program, since a control system designer necessarily has some expectations on the effects of action executions. The question is how "well" the model is represented. By "well" we mean how *precise* the modeling formalism enforces the models to be, with respect to the possible predictions from the model. We call this dimension "Strength of Closed World Assumption" (SCWA)[1], depicted in Figure 2.2, and it is one of the central issues of this thesis. The SCWA dimension has been introduced independently in [Falkenroth, 2000] as one of the three dimensions of the functionality of simulation data systems. The reader should note that SCWA does not measure the relation between a model and the plant that it models. It is a measure on *formalisms*, and the precision a formalism enforces when it is used for modeling. We will say that that the SCWA *increases* (*decreases*) as we move to the right (left) on the SCWA axis. Formalisms on the far right (left) end on the axis are said to have *high* (*low*) SCWA.

It should be noted that the term "representation" is intended to encompass both syntax and *execution semantics* of a modeling formalism. Execution semantics is the description of how the evolution of a system manifests itself in terms of the syntax, that is, precise notions on when the system is in a particular state, when actions are executed, and when state transitions occur. For some formalisms, such as some versions of differential/difference equations and the situation calculus, the execution semantics is clear, since it is assumed that the system dynamics is perfectly

[1]We are using the notion CWA in a more abstract way than in the original work by Reiter [1978], or any of the extensions (see [Łukaszewicz, 1990] for a survey).

modeled[2]. For other formalisms, such as hybrid automata, a "simulation semantics" is specified, which does not specify how sensor signals from outside the simulator are to be handled. For state transition diagrams there are numerous ways in which execution semantics can be defined. If we look at the state transition diagram in

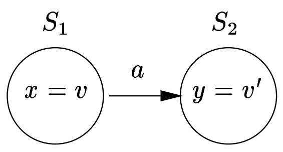

Figure 2.3: A small state transition diagram.

Figure 2.3 it will be interpreted as "if $x = v$ (we are in state S_1) and we execute the control action a, we will *eventually* end up in a state where $y = v'$ (S_2)". Two semantic questions that immediately arise are what the value of y is in S_1 and what the value of x is in S_2, since neither of the variables are explicitly represented in those states. If we, for example, assume that *nothing changes unless the change is explicitly represented*[3] we would have a semantics where the value of y in S_1 is unknown but that $x = v$ in S_2. Another assumption could be that *nothing is explicitly represented unless it changes* which would imply that $y \neq v'$ in S_1 and that $x = v$ in S_2. However, in this thesis neither of the two last assumptions are used for state transition diagrams. The consequence of this is, in our example, that the values of y at S_1 and of x at S_2 are unknown. We state the assumption we do use as follows:

Assumption 2.2.2 For a transition due to a control action in a state transition diagram the control action is started when all explicitly represented propositions in the starting state are true, and eventually ended in a state where all the explicitly represented proposition in the ending state are true.□

Assumption 2.2.2 does not restrict the length of a control action, or give a precise meaning of when state transitions occur. These are problematic issues that are discussed in Chapter 4. In Reiter's terminology [Reiter, 1978], Assumption 2.2.2 is an instance of the *Open World Assumption.*

In the "low" end of the SCWA axis we find "no explicit model" and the higher the SCWA gets, the more precision the modeling formalism enforce, as depicted in Figure 2.2. Examples of "no explicit model" are Brooks subsumption architecture [Brooks, 1991] and PENGI [Agre and Chapman, 1987]. Note that underlying these "reactive" control architectures, there certainly are models, but not *explicitly represented* models that can be used and manipulated as "1st class citizens". This

[2]By "perfectly modeled" we mean that the dynamics of the system as well as all potential disturbances and faults are incorporated in the model.

[3]This is an assumption used in many approaches to *Logics of Action and Change* and is discussed in Chapter 7.

means that no predictions about the behavior of the closed-loop system can be made. These systems are of no immediate interest to us, but in Chapter 4 we will describe experiences from a project where a model was (semi-automatically) extracted from a (data-flow based) control program that lacked an explicit model. The modeling formalism used in that Chapter is based on state transition diagrams which enforces predictions only on the expected next state of a control action executed in a particular state, and not on the evolution of state variables over time, following Assumption 2.2.2.

If we now look at the high end of the SCWA in Figure 2.2, we find "complete" modeling formalisms, that is, formalisms that enforce precise modeling of how state variables change over time due to control actions. This means that we can predict the value of every state variable at any time point. In Chapter 7 we use GOLOG and underlying theories in the Situation Calculus to represent this other extreme. Other examples of very precise formalisms are *planning* formalisms (such as STRIPS [Fikes and Nilsson, 1971]), where the exact effects on every state variable are modeled.

A final example of precise formalisms is differential/difference equations, which are the main modeling tool for conventional continuous systems.

From the definition of a "model" it is clear that our purpose for using models at all is to make predictions about effects of control actions. We have informally defined the SCWA as a measure on the precision of the predictions than can be made from the formalism. This measure is also interesting from other perspectives, and here we list some of them:

Procedural/Declarative representation

With a slight simplification it is possible to argue that industrial-type controllers basically consist of *production rules*.[4] This implies that if there is no explicit model (the far left on the SCWA axis) the production rules constitute all the knowledge representation we have. This can be called a purely *procedural representation of knowledge*, in the sense that the only kind of reasoning that can be performed is "what do I *do*, given this input?". For example, consider the following controller (that actually could be the controller for the state transition diagram in Figure 2.3):

$$\begin{gathered} x = v \Rightarrow a \\ y = v' \Rightarrow a' \end{gathered}$$

We assume that the control actions (a and a') are executed whenever the corresponding preconditions ($x = v$ and $y = v'$, respectively) are satisfied by inputs to the controller. Obviously, there is no information about the expected effects of the control actions a and a'. The only thing we can say is that whenever $x = v$ is satisfied then a is executed, and whenever $y = v'$ is satisfied, a' is executed.

[4]Industrial-type controllers necessarily need to react very rapidly to sensor inputs and this places strong limitations on the potential use of memory or more intelligent control mechanisms than those used by productions rules. For example, all the PLC languages in the IEC 1131-3 standard [Lewis, 1997] are of this kind.

However, in some cases it is possible to do somewhat better: In Chapter 4 we will extract a richer representation from a control program. The extraction algorithm is based on engineering intuitions, that is, an assumption that the programs are written according to good engineering practice. Thus, there is information about expected effects of actions implicit in the control program. In terms of SCWA such programs are a little higher on the score than the lowest possible SCWA.

At the high end of the SCWA axis we have *declarative formalisms*, where knowledge about the dynamics of the modeled system is given independent of any specific reasoning system. This does not mean that the controller there is different from the controllers to the far left, but that there exists explicitly represented knowledge about the controlled system that the controller or an execution monitoring system can use.

There has been a discussion between advocates of procedural and declarative knowledge representation in AI almost for as long as the area has existed (see for example [Rich and Knight, 1991]). We do not believe that these two concepts provide a dichotomy. Instead the SCWA dimension shows that, for a given control system, the representation used is procedural (or declarative) to a certain degree.

Explicit plant representation

Another aspect, related to the previous, is the precision with which the controlled system (the plant) is modeled. In conventional control theory (see for example Faurre and Depeyrot [1977]) the standard approach for controller design (and automatic synthesis of controllers) is begun by modeling the open loop system, that is, finding the differential equations governing the input/output behavior of the plant. By introducing some desired property (such as a reference value for stabilizing control, or a goal), mathematical methods are then used to construct the controller (such as the Ziegler-Nichols rules for PID controllers). In the context of this Section, the input/output behavior precisely represents the plant.

On the low end of the axis, there is an approach that is not uncommon in industrial process control: A control program is written by an engineer with good domain knowledge, and the program is then simulated and/or tested until a certain level of reliability is reached. In this case the plant dynamics is taken into consideration during controller design, but is never explicitly represented.

If the representation of the plant is done using state transition diagrams, following Assumption 2.2.2, we do not have as high precision as in the conventional control theory case, but higher than in the industrial process control case.

Action type expressivity

Another aspect of the SCWA dimension is the *expressivity of representation of action types*. For the "no explicit model" end of the dimension, there is no real notion of an action type; actions are executed when their preconditions are satisfied by inputs to the controller. There is no point in distinguishing between, for example, context-dependent actions, nondeterministic actions, concurrent actions, actions with side-effects, instantaneous actions, actions with duration, and so on.

As the SCWA increases, it is possible to see that such distinctions do become possible, and interesting. For example, for hybrid automata, the same action can switch the system into different continuous behaviors, depending on the context in which it is executed.

At the high end of the axis, the notion of action types is important. The complexity of the action types and their possible interaction greatly influences the robustness (and complicates the modeling) of the prediction mechanisms associated with systems that use such actions. The logical mechanisms necessary to model, for example, concurrent actions are quite different from those required to handle sequential (non-concurrent) actions (see Chapter 7 and Appendix B).

Modeling pragmatics

The task of using the formalisms becomes increasingly difficult as the SCWA is increased. It is, for many reasons, desirable to have as complete a model as possible, but when the complexity of the control system increases, the feasibility of modeling it precisely decreases radically. This implies that a control designer must handle a difficult trade-off: The model should be as rich as possible, but the modeling task should not be too time consuming. The intermediate formalisms in Figure 2.2 (state transition diagrams and hybrid automata) do not enforce complete models, even though it is possible to construct such models in them. This means that a designer has the possibility of adding as much information as he or she can.

2.3 Execution monitoring

There is no consensus in the literature on how to define execution monitoring. For example, Dean & Wellman in their book "Planning and Control" [1991] choose the following definition:

> "In robot planning, the process of sensing the state to influence subsequent action is called execution monitoring."

Another, more unorthodox, definition is due to Saffiotti [1998]:

> "Thus, monitoring does not consist in matching the observed execution against some "expected" course; rather, it should distinguish situations for which the information in the plan is *relevant* from situations for which it is not."

Saffiotti's definition is motivated by the fact that there is no crisp notion of "next state" that can be generated from his representation. His approach will be discussed at more length in Chapter 3.

In this thesis we turn to a definition that will be of more use to us as a methodology for designing execution monitors as well as for comparing existing approaches to each other.

Following De Giacomo *et al.* [1998] we choose the following augmented definition[5]:

[5]The original definition by De Giacomo *et al.* excludes the *classification* of discrepancies.

Definition 2.3.1 (Execution monitoring)
Execution monitoring is an agent's process of identifying discrepancies between observations of the actual world and the predictions and expectations derived from its representation of the world, classifying such discrepancies, and recovering from them.□

The definition of "model" (Definition 2.2.1) is motivated by this definition of execution monitoring. It should also be noted that "the execution of control actions" is not mentioned in definition 2.3.1. The reason for this is so that other techniques from related areas such as Estimation Theory, System Identification, and Model-Based Reasoning, that not necessarily *directly* concern controller execution, remain consistent with the definition we propose, and can provide valuable insights and perspectives to execution monitoring.

2.3.1 Meta- vs. Object-level execution monitoring

In the early seventies, the distinction between meta-level and object-level execution monitoring appeared in the AI literature. The ground breaking work on the STRIPS planner by Fikes and Nilsson [1971] at SRI was applied to their robot Shakey. In [Fikes *et al.*, 1972] they proposed a special *plan execution mechanism*, PLANEX, with the purpose of executing and monitoring STRIPS plans. PLANEX was a separate architectural entity in the robot control system that detected discrepancies and performed plan repair using Nilsson's *triangle tables* (see for example [Nilsson, 1982]).

At the same time, also at SRI, Munson [1971] suggested that monitoring should be incorporated in the plan, that is, that the robot plan should be interleaved with *monitor formulas*. The monitor formulas should detect, and possibly classify, discrepancies, and alert a re-planning mechanism if necessary.

Meta-level execution monitors, as we choose to name execution monitors that are separate entities connected to the controller of a system, with access to the inputs and outputs of the controller are represented in the literature by the work of Sacerdoti [1977], Broverman and Croft [1988], Ambros-Ingerson and Steel [1988], Hammond [1990], Beetz and McDermott [1994], Lyons and Hendriks [1995], Simmons *et al.* [1997], and by Earl and Firby [1997]. A body of work on object-level execution monitoring, where execution monitoring is interleaved with the control program, and handled by the control language constructs, can be found in work by Munson [1971], Doyle *et al.* [1986], Abramson [1991], Musliner *et al.* [1995], and by DeGiacomo and Levesque [1999].

It is difficult to find a crisp distinction between meta- and object-level execution monitoring. Tentatively, we propose that the distinction can be made in terms of how the model relates to the execution monitor. That is, if execution monitoring mechanisms are included in the model of the closed-loop system, we have object-level execution monitoring, and if the the execution monitoring mechanisms can manipulate and reason about the model, we have meta-level execution monitoring. One can imagine a system that cannot be distinctly classified as being either meta-

or object-level according to this criterion. However, the distinction above suffices as a basis for the subsequent discussions of this thesis.

There are advantages and disadvantages to the two approaches. Object-level execution monitoring follows, in a sense, a control theoretic tradition where a major goal is to describe the entire control system within one theoretical framework. Conventional control systems can be described as systems of differential equations, which incorporate various forms of execution monitoring and diagnosis in itself. In similar spirit, if monitoring information is included in plans, we have planning systems where only the plans generated by the planner have to be analyzed. The system is independent of the architecture, which supposedly implies easier and more coherent possibilities of modeling, design and analysis.

Meta-level execution monitoring, on the other hand, solves the problem of plan execution more in line with traditional applied AI and robotics. A major goal there is generality and modularity (and not necessarily coherent theories). However, there does exist theoretically oriented AI research on combining different formalisms: For example in the area *Hybrid Knowledge Representation.* See for example [Chittaro *et al.*, 1993].) An execution monitoring system should be applicable in a wide variety of domains, and therefore computations should be organized so that a minimal number of changes are necessary when switching domains.

The distinction between object- and meta-level execution monitoring is made from a modeling (or, knowledge representation) point of view. In this thesis we are in favor of meta-level execution monitoring, though we will discuss approaches to object-level execution monitoring in some detail.

2.3.2 A functional view of execution monitoring

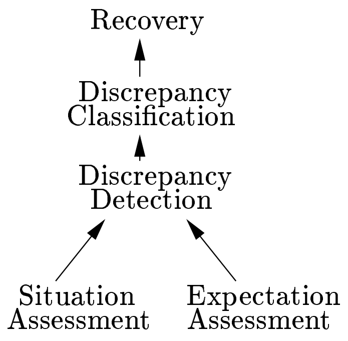

Figure 2.4: The functional view of execution monitoring.

It is possible to extract five distinct *functions* that constitute an execution monitor from Definition 2.3.1 (see Figure 2.4).

- **Situation Assessment**: A (partial) function from inputs to the controller, to states in the model. Answers the questions: "In what state is the system right now, w.r.t. the model"?

- **Expectation Assessment**: A (partial) function from states and control actions to new states according to the model. Embodies the expected effects of control actions. Answers "In what state is the system expected to be"?
- **Discrepancy Detection**: A function that from situation and expectation assessments decides whether a discrepancy has occurred. Answers "Did something not go as expected"?
- **Discrepancy Classification**: A function that yields an explanation (or possibly, a cause) of a detected discrepancy. Answers "What went wrong"?
- **Recovery**: A function that returns a sequence of control actions to be executed. As side-effects the recovery function may change the model, the control program, or it may simply start an alarm to alert an operator. Answers "How do we continue the execution"?

These five functions have been addressed in the AI literature since the emergence of the field in the 1950's and each function still contains open problems. Below, we will briefly explain the functions, but postpone more detailed discussions on related work to Chapter 3.

Situation assessment is about mapping the sensory signals to states in the model, given a model at a different conceptual level than the sensory signals. In Control Theory an example of situation assessment would be mappings between the measurement space and the state space of a system. In AI and Robotics, an important part of a situation assessment mechanism could be *anchoring*, for example studied by Coradeschi and Saffiotti [2000], which is defined as the process of creating and maintaining the correspondence between symbols and sensor signals that refer to the same physical object.

In control theory, as in many AI type execution systems, situation assessment is occasionally not addressed as a particular problem. Then, it is common to assume that the mapping from inputs to states in the model is straightforward. The problem of handling, e.g., noisy sensors is not defined as a situation assessment problem, but rather as a discrepancy detection problem, where the noise needs to be filtered for an accurate detection of "real" discrepancies.

By *expectation assessment* we mean temporal prediction, that is, a mechanism that can predict the next state (or, sequence of states) of the system given the previous state(s) and (possibly) a control action. In certain systems expectation assessment and situation assessment are indiscernible functions, where the inputs and the predictions are both used to compute the most probable current state (this is for example done with *observers* [O'Reilly, 1983, Misawa and Hedrick, 1989] and Kalman filters [Kalman, 1960, Sorensen, 1985] in control theory).

Discrepancy detection is the task of finding discrepancies between the current state and the expected state, and in its simplest form it consists of comparing the input signals to a reference vector which may yield a *residual* (a description of the possible discrepancy). We would like to view it as the task of comparing the result from the situation assessment to the result of the expectation assessment. For example, discrepancy detection could be the logical task of finding out if the logical formula describing the assessed situation is simultaneously satisfiable with the formula generated by the expectation assessment (this approach is used in Chapters 4, 6, and 7).

By *discrepancy classification* we mean the task of finding a *cause* for a detected discrepancy. This could imply identifying faulty components, actions (or plan segments) that failed, or, as we do in Chapters 4 and 7, distinguishing between disturbances and inadequate models (faulty expectations).

Finally, we turn our attention to *recovery*, which is the most difficult function to characterize. Recovery may be interpreted as meaning a number of different things, but the most important is to ensure that a system that has experienced a discrepancy continues to control the plant in a satisfactorily manner. That is, the most important output of a recovery function is a new (sequence of) control action(s). This may, however, not be enough. For example, in a setting of component-based diagnosis, if the classification function has identified a faulty component, let's say a valve that is stuck closed, it is necessary to take that information into consideration during continued control. This may imply that the controller needs to use some other valve to achieve the same effects as before the valve broke. In this case the controller (control program or plan) needs to be changed, as a side-effect of the task of finding a new appropriate control action. If the classification has identified that the current model is inadequate, on the other hand, it may be necessary to recover by updating the model.

2.4 Research issues

In this Section we will look a some of the research issues that arise in the context of execution monitoring of discrete closed-loop systems. The issues we examine are

- **Purpose**: The purpose of execution monitoring may, at a low level of granularity, be one of the areas of functionality presented in Chapter 1. In more detail, the research issue here is to identify different types of discrepancy classifications that may be of interest, and examine the circumstances under which they can be used. We will look at two types: *Ontological Control* and *Stability-Based execution monitoring.*

- **Design/Synthesis**: Following the tradition of model-based reasoning, we are interested in developing domain-independent engines for execution monitoring. The engines then take the domain-dependent model of the closed-loop system as an input. The research issue of interest here is how to *design* such engines and how to *synthesize* models (from scratch or from already existing models in other formalisms) appropriate for certain types of execution monitoring.

- **Analysis**: For complex systems we cannot expect to find a general solution to the problems posed in Chapter 1. It is clear that only subclasses of systems can be subject to any type of execution monitoring. An important research issue is therefore to *analyze* and find *tools for analysis* of the applicability of given execution monitoring paradigms.

- **Application**: Even though a user-friendly set of generic synthesis and analysis tools may be available, it is typically non-trivial to apply the theories to an actual system. It is, thus, of importance to apply the theories to systems in systematical ways and to document and discuss the application in detail.

We will now more closely discuss how the four research issues are addressed in the rest of this thesis.

2.4.1 Purpose

A natural question that arises for execution monitoring is: *What should be monitored?*. A coarse set of possible answers to that question can be found in Chapter 1, in the list of the seven areas of functionalities. But even if we state that we would like to detect deviations from predictions by a model, there are still questions to be answered. First, as mentioned in Chapter 1, we have assumed that the operating domains are too complex or uncertain for precise mathematical modeling. This implies that there always will be discrepancies between predictions by the model and plant signals. However, not all such discrepancies require action. As an example we can take a UAV where a certain trajectory is predicted by a model and where a gust of wind suddenly moves the UAV away from that trajectory. Within certain bounds the system should be able to recover from such a discrepancy itself, while an abrupt change of location outside the bounds should be treated more seriously.

The particular type of classification and recovery schema we will investigate is *Benign/Malignant* (BM) classification and *Model Tuning* (MT) recovery. The basic idea is that every detected discrepancy should be classified as either being benign (in the sense "harmless") or malignant (in the sense "possibly harmful"). For benign discrepancies we then recover by using the current model, perhaps with re-planning to get the system back on track. For malignant discrepancies we also need to rely on some form of re-planning, but we will also modify the model to make sure that the same discrepancy will not appear again. This modification technique will be called *model tuning*.

In this thesis we will look at two particular instantiations of BM classification: *Ontological Control* and *Stability-Based execution monitoring.*

Ontological control

Ontological control is an execution monitoring paradigm developed for sequential control, primarily for industrial process controllers. In such controllers a sequence of states has to be traversed. If an expected state does not materialize when it should, the controller typically invokes a recovery action to force the system to redo earlier steps of the sequence. For example, if it is important that a tank has a certain pressure before a chemical is introduced in a chemical process plant, and the pressure suddenly drops, the controller will force the system to an earlier state in the system where the pressure is re-established. Ontological control emerged from the need to detect and handle *infinite recovery cycles*, that is, infinite sequences of states where the controller is trying to recover from a discrepancy. One cause of such cycles is *violations of ontological assumptions*. That is, all sufficiently complex control systems rely on some underlying unmodeled assumptions that are necessary for the validity of the controller. Such assumptions (*ontological assumptions*) may be violated. By introducing the notion "perfect sub-model" we are able to detect such violations and distinguish them from benign discrepancies.

In Chapter 4 we will develop a theory of Ontological Control and report a set of experimental results from an application of the theory. In Chapter 7 we will reformulate ontological control in the Situation Calculus framework.

Stability-based execution monitoring
For stabilizing controllers we have an almost completely different situation than for sequential controllers. Cycles of states are not a problem; it is in fact the goal of the control, in the sense that the controller is forcing the plant to remain within a certain set of states, call it S. Now, any discrepancy will either make the system remain within S, or take it outside. In the first case we have a benign discrepancy (since the control goal is still achieved), and in the second the stability is violated and we have a malignant discrepancy.

In Chapter 5 we investigate the stability notion and present a weaker notion: "maintainability" that appears to be more applicable to robotics applications than "stabilizability". These notions are applied in Chapters 6 and 7.

2.4.2 Design and synthesis

We are interested in designing generic *engines* for execution monitoring that are able to handle models in formalisms with a certain form. For ontological control the principles behind this are studied in Chapter 4, and for stability-based execution monitoring the principles are studied in Chapter 5.

A problem we have discussed earlier in this Chapter is how to handle procedural models. For industrial sequential controllers it is not uncommon that the only model available is the control program itself. For control programs in general, it is impossible to generate a model with a higher SCWA (as argued above) than the program itself. But, for real process controllers more systematic program design schemas are used, that is, for well-engineered systems it is possible to extract models with more predictive power than the program has itself. We show how this can be done semi-automatically in Chapter 4.

Another problem arises when we have a *specification* of the closed-loop system. There are typically a number of different ways in which a specification can be implemented, so the goal is then to synthesize a model of the closed-loop system that can handle any control program implementation. In Chapter 6 we show how a state-transition model can be synthesized from a Hybrid Automata specification.

2.4.3 Analysis

Neither ontological control nor stability-based execution monitoring apply to all controllers. This means that an important research issue is to investigate under what conditions the theories can be applied.

For ontological control we (in Chapter 4) identify a set of restrictions on sequential controllers that are necessary for successful application of the theory. The restrictions are

- *Perfect sub-model*: Violations of ontological assumptions imply that the current model of the system is inadequate. To detect this we cannot, of course, use the model itself as a reference point. The solution to this is to assume that some sub-system is working perfectly (such as the actuators), and that the model contains a sub-model of this sub-system. This sub-model is then used as the reference.

- *Non-logging*: In the model, there may be state transitions that only are used for tracking the execution of one particular action. This is called *logging*. Logging prevents precise distinction between states during execution, and therefore makes prediction very difficult. In ontological control we assume that the system is non-logging.

- *Energized actions*: To be able to precisely predict when state changes occur we require that all actions are energized (which are defined above).

For stabilility-based execution monitoring we have a slightly different problem. In principle we can apply this paradigm to any controller; the form of the models are not very important. However, we require that the system is *stable* with respect to some set of states, S. This is the analysis problem of the paradigm. In Chapter 5 we define "stability" and a new, more general, concept, "maintainability", that is more applicable for active databases and robotics than "stability". We also present some algorithms for the analysis of maintainability.

2.4.4 Application

Ontological control is applied to an industrial process controller in Chapter 4, where we discuss an implementation and some experimental results. In Chapter 7 we let most issues discussed earlier be placed in a single framework: GOLOG. GOLOG is a high-level agent programming language that uses an explicit model of the operating environment. One of the most useful properties of GOLOG is that it is based on formal logic which provides a formal semantic theory and a formal modeling language (the Situation Calculus, SitCalc). Thus, correctness proofs of GOLOG programs for particular environments can be constructed deductively. In Chapter 7 and the two Appendices A and B we address the following issues:

- *Restrictions for ontological control*: We show how a SitCalc model can be analyzed to guarantee successful application of ontological control.
- *Stability analysis*: We are interested in deductively proving that a SitCalc theory is stable. We therefore develop a formulation of the stability concept introduced in Chapter 5 in the logic itself, which is presented in Appendix A
- *Controller synthesis*: We show how a *stabilizing* GOLOG program can be automatically synthesized from an unstable (Markovian) SitCalc theory, that is, the closed-loop system involving the SitCalc theory and the synthesized controller is stable.
- *Situation and expectation assessment*: These two functions are already implemented in the Situation Calculus and GOLOG. The first is assumed and the second is handled by *goal regression* [Reiter, 1991].
- *Discrepancy detection*: Both ontological control and stability-based execution monitoring perform discrepancy detection on *states* of the system. In the Situation Calculus it is often convenient to reason with sequences of executed actions (*situations*), similar to the Ramadge-Wonham theory of Discrete Event Systems (see for example [Kumar and Garg, 1995]). We address the problems of execution monitoring both with the state view and the situation view and implement ontological control and stability-based execution monitoring.

  We are interested in deductively proving that a discrepancy occurs. The problem with SitCalc is that it is to the very far right of the SCWA axis, and that discrepancies entail an inconsistent logical theory. We solve this problem by relaxing the SCWA (technically we relax the inertia assumption of SitCalc) to allow for discrepancies. In the terminology of the *Logics of Action and Change* area, we handle *surprises.* We extend SitCalc to handle surprises in Appendix B.
- *Discrepancy classification*: This is not particularly problematic for either ontological control or stability-based execution monitoring.
- *Recovery*: we develop a technique for changing the SitCalc model with respect to detected malignant discrepancies. The idea is to make the change parsimonious, that is, we want to ensure that exactly the same discrepancy does not occur again.

2.5 Abstract formal framework

In this Section we will present a summary of an abstract theory of execution monitoring, a theory that will be expanded upon in Chapters 4, 6, and 7.

An execution monitor system is a twelve-tuple

$$\langle P, A, Cont, S, M, Sit, Exp, D, Det, L, Class \rangle,$$

with the following interpretation:

- We assume that we have a set of *plant states*, P, that represent vectors of sensor readings as well as vectors of other signals that are inputs to the controller.[6]
- The possible control actions belong to a set A.
- A *controller* is a relation $Cont$, possibly between plant states or sequences of plant states (depending on how much of the history the controller takes into account) and control actions. A sequence of plant states, in this context, is assumed to correspond to one dynamical development of the controlled system. A controller defined as $Cont : P \rightarrow A$ is said to be *deterministic*.
- The set of possible *models* of the closed-loop system is denoted $\mathcal{M}$ with $M \in \mathcal{M}$.
- We assume that we have a set of *internal states* (or, simply, *states*), S.
- A function Sit from sequences of plant states, internal states and models to internal states is called a *situation assessment function*. Intuitively, the sequence of plant states represent the development of the system including the current sensor reading and the sequence of internal states the previously "visited" internal states.
- An *expectation assessment function*, Exp, is function that predicts the state given a sequence of previous states, a control action, and a model.
- We assume that we have a set of *discrepancies*, D (this set may for example contain states or sequences of states).
- A *discrepancy detection function Det* maps the current situation (perhaps in terms of the previous state, the executed action, the actual current state, and the predicted state) to discrepancies or to *null* if there is no discrepancy between the two states.
- We assume a set of discrepancy classifications, L and that L always contains a symbol *none* that marks cases where no known explanation for a discrepancy exists.
- A *discrepancy classification function*, $Class$ maps discrepancies to discrepancy classes.

A *recovery function* can be defined to map the current situation and execution monitoring system to a new execution monitoring system and a control action.

[6]From a control theoretic perspective, we thus assume that only the output from the plant is what is modeled, that is, that we do not assume to know anything about the internal state of the plant. This implies that the observable/unobservable dichotomy is meaningless in our setting.

Chapter 3

Related work

3.1 Introduction

Work on execution monitoring can be found in the AI, Computer Science, and Control Theory literature. The amount of papers concerning execution monitoring as a particular problem in its own right is fairly low. Instead, it is more common to find either papers describing entire control architectures, where execution monitoring is one component, or papers on particular techniques that may be used for the purpose of execution monitoring. This poses a problem when surveying the "area" of execution monitoring. In this chapter we have chosen to present a number of representative approaches to execution monitoring in Control Theory, Computer Science and AI. After that we discuss techniques developed in various subareas of AI and control theory that can be adapted to the framework described in Chapter 2.

3.2 Control theory

In this section we intend to convey some of the basic ideas, as well as some recent approaches, of monitoring, supervision and diagnosis from the control theory community. Although the amount of existing work on these topics is huge, the focus is typically different from ours. As described in the previous chapter we assume that the controller is given and we wish to add an execution monitor engine to the controller, and then feed the engine with a model of the closed-loop system. In control theory the problem would usually be to start by modeling a plant in terms of constraints (typically as differential/difference equations) and synthesize a controller from the model and a control goal, where the goal should contain the aspects that should be monitored. However, below we will present a control theoretic framework where it is assumed that the architecture contains meta-level reasoning mechanisms for diagnosis and supervision.

3.2.1 Fault detection and identification (FDI)

Fault detection and identification (FDI) is a problem that has been studied quite extensively by control theorists for the past thirty years[1](see e.g. Basseville and Nikiforov [1993], Frank [1990], or Chen and Patton [1999]). A difference between that work and the approaches discussed above is that FDI has mainly focused on continuous systems, while the mainstream approaches in AI have been concerned with discrete systems (Saffiotti's work [1998] being an example of an exception).

Situation assessment is, typically, not considered a particular problem, while expectation assessment would correspond to (the prediction part of) *state estimation* with *observers*, e.g. O'Reilly [1983], and Misawa and Hedrick [1989] or Kalman filters [Kalman, 1960, Sorensen, 1985], that yield an estimated current state $\hat{x}$. In a simple version of FDI the estimated current state is compared to a reference value, r, and generates a *residual*, ρ, for example by computing $\rho = |r - \hat{x}|$. Fault detection is then a comparison between the residual and some threshold where, for example, $-\epsilon \leq \rho \leq \epsilon$ could indicate that no discrepancy has occurred, for a suitable ϵ, and that $-\epsilon > \rho$ and $\rho > \epsilon$ indicates two distinct classes of discrepancies. There is an abundance of variants on this theme (see the recent survey [Chen and Patton, 1999] for details).

The large amount of work on FDI for continuous systems has almost exclusively addressed fault detection in the open-loop part of the systems, that is, the controller has not been considered in the FDI process[2]. Due to this the work on FDI in general does not apply to the problems addressed in this thesis.

3.2.2 Fault-tolerant control

In a recent paper, Blanke *et al.* [2000] presented the novel control paradigm *Fault-Tolerant Control*. This term denotes a set of techniques that were developed to increase plant availability and reduce the risk of safety hazards. We will not go into the details of these techniques in this thesis, instead we present an abstract framework of control (introduced in [Blanke *et al.*, 2000]) that demonstrates the relation between classical control, robust and adaptive control and fault-tolerant control.

Constraints of a dynamical system are functional relations that describe the behavior of the system. In our terminology, a set of constraints is an open-loop model of the system. In control theory the most common way of describing constraints is with differential/difference equations. A set of constraints, C, for a plant defines a *structure* S and *parameters* θ of the system. For example, for the constraint

$$\dot{x} = Ax + Bu$$

the parameters are the matrices A and B, and the structure is a first-order linear system. A distinguished set of constraints are called *control objectives*, and are denoted O. A function from plant output to a control order u (which is one of the

[1]The origin of *model-based* FDI is usually credited to Beard's PhD thesis [1971].

[2]There are, of course, exceptions to this. See for example [Jacobson and Nett, 1991].

inputs to the plant) is called a *control law.* The class of plausible control laws is denoted by $\mathbb{U}$. We can now define the *control problem*:

Control: Solve $\langle O, S, \theta, \mathbb{U}\rangle$, that is, find a control law in $\mathbb{U}$ that achieves O while satisfying C (i.e. while not violating the constraints).

If we now assume that we do not know the parameters precisely, for example due to uncertainty, but only a set, Θ, of plausible parameter values. the problem is then to achieve O under constraints with structure S and whose parameters belong to Θ. Two approaches to this problem can be defined: *Robust control* where the discrepancies over Θ are minimized, and *Adaptive control* where a parameter $\hat{\theta}$ is estimated before the control problem is solved. Formally,

Robust control: Solve $\langle O, S, \Theta, \mathbb{U}\rangle$, that is, find parameters in Θ and a control law in $\mathbb{U}$.

Adaptive control: Solve $\langle O, S, \hat{\theta}, \mathbb{U}\rangle$ where $\hat{\theta} \in \Theta$ is estimated.

Next, faults are taken into consideration. From the point of view of this framework, a fault is a change in the given constraints. This means that given a fault, a diagnosis computation should yield estimated sets of possible structures, $\hat{\mathbb{S}}$, and parameters, $\hat{\Theta}$. Formally, the task of *Fault-tolerant control* can then be described as

Fault-tolerant control: Solve $\langle O, \hat{\mathbb{S}}, \hat{\Theta}, \mathbb{U}\rangle$.

In our functional framework from Chapter 2, fault-tolerant control involves situation and expectation assessment as well as discrepancy detection and classification. The former can be handled with state estimation techniques such as Kalman filters, while the latter involves diagnosis techniques. However, the work on fault-tolerant control has yielded two distinct specializations for discrepancy classification that fit nicely into our BM (Benign/Malignant) framework.

In the benign case a diagnosis system estimates the actual constraints, that is, generates $\hat{S}$ and $\hat{\theta}$ and then solves the control problem. We call this *fault accommodation* and formally we have

Fault accommodation: Solve $\langle O, \hat{S}, \hat{\theta}, \mathbb{U}\rangle$.

If it is not possible to do fault accommodation, we cannot hope to handle the problem by estimations that take the entire plant (including parts that have been diagnosed to be faulty) into consideration. Then we have a malignant case and try to solve the control problem by using constraints *not* involving faulty parts. Formally,

Reconfiguration: Find $\Sigma \in \mathbb{S}$ and $\tau \in \Theta$ (where $\mathbb{S}$ and Θ are restricted to non-faulty parts of the system) such that $\langle O, \Sigma, \tau, \mathbb{U}\rangle$ has a solution.

The difference between fault accommodation and reconfiguration is that for the latter the input-output relations between the controller and the plant have not changed, while this is a possibility in the second case.

The most general problem in this framework involves monitoring of the control objectives. If neither fault accommodation nor reconfiguration is possible, we can still relax the control objectives. This is called *supervision* and is described as

Supervision: Monitor the triple $\langle O, S, \theta \rangle$ to determine whether the control objective is achieved. If this is not the case, and the fault-tolerant control problem does not have a solution, then find a relaxed objective ? $\in \mathbb{O}$ and $\Sigma \in \mathbb{S}$ and $\tau \in \Theta$ such that the relaxed problem $\langle ?, \Sigma, \tau, \mathbb{U} \rangle$ has a solution.

Blanke *et al.* suggest that a fault-tolerant system with supervision should be implemented as a hybrid (discrete/continuous) system where diagnosis and controller synthesis is done within the continuous loop and fault accommodation, reconfiguration, and supervision is handled by meta-level mechanisms using automata representations. This is natural since those three tasks normally produce discontinuities in the state trajectory of the system.

3.3 Computer science

The idea of adding probes to a computer program to monitor its execution probably originates with the advent of debuggers in the early 1960's. Today, advances in high performance computing, communications, and user interfaces are enabling developers to construct increasingly interactive applications. This implies a large increase in the complexity of such systems, and a lot of work has been put into development of tools that can support the management of large-scale parallel codes. In this section we will examine a sophisticated tool for *on-line monitoring* and *steering* of such software systems: *Falcon* [Gu *et al.*, 1997].

The notion *program steering* can be defined as "*the capacity to control the execution of long-running, resource-intensive programs*" ([Gu *et al.*, 1994]). The control of the execution may involve control over parameters of the program, usage of modules (it may for example be of interest to be able to switch between different algorithms at run time), and usage of hardware resources (for example load balancing of processors). Program steering is in [Gu *et al.*, 1994] described as consisting of two distinct tasks: monitoring program or system state and enacting program changes made in response to observed state changes. From our point of view the monitoring task is equivalent to situation assessment, while steering involves expectation assessment, discrepancy detection and classification, as well as recovery. However, program steering may be used for other purposes than handling discrepancies, for example, managing data output in an efficient manner. The reader is encouraged to read the survey [Gu *et al.*, 1994] for a more detailed account of the vast literature in the area.

Another related concept from Computer science is *self-stabilization* (introduced by Dijkstra [1974]) which has been thoroughly investigated by the distributed systems community. A self-stabilizing system can achieve its goals from any initial state and under the influence of any transient fault (unexpected state change). This notion is strongly related to the work on maintainability, in Chapter 5, and will be discussed below.

3.3.1 Falcon

The Falcon project [Gu *et al.*, 1997] was initiated as a step towards *distributed laboratories*, where multiple and distributed end users can collaborate with each other as if they were co-located in a single laboratory setting. In this distributed setting, the users can solve complex scientific or engineering problems by jointly experimenting with multiple coupled and distributed simulations all of which may be monitored and steered by the users and with algorithms.

Falcon is a step in this direction, which consists of a set of tools that jointly support three tasks: The first task is on-line capture, collection and analysis of the application-level program and performance information required for program steering and for display to end users. The second task is the analysis, manipulation, and inspection of such on-line information, by human users and/or programs, based on which decisions concerning program steering may be made. The third task is the support of steering decisions and actions, which typically result in on-line changes to the program's execution. In our functional framework, the first task corresponds to situation assessment, the second task to expectation assessment and discrepancy detection and classification, and the third to recovery.

In the implementation of Falcon, the original application source code is compiled together with a sensor and steering specification. This corresponds to object-level monitoring since the extra information is added directly to the source code. During run-time, the instrumented application program is running with a trace data collector and analyzer and a user interaction controller. The analyzed trace data is sent to a filter mechanism and then displayed to the user. The user has the possibility of steering the execution of the program via the interface which controls a steering server that can control the execution of the running application. The running application can also access the steering server for algorithmic steering actions.

3.3.2 Self-stabilization

In his seminal paper [Dijkstra, 1974] Dijkstra introduced the notion *self-stabilization* of distributed systems. The problem posed was whether it was possible to construct a protocol for a set of interconnected processors that would maintain a certain set of global states. Dijkstra proposed a solution for a class of systems[3]. The problem was intensively studied during the 80's as a part of the development of *fault tolerant*

[3]The class was token rings of non-prime length. In the CACM paper he did not provide proofs of his suggestion, stating that they where left as an exercise for the reader. 11 years later the proofs were published in [Dijkstra, 1986].

systems. It was clear already to Dijkstra that self-stabilization was a difficult problem, but the fact that the problem is undecidable was not shown before 1994 (see [Abello and Dolev, 1997] for the full length paper).

The formal definition of self-stabilization can be stated as follows:

Definition 3.3.1 Let P be a set of states for a system S. S is *self-stabilizing* to P iff the following two conditions hold:

- P is closed in S, i.e. every state in a computation of S that starts at a state in P, belongs to P.
- Every computation of S has a finite prefix such that the following states belong to P.

$\square$

The first condition is called *Closure* and the second *Convergence.* In control theory the notion of "stability" is a core concept, and it comes in many flavors. One flavor of particular interest to us, from [Özveren *et al.*, 1991], will be presented in detail in Chapter 5. That approach coincides only with the convergence condition of self-stabilization, thus relaxing the concept.

3.4 Artificial intelligence

The first application of execution monitors in the AI literature appear, as mentioned in Section 2.3.1, with the application of the STRIPS planner to Shakey the robot [Fikes *et al.*, 1972]. The planning community has strongly focused on developing planners that, given a "plant model" (in terms of planning operators) an initial state, and a goal, generates a sequence that, according to the model, will lead the plant from the initial state to the goal. When such planners have been put to use in applications, it has been necessary to develop execution monitors to handle contingencies not predicted by the model. Two examples of such systems are IPEM [Ambros-Ingerson and Steel, 1988] and GRIPE [Doyle *et al.*, 1986], where the first contains a meta-level execution monitor, and the second represents object-level execution monitoring. These systems will be presented below in sections 3.4.1 and 3.4.2.

There are other approaches to robot control where the formalisms used lack any adequate notion of "predicted next state". This is clearly problematic from our point of view, so instead of relying on predictions other means have to be developed to assess the execution of such system. In Section 3.4.3 we review a robot control system based on *fuzzy logic* ([Saffiotti, 1998]).

The last, and most recent, approach to execution monitoring developed in the AI community has inspired a lot of ideas in this thesis. The system, *Livingstone* ([Williams and Nayak, 1996]), was developed at NASA AMES as a central component in the "New Millennium Remote Agent" (NMRA) architecture, and it was tested on a space probe in July 1999. Livingstone is a model-based execution monitoring system which acts as a low-level situation assessment and recovery mechanism with no direct connections to the planning/scheduling system in the NMRA architecture. This approach is discussed in Section 3.4.5.

3.4.1 IPEM

Integrated Planning, Execution and Monitoring (IPEM) is a framework developed by Ambros-Ingerson and reported in [Ambros-Ingerson and Steel, 1988]. The idea is to use techniques similar to those of the partial-order planner TWEAK [Chapman, 1987] to generate a plan which is then executed. One difference in the planning process, compared to TWEAK is that actions (planning operators) may be complex, that is, they may in themselves be plans. This relates to one important body of work on planning: Hierarchical Task Nets [Sacerdoti, 1977].

Initially, a partial plan is provided consisting only of the initial state and the goal. Planning is then performed by identifying *flaws* and transforming the plan according to the corresponding *fixes*:

FLAW	FIX
Unsupported Precondition: A precondition of an action in the plan is not supported by (or, does not logically imply) a postcondition of an earlier action.	**Reduce:** Establish a support from an action already in the plan, or add a new action.
Unresolved Conflict: A precondition of an action is clobbered by a postcondition of an earlier action.	**Linearize:** Place the clobberer in the plan where it does no harm.
Unexpanded Action: An action that is not primitive, that is, represents a plan, exists in the plan.	**Expand:** Replace occurrences of the action with the plan it represents.

During execution the plan is then elaborated (fixed) if a monitor detects errors (flaws). For monitoring we have the following examples of flaws and fixes:

FLAW	FIX
Unexecuted Action: An action is ready to be executed.	**Execute:** Execute the action.
Timed Out Action: No further effects are expected to come about as consequence of an execution of an action.	**Excise Action:** Remove the action from the plan.
Redundant Action: There exists an action whose postcondition does not support the goal or any precondition of other actions in the plan.	**Excise Action:** Remove the redundant action from the plan.

There are two more flaws that can be detected during execution: **Unsupported Range** and **Unextended Range**. The first occurs when support by a postcondition for a precondition is broken, that is, when that support is no longer a consequence of the current observations. The fix suggested is to *re-instantiate* the precondition by some other part of the postcondition. For example, assume that a precondition $clear(x)$ is initially supported by a postcondition $clear(\mathbf{a})$ in action A and A is executed. Now, if $clear(\mathbf{a})$ is not true after the execution, we have an unsupported range flaw. If A also has $clear(\mathbf{b})$ as a postcondition, then by re-instantiating x to $\mathbf{b}$ we have fixed the flaw. The second occurs when there exists a support by a postcondition in action A for a precondition, but some other action occurs *before* A in the plan and also provides the same support. In this case the fix is to let the earlier action provide the support, possibly making A redundant.

From the perspective introduced in Chapter 2, it is possible to view IPEM as a meta-level execution monitor, since the flaws-and-fixes system is separate from the actual planning and plan execution system. Moreover, we can see that the problems of situation and expectation assessment are not addressed. The flaws and fixes defined for plan generation are not particularly interesting to us; they are supposed to be invoked prior to execution. Discrepancy detection in IPEM consists of checking whether any of the flaws has occurred, which will require a detection function that is different from the abstract function Det defined in Section 2.5, since it depends on the current plan and the current set of support instantiations, as well as on the current and expected state. It is unclear from [Ambros-Ingerson and Steel, 1988] how the Timed Out Action flaw is detected. It is hinted that there is notion of execution completion time for actions, but how that notion is used in the framework is not discussed. Clearly, discrepancy classification is integrated in the detection mechanism (IPEM detects particular classes of discrepancies), which in the abstract theory means that the set of discrepancies D is equal to the set of classifications L, and, thus, that $Class$ can be the identity function.

If we view the currently executing plan as a controller and the set of support instantiations (and possibly the execution completion time information) as the model, recovery corresponds to fixes, where the controller may be changed for certain flaws (like Excise Action) and the model may be changed for others (such as Unsupported Range).

The ideas from IPEM have been extended by Knoblock [1995] to include sensing and concurrent execution of non-conflicting actions.

3.4.2 GRIPE

As an example of an object-level execution monitor we choose GRIPE (Generator of Requests Involving Perceptions and Expectations) [Doyle *et al.*, 1986]. From the object-level point-of-view the authors argue that there are four tasks that must be accomplished by an execution monitor:

- **Selection:** The task of determining which actions (or which effects of actions) in the plan require monitoring.
- **Generation:** The task of determining which sensors to employ to verify assertions, and the expected sensor values.
- **Detection/Comparison:** The task of recognizing significant events on sensors and comparing those events to the corresponding expectations.
- **Interpretation:** The task of explaining failed expectations.

The basic idea behind GRIPE is to use a special set of plan operators, *verification operators*, that are interleaved with the plan to perform run-time plan verification. The verification operators are automatically generated by using a sensor model and the plan. An important part of the generation and interleaving of verification operators is context (or, intent-) analysis. For example, if a robot arm is holding an object and is supposed to move the object to a particular position, the move action needs to be carefully monitored, while if the arm is not holding anything and is supposed to move to a resting position, the move action does not have to be as precise as in the previous case. Thus, the verification operators in the two cases should be different. The authors do not provide any general principle of analysis of context; that task is viewed as being domain-dependent.

It is easy to see the resemblance between the five functions in Chapter 2 and the four tasks described in [Doyle *et al.*, 1986]. GRIPE's notion of Generation corresponds to expectation assessment, and Detection/Comparison is a combination of situation assessment and discrepancy detection. Interpretation clearly corresponds to discrepancy classification. The differences between GRIPE and the framework is that GRIPE does not address recovery, and that the framework does not consider sensor selection. Sensor selection is important whenever sensing is a limited resource in a system, which we do not assume in this thesis. The intended application domain for GRIPE was the NASA/JPL Telerobot servicing the US space station, that is, a domain where basically everything is a limited resource.

3.4.3 Saffiotti's approach

An approach that is difficult to compare to many others in this thesis is the work by Saffiotti [1998]. He is interested in the problem of robot navigation and has chosen fuzzy logic as the basic framework. The system consists of a planner, and an execution system, where the planner uses *behavior templates* to produce plans. The behavior templates are planning operators with some extra slots, especially a slot that states the *goodness* of the particular behavior (a real number between 0 and 1) that quantifies the confidence of the expectation on the effects of the action. The planner is a simple regression-based planner that finds a plan and then gathers the operators into a set of fuzzy rules, that is, the operators are transformed to **if** *precondition* **then** *action* rules. The execution system then interprets the rules as fuzzy rules, so that the precondition of a rule is satisfied to a certain degree (degree of desirability). The goodness of an invocation of an action is then a function of the degree of desirability of the precondition and the goodness of the operator. Since it is possible that a number of fuzzy rules are all satisfied to some degree simultaneously, *action blending* is performed, which basically is concurrent execution of all actions with satisfied preconditions but to a degree proportional to the degree of desirability of the particular precondition.

In Saffiotti's system the following adequacy measures are monitored:

- The degree of *goodness* of the current goal, i.e. how well the current plan will achieve the current goal. For example, if a robot is in idle mode and receives a new goal, e.g. to move into room R5, the goodness of the plan of being idle should be very low, which the monitor should detect.
- The degree of *conflict* between behaviors, i.e. how well the involved behaviors interact. For example, a plan consisting of behavior $moveRight$ and $moveLeft$ to be executed concurrently, should have a high degree of conflict.
- The degree of *competence* of the plan in the current situation, i.e. to what degree the involved behaviors are active in the current situation. For example, if the preconditions of the involved behaviors are all satisfied to a low degree, then the activation level of all the actions is low, which is a sign of a low degree of competence.

One of the reasons why comparisons to other approaches are difficult for this approach is that there is no crisp notion of *state* in a fuzzy setting. However, the framework discussed in Chapter 2 does not have a problem characterizing the approach.

Situation assessment in fuzzy systems is defined in terms of membership functions, that is, functions that map sensor inputs to the degree of membership in a particular state. Expectation assessment is more complicated, and in Saffiotti's approach one can say that it is missing; there is no mechanism that predicts the next state of the system. As in IPEM (Section 3.4.1) discrepancy detection and classification is done simultaneously. Low degrees of goodness and competence, and high degrees of conflict are detected by computing the three degrees and comparing them to predefined threshold values. Recovery (or, repair in Saffiotti's terminology) is performed by representing a plan (a set of fuzzy rules) as a tree, and detecting problems with goodness, conflict, or competence in subtrees, and thereafter only replace "bad" subtrees.

3.4.4 Action-based diagnosis

In the AI sub-area "Model-based diagnosis" various diagnosis problems have been studied following the ground breaking work of Reiter [1987] and de Kleer and Williams [1987]. Most of this work is fairly recent, and can be differentiated with respect to the expressive power of the language used to model the domain (in the same sense as our SCWA dimension), how the notion of diagnosis is defined (e.g. in terms of fault models, sequences of actions, sets of abnormal components, or probabilistic criteria), how observations (measurements) are expressed, whether the diagnosis is on-line or off-line, and what aspects of diagnostic problem solving, beyond diagnosis, are addressed (e.g. recovery).

The work most strongly related to the topic of this thesis is "Action-Based Diagnosis" where the aim is to compute a sequence of actions (controllable or uncontrollable) than can explain a discrepancy (or, fault). In our terminology, this means that discrepancy classification is defined as finding a sequence of actions that explains (that is, is logically consistent with) an observed discrepancy. The most comprehensive account of this work can be found in [McIlraith, 1997, McIlraith, 1998], and similar work in [Thielscher, 1997, Baral *et al.*, 2000].

3.4.5 Livingstone

A fairly recent example of a model-based execution monitor is *Livingstone*, by Williams and Nayak, [1996], which is an important part of the *New Millennium Remote Agent* (NMRA) architecture developed at NASA Ames (see for example [Muscettola *et al.*, 2000]) for the purpose of spacecraft control. Livingstone's sensing component, *Mode Identification* (MI), uses a model to identify the most likely (in a probabilistic sense) spacecraft states and reports all inferred state changes to the controller (or execution mechanism, EXEC), enabling the EXEC to reason purely in terms of spacecraft states. Input to MI is sensor values and commands sent by EXEC to the real-time system. For example, a particular combination of attitude errors may allow MI to infer that a particular thruster has failed. EXEC is only informed about the failed state of the thruster, and not about the observed low-level sensor values.

The command component of Livingstone, *Mode Reconfiguration* (MR), uses a model of the spacecraft to find a least-cost command sequence that establishes or restores desired functionality by reconfiguring hardware or repairing failed components. MR is invoked by EXEC with a recovery request that specifies a set of constraints to be established and maintained. In response, MR produces a recover plan that, when executed by EXEC, moves the spacecraft from the current state (as inferred by MI) to a new state in which all the constraints are satisfied.

There is no real distinction in the MI component between situation and expectation assessment, and discrepancy detection, since mode identification is performed based on a normal-behavior model and a fault model. That is, the set of possible current states is chosen as the most likely one according to the normal-behavior model with a minimal number of faults. It is then up to EXEC to decide whether any reported faults call for special purpose (hard-coded) recovery actions, or for an invocation of MR (which performs recovery actions in a model-based manner). In our terminology, EXEC performs discrepancy classification and some recovery, and MR performs model-based recovery.

3.4.6 Other examples

An interesting approach to execution monitoring in the AI community is the work by De Giacomo *et. al.* That approach will be analyzed and extended in Chapter 7.

In the AuRA architecture [Arkin, 1990] a *homeostatic control subsystem* is connected to a robot's hardware interface, and monitors the internal conditions of the robot (which we will call the *configuration* of the control system in subsequent chapters). This is, in fact the only execution monitoring that occurs in AuRA, and motivation stems from an analogy to mammalian control systems that allows for dynamic re-planning in hazardous environments.

In Ferguson's *Touring Machines* [Ferguson, 1992] execution monitoring, in our sense, is the principle reasoning mechanism. A Touring Machine consists of three layers: A reactive layer, a planning layer, and a model layer. The reactive layer provides the system with fast, reactive capabilities for coping with events its higher level have not planned for or modeled. Whenever such an event occurs a rule is triggered, some action is executed and the model layer is alerted. The purpose of the planning layer is to generate and execute plans. This layer has some capabilities to detect plan failures, but decisions regarding recovery are taken by the modeling layer. In the modeling layer capabilities for reflection and prediction reside. The basic idea is that the modeling layer gets input from sensors, resource monitors and other layers and uses this input to construct a new model from a library of model templates. Reasoning from a model is then discrepancy detection, in our terminology, where the chosen model is used for expectation assessment. Recovery may then mean orders to the planning level to re-plan to the old goal, or planning from scratch to a new goal.

3.5 Techniques for execution monitoring

From the viewpoint of the framework presented in Chapter 2, we will now examine contributions of techniques, not necessarily designed for execution monitoring purposes, that can be used for the five constituting functions.

3.5.1 Situation and Expectation assessment

Successful approaches to expectation assessment have been developed in the framework of Partially-Observable Markov Decision Processes [Cassandra *et al.*, 1994], where the domain is modeled with partially observable stochastic processes, and probabilistic techniques are used to predict the behavior of the controlled system.

The Qualitative Reasoning (QR) community has focused on qualitative modeling, analysis, and simulation of physical systems. For example, a model of a system may only consist of the sign of the derivatives at particular time points (see e.g. [Dvorak and Kuipers, 1989, Dvorak and Kuipers, 1991]). Such information can be used to monitor a physical system, and an interesting recent approach to qualitative monitoring can be found in [Rinner and Kuipers, 1999], where a simulator produces an expectation assessment, and where a tracking system refines the model whenever the observations allow bounds on variables to be decreased. In this setting a discrepancy is an observation of a value of a variable outside its bounds, and this is not handled by the system.

Similarly to the work on FDI (Section 3.2.1) the work on monitoring in QR has focused on open-loop systems.

Another example is planning operators consisting of preconditions and effects, where estimation of the expected current state is performed in terms of plan projection or regression [Fikes *et al.*, 1972, Sacerdoti, 1977, Bjäreland and Karlsson, 1997].

3.5.2 Discrepancy Detection and Classification

There exists sophisticated discrepancy classification approaches for *simulation* purposes, for example [Hammond, 1990, Beetz and McDermott, 1994], which are semi-domain independent. Other examples of discrepancy classification approaches for planning systems are [Ambros-Ingerson and Steel, 1988, Knoblock, 1995]. In our group we have investigated the problem of classifying discrepancies distinctly as being due to external disturbances, or due to model inadequacies (faulty expectations) [Bjäreland and Fodor, 1998, Bjäreland, 1999b].

For classification of discrepancies a body of work exists in the area of *Model-Based Diagnosis* [Reiter, 1987, Struss, 1997] where, typically, a model of the *components* of a system is assumed, and a *diagnosis* is a subset of the set of components, which are presumably faulty. For execution monitoring purposes, the value of a component-based model is not clear, since it is the state transitions that are subject to diagnosis for EM.

3.5.3 Recovery

Recovery from discrepancies is often mentioned as an important component of control systems. It can for example mean *plan repair* as for Hammond [1990], Beetz and McDermott [1994], Ambros-Ingerson and Steel [1988], and Knoblock [1995], *re-planning* as for Fikes *et al.* [1972], and Sacerdoti [1977], or *control program elaboration* as for [Lyons and Hendriks, 1995].

A part of the Machine Learning community has studied the problem of "Learning Planning Operators" [Benson, 1996, Gil, 1992, Shen, 1989, Wang, 1994] where the basic idea is to observe the effects of executing planning operators and then use learning techniques to improve them. This is an interesting idea, but since the current approaches rely on multiple failures or experimentation, they are of limited interest outside a laboratory environment.

Chapter 4

Ontological control

In chapter 2 we introduced the notion "benign/malignant classification" and discussed its applications. In this chapter we will present one particular instance of BM classification named *ontological control.* Ontological control relies heavily on engineering intuitions about sequential process controllers and was introduced by George Fodor [Fodor, 1995, Fodor, 1998].

In modern industrial process controllers it is very difficult to construct complete mathematical models of the closed-loop system. This is due to complexity that arises as there typically are hundreds of programmable controllers (PCs) that influence each other, actuators, databases, and Ethernets. These entities are generally run as asynchronous concurrent processes. The set of PCs is often modularized so that a subset of PCs is responsible, for example, for one actuator system, and every PC in the subset has its own responsibilities, such as numerical computations or continuous control. It is not uncommon that such a subset of PCs is organized in a hierarchy. For such systems the term "sequential control" should be understood as the high-level task of forcing the system through a sequence of discrete state transitions. A state can, for example, represent the normal continuous control of the system, where previous states must materialize to fulfill conditions necessary for the continuous control. States occurring after that state in the sequence should materialize to safely stop the process. So, the primary role of the sequential controller may not be to achieve the "real" control goal (the quality goal of the process), but to ensure that the configuration of the closed-loop system permits that the "real" control can be performed.

The strong dependencies between modules and PCs in modules need to be considered by the PC programmer. This means that the sequential controller should be able to produce a control action in whatever state the system may be in, as well as it should ensure that the control sequence is promptly executed. Usually, contingencies are taken care of within the program and discrepancies are detected and recovered from by specialized hard-coded procedures in the program. However, in sufficiently complex systems the problem of *infinite recovery cycles* occurs. That is, the execution runs through the sequence as expected up to a point where a discrepancy occurs and the hard-coded recovery strategy sets in and forces the system to a state earlier in the sequence. Then the execution runs as expected again until the same discrepancy occurs again, and the recovery mechanism again is invoked, etc. We then have a cycle of states that is materialized *ad infinitum.* Ontological control was invented for the purpose of detecting and classifying such *systematic discrepancies*, as opposed to temporary disturbances.

When do these systematic discrepancies occur? Before we go into detail, the question can briefly be answered as follows: *When an expected state transition should have occurred but did not.* That is, if a control action is interrupted prematurely, and the plant state does not satisfy the expected next state, we have a disturbance. In the terminology of Ontological Control we have a discrepancy due to an *External Action* (EA). Such control situations (that is, when control actions are interrupted) are not at all uncommon in process control, and their occurrences can normally be explained with some other PC overriding the invoked control action (hence the term External Action). These discrepancies do not necessarily (or even usually) entail that something is wrong with the control system.

The more problematic case occurs when a control action is completely executed, but the expected next state does not materialize. This may imply that the problem is not in the closed-loop system, but in the *expectations* of the effects of the control actions. This holds when it is possible to precisely keep track of actual and expected state transitions, as well as a precise notion of action execution. That is, if we know that an action was executed as expected and a state transition occurred as expected, but the expected next state did not materialize, then it is very unlikely that an external action has occurred (since we will assume that the sampling interval is short enough for us to able to detect every state transition). Such discrepancies will be called *Violations of Ontological Assumptions.*

In this chapter we will begin by presenting the intuitions of violations of ontological assumptions more carefully, and then exemplify ontological control and problems related to it. Then we informally present the limiting assumptions of the theory in Section 4.4. The theory itself is developed along the lines of the abstract theory in Section 2.5. We formally show that the assumptions make it possible to distinctly classify any discrepancy as either being an external action or a violation of ontological assumption. In Section 4.5 we will describe how the theory was implemented and applied to a real process control system, ABB's STRESSOMETER. As it turned out the semi-automatically generated model used there did not satisfy the assumptions of ontological control, so an important issue addressed in that section is how to extract information from the model to make ontological control possible. In Section 4.6 we compare the original work by Fodor to the work presented here.

4.1 Violations of ontological assumptions

In any sufficiently complex control system there are always assumptions that are not checked by the controller, but that are necessary for the validity of the control system. We call such assumptions *ontological assumptions.* A simple example of this is a controller that invokes actions based on measurements of a voltage computed from a current under the assumption that the resistance is constant (according to Ohm's law). However, under varying temperature conditions, this assumption does not hold. Thus, the ontological assumptions may be *violated.* The detection and correct classification of violations of ontological assumptions, VOAs, and especially the distinction between disturbances and VOAs (or, the *disturbance decoupling problem* in FDI terminology) is important for industrial process control. It is a distinct possibility that VOAs cannot be detected by assessing output quality of the process only. Industrial process control systems are normally very robust, and even though the execution of the controller does not comply with the designer's intentions, it may still achieve its control goals. However, this means that undetected VOAs may make the system behave sub-optimally, such as in the case of infinite recovery loops, and that the life-span of the system is considerably shortened.

The fundamental question of ontological control is

> *Given a discrepancy, when can we distinguish between the cases when the discrepancy is caused by an external action (disturbance) and caused by a VOA (model inadequacies)?*

This chapter is devoted to giving a formal answer to this question. The reader should note that in the terminology introduced in 2.4, external actions correspond to benign discrepancies, and VOAs to malignant discrepancies. The next obvious question, "How do we recover from these classes of discrepancies?", is not addressed in detail in this chapter. The issue is discussed in Section 4.4.10, but mainly to argue that to find general principles of recovery we need to employ modeling formalism with a higher SCWA than what is used in this chapter. Such a formalism (the situation calculus) is introduced and applied in Chapter 7.

4.2 Two examples

In this section we will attempt to convey the basic intuitions behind ontological control by giving two examples. The first example is a diesel engine controller where we establish an informal account of VOAs, and the second example is a formal blocks-world example where a robot arm moves blocks from a store to two trolleys, which can represent sequential control, that is, achieving a goal by executing a sequence of control actions.

4.2.1 Diesel engine

The following example was introduced in [Fodor, 1995]. In figure 4.1 a closed-loop control system for a diesel engine is depicted. The speed controller determines the engine speed using two physical principles: When the engine has low speed, a proximity sensor reads the rotation speed from a teeth wheel placed on the engine shaft. The variable N_1 (in *rpm*) represents this value. This is an inaccurate measure, so when the diesel engine rotates with a speed larger than a specified limit N_{lim},

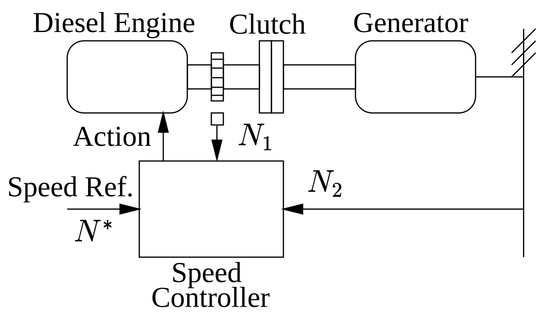

Figure 4.1: The diesel engine example

the speed is measured more precisely using the frequency of the generated electrical current[1]. The variable N_2 represents this value.

The goal for the sequential controller is to bring the engine to speed where "normal" continuous control can be used. This requires two distinct modes:

1. At start-up, the speed controller accelerates the diesel engine according to a predefined start-up trajectory, using the speed value N_1. This mode is used for speed values $N_1 < N_{lim}$.

2. When the diesel engine reaches the speed value N_{lim} the controller enters control mode 2, where N_2 is used. In this mode the goal is to maintain the reference speed N^*.

One ontological assumption present in this application is that *the diesel engine shaft and the generator shaft have the same rotation speed, since they are linked.* If this assumption is violated and the speeds of the two shafts are different, then the control schema presented above is invalid.

So, let us assume that this ontological assumption is violated, for example by assuming that the clutch is slippery. This implies that $N_1 > N_2$. Let us follow a control sequence for the diesel engine.

The controller starts the diesel engine in mode 1 using the predefined speed trajectory. When the controller observes that $N_1 = N_{lim}$ it switches to mode 2. However, at this moment, the controller starts using N_2 and observes that $N_2 < N_1 = N_{lim}$ which causes a switch back to mode 1. In mode 1 the controller again observes $N_1 = N_{lim}$ and consequently switches to mode 2, *etc.* **The system is stuck in an infinite recovery loop.**

[1]This is done according to the formula $N_2 = \frac{2 \times 60 \times f}{p}$ where f is the frequency and p is the number of magnetic poles of the generator.

4.2.2 Trolleys

The following pedagogical example has been used in both [Bjäreland and Fodor, 1998] and [Bjäreland, 1999b] to illustrate ontological control.

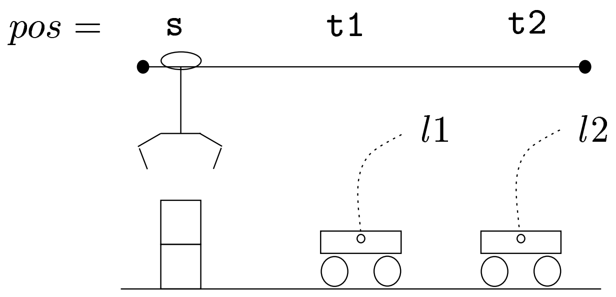

Figure 4.2: The trolley example domain.

In Figure 4.2 the plant with one actuator (the robot arm) and three sensors (the position of the robot arm and the two pressure sensors on the trolleys) is depicted. We control the robot arm that is supposed to move boxes from `s` to the trolleys `t1` and `t2`. The variable *pos* (for the position of the robot arm) takes values from the domain $D_{pos} = \{\mathtt{s}, \mathtt{t1}, \mathtt{t2}\}$, and the other two variables (that measure the number of boxes on each trolley), $l1$ and $l2$, that take values from domains $D_{l1} = D_{l2} = \{0, 1, 2\}$. We have three control actions, MS for the action of moving the arm to the store, $M1$ for the action of gripping a block and moving it to position `t1`, and $M2$ for gripping and moving a block to position `t2`. We choose to trust the actuator completely in this example, that is, the execution of control actions will serve as the reference point which we use to classify discrepancies. This means that we can construct a *sub-model* induced by only looking at sub-states involving the variable *pos*. Since we completely trust the actuator, this sub-model gives the precise means of tracking the execution of the system. Formulas belonging to such sub-states are called *control configurations*. Formulas not belonging to such sub-states are called *plant formulas*. If it is a physical fact that no more than two blocks are in the system simultaneously, the relevant states can be found in Table 4.1. For readability, the indices of the states denote the values of the variables, e.g.

$$y_{11}^2 = \overbrace{pos = \mathtt{t2}}^{c^2} \land \overbrace{l1 = 1 \land l2 = 1}^{z_{11}}$$

The first state is the initial state of the system, and the three last states are the goal states. The remaining states (for example y_{20}^2) represent "non-relevant" states. As this is an example of sequential control, we have three sequences of states and control actions (*goal paths*) that are desirable (we consider all other paths to be

State	*pos* =	*l1* =	*l2* =
y^s_{00}	`s`	0	0
y^s_{10}	`s`	1	0
y^s_{01}	`s`	0	1
y^1_{10}	`t1`	1	0
y^1_{20}	`t1`	2	0
y^2_{01}	`t2`	0	1
y^2_{02}	`t2`	0	2
y^2_{11}	`t2`	1	1
$y_{g_{20}}$	`s`	2	0
$y_{g_{11}}$	`s`	1	1
$y_{g_{02}}$	`s`	0	2

Table 4.1: The relevant states.

non-desirable):

$$y^s_{00} \overset{M1}{\Longrightarrow} y^1_{10} \overset{MS}{\Longrightarrow} y^s_{10} \overset{M1}{\Longrightarrow} y^1_{20} \overset{MS}{\Longrightarrow} y_{g_{20}}$$
$$y^s_{00} \overset{M1}{\Longrightarrow} y^1_{10} \overset{MS}{\Longrightarrow} y^s_{10} \overset{M2}{\Longrightarrow} y^2_{11} \overset{MS}{\Longrightarrow} y_{g_{11}}$$
$$y^s_{00} \overset{M2}{\Longrightarrow} y^2_{01} \overset{MS}{\Longrightarrow} y^s_{01} \overset{M2}{\Longrightarrow} y^2_{02} \overset{MS}{\Longrightarrow} y_{g_{02}}$$

By merging the goal paths we can construct a model of the closed-loop system,

$$\begin{tikzcd}
 & & & y^1_{20} \arrow[r] & y_{g_{20}} \\
 & y^1_{10} \arrow[r] & y^s_{10} \arrow[ur] \arrow[dr] & & \\
 & & & y^2_{11} \arrow[r] & y_{g_{11}} \\
y^s_{00} \arrow[uur] \arrow[dr] & & & & \\
 & y^2_{01} \arrow[r] & y^s_{01} \arrow[r] & y^2_{02} \arrow[r] & y_{g_{02}}
\end{tikzcd}$$

Figure 4.3: The model of the trolley example.

depicted in Figure 4.3. It is not necessarily clear that the control actions in this example are energized, but if we assume that the variable *pos* maintains its value until it reaches a new position, they are. Moreover, the system is *non-logging* since configurations of consecutive states are mutually exclusive. This would imply that discrepancy classification should be possible.

Assume that the current goal is $y_{g_{11}}$ and that the state y^s_{10} has materialized. According to the model the action $M2$ should be invoked and the expected next state is y^2_{11}.

EA: If an operator would move the box on the first trolley to the second, the state y^s_{01} would materialize. Since the states have the same control configuration ($pos = s$) it is clear that the execution of the control action did not perform as expected and the conclusion is that the discrepancy is due to an EA. That is, the control action was disturbed during its execution and did not complete, but the state is different from the state the action was executed in.

VOA: One of the ontological assumptions in our examples is that the sensors $l1$ and $l2$ on the trolleys correspond to the positions `t1` and `t2`, respectively. If the trolleys would change places while executing $M2$, the resulting state would be y^2_{20} (since the arm would put a block in position `t2`, where, now, the sensor $l1$ is situated), which is not explicitly represented in the model. But, since the expected control configuration $pos = 2$ has materialized, but the expected plant formula (the part of the state not concerning pos) did not, the conclusion is that a VOA has occurred. That is, the control action was completely executed, but the resulting state differed from the expected state.

In this example there are no recovery actions, and thus no infinite recovery loops. If we would add more actions to the example, such as actions that can move objects between the trolleys, and from the trolleys to the store, the system could get stuck in infinite loops, in the case of the VOA presented above.

4.3 Summary of assumptions

As discussed above, the idea of detecting and classifying faults due to inadequate models requires some assumptions on both the model and the plant. The main idea is that contrary to, for example, the work on model-based diagnosis, we do not use the entire model as the reference point of the monitor system. Instead, if we trust a part of the model that marks the complete execution of control actions, we then need to know exactly when state transitions occur, and we have chosen to achieve this by assuming that actions are energized (as argued in Chapter 2. Finally, the classification scheme requires that the configuration parts (which intuitively is the part of the states concerning variables that are completely trusted) of all consecutive states are mutually exclusive. Below we will discuss the assumptions in more general terms.

4.3.1 Perfect sub-model

In the original work on ontological control [Fodor, 1995, Fodor, 1998] it was assumed that the execution monitor had access to the configuration of the actuators of the controlled system. The sub-model (that is, that model induced by the configuration parts of the states) only regarding the actuator sensor was then assumed to be perfect, that is, if the actuator sensor values changed state, that state change was completely trusted.

In this thesis we use a slight generalization of Fodor's assumption of trusted actuator sensors. We construct the models in such a way that we easily can extract a perfect (or, completely trusted) sub-model, but without any assumption of the physical realization of it. The reason for this is that in actual industrial process control systems, the inputs to the controller do not necessarily correspond to sensor readings, and outputs do not necessarily correspond to actuator invocations. A controller in such an environment is typically reading data from various sources, such as databases, Ethernets, sensors, and many other controllers. The controller then sends output to various receivers, such as databases, Ethernets, actuators, and many other controllers. A complicating factor is that both the internal components of a controller and all processes in its environment are run concurrently and asynchronously. This means that the models cannot only be confined to the input/output behaviors of the controller, but also have to consider internal computations, and actually consider such computations as providing inputs and outputs to the controller itself.

A perfect sub-model is, thus, not necessarily a model of the input/output behavior of the actuators, but a model of the system that can be trusted. An example may be a sub-model that only reflects computations performed by the controller. In fact, such a sub-model is constructed for the application described in Section 4.5.

4.3.2 Non-logging

It is not uncommon that parts of a control program are constructed for the purpose of *logging*, that is, that state changes are recognized, but that they do not trigger new control actions. An example could be robot arm, where there is a state when the angle from some plane is $< 30°$, and a distinct consecutive state checking that the angle now is $\leq 45°$.This is somewhat problematic from an execution monitoring perspective. To do execution monitoring in the state-based manner of this thesis, it is necessary to be able to distinctly recognize state changes. That is, with a vector of input values to the controller we need to be able to uniquely determine the internal state of the controller. For stabilization controllers we have no assumptions on exactly how the model is constructed, but for sequential controllers we assume the following:

The model is organized in sequences of states, *goal paths*, where transitions are assumed to occur due to control actions. The goal paths have different priorities, and we assume that the controller will try to follow the highest prioritized goal path possible. Next, we assume that consecutive states on the goal paths are mutually exclusive (that they cannot be materialized simultaneously), and this assumption is enforced by assuming that consecutive states in the perfect sub-model are mutually exclusive (see Section 4.4.2 for the details).

This is not a particularly restrictive assumption from a practical point of view, since the controllers we are interested in typically are reactive, and thus states triggering certain control actions are mutually exclusive to states triggering different actions.

4.3.3 Energized actions

The last assumption we need to make is how the execution of actions can be modeled. As discussed in Section 2.2.1, we will assume that actions are energized, that is, we assume that our models are constructed in such a way that actions are executed exactly as long as the state in which they were invoked is materialized.

4.4 Theory

From Section 2.5 we recall the definition of an execution monitoring system: A twelve-tuple

$$\langle P, A, Cont, S, M, Sit, Exp, D, Det, L, Class, Recover\rangle,$$

where P is a set of plant states, A a set of control actions, $Cont$ a function representing the controller, S a set of internal states, M a model, Sit the situation assessment function, Exp the expectation assessment function, D a set of discrepancies, Det the discrepancy detection function, L a set of discrepancy classifications, $Class$ the discrepancy classification function, and $Recover$ the recovery function.
In this section we will concretize the abstract theory and give a simple example.

4.4.1 Plant states, control actions, and internal states

We assume a finite set $X = \{x_1, \ldots, x_n\}$ of variables, representing input signals, and a set of domains $\Delta = \{\delta_1, \ldots, \delta_n\}$ representing the possible readings of the corresponding input signals. We assume a fixed ordering, $\langle x_1, \ldots, x_n\rangle$, on the variables (where δ_i is the domain of x_i).

The set of plant states is defined as the set $P = \delta_1 \times \delta_2 \times \ldots \times \delta_n$, that is, all n-tuples of values from the variable domains according to the ordering.
We assume a set $A = \{a_1, a_2, \ldots\}$ of control actions.

An expression $x_i \odot d$, with $x_i \in X$, $\odot \in \{<, =, >\}$, and $d \in \delta_i$, is called a *constraint*. A boolean combination of constraints[2] will be called a *formula*. The set of all formulas over X and Δ is denoted by F. The set of internal states is $S \subseteq F \times F$, and for sequential control we denote the distinguished set of *goal states* by $S^g \subseteq S$. For an element $\langle c, z\rangle \in S$ the first component, c, is called the *configuration*, and the second, z, is called the *plant formula*. For states $y = \langle c, z\rangle$ we define the projections $conf(y) = c$ and $plant(y) = z$.

[2]That is, a combinations of constraints constructed $\neg$ (negation), $\wedge$ (conjunction), and $\vee$ (disjunction)

4.4.2 Models

We assume that models are constructed from goal paths, in the following manner:

A *goal path* is a directed graph $G = \langle V, E \rangle$ with labeled edges, where $V \subseteq S$, $E \subseteq V \times A \times V$ (where the control action is the label on the edge), and where the following holds:

- There exists a unique internal state in V, s, such that no edge $\langle s', a, s \rangle \in E$ exists, for any internal state s' and control action $a \in A$. This internal state is called the *initial state* of the goal path.
- There exists a unique internal state in V, $s \in S^g$, such that no edge $\langle s, a, s' \rangle \in E$ exists, for any internal state s' and control action $a \in A$. This internal state is thus the *goal state* of the goal path.
- For any internal state $s \in V$, *except* for the goal state, there is exactly one edge $\langle s, a, s' \rangle \in E$ for some control action a and internal state s'.

Thus, a goal path is a total ordering of a subset of internal states, and the length of a goal path, G, (that is, the number of internal states in the ordering) is denoted $length(G)$.

A *model* is a tuple $M = \langle \mathcal{G}, o \rangle$ where $\mathcal{G}$ is a set of goal paths, and o is a total ordering (the *priority ordering*) of $\mathcal{G}$. Since all goal paths are totally ordered we define the function $index : V \times \mathcal{G} \to \mathbb{N}$ such that $index(s, G) = k$ iff s is the kth internal state in the goal path G. If s is the initial state in G, then $index(s, G) = 1$. Moreover, we define functions $initial(G)$ and $goal(G)$ to return the initial and goal states, respectively, of a given goal path G.

Clearly, it is possible to view the first component of a model (the set of goal paths) as one single graph if we look at the union of the vertices and edges in the goal paths. This graph will be referred to as the *control graph.* The control graph is useful for recovery, which will be discussed below. We say that two internal states, y and y', are *consecutive* if one of the the edges $\langle y, a, y' \rangle$ or $\langle y', a, y \rangle$ exist in the control graph.

By restricting a control graph to configurations, that is, by restricting all goal paths to configurations, we construct a *sub-model.* A perfect sub-model is a sub-model where all consecutive states are mutually exclusive (this concept will be properly defined in Section 4.4.3.). In figure 4.4 the perfect sub-model of the trolley example is depicted.

4.4.3 Interpretations

A *plant state*, $\sigma \in P$ can be viewed as a projection from variables to the corresponding domains, where $\sigma(x_i) = d_i$ iff $\sigma = \langle d_1, \ldots, d_i, \ldots, d_n \rangle$.

The *interpretation*, ζ, is a function from plant states and formulas to $\{0, 1\}$, and it is defined as follows: For $x_i \in X$, $\odot \in \{<, =, >\}$, $d_i \in \delta_i$, and formulas α and β,

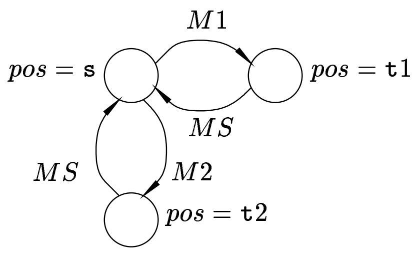

Figure 4.4: The perfect sub-model of the trolley example.

we have

$$
\begin{aligned}
\zeta(\sigma, x_i \odot d_i) = 1 &\quad \text{iff} \quad \sigma(x_i) \odot v_i \text{ holds} \\
\zeta(\sigma, \alpha \wedge \beta) = 1 &\quad \text{iff} \quad \zeta(\sigma, \alpha) = 1 \text{ and } \zeta(\sigma, \beta) = 1 \\
\zeta(\sigma, \alpha \vee \beta) = 1 &\quad \text{iff} \quad \zeta(\sigma, \alpha) = 1 \text{ or } \zeta(\sigma, \beta) = 1 \\
\zeta(\sigma, \neg\alpha) = 1 &\quad \text{iff} \quad \zeta(\sigma, \alpha) = 0
\end{aligned}
$$

We say that a formula, α, is *consistent* iff there exists a $\sigma \in P$ such that $\zeta(\sigma, \alpha) = 1$. The plant state σ is then said to *satisfy* α.

We overload ζ to apply to internal states, so that $\zeta(\sigma, \langle c, z\rangle) = 1$, iff $\zeta(\sigma, c) = 1$ and $\zeta(\sigma, z) = 1$. Consistency and satisfaction is similarly defined for internal states as for formulas.

If we have a plant state σ such that for an internal state y, $\zeta(\sigma_t, y) = 1$, we say that y *materializes* by σ.

Two internal states, y and y', are said to be *logically equivalent* iff, for every plant state σ, we have $\zeta(\sigma, y) = \zeta(\sigma, y')$. Two states are *mutually exclusive* iff they are not simultaneously satisfied by any plant state.

4.4.4 Controller

A controller is a function from a current internal state and a goal state to a control action. Formally, a controller is a function $Cont : S \times S^g \to A$. The goal state is necessary to distinguish between the goal paths.

4.4.5 Embeddings

As we will need to put restrictions on the actual plant we need to simulate it somehow. We do this by introducing the concept *embedding*, $\mathcal{E} : P \times A \to P$, a function from plant states and control actions to plant states. Intuitively, $\mathcal{E}$ models the behavior of the plant under the influence of control actions.

Every embedding defines a set of traces $\tau_{\mathcal{E}} = \sigma_0 a_0 \sigma_1 a_1 \sigma_2 \ldots$ such that $\mathcal{E}(\sigma_i, a_i) = \sigma_{i+1}$. We drop the subscript of $\tau_{\mathcal{E}}$ whenever no ambiguity can arise. We use the symbol $\in$ to denote the substring relation.

Definition 4.4.1 (Non-logging systems)
We say that a system is *non-logging* w.r.t. an embedding $\mathcal{E}$ and a corresponding trace τ iff, for every triple $\sigma a \sigma' \in \tau$ and

$$c, c' \in \{c'' \,|\, \langle c'', z\rangle \in S\},$$

$\zeta(\sigma, c) = \zeta(\sigma', c') = 1$ holds exactly when c and c' are mutually exclusive.□

Intuitively, for all subtraces $\sigma a \sigma'$ in a trace, any pair of configurations c, c' that are satisfied by σ, σ' respectively, must be mutually exclusive. Thus, we ensure that state changes can be detected by looking at materializations of configurations only.

4.4.6 Situation assessment

The situation assessment function is, in this chapter, a function from a plant state, a model, and a goal state to an internal state, that is, $Sit : P \times \mathcal{M} \times S^g \to S$.

Here, we exploit the goal path structure. As discussed in Chapter 2, the important issue is not to satisfy a goal state as quickly as possible, but to ensure that all control actions in the sequence are properly executed. The idea is that the situation assessment function picks out the internal state with the lowest index on the highest prioritized goal path that is satisfied by the given plant state. Formally, for a set of goal paths $\mathcal{G} = \{G_i\}_{i=1}^m$ and a state $y = \langle c, z\rangle$, we have $Sit(\sigma, \langle \mathcal{G}, o\rangle) = y$, for some ordering o, by the following algorithm:

1. Let $Y = \{y' \in S \,|\, \zeta(\sigma, y') = 1\}$. If $Y = \emptyset$ then return the dummy state $y = \mathbf{nosat}$.
2. Let $\mathcal{G}' = \{G_i \in \mathcal{G} \,|\, \exists y' \in Y. y' \in G_i\}$. $\mathcal{G}'$ cannot be empty by the definition of "model".
3. Pick out the highest prioritized member of $\mathcal{G}'$, that is, take $G \in \mathcal{G}'$ such that $o(G) = max\{o(G') \,|\, G' \in \mathcal{G}'\}$.
4. Set y to be the state in Y with the lowest index on G, that is $index(y) = min\{index(y') \,|\, y' \in G\}$.

It is now possible to define the notion of energized actions.

Definition 4.4.2 (Energized actions)
We say that an action a is *energized* w.r.t. an embedding $\mathcal{E}$ and a controller $Cont$ iff $\mathcal{E}(\sigma, a)$ is defined exactly when $Cont(Sit(\sigma, M), g) = a$, for some goal state g. A system is energized if all control actions of the system are.□

4.4.7 Expectation assessment

Expectation assessment is function $Exp : S \times A \times \mathcal{M} \to S$ from an internal state, control action, and a model to an internal state, the *expected* state. The computation of an expected state is done from a plant state and a control action in an embedding, that is, for $\sigma a \in \tau$, $Exp(Sit(\sigma, M), a, M) = y$ where the edge $\langle Sit(\sigma, M), a, y\rangle$ is an edge on the highest prioritized goal path possible.

4.4.8 Discrepancy detection

Let τ be trace of an embedding and pick $\sigma a \sigma' \in \tau$, such that

$$Exp(Sit(\sigma, M), a) \neq Sit(\sigma', M).$$

Then we say that *the discrepancy $\sigma a \sigma'$ has been detected.*

4.4.9 Discrepancy classification

The main goal for ontological control is to detect and classify discrepancies due to model inadequacies and disturbances. With the concepts configuration, non-logging and energized systems this is not particularly difficult.

Theorem 4.4.3 We assume a given model, controller, and embedding, such that the system is non-logging and energized. Let $\sigma a \sigma'$ be a detected discrepancy. Then, exactly one of the following statements hold:

1. $conf(Exp(Sit(\sigma, M), a))$ is logically equivalent to $conf(Sit(\sigma', M))$.
2. $conf(Exp(Sit(\sigma, M), a))$ is mutually exclusive with $conf(Sit(\sigma', M))$.

.

Proof: Since $\sigma a \sigma'$ is a discrepancy we have

$$\langle c, z \rangle = Exp(Sit(\sigma, M), a) \neq Sit(\sigma', M) = \langle c', z' \rangle.$$

As the system is energized, a change of internal state must have occurred due to the discrepancy, that is, σ and σ' must be mapped to different states by Sit. The non-logging property then gives us that either $\zeta(\sigma, c) = \zeta(\sigma', c') = 1$ holds, or it does not. When it holds, we have case 2, otherwise we have case 1.□

Case 1 in Theorem 4.4.3 will be referred to as a discrepancy due to EA, and case 2 as a discrepancy due to VOA.

4.4.10 Recovery from discrepancies

When a discrepancy is detected during execution we need to do two things. First, we need to figure out what state we have come to and find an appropriate control action from that state, in line with the purpose of the control system. Secondly, we may have to adjust the model to make it more appropriate for our particular application.

We address neither of these issues in this chapter. The low SCWA of our representation makes it very difficult to predict the effects of any changes of the model. In chapter 7, however, the issues will be discussed in detail.

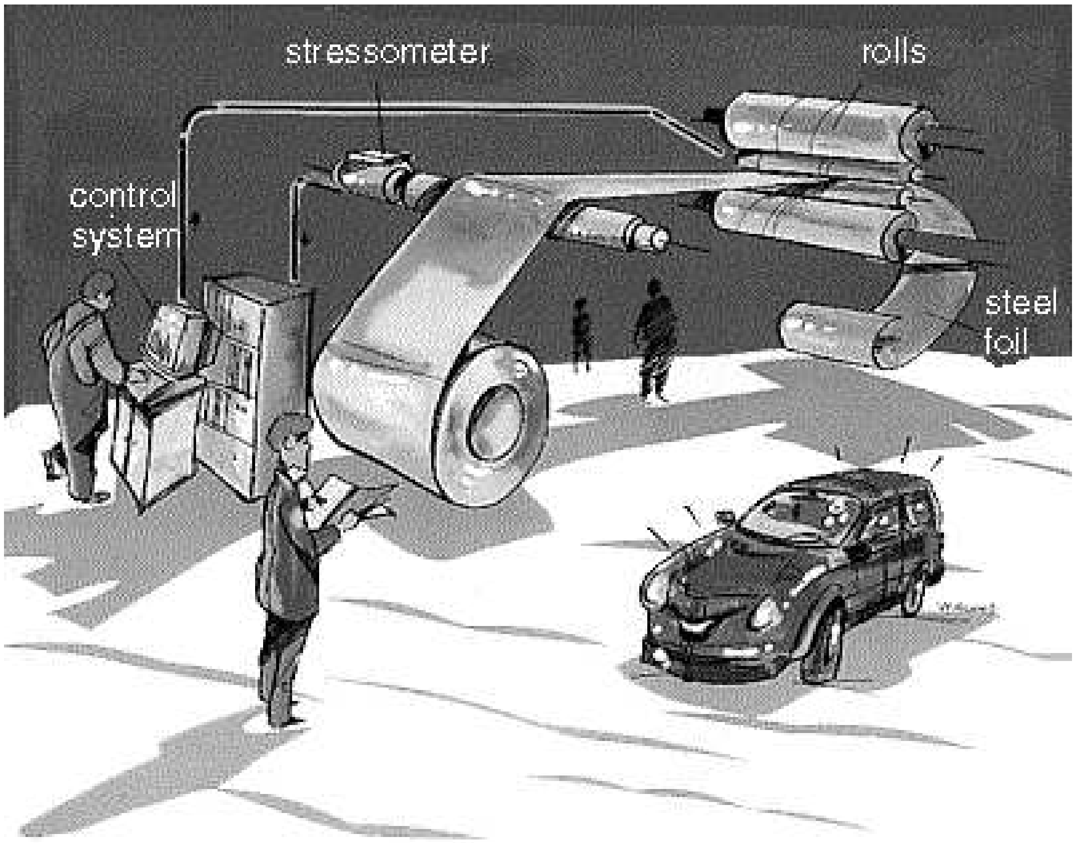

Figure 4.5: The STRESSOMETER flatness control system. Drawing by Lasse Widlund.

4.5 Experiments

The theory described above has been applied to a real industrial process control system. In this section, we will describe the application, the implementation, the experimental setup, and some of the results. This is an extension of the paper [Bjäreland and Fodor, 2000].

4.5.1 Application: ABB STRESSOMETER

The application is the market leading flatness control system for cold steel mills, the ABB STRESSOMETER (see Figure 4.5). In cold-rolling mills, a metal strip that is subject to different degrees of reduction across its width will be elongated in varying length over different sections. A special measuring roll (in this case, a STRESSOMETER System [ABB, 1999]) can gauge the flatness error of the strip. Several different methods have been developed to correct the flatness error; all of them are based upon local modifications of the gap between rolls. We consider here a system with two actuators. The first one is a roll bending actuator that performs a course-granularity symmetrical compensation of the error by applying bending forces on the rolls. The second one is a so-called multi-step cooling actuator. This produces small changes of the rolling diameter in well-defined zones of the rolls, where cooling liquid is applied such that the flatness error is reduced. The cooling unit performs a fine-granularity, possibly asymmetrical, compensation. Normally the effect of the bending actuator on the strip is known in advance (given by a so-called *evaluation curve*). To eliminate the error, a control algorithm is implemented as a two-step control loop: First, the expected effect of the bending actuator is subtracted from the measured error. Secondly, the error that is left over is further reduced by the multi-step cooling unit.

A discrete state set for this type of control is a sequence $S_0, \dots, S_3$ as follows:

S_0 The controller computes the flatness error as a multi-variable distance (norm) along the strip width, between the measured and the reference strip profile.

S_1 The controller predicts what will be the effect of the bending force on the error.

S_2 The error input for the cooling input is computed and other discrete conditions are considered (e.g. manual cooling zones are excluded).

S_3 The actuator commands for the cooling unit are determined.

Some of the commonly expected external actions (disturbances) that may occur and change the state transition sequence described above are the following.

- The error to be compensated by the cooling unit is too high (the roll has been cooled down too much and cannot act anymore).
- One or several cooling nozzles deliver less coolant liquid than specified, for example due to sediments.

- The bending actuator has changed its last characteristics since its maintenance and acts in a lesser, or higher, degree.

In industrial applications, it is essential that trends for such expected external actions are detected early. Ontological control can be applied both for slow continuous trends as well as for discrete, abrupt changes such as to detect a sudden cross connection of nozzles.

It should be noted that this system is non-linear and that the vast complexity prohibits the construction of a mathematical model of it, from an economical point of view. This implies that attempts to model the system and then formally *verify* its correctness are futile (as are any attempts to verify industrial-scale control systems). The control algorithms are implemented in "ABB Master Piece Language" (AMPL)

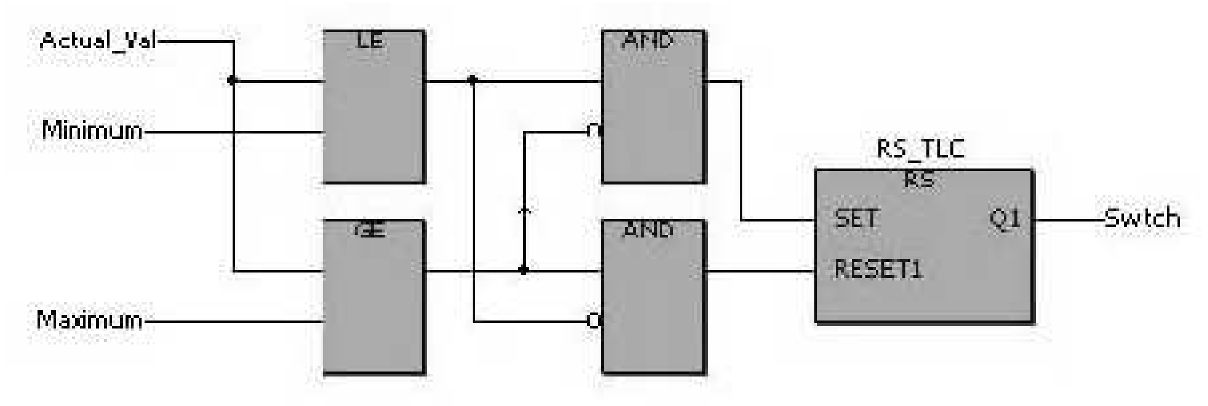

Figure 4.6: A small FBD program.

that is an instance of "Function Block Diagrams" (as described in the standard IEC 1131-3 [Lewis, 1997]). AMPL is thus data-flow based, and a program consists of a number of *elements* with input and output ports connected to each other (see Figure 4.6 for a small example of a FBD program). To receive input, an AMPL program relies on a number of communication elements and database query elements. It is seldom the case that a program receives sensor readings directly; it is far more common that the source of input is another controller or a database. The same holds for output, where it is far more common for a program to send computation results to other programs than to actuators. This means that the theoretical notions "input variable" and "actuator invocation" should be viewed as inputs and outputs to a control program without attaching any physical realization to them.

It is also the case that the programs are not *I/O switching*, that is, all computations within a program are not necessarily performed during one sampling interval.[3]

[3]I/O switching systems run in a read-compute-write loop, and finishes the loop at every sampling instance.

This implies that the states of a model cannot only reflect variable changes *outside* the program, but also need to model the internal execution. In fact, in the program we studied there are 22 sequential processes running concurrently (and partly asynchronously).

In this experiment we looked at one program, program 9 (of 25), for the cooling unit. Program 9 was chosen since it is fairly small (7000 - 8000 lines of code), and since it is a "true" legacy program, in the sense that it was coded a number of years ago, and it is not entirely clear how it executes.

4.5.2 Goal path and control graph generation

The input for the goal path and control graph generation is an AMPL program source code listing in ASCII text format. For program 9, the generation was done by hand, but it can be semi-automated. After a number of unsuccessful attempts at finding suitable parsing rules, the following "algorithm" was invented:

1. Each element of the program is associated with one of the following four types:
   - t_1 Control actions: Elements that influence devices outside the program, such as elements producing external actions, communication elements, database storage elements etc. Can be viewed as output elements.
   - t_2 Logical: Logical gates.
   - t_3 Input plant formula: Input elements.
   - t_4 Data-flow: All other elements.

   Only boolean signals are considered.
2. A restricted program is generated, in terms of $t_1, \dots, t_4$ and boolean variables only. For program 9 we identified 94 variables.
3. For every control action element the corresponding precondition configuration is determined, using the t_2 elements. This gives us the first two members of the $\langle c, a, c' \rangle$ triplets, and the number of states. The plant formula part of the states comes from the t_3 elements. In program 9 we identified 125 states.
4. The preconditions and the data-flow elements are then used to determine the postconditions of the actions, which yields the goal paths. In program 9 there are 46 goal paths.
5. The goal paths are then merged into the control graph.

The structure of the control graph is depicted in Figure 4.7, to show the complexity of the application. The vertices in the picture represent states and the edges represent transitions due to control actions.

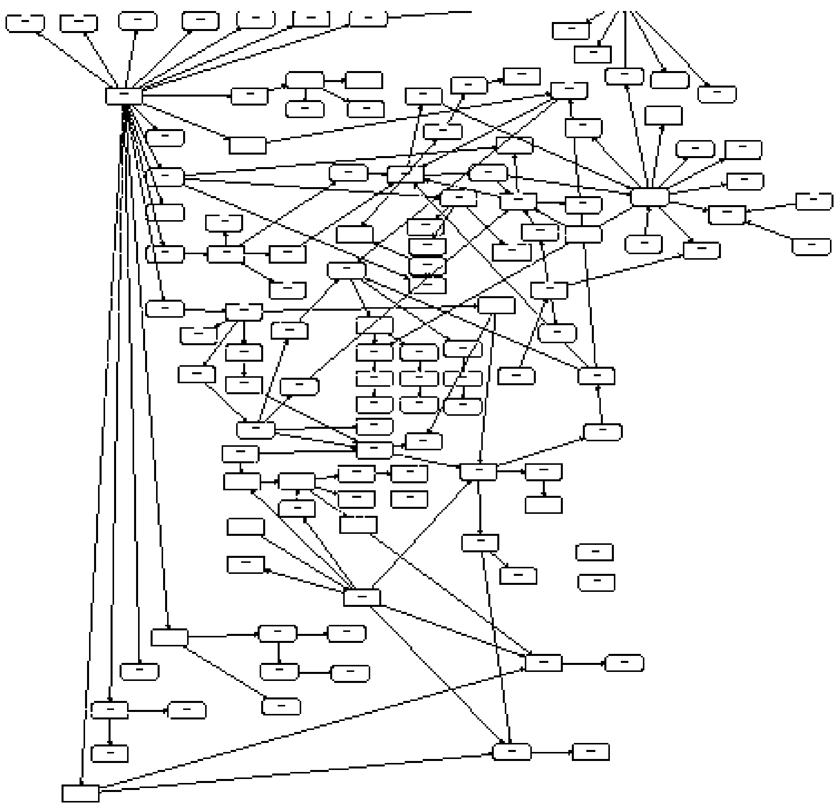

Figure 4.7: The control graph generated from program 9.

4.5.3 Implementation

Based on the theory above, a prototype implementation was developed by Lewau [Lewau, 1999]. The main part of the system is a domain-independent execution monitoring engine that given a list of variables, a list of goal paths, and the control graph, can monitor the execution of the corresponding controller. The prototype is object-oriented and implemented in JAVA and is thus easy to move between platforms. This also ensures that different means of communication with the execution monitor are easy to implement (whether it is through sockets, Internet, RMI, or Corba).
Below we refer to the prototype as the "OC system".

4.5.4 Experimental setup and results

The experimental setup consists of an industrial-grade STRESSOMETER system with a simulated process, which is normally used for factory acceptance tests. By probing relevant variables during execution, we recorded 11 traces (sequences of samples), with various faults, and stored them into text files.

The experiments have been done as follows: files with variable lists, goal paths (with priorities), and the control graph were given to the OC system. The system was tested with the 11 traces, of which one was sampled during optimal execution of the STRESSOMETER, and the rest sampled with some fault introduced. We analyzed the following relevant traces:

Trace 5: Cooling did not have effect due to high compensation.
Trace 6: Exception 1, data invalid at initiation phase.
Trace 7: Exception 1, communication error at initiation phase.
Trace 8: Exception 2, data invalid during processing.
Trace 9: Connection fault.
Trace 10: Internal communication error.
Trace 11: One cooling zone failure.
Except for the faults in trace 11, all other faults were detected, and none of the faults had previously been explicitly detected. The granularity of the model prohibited detection of the fault in trace 11.

The faults were also classified correctly, but since the system did not obey the non-logging property, it is hard to say anything about the distinction between EAs and VOAs for the given traces.

4.6 Comparison to Fodor's original work

In previous sections there are claims that the presentation of ontological control in this thesis is more general, more formal, and extended compared to the original work of Fodor [1995, 1998]. In this section we will present the main differences between the work in this thesis and Fodor's work, and discuss them.

4.6.1 Discrepancy classes

In this chapter we have considered only two causes for discrepancies: EA and VOA. Fodor, on the other hand, distinguishes between the following four:

- VOA,
- Expected External Action (EEA),
- Unexpected External Action (UEA), and
- Timing Errors (TE).

Fodor defines VOAs in a similar way as we, but chooses a more refined definition of EA. He assumes that there is a second kind of actions, *expected external actions*, in the model, that may change the state when they are executed, but that are uncontrollable by the monitored controller. In this way it is possible to model known possible interferences with other controllers. Without the presence of such expected external actions, the classes EEA and UEA coincide, and since they are detected, classified and recovered from in exactly the same way, we have chosen to view them as the same class.

Timing errors occur when the sampling time is too long, that is, when the execution monitor fails to notice a state change as it happened between two sampling instances. This means that the execution monitor may erroneously detect a discrepancy. It is difficult (probably impossible within this framework) to devise a theory that can distinguish between VOAs and TEs, and since the experiences show that normal sampling frequencies of 50 - 100 Hz suffices to detect all state transitions, we assume that the frequencies can be set to be sufficiently high. Thus, we do not need to to consider TEs.

4.6.2 States, goal paths, and control graphs

A fundamental difference is the notion of *states*. In Fodor's work a *controller state* is a tuple $\langle y, u \rangle$ where y is a pair $\langle z, c \rangle$ exactly corresponding to our notion of internal state, and where u is a control action. A goal path in this setting is a path with controller states as nodes and with unlabeled edges, while in our setting the internal states are connected by edges corresponding to control actions. The advantage to having control actions within the states is that then the "energized action property" is built into the system (the system is in a state as long as the internal state is satisfied *and* the control action is executed). However, there certainly are situations where an action is more naturally modeled as being ballistic, and these cases cannot be handled in a straightforward way in Fodor's setting. Another difference is how different actions with the same precondition are modeled. In our version this can be handled by letting two control actions lead from the same state to two different states, as shown in Figure 4.8. In Fodor's setting Figure 4.8 must be interpreted as nondeterministic effect of an action, which traditionally is avoided in process control

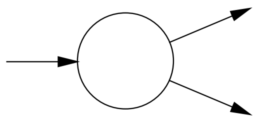
Figure 4.8: A state with two outgoing edges.

systems[4]. Instead, different actions with the same precondition must be modeled with two different states, one for each action, and subsequently, two different paths leading up to those.

4.6.3 Execution monitoring

Another difference lies in how situation assessment is assumed to work. We represent states as formulas and use samples to satisfy states on the way to determine the current state. In Fodor's work the job of finding the current state is achieved by the controller, that is, the controller decides the current state and then sends the *name* of the state to the ontological controller. This implies that the execution monitor does not have access to the vector of variable values. The reason for this is computational: Fodor argues that it is unnecessary to perform the situation assessment computation in the ontological controller since it is already done in the controller. In our setting the value vectors are used to detect and to classify discrepancies, so Fodor uses a different strategy: Discrepancy detection is done by comparing the name of the state sent by the controller to the name of the expected state, and if these are not the same a discrepancy is detected. For classification, Fodor relies on sets of states related to every state in the model, the *collateral states.* The collateral states of a state, $y = \langle c, z \rangle$, are all states that have the same c component. Thus, for a detected discrepancy it suffices to check whether it belongs to the collateral set of the previous state or to the collateral set of the expected state to determine whether it is due to an EA or a VOA, respectively. This idea is simpler than ours. However, it relies on the assumption that all physically possible states are represented in the model, that is, that for any possible sample there is a state that is materialized by the sample. We consider this assumption to be both inconvenient for the model designer and unnecessarily restrictive.

4.6.4 Well-determined state sets

In Fodor's work a state transition diagram model can only be used for ontological control if it is *well-determined*, that is, if the following four restrictions are met:

[4]This is a possibility that we have not considered in this thesis. We acknowledge the need for modeling uncertainty, but believe that there are other (and perhaps better) ways of doing this than to have nondeterministic actions. For example, since we are only interested in using the models for prediction, there is a perfect duality between uncertainty of the effects of control actions, and the uncertainty of sensor readings (see e.g. [Sandewall, 1994, Lin, 1996] for discussions on this duality). The latter fits nicely into our framework.

- Control action integrity: No state can be reached by both a control action and an expected external action.
- Specificity of configurations: There cannot be two different control actions whose effects satisfies the same configuration formula.
- Specificity of control actions: Control actions are deterministic.
- Syntactically complete state sets: There is a control action for every output of the plant.

Fodor shows that these four restrictions enable solutions to the fundamental problem of ontological control. The reader should note that it is difficult to compare Fodor's restrictions to the restrictions in this chapter. Fodor's restrictions heavily rely on his definition of "state" and the fact that his ontological controller does not perform situation assessment by attempting to satisfy a logical formula, but instead receives a pointer to the currently materialized state. However, we claim that our restrictions are weaker, in the sense that more systems can satisfy our restrictions.

Clearly, the control-action-integrity restriction is not relevant to the work in this chapter, since we have no explicit notion of "external action". The specificity-of-configuration restriction has been removed, while the specificity-of-control-actions restriction remains, but more as a desired property of control systems in general than as an explicit restriction. Finally, the syntactically-complete-state-sets restriction is very strong, since it assumes that there exists an explicitly represented state for every possible sample, and that the controller can handle every such state. This assumption has been relaxed, because it has more to do with the design of robust controllers then with execution monitoring of controllers.

Chapter 5

Stability-based execution monitoring

In section 2.4 we discussed various research issues involved in execution monitoring, one of which was "purpose". As mentioned there we investigate two purposes of execution monitoring in this thesis: ontological control and stability-based execution monitoring. The former concerns sequential control systems and was developed to hedge infinite recovery cycles in such systems (as presented in chapter 4). If we instead consider control systems with the control task of making a plant *stable*, that is, to ensure that the actual plant state always belongs to a predetermined set of states, we have a different situation. It may be interesting to detect infinite cycles for such systems as well, but as such controllers typically are continuously running through a cycle of states, it is far more important to ensure that the system does not jump out of that cycle, and if it does, that it is possible to get back into the cycle again, in a safe manner.

In this chapter we will formally present the standard notions "stability" and "stabilizability" for discrete system. We will argue that these notion may be very restrictive for a number of domains, such as autonomous systems and robotics, and propose the notion "maintainability". The difference between these concepts lie in the explicit representation of exogenous (uncontrollable) events that appear for maintainability but not for stabilizability. A system is stabilizable w.r.t. a set of states E if we can guarantee that we by control actions only can reach E from any state in the system in finite time. We say that a system is k-maintainable w.r.t. E if we during a time interval of length k with no occurrences of exogenous actions, can reach E with control actions only. It is easy to see that maintainability and stabilizability are incomparable (neither property implies the other), so we propose a more general notion, (k, l)-maintainability, meaning that any state trajectory of length k with at most l occurrences of exogenous actions will take the system to E. We show that stabilizability is equivalent to the existence of an m such that the system is (m, m)-maintainable, and that k-maintainability is equivalent to $(k, 0)$-

maintainability.

We formally compare these notions in a framework of discrete finite automata. We then present algorithms to verify maintainability, and to construct controllers to make a system maintain a set of states. Finally we discuss how these notions can be monitored.

The theory developed in this chapter will be applied in chapters 6 and 7.

5.1 Introduction

The concept of stability has undergone extensive investigation in both the Computer Science (see Section 3.3.2 and the on-line bibliography [Herman, 1999]) and the Control Theory community (see [Passino and Burgess, 1998]), both for continuous systems (e.g. Lyapunov stability and asymptotic stability) and Discrete Event Dynamic Systems (DEDS) [Ramadge and Wonham, 1989], [Özveren *et al.*, 1991]. All these notions can be summarized as in [Passino and Burgess, 1998]:

> We say that a system is stable if when it begins in a good state and is perturbed into any other state it will always return to a good state.

The appropriate stability notion in a particular case depends on how the notions "system", "begins", "state", "good", and "perturbed" are defined. For DEDS the mainstream definition can be found in [Özveren *et al.*, 1991] (see Definition 5.3.3, and that definition is the one we use in this chapter. A related, more general notion which we call *maintainability* is introduced in this chapter, and we argue its importance, particularly for high level control of agents.

Intuitively, we can view stabilizability as a hard constraint on the system while maintainability is a softer constraint. In both maintainability and stabilizability our goal is that the system should be among a given set of states E as much as possible. In stabilizability, we want a control such that regardless of where the system is now and what exogenous actions may occur, the system will reach one of the states in E within a finite number of transitions and keep visiting it infinitely often after that. In maintainability, we have a weaker requirement where the system reaches a state in E within a finite number of transitions, provided it is not interfered with during those transitions. Thus in maintainability, we admit that if there is continuous interference (by exogenous actions) we can not get to E in a finite number of transitions. Such a system will not satisfy the condition of stabilizability, but may satisfy the condition of maintainability.

Many practical closed-loop systems are not stabilizable, but they still serve a purpose and we believe that such systems can be specified by using the weaker notion of maintainability. An example is a mobile robot [Brooks, 1986] which is asked to 'maintain' a state where there are no obstacles in front it. Here, if there is a belligerent adversary that keeps on placing an obstacle in front of the robot, then the robot can not get to a state with no obstacle in front of it. But often we will be satisfied if the robot avoids obstacles in front of it when it is not continually harassed. Of course, we would rather have the robot take a path that does not have such an adversary, but in the absence of such a path, it would be acceptable if it takes an available path and 'maintains' states where there are no obstacles in front.

Other examples include agents that perform tasks based on commands. Here, the correctness of the agent's behavior can be formalized as 'maintaining' states where there are no commands in the queue. We can not use the notion of stability because if there is a continuous stream of commands, then there is no guarantee that the agent would get to a state with no commands in its queue within a finite number of transitions.

Another important aspect of maintainability is that in reactive *software* systems, if we know that our system is k-maintainable, and each transition takes say t time units, then we can implement a transaction mechanism that will regulate the number of exogenous actions allowed per unit time to be $\frac{1}{k \times t}$. This will also be useful in web-based transaction software where exogenous actions are external interactions and the internal service mechanism is modeled as control laws. On the other hand, given a requirement that we must allow m requests (exogenous actions) per unit time, we can work backwards to determine the value of k, and then find a controller to make the system k-maintainable. In general, since in high level control we may have the opportunity to limit (say through a transaction mechanism) the exogenous actions, we think 'maintainability' is an important notion for high level control.

5.2 Running example: Two finite buffers

We imagine a system with two finite buffers, b_1 and b_2, where objects are added to b_1 in an uncontrollable way. An agent then moves objects from b_1 to b_2 and processes them there. When an object has been processed it is automatically removed from b_2. This is a slight modification of a finite buffer example from [Passino and Burgess, 1998] and generalizes problems such as ftp agents maintaining a clean ftp area by moving submitted files to other directories, or robots moving physical objects from one location to another.

For simplicity, we assume that the agent has three control actions $\mathbf{M}_{12}$ that moves an object from b_1 to b_2 (if such an object exists), the opposite action, $\mathbf{M}_{21}$ that moves an object from b_2 to b_1, and **Pro** that processes and removes an object in b_2. There is one exogenous action, *Ins* that inserts an object into buffer b_1. The capacities of b_1 and b_2 are assumed to be equal.

If the control goal of this system is to keep b_1 empty, the system is not stabilizable, since an object can always be inserted and violate the goal. However, if no insertions are performed for a certain window of non-interference, the agent can always empty b_1. This implies that the system is maintainable but not stabilizable.

We will formalize this example below.

5.3 Reviewing stability and stabilizability

In this section we review the notions of stability and stabilizability adapted from the definitions in [Özveren *et al.*, 1991].

5.3.1 Stability and aliveness

Definition 5.3.1 A system A is a 4-tuple (X, Σ, f, d), where X is a finite set of states, Σ is a finite set of actions, $d : X \rightarrow 2^{\Sigma}$ a function listing what actions may occur (or are executable) in what state, and $f : X \times \Sigma \rightarrow 2^{X}$ the non-deterministic transition function. We overload the transition function to handle sequences of actions as well, that is, for a sequence of actions $A = [a_0, a_1, \ldots, a_n]$ we define $f(x, A)$

$$f(x, [a_0, a_1, \ldots, a_n]) = \bigcup_{x' \in f(x, a_0)} f(x', [a_1, \ldots, a_n]).$$

□

Running Example, Cont'd
The transition diagram of the buffer example is depicted in figure 5.1. We assume

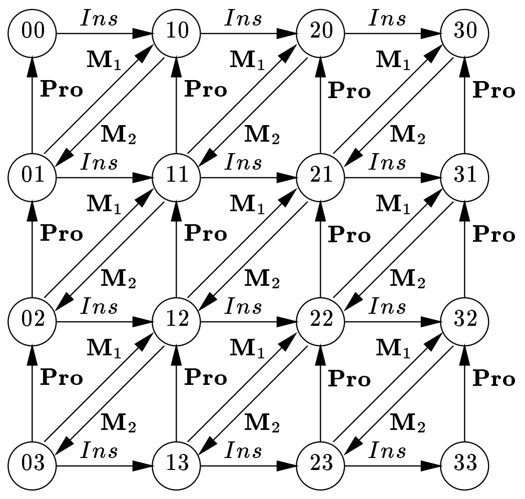

Figure 5.1: The transition diagram of the system for the buffer example.

that the maximum capacity of the buffers is 3, and model the state space by letting every state in X represent the number of objects in b_1 and in b_2, that is, a state is identified by a pair of integers $\langle i, j \rangle$ (ij for short) where i denotes the number of objects in b_1 and j the number of objects in b_2. With the maximum capacity assumed to be 3, the state space consists of $4 \times 4 = 16$ states. That is

$$X_b = \{0,1,2,3\} \times \{0,1,2,3\}.$$

The set of actions is $\Sigma_b = \{\mathbf{M}_{12}, \mathbf{M}_{21}, \mathbf{Pro}, Ins\}$.

We assume that the transition function is deterministic (that is $f_b : X_b \times \Sigma_b \to \Sigma_b$) and is defined as follows:

$$\begin{aligned}
f_b(\langle i,j\rangle, \mathbf{M}_{12}) &= \langle i-1, j+1\rangle \\
f_b(\langle i,j\rangle, \mathbf{M}_{21}) &= \langle i+1, j-1\rangle \\
f_b(\langle i,j\rangle, \mathbf{Pro}) &= \langle i, j-1\rangle \\
f_b(\langle i,j\rangle, Ins) &= \langle i+1, j\rangle
\end{aligned}$$

The enabling function, d_b, is defined as

$$\begin{array}{lll}
\mathbf{M}_{12} \in d_b(\langle i,j\rangle) & \text{iff} & i \geq 1 \text{ and } j \leq 2 \\
\mathbf{M}_{21} \in d_b(\langle i,j\rangle) & \text{iff} & i \leq 2 \text{ and } j \geq 1 \\
\mathbf{Pro} \in d_b(\langle i,j\rangle) & \text{iff} & j \geq 1 \\
Ins \in d_b(\langle i,j\rangle) & \text{iff} & i \leq 2
\end{array}$$

The system A_b is then defined as $A_b = \langle X_b, \Sigma_b, f_b, d_b\rangle$

Definition 5.3.2 An alternating sequence of states and actions

$$x_0, a_1, x_1, a_2, \ldots, x_k, a_{k+1}, x_{k+1}, \ldots$$

is said to be a *trajectory consistent with a system A* if:

- $x_{k+1} \in f(x_k, a_{k+1})$, and
- $a_{k+1} \in d(x_k)$.

□

Definition 5.3.3 Given a system A and a set of states E, a state x is said to be *stable* in A w.r.t. E if all trajectories consistent with A and starting from x go through a state in E in a finite number of transitions and they visit E infinitely often afterwards.

We say $A = (X, \Sigma, f, d)$ is a stable system if all states in X are stable in A w.r.t. E. □

Alternatively, A is stable w.r.t. E if, for any state $x \notin E$, every infinite trajectory starting with x will lead to E in a finite number of steps. This alternative definition will be used in chapter 7.

It is not possible to pick any strict subsets S and E of X_b such that S is stable w.r.t. E. Thus, A_b is only trivially stable, that is, any set of states S is only stable w.r.t. X_b.

Definition 5.3.4 $R(A, x)$ denotes the set of states that can be reached from x by some trajectory consistent with A.

A state x is said to be *alive* if $d(y) \neq \emptyset$, for all $y \in R(A, x)$. (That is, we can not reach a state y from x, where no action is possible.)

We say $A = (X, \Sigma, f, d)$ is *alive* if all states in X are alive.$\Box$

As an aid in future proofs we state the following characterization of stability:

Lemma 5.3.5 Let A be a system and S and E sets of states. S is stable w.r.t. E iff there exists a natural number n such that every trajectory consistent with A starting from x meets E in at most n transitions, for every $x \in S$.
Proof: $\Leftarrow$) Straightforward from the definitions.
$\Rightarrow$). Assume that S is stable w.r.t. E and that there is trajectory consistent with A starting in a state $x \in S$ with length $> |X|$ where no member of the trajectory belongs to E. Then some state in X must occur more than once on the trajectory, that is, there is a cycle on the trajectory not meeting E. But then there is an infinite trajectory starting from x not meeting E at all, which contradicts the assumption that S was stable w.r.t. E. Therefore, no trajectory starting in S can be longer than $|X|$, which proves the lemma.$\Box$

It should be noted that Lemma 5.3.5 depends heavily on the finiteness of the set of states. If X is infinite the Lemma fails.

5.3.2 Stabilizability

We now consider control and exogenous actions. The set of control actions U is a subset of Σ, that can be performed by the agent. A particular control law[1] $K : X \to U$ is function from states to control actions, such that $K(x) \in d(x)$. The set of exogenous actions that can occur in a state (and that are beyond the control of the agent) is given by a function $e : X \to 2^{\Sigma}$, such that $e(x) \subseteq d(x)$. We assume that $K(x) \notin e(x)$ for any state x.

For our running example we have assumed that the only exogenous action is *Ins*.

Definition 5.3.6 Let $A = (X, \Sigma, f, d)$ be a system. In presence of e, U, and K, the *closed loop system of* A, denoted A_K, is defined as the four-tuple (X, Σ, f, d_K), where $d_K(x) = (d(x) \cap \{K(x)\}) \cup e(x)$. $\Box$

Definition 5.3.7 Given a system A, a function e, and a set of states E, we say $S \subseteq X$ is *stabilizable* with respect to E if there exists a control law K such that for all x in S, x is alive and stable with respect to E in the closed loop system A_k. If $S = X$, we say A is stabilizable with respect to E. $\Box$

[1]It is also referred to as 'feedback law', 'feedback control' or 'state feedback' in the literature.

Running Example, Cont'd
It is easy to see that for $S = \{00\}$ (no objects in the buffers) and $E = \{00, 01, 02, 03\}$ (that is, we want to keep b_1 empty) S is not stabilizable w.r.t. E, since the exogenous action *Ins* can always interfere the task of bringing the system back to E.

On the other hand, $S = \{00\}$ is in fact stabilizable w.r.t. $E = \{0, 1, 2\} \times \{0, 1, 2, 3\}$ (that is, we allow at most two objects in b_1 at any time), since we can go from any of the states in $\{30, 31, 32, 33\}$ to E with the execution of at most two control actions, while the exogenous action is not defined for those states. If we would introduce a failure state, for example for the case when b_1 is full and *Ins* is executed, S would no longer be stabilizable w.r.t. E. In that case A_b would be only trivially stabilizable.

5.4 Maintainability

Our intuition behind maintainability is that we would like our system to 'maintain' a formula (or a set of states where the formula is satisfied) in the presence of exogenous actions. By 'maintain' we mean a weaker requirement than the temporal operator *always* ($\Box$) where $\Box f$ means that f should be true in *all* the states in the trajectory. The weaker requirement is that our system needs to get to a desired state within a finite number of transitions provided it is not interfered in between by exogenous actions. The question then is what role the exogenous actions play.

Our definition of maintainability relates a set of initial states S, that the system may be initially in, a set of desired states E, that we want to maintain, a system A and a control law K. Our goal is to formulate when the control law K maintains E assuming that the system is initially in one of the states in S. We account for the exogenous actions by defining the notion $Closure(S, A)$ of a closure of S with respect to A. This closure is the set of states that the system may get into starting from S. Then we define *maintainability* by requiring that the control law be such that if the system is in any state in the closure and is given a window of non-interference from exogenous actions then it gets into a desired state.

Suppose the above condition of maintainability is satisfied, and while the control law is leading the system towards a desired state an exogenous action happens and takes the system off that path. What then? The answer is that the state that the system will reach after the exogenous action will be a state from the closure. Thus, if the system is then left alone (without interference from exogenous actions) it will be again on its way to a desired state. So in our notion of maintainability, the control is always taking the system towards a desired state, and after any disturbance from an exogenous action, the control again puts the system on a path to a desired state.

We now formally define the notions of closure and maintainability.

Definition 5.4.1 (Closure)
Let $A = (X, \Sigma, f, d)$ be a system and S be a set of states. The *closure of A w.r.t. S*, denoted $Closure(S, A)$, is defined as

$$Closure(S, A) = \bigcup_{x \in S} R(A, x),$$

that is, the set of all states reachable from any member of S in the system A.□

Definition 5.4.2 With $Closure(S, A_K)$ and a set of states E we associate the set $Seq(S, A_K, E)$ of sequences $x = x_0, x_1, \ldots, x_{|X|}$, one for each $x \in Closure(S, A_K)$, where $x_{k+1} = x_k$ if $x_k \in E$, and $x_{k+1} \in f(x_k, K(x_k))$ otherwise. We also define

$$Seq_{\cup}(S, A_K, E) = \{\{x_0, \ldots, x_{|X|}\} \mid x_0, \ldots, x_{|X|} \in Seq(S, A_K, E)\}.$$

□

Definition 5.4.3 Let A be a system, x a state, and E a set of states. We call the sequence of control actions $a_1, \ldots, a_n$ a *plan from x to E* iff $f(x, [a_1, \ldots, a_n]) \subseteq E$.□

We note that the possible sequences of states that can occur between actions in a plan are members of $Seq(\{x\}, A_K, E)$, for some control law K, if we repeat the last state (which is in E) sufficiently many times.

Definition 5.4.4 Given a system $A = (X, \Sigma, f, d)$, a set of control action $U \subseteq \Sigma$, a specification of exogenous actions e, and a set of states E, we say a set of states S is *k-maintainable with respect to E* if there exists a control law K such that from each state x in $Closure(S, A_K)$, we can get to a state in E with at most k transitions, where each transition is dictated by the control K.

If there exists an n such that S is *n-maintainable* with respect to E, we say S is *maintainable* with respect to E.

If $S = X$, then we say A is maintainable with respect to E.□

Running Example, Cont'd
Above we showed that in A_b, $S = \{00\}$ is not stabilizable w.r.t. $E = \{00, 01, 02, 03\}$. Is then S maintainable w.r.t. E? Yes, since for the worst case system state, 33, a control law can move the system to 30 (by three transitions due to **Pos**) without the risk of interfering occurrences of exogenous actions. If there then is three transitions without interference the control law can apply $\mathbf{M}_{12}$ three times and be in 03. This implies that S is 6-maintainable w.r.t. E. We can with a similar argument show that S is 9-maintainable w.r.t. $\{00\}$. However, we have that S is *not* maintainable w.r.t., for example, $\{03\}$ (Since we cannot go from, for example, $\{00\}$, to $\{03\}$ with control actions only).

5.5 Algorithms

In this section we provide two algorithms to verify maintainability, and to generate control for maintainability, with correctness proofs.

5.5.1 Testing maintainability

Input: A system $A = (X, \Sigma, f, d)$, a set of states E, a set of states S, and a control K.
Output: To find out if S is maintainable with respect to E, using the control K.
Algorithm:
Step 1: Compute $Closure(S, A_K)$.
Step 2: Compute $Seq(S, A_K, E)$.
Step 3: If for all $Y \in Seq_\cup(S, A_K, E)$, $Y \cap E \neq \emptyset$ then S is maintainable with respect to E, using the control K; Otherwise it is not maintainable with respect to E, using the control K. $\Box$

For the correctness of this algorithm we show the following characterization of maintainability.

Proposition 5.5.1 Let A be a system, E a set of states, and K a control law. For every $Y \in Seq_\cup(S, A_K, E)$, $Y \cap E \neq \emptyset$ iff S is maintainable w.r.t. E, using the control law K.
Proof: $\Rightarrow$) Straightforward, since the control law can move the system from any state in $Closure(S, A_K)$ to a state in E in at most $|X|$ transitions.
$\Leftarrow$) If S is maintainable w.r.t. E using K, there is a natural number n such that from every state in $Closure(S, A_K)$ there exist a sequence of states ending in E of length at most n, where every transition between states is due to K. Clearly, $n \leq |X|$ so we repeat the last state in all sequences so that all sequences are of length $|X|$ and gather all such sequences into one set. This set is clearly equal to $Seq(S, A_K, E)$. Since every such sequence contains a member of E the proposition holds.$\Box$

The complexity of this algorithm depends on the computations in steps 1 and 2. Let $n = |X|$ and $S = X$ (this maximizes $Closure(S, A_K)$ and is, thus, the worst case). Clearly, we can compute $R(A_K, x)$, for every $x \in S$, in polynomial time (e.g. by building the depth-first search tree, and simply collecting the vertices), and since we only need to do this n times, the computation is polynomial. In the same manner we can compute $Seq(S, A_K, E)$ by recording the sequences that reach E during the depth-first search.

5.5.2 Generating a control law for maintainability of a set of states

Input: A system $A = (X, \Sigma, f, d)$, a set of states E, and a set of states S.
Output: Find a control K such that S is maintainable with respect to E, using the control K.
Algorithm:
Step 0: $S_{in} := S$, $S_{out} = \emptyset$.
Step 1: While $S_{in} \neq S_{out}$ Do.
Pick an x from $S_{in} \setminus S_{out}$. Find a *plan of minimal length* from x to a state in E using only control actions.

If no such plan exists then EXIT and return(FAIL).
Let a be the first action of that plan.
Assign $K(x) = a$.
$S_{out} := S_{out} \cup \{x\}$
$S_{in} := S_{in} \cup f(x, a) \cup \{y : y \in f(x, b)$, for some $b \in e(X)\}$.
Step 2: If $S_{in} = S_{out}$, return(S_{out}, K).

Proposition 5.5.2 If the above algorithm terminates by returning S' and K, then: (i) $S' = Closure(S, A_K)$, and (ii) S is maintainable with respect to E, using the control K.
Proof: We assume that for a particular S the algorithm has returned S' and K.
(i) We start by showing that $S_{out} \subseteq Closure(S, A_K)$ is an invariant of the algorithm by induction over the number of iterations, n, of the algorithm, and write S_{out}^n and S_{in}^n to denote the corresponding sets of states after n iterations. Trivially, $S_{out}^0 \subseteq Closure(S, A_K)$, and $S_{out}^1 \subseteq Closure(S, A_K)$ holds since a member of S_{out}^1 necessarily is a member of S. We assume that $S_{out}^k \subseteq Closure(S, A_K)$ and inspect a state $x \in S_{out}^{k+1}$ such that $x \notin S_{out}^k$. Again, if $x \in S$ we have $x \in Closure(S, A_K)$ by the definition of closure. Otherwise, there exists a state $x' \in S_{in}^k$ and an action a such that $x \in f(x', a)$, from the construction of S_{in}^k. By the induction hypotheses we have that $x' \in Closure(S, A_K)$ and since x is reachable from x' we also have $x \in Closure(S, A_K)$.

For the other inclusion we take a state $x \in Closure(S, A_K)$. By the definition of closure there exists a state $x' \in S$ such that x is reachable from x' which implies that there is a trajectory consistent with A_K: $x' = x_0, a_1, x_1, \ldots, a_n, x_n = x$ where $x_{k+1} \in f(x_k, a_{k+1})$ and $a_{k+1} \in d_K(x_k)$. We prove that $x \in S'$ by induction over the length of the trajectory.

For $n = 0$ we have $x \in S \subseteq S'$. We assume that x_k belongs to S' for all $k < n$. By the definition of trajectories consistent with A_K we have that $a_{k+1} \in d_K(x_k)$ which implies either that $K(x_k) = a_{k+1}$ (1) or $a_{k+1} \in e(x_k)$ (2). By the induction hypotheses, x_k is added to S_{out} at some iteration of the algorithm, and in the same iteration all members of the set $L = f(x_k, a_{k+1}) \cup \{y : y \in f(x, b)$, for some $b \in e(X)\}$ are added to S_{in}. From (1) or (2) we get that x_{k+1} belongs to L, which means that it will be picked by the algorithm in a later iteration, and added to S_{out} and thus belongs to S'. Therefore, $S' = Closure(S, A_K)$.

(ii) From (i) we know that $S' = Closure(S, A_K)$, so we may construct $Seq(S, A_K, E)$. Since the algorithm selects states reached by (sub-) plans of minimal length, the same state (except for the last that belongs to E) cannot occur more than once in every sequence in $Seq(S, A_K, E)$. It is also clear that in every such sequence there is a state belonging to E. Thus, we can apply proposition 5.5.1, which proves this proposition. $\square$

For the complexity of this algorithm, we begin by observing that our systems easily translates into *generalized Büchi automata*, and that such automata can be translated into our systems. This implies that we can exploit the following result by De Giacomo and Vardi [2000]:

Theorem 5.5.3 Planning in generalized Büchi automata is PSPACE-complete[2].□

This means that the algorithm has to solve a PSPACE-complete problem in every iteration. However, if we would restrict systems to only allow *deterministic* transition functions (that is, $f : X \times \Sigma \to X$) the phrase in italics in step 1 of the algorithm could be replaced by "*a minimal length trajectory consistent with A*". The correctness and completeness proofs are easily adaptable to accommodate this change. In this case, the systems could be reduced to "standard" Büchi automata, and *vice versa.* Then we can exploit the following result by De Giacomo and Vardi:

Theorem 5.5.4 Planning in standard Büchi automata is NLOGSPACE-complete.□

At this point we would like to point out the relation between our work here and some research on reactive and situated agents [Kaelbling and Rosenschein, 1991]. In [Kaelbling and Rosenschein, 1991], they state that in a control rule 'if c then a', the action a, must be the action that *leads* to the goal from any situation that satisfies the condition c. The above algorithm interprets the notion of 'leading to' as the first action of a minimal cost plan.

5.6 Generalization: (k,l)-Maintainability

In this section we generalize the notion of maintainability and show that the notion of stabilizability is a special case of this generalization. Our generalization is based on the intuition that perhaps, we can allow a limited number, l, of exogenous actions during our so called 'window of non-interference', k, and still be able to get back to a state in E. We refer to this general notion as *(k,l)-maintainability.*

Definition 5.6.1 Given a system $A = (X, \Sigma, f, d)$, a set of agents action $U \subseteq \Sigma$, a specification of exogenous actions e, and a set of states E, we say a set of states S is *(k,l)-maintainable* ($l \leq k$) with respect to E if there exists a control law K such that for each state x in $Closure(S, A_K)$, all trajectories – consistent with A_K – from x whose next k transitions contain at most l transitions due to exogenous actions and the rest is dictated by the control K, reach a state in E by the k-th transition. □

Proposition 5.6.2 $(k,0)$-maintainable is equivalent to k-maintainable. (A set of states S is $(k,0)$-maintainable with respect to a set of states E if and only if S is k-maintainable with respect to E.)
Proof: Since we will have some time that we do not allow any exogenous actions to interfere the system so that all trajectories which are consistent with A_K from x in $Closure(S, A)$ whose next k actions contain at most 0 transitions due to exogenous

[2]The reader should note that this, in fact, is good news, since planning in general with non-deterministic operators is EXPSPACE-complete. The reason for the low complexity classification is that the automata provide a *sparse representation* of the problem.

actions, reach a state in E within k transitions. And a control law K guarantees that there is at least one (and exactly one) action to be taken at each state of $Closure(S,A)$, thus guarantees an existence of at least one trajectory from each state x of $Closure(S,A)$. So S is k-maintainable.

($\Leftarrow$) Suppose that S is k-maintainable with respect to E. Then there exists a control law K such that for each state x in $Closure(S,A)$, we can get to a state in E within k transitions, where each transition is dictated by the control K, and is not an exogenous action. I.e. at most 0 transition is due to exogenous actions. Since k actions which do not contains any exogenous actions determine only one trajectory from a state x to a state in E (since the control law has only one feasible action at each state), we can say that all trajectories (actually there is only one in this case) which are consistent with $A_K = (X, \Sigma, f, d_K)$ from a state x whose next k actions contain at most 0 exogenous action reach a state E within k transitions. Thus S is $(k,0)$-maintainable with respect to E.

Therefore $(k,0)$-maintainable is equivalent to k-maintainable. □

Proposition 5.6.3 A set of states S is stabilizable iff S is alive and there exists an integer m such that S is *(m,m)-maintainable* with respect to E.
Proof: ($\Rightarrow$) Suppose a set S is stabilizable with respect to E. So there is a control law K such that S is alive and stable with respect to E in the closed loop system A_K. Thus all trajectories consistent with A_K and starting from x go through a state in E in a finite number of transitions and they visit E infinitely often afterwards.
Case 1. Let $x \in S$. Then by assumption, all trajectories from x go through a state in E within finite transition. Since there is a finite number of such trajectories, take n_x to be the maximum number of transition of the trajectories from x to a state E. This includes some actions due to exogenous actions, but at most n_x of them.
Case 2. If $x \in Closure(S, A_K) \backslash S$, then the state x is in a trajectory T from a state, say y, in S. Since all trajectories consistent with A_K reach from a state y in S to a state in E with finite number of transitions and visit E infinitely often afterwards, this partial trajectory T' which starts from a state x and follow the trajectory T afterwards will visit E infinitely often. So through this trajectory T', we can reach from x to a state in E within finite transitions, say n_x. This includes at most n_x transitions which are due to exogenous actions.

Then take $m = max\{n_x | x \in Closure(S,A)\}$. Since $Closure(S,A)$ is finite, m exists ($m < \infty$). From each state x in $Closure(S, A_K)$, all trajectories, consistent with A_K, whose next m transitions contain at most m transitions due to exogenous actions and the rest is dictated by the control K reach a state in E within m transitions. Thus S is (m,m)-maintainable with respect to E. □.

($\Leftarrow$) Suppose that S is alive and there exists an integer m such that S is (m,m)-maintainable with respect to E. Thus there exists a control law K such that from each state x in $Closure(S, A_K)$, all trajectories, consistent with A_K, from x whose next m actions contain at most m transitions due to exogenous actions reach a state in E within m transition. Thus all trajectories, consistent with A, starting from x go though a state y in E, any state, say z, which we can reach from that state y in E belongs to $Closure(S, A)$ since that state z can be reached from a state x in S though a state y in E. And with assumption, all trajectories which are consistent with A_K from a state z whose next m actions contain at most m transitions due to exogenous actions reach a state in E within m transitions again. Hence those trajectories visit E within m transitions every time it leaves E to a state outside of E afterwards. I.e. they visit E infinitely often afterwards. Thus S is stable with respect to E in the closed loop system A_K. Since S is alive by assumption, S is stabilizable with respect to E.

Therefore a set of states S is stabilizable with respect to a set of states E if and only if S is alive and there exists an integer m such that S is (m, m)-maintainable with respect to E.$\Box$

5.7 Stability-based execution monitoring

The obvious monitoring task for a stabilizing controller is to detect whether the system is stable or not after a detected discrepancy. By viewing a discrepancy as a new exogenous action, can the controller still stabilize the system? Depending on the answer to that question we again have the benign/malignant classification, where a discrepancy that leads to a state within the range of the current controller is benign, and all other discrepancies are malignant. For malignant discrepancies we recover applying the following three steps, if possible.

1. Incorporate the discrepancy in the plant model as a new effect of the executed action (model tuning recovery), then,
2. construct a new controller from the new plant model, and finally,
3. apply the new controller to the current state.

We assume that we have a closed-loop system $A_K = \langle X, \Sigma, f, d_K \rangle$ for a controller K. Let x be a state in which the control action a was executed, and let x' be the measured state after the execution. Clearly, if $x' \notin f(x, a)$ we have a discrepancy, since f is the prediction function. However, for the current set of initial states S and goal states E, if $x' \in Closure(S, A_K)$ we know that the controller will be able to force the system to a state in E. This is a benign discrepancy, and we take no further action. Instead, if $x' \notin Closure(S, A_K)$, we do not know whether E is reachable anymore, so perform the following:

1. Set $f' = (f - \langle x, a, f(x, a) \rangle) \cup \langle x, a, (f(x, a) \cup \{x'\}) \rangle$, that is, add the new effect to the transition function. Construct $A' = \langle X, \Sigma, f', d \rangle$ and $S' = S \cup \{x'\}$, that is, add the new state as a possible initial state.

2. For A', S', and E, construct a new controller K', for example by using Algorithm 5.5.2. If there is no such controller, we have a fatal error and the execution should be stopped.

3. Otherwise, execute action $K'(x')$.

This algorithm performs a parsimonous model-tuning recovery, in the sense that the same discrepancy with the same classification cannot occur again.

Chapter 6

Execution monitoring of hybrid-systems controllers

6.1 Introduction

In Chapter 2 it was noted that the idea of automated generation of a state transition model from a rule-based control program is impossible in the general case, and very difficult even under the limiting assumptions given by control designers. In this chapter we will present work that remedies some of those problems: We will instead of a control program begin with a *specification* of a closed-loop system written in a language that will enable us to generate a state transition diagram more easily.

As mentioned in Chapter 2 there are numerous modeling formalisms for control systems. On one hand we have purely discrete formalisms such as finite automata (e.g. [Ramadge and Wonham, 1989]) or planning formalisms (e.g. Fikes and Nilsson [1971]), and on the other we have the purely continuous differential equations (e.g. Faurre and Depeyrot [1977]). However, it is clear that most actual control systems are *hybrid*, that is, that they contain both discrete and continuous elements.

There is a large body of work on the problem of hybrid controller *synthesis* being pursued in AI research (e.g. [Bjäreland and Driankov, 1999]), hybrid systems (e.g. [Zhang and Mackworth, 1995, Henzinger and Kopke, 1997]), and control theory (e.g. [Lennartsson *et al.*, 1996]) communities. Typically the synthesis formalisms are designed for *verification* purposes, and the synthesized controllers are often specifications of controllers rather than actual implementations of these. One of the reasons for this gap between verification and use is that controllers in use necessarily need to be able to handle many more cases than what normally is covered by the verification. According to the folklore of process control, more than 70% of a medium sized PLC program is devoted to fault detection, isolation and recovery. The remaining 30% is taking care of the actual control, and is the part that is suited for verification. The verification-use gap is unfortunate, to say the least, since the purpose of verification is to show that the verified systems can be used in a safe way.

We believe that we are taking one step in closing the verification-use gap by, in this chapter, proposing not only that controller programs should be synthesized from specifications, but also that representations useful for execution monitors should be synthesized from the same specifications. We do this by utilizing closed-loop *Hybrid Automata* (HA) specifications [Alur *et al.*, 1992], assuming that a (reactive) control program has been synthesized, and constructing a rich model where it is possible to reason about the expected effects of control actions.

Rectangular hybrid automaton (RHA) is a modeling formalisms where the continuous behavior is modeled by differential inequalities (e.g. $0 \leq \dot{x} \leq 1$), and the discrete behavior in terms of (instantaneous) mode switches. We will assume that the RHAs we study are specifications of controllers, and that every mode switch is due to an invoked control action. Moreover, we will restrict ourselves to handling controllers for which the control goal can be formulated as safety requirements (that some condition should, or should not, be maintained during the control).

6.2 Preliminaries

In this section we will present the formal definitions of rectangular hybrid automata. The following definitions are taken verbatim from [Henzinger and Kopke, 1997].

6.2.1 Rectangular hybrid automata

Definition 6.2.1 Let $X = \{x_1, \ldots, x_n\}$ be a set of real-valued variables. A *rectangular inequality* over X is an expression of the form $x_i \sim c$, where c is an integer constant, and $\sim \in \{<, \leq, \geq, >\}$. A *rectangular predicate* over X is a conjunction of rectangular inequalities. We denote the set of all rectangular predicates over X with $Rect(X)$. The set of vectors $\vec{z} \in \mathbb{R}^n$ that satisfies a rectangular predicate is called a *rectangle*. For a particular rectangular predicate ϕ, we denote the corresponding rectangle with $[\![\phi]\!]$. By writing ϕ^i, for a rectangular predicate ϕ, and a variable index i, we denote the conjunction of all rectangular inequalities in ϕ only involving the variable x_i. For a set of indices, I, we define $\phi^I = \bigwedge_{i \in I} \phi^i$.□

Definition 6.2.2 (Rectangular automaton)
A *rectangular automaton* A consist of the following components.
Variables. The finite set $X = \{x_1, \ldots, x_n\}$ of real-valued variables representing the continuous part of the system. We write $\dot{X} = \{\dot{x_i} \mid x_i \in X\}$ for the set of dotted variables, representing the first derivatives. For convenience, we write X' to denote the set $\{x'_i \mid x_i \in X\}$ (which we will use to connect variable values before and after mode switches).
Control Graph. The finite directed multi-graph $\langle V, E\rangle$ represents the discrete part of the system. The vertices in V are called *control modes* which we also will refer to as *locations*. The edges in E are called *control switches*. The switches will sometimes be viewed as functions, i.e. we can say that $e(v) = v'$ iff $e = \langle v, v'\rangle$. In a graphical representation of an automaton the locations correspond to the boxes and

the switches to the arrows between boxes.
Initial Conditions. The function $init : V \to Rect(X)$ maps each control mode to its *initial condition*, a rectangular predicate. When the automaton control starts in mode v, the variables have initial values inside the rectangle $[\![init(v)]\!]$.
Invariant Conditions. The function $inv : V \to Rect(X)$ maps each control mode to its *invariant condition*, a rectangular predicate. The automaton control may reside in mode v only as long as the values of the variables stay inside the rectangle $[\![inv(v)]\!]$. We define $inv(A)$ as $inv(A) = \bigwedge_{v \in V} inv(v)$. Below, we will apply ontological control in this setting and will need to distinguish between the configuration and plant formula parts of the invariants. Thus, inv_{conf} maps a control mode to its configuration invariant, and we let $inv_{plant} = inv - inv_{conf}$ (in a set theoretic understanding of the functions).
Jump Conditions. The function *jump* maps each control switch $e \in E$ to a (non-rectangular) predicate $jump(e)$ of the form $\phi \wedge \phi' \wedge \bigwedge_{i \notin update(e)} x_i = x'_i$, where $\phi \in Rect(X)$, $\phi' \in Rect(X')$, and $update(e) \subset \{1, \ldots, n\}$. The jump condition $jump(e)$ specifies the effect of the change in control mode on the values of the variables: each unprimed variable x_i refer to the corresponding value before the control switch e, and each primed variable x'_i to a corresponding value after the switch. So the automaton may switch across e if

1. the values of the variables are inside $[\![\phi]\!]$, and
2. the value of every variable x_i with $i \notin update(e)$ is in the rectangle $[\![{\phi'}^i]\!]$.

Then, the value of every variable x_i with $i \notin update(e)$ remains unchanged by the switch. The value of every x_i with $i \in update(e)$ is assumed to be updated nondeterministically to an arbitrary value in the rectangle $[\![{\phi'}^i]\!]$. For a jump condition $jump(e) \equiv \phi_e \wedge \phi'_e \wedge \bigwedge_{i \notin update(e)} x_i = x'_i$, we define $jump'(e) \equiv \phi_e$, to denote the actual condition that forces the switch e. Below, we will apply ontological control in this setting and will need to distinguish between the configuration and plant formula parts of the jump conditions. Thus, $jump'_{conf}$ maps a control switch to its configuration jump condition, and we let $jump'_{plant} = jump' - jump'_{conf}$ (in a set theoretic understanding of the functions).
Flow Conditions. the function $flow : V \to Rect(\dot{X})$ maps each control mode v to a *flow condition*, a rectangular predicate that constrains the behavior of the first derivatives of the variables. While time passes with the automaton control in mode v, the values of the variables are assumed to follow nondeterministically any differentiable trajectory whose first derivative stays inside the rectangle $[\![flow(v)]\!]$.
Events. Given a finite set Λ of *events*, the function $event : E \to \Lambda$ maps each control switch to an event.

Thus, a rectangular automaton A is a nine-tuple

$$\langle X, V, E, init, inv, jump, flow, \Lambda, event \rangle$$

□

6.2.2 Example – Railroad crossing

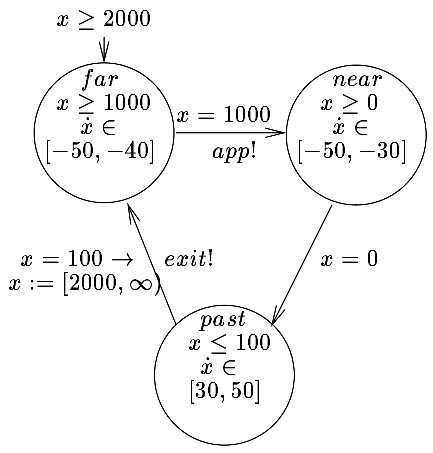

Figure 6.1: Hybrid automaton modeling the train in the train-and-gate example.

Our running example is a HA model of a railroad crossing presented by Alur *et al.* [1993]. The model consists of three sub-models, a train, a gate, and a controller. In [Alur *et al.*, 1993] the model was used to verify that the controller guaranteed certain safety properties for the overall system. Our use of this example is to illustrate how execution monitoring of the controller can be performed by using information embedded in the HA sub-model for the train.

The HA model of the train (Fig. 6.1) has three locations: far, $near$, and *past*, defined as the distance, x, between the train and the gate. That is, $x \geq 1000$ is the invariant for the location far, $x \geq 0$ for $near$, and $x \leq 100$ for *past*. Furthermore, when the jump condition $x = 1000$ is satisfied in location far, the train HA switches from far to $near$, and sets the boolean variable *app* to true. This variable (and the variable $exit$) occur with exclamation marks in this HA to denote that they are set to true during one sampling interval (and reset afterwards) as a result of an action in this part of the system. In the HA model of the controller we will see the same two variables with question marks to denote that switches are *caused* by changes in their values.[1] When the HA is in $near$ and $x = 0$ is satisfied, the HA switches from $near$ to *past*. Finally, when the current location is *past* and $x = 100$, then x is reassigned to an arbitrary value in $[2000, \infty)$, and the variable *exit* is set to true. The HA model of the gate (Fig. 6.2) has four location: *up*, *open*, *down*, and *closed*. Initially, the gate is open and the angle of the gate to the ground is 90° (the angle is measured by g). Whenever the variable *lower* is set to true (by the controller) the automaton switches to *down* where the gate is lowered a rate of 9° per second.

[1]The exclamation and question marks are in fact HYTECH syntax [Henzinger *et al.*, 1997], and are used to denote that the automatons are *synchronized* by such variables.

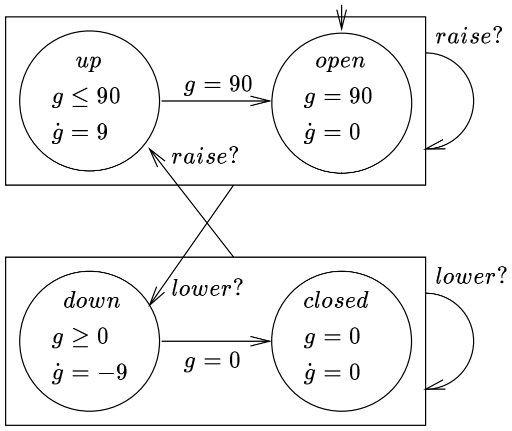

Figure 6.2: Hybrid automaton modeling the gate in the train-and-gate example.

When the angle is 0° a switch to *closed* occurs. If the gate is in location *down* or *closed* and the variable *lower* is set to true (by the controller), the location will be the same. If *raise* is set to true, a switch to *up* will occur, and the gate will start to raise by 9° per second until it reaches 90° where a switch to *open* occurs.

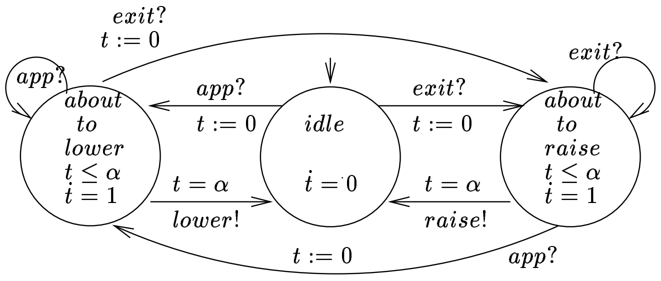

Figure 6.3: Hybrid automaton modeling the controller in the train-and-gate example.

The HA model of the controller is given in Fig. 6.3. The idea is that the controller obtains information from the train automaton whether the train is approaching or exiting (i.e. from the variables *app* and *exit*), then waits for a period of time, α, and sets *lower* or *raise* to true, respectively, which is then handled by the gate.

The system with the three automata has been verified in HYTECH by Henzinger *et al.* [1997] to prove that for $\alpha < 49/5$, when the train is within 10 meters from the gate, the gate is always fully closed.

6.3 Automatic generation of models for execution monitoring

We will now demonstrate how a model for execution monitoring can be generated for a particular example: A train-and-gate example from [Alur *et al.*, 1993].

To be able to formally describe the translation of an HA to a state transition diagram, and to formally define execution monitoring on that representation, we will formally define state transition diagrams.

Definition 6.3.1 (State Transition Diagram)
A *state transition diagram* is a digraph $G = \langle V, E\rangle$ with labeled vertices and edges. For a vertice $v \in V$ we define a function $formula(v)$ that maps the vertice to a boolean formula. Similar to the configuration and plant formula versions of the jump conditions of control modes, we define $formula_{conf}$ and $formula_{plant}$. Moreover, there is a relation $Idle(v)$ that holds iff the vertice denotes an idle state. For an edge $e \in E$ we define $start(e)$ to be the starting vertex of the edge, $end(e)$ to be the ending vertex, and $action(e)$ to be a conjunction of control actions, or the dummy label **nop**.□

Definition 6.3.2 (Transformed hybrid automata)
Let $A = \langle X, V_A, E_A, init, inv, jump, flow, \Lambda, event\rangle$ be rectangular automaton. Below, we define a state transition diagram $\mathcal{T}(A) = \langle V_{\mathcal{T}(A)}, E_{\mathcal{T}(A)}\rangle$.

1. For every $v \in V_A$ there is a vertice $\mathcal{T}(v) \in V_{\mathcal{T}(A)}$ such that
   - $formula(\mathcal{T}(v)) = inv(v)$ $(formula_{conf}(\mathcal{T}(v)) = inv_{conf}(v))$, and
   - $Idle(\mathcal{T}(v))$ holds.
2. For every $e \in E_A$ there is a vertice $\mathcal{T}(e) \in V_{\mathcal{T}(A)}$ such that
   - $formula(\mathcal{T}(e)) = jump'(e)$, $(formula_{conf}(\mathcal{T}(e)) = jump'_{conf}(e))$ and
   - $\neg Idle(\mathcal{T}(e))$ holds.
3. No other elements exist in $V_{\mathcal{T}(A)}$ or $E_{\mathcal{T}(A)}$ than the ones described above.
4. For $v \in V_{\mathcal{T}(A)}$ such that $Idle(v)$ holds, we know by construction that there exists a $v' \in V_A$ where $v = \mathcal{T}(v')$. For every such v, there are edges $e \in E_{\mathcal{T}(A)}$ such that $action(e) = \mathbf{nop}$, $start(e) = v$ and $end(e) = \mathcal{T}(e_A)$ for every $e_A \in E_A$ where $e_A(v')$ is defined.
5. For $v \in V_{\mathcal{T}(A)}$ such that $\neg Idle(v)$ holds, we know by construction that there exists a $e_A \in E_A$ where $v = \mathcal{T}(e_A)$. For every such v, there are edges $e \in E_{\mathcal{T}(A)}$ such that $action(e) = event(e_A)$, $start(e) = v$, and $end(e) = \mathcal{T}(v_A)$, for every $v_A \in V_A$ where there exists a vertice $v'_A \in V_A$ for which $e_A(v'_A) = v_A$.

□

The basic assumption governing the transformation in Definition 6.3.2 is that *discrete change is controlled change, and vice versa.* Thus, for the purpose of execution monitoring of hybrid systems in this thesis, we only monitor the discrete state transitions.

6.3.1 Railroad crossing cont'd

We will now construct a state transition diagram for the railroad crossing example. The problem is that it is impossible to clearly distinguish between the locations *idle*, *about_to_lower*, and *about_to_raise* by only looking at the invariants of the locations. That is, the "names" of the locations play an important role in the model. As we have argued in previous chapters, situation assessment relies on some kind of "uniqueness" of materialized states, which, for example, can be achieved by forcing states to be mutually exclusive. We will do this for this example too, as follows: We begin by constructing a set of production rules that could be a control program for the HA in Figure 6.3, where control actions are executed if a jump condition from a location is satisfied while the system is in that location. That controller will not have mutually exclusive conditions for the control actions. Then by exploiting the synchronization information in the controller HA, the train HA (Figure 6.1), and the gate HA (Figure 6.2), and by viewing the locations in the train and the gate HAs as sensor signals to the controller, we can construct a control program with mutually exclusive conditions. It is then possible to add those conditions to the original controller HA as jump conditions, add more information to the invariants, and then to construct the state transition diagram as in Definition 6.3.2.

We define an *HA-state* of a HA to be a tuple $\langle l, inv \rangle$, where l is a location (a symbol) and $inv(l)$ is the invariant of l.

A *control rule* is an expression $S \wedge j \Rightarrow a$, where S is an HA- state, j a jump condition, and a a control action. The conjunction $S \wedge j$ is the *precondition* of a control rule. A control program is a set of control rules.

The control program for the controller in Fig. 6.3 will then be as follows:

$$\begin{aligned}
&\langle idle, true \rangle \wedge exit \Rightarrow start\_clock \\
&\langle idle, true \rangle \wedge app \Rightarrow start\_clock \\
&\langle about\_to\_lower, t \leq \alpha \rangle \wedge exit \Rightarrow start\_clock \\
&\langle about\_to\_lower, t \leq \alpha \rangle \wedge t = \alpha \Rightarrow \\
&\qquad lower \wedge stop\_clock \qquad (6.1) \\
&\langle about\_to\_raise, t \leq \alpha \rangle \wedge app \Rightarrow start\_clock \\
&\langle about\_to\_raise, t \leq \alpha \rangle \wedge t = \alpha \Rightarrow \\
&\qquad raise \wedge stop\_clock \qquad (6.2)
\end{aligned}$$

In the example (Fig. 6.3) we choose to view every resetting of the clock, t, to be a control action, *start_clock*, which is assumed to reset and start the clock. Moreover, we introduce the control action *stop_clock* that is assumed to stop the clock. The action *start_clock* replaces $t := 0$ as an action on edges, and *stop_clock* is introduced for every edge where the resulting location includes $\dot{t} = 0$. [2]

It is clear that the control program above is valid if it is started in the *idle* location, the train is started in *far*, and the gate in *open*, i.e. the control program will behave well under the exact circumstances the automata are verified for. However, in engineering practice it is unrealistic to assume anything about the initial state. Thus, it is necessary to base the execution monitoring on *external information* (such as sensor or actuator information), rather than an assumption on the current state of the controller. Also, we can see that the preconditions of control rules (6.1) and (6.2) cannot be distinguished *logically* based on their invariants. Thus, we require the control rules to be "physically" mutually exclusive, i.e. that it is physically impossible for two preconditions to be satisfied simultaneously.[3]

Now, we will take advantage of two assumptions: First, that we know whether the clock is running or not (the truth-value of a variable *clock_on*). Secondly, that we can sense the current location of the train (that the controller has access to the truth-values of the variables *far*, *near* and *past*).

To characterize the controller locations we begin in the verified initial location *idle* and note that the clock is off, and that the corresponding initial location in the train HA is *far*. For *about_to_lower* we can see that the previous train location may have been *far*, and that a switch in the train HA, from *far* to *near* must have occurred, that is, when the controller is in *about_to_lower* it is possible that the train is in *near*. It is also possible that the train switches to *past* before $t = \alpha$ is satisfied. Thus, for *about_to_lower* we know that the clock is on, and that the train is in either *near* or *past*. If the condition $t = \alpha$ eventually is satisfied the controller switches to *idle* which implies that the train is in *far* (from the initialization), *near* or *past* when the controller is in *idle*. Similarly, we note that *about_to_raise* can be characterized by $clock\_on \wedge far$. To summarize, we can characterize the location *idle* with the formula $\neg clock\_on \wedge (far \vee near \vee past)$ which we may translate to $\neg clock\_on$, *about_to_lower* with $clock\_on \wedge (near \vee past)$. But these characterizations only consider the activations of control actions, and to perform execution monitoring we also need to represent the states in which the controller is waiting in a location. For example, when the train is approaching, and the clock is running, there is a time interval when no control action is invoked by the controller.

By replacing the HA-states in the control program above by the characterizations we obtain the set of rules below.

[2]To be able to verify a system with this change, we would have to model the clock too with HA. However, this fairly simple modeling task is not included here.

[3]One way of constructing physically mutually exclusive preconditions is to ensure that they are logically mutually exclusive. However, this may be very inconvenient for the control designer.

$$\begin{array}{ll}
(S1) & \neg clock\_on \wedge exit \Rightarrow start\_clock \\
(S2) & \neg clock\_on \wedge app \Rightarrow start\_clock \\
(S3) & clock\_on \wedge (near \vee past) \wedge exit \Rightarrow \\
 & start\_clock \\
(S4) & clock\_on \wedge (near \vee past) \wedge t = \alpha \Rightarrow \\
 & lower \wedge stop\_clock \\
(S5) & clock\_on \wedge far \wedge app \Rightarrow start\_clock \\
(S6) & clock\_on \wedge far \wedge t = \alpha \Rightarrow \\
 & raise \wedge stop\_clock
\end{array}$$

The preconditions of the rules above can now be added to as jump conditions to the automata, in the obvious way. Below, we will use the labels $S1$ – $S6$ to denote the edges in the HA. For example, $jump'(S1) = \neg clock\_on \wedge exit$ and $event(S1) = start\_clock$.

The characterizations is also used to construct new invariants for the locations, with the following result.

$$\begin{array}{ll}
(IS1) & \neg clock\_on \\
(IS2) & clock\_on \wedge (near \vee past) \wedge t \leq \alpha \\
(IS3) & clock\_on \wedge far \wedge t \leq \alpha
\end{array}$$

These three rules model the cases where the controller is waiting in a location *idle*, *about_to_lower*, and *about_to_raise*, respectively, and we will use the labels $IS1$ – $IS3$ to denote the locations in the HA. For example, $inv(IS1) = \neg clock\_on$.

By performing the transformation in Definition 6.3.2 we obtain a state transition diagram resembling that in Figure 6.4, where the formulas of the states are $inv(Sx)$ and $inv(ISx)$, respectively, and where the unlabeled edges should be marked by **nop**.

In general, we can note that if the control program has n rules, and the original HA has m locations, the state transition diagram will have $n + m$ states (m is the number of idle states).

6.4 Execution monitoring

We begin by defining the execution semantics of the particular control programs used in this chapter. If $V = \{v_1, \dots, v_n\}$ is the set of variables of (or inputs to) the controller (in our example we have $V = \{clock\_on, exit, app, near, past, far, t = \alpha, t \leq \alpha\}$.[4]) a vector $\sigma = \langle b_1, \dots, b_n \rangle$, with $b_i \in \{0, 1\}$, is called a *sample*. The set

[4]Since t only is used in two particular comparisons to α we choose to view the comparisons as a boolean variable. We could easily extend the semantics to handle real-valued variables, as well.

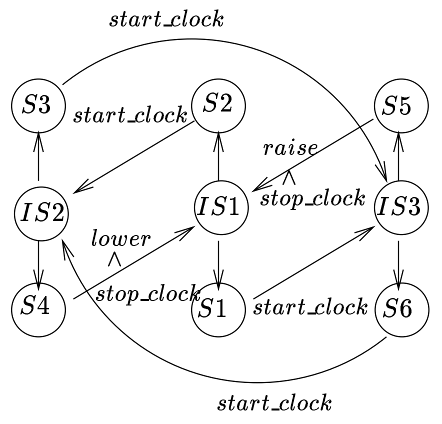

Figure 6.4: A state transition diagram representing the controller.

of all samples for an application is denoted Σ. We determine the truth-value of a state in a given sample as usual in propositional logic. During the execution of a controller, it will receive a stream of samples.

Now, we define how the four first functions of execution monitoring are implemented in this work.
Situation assessment: When a sample is received by the controller, there are only three distinct possibilities:

1. either exactly one idle state, is, (that is $Idle(is)$ holds) and no other state is satisfied, or

2. one idle state, is, and a state, s, where there exists an unlabeled edge from is to s, is satisfied, or

3. no state is satisfied.

The reason why no other cases can occur is that we have physically mutually exclusive jump conditions and invariants. The first case corresponds to when the controller is inactive waiting for something to happen), the second to when the condition of a control rule is materialized (note that in this case an idle state and an non-idle state are simultaneously satisfied, since the invariant of a location and a jump condition from that location are not physically mutually exclusive. However, two different jump conditions from the same location are physically mutually exclusive.) Since we do not assume that all conceivable samples can be handled by the controller, the third case is possible.

The situation assessment function $Sit : \Sigma \to V_{\mathcal{T}(A)}$ is defined as

$$Sit(\sigma) = \begin{cases} is & \text{for case 1,} \\ s & \text{for case 2, and} \\ s_{\text{fault}} & \text{for case 3,} \end{cases}$$

where s_{fault} is a dummy state. The motivation for cases 1 and 3 should be clear, and for case 2 we set the current state to s since we know that then the controller will invoke a control action.

Expectation assessment: When the initial sample is received by the controller the expectation assessment mechanism is idle.

We define the function $Exp : S \times A \to 2^S$, where S is the set of states, and A is the set of control actions. The set of states in our example is the set of $S1 - S6$ and $IS1 - IS3$, the set of control actions is $A = \{start\_clock, raise \wedge stop\_clock, lower \wedge stop\_clock\}$. From the generated state transition diagram (in our example, Fig. 6.4) we can obtain the definition of Exp. If the controller executed an action a in the state s, there are two possibilities:

- $a = \mathbf{nop}$ which means that s is an idle state. Let $M = \{s'_1, \ldots, s'_m\}$ be the set of all states such that there is an edge from s to s'_i labeled **nop**, for $1 \leq i \leq m$. Then $Exp(s, a) = M \cup \{s\}$, that is, we expect that any of the states connected to s, and s can be satisfied next.

- $a \neq \mathbf{nop}$ which means that a "real" control action is executed, and that s is a non-idle state. If $M = \{s'_1, \ldots, s'_m\}$ is the set of all states such that there is an edge labeled with a from s to s'_i, then $Exp(s, a) = M$.

Discrepancy detection: This is not particularly problematic: If the controller performs action a in state s, and the controller reads a sample σ such that $Sit(\sigma) \neq s$, we say that we have a discrepancy iff $Sit(\sigma) \notin Exp(s, a)$.

The assumption that $Sit(\sigma) \neq s$ encodes a precise notion of when state transitions occur. That is, if s is an idle state, then a state transition occurs exactly when a non-idle state materializes, and when s is non-idle a state transition occurs exactly when an idle state materializes.

Discrepancy classification: The two classification methods introduced in Chapters 4 (ontological control) and 5 (stability-based execution monitoring) can (fairly easily) be adopted to the HA framework. Below we address the two methods separately.

In Chapter 4 the classification scheme of ontological control required energized actions. In that setting this meant that an action always was active, which coincides with the execution semantics of rule-based control languages. The reason for this restriction was that we needed precise knowledge about when state changes occurred. Here, we have distinguished between states that trigger actions and idle states which means that we can model a broader class of controllers. However, we still need the "non-logging" assumption, which in our case means that if there is an edge from a non-idle state, s, to an idle state is, we have that $formula_{conf}(s)$ and

$formula_{conf}(is)$ are mutually exclusive (or that their respective rectangles have an empty intersection).

Assume that a discrepancy has occurred, that is $Sit(\sigma) \notin Exp(s, a)$. Recall that the distinction between EA and VOA was made based on whether the configuration had changed or not. That is, if

$$formula_{conf}(Sit(\sigma)) \equiv \bigwedge_{s' \in Exp(s,a)} formula_{conf}(s')$$

is satisfiable (that is, that the respective rectangles of the two formulas are equal) then we have an EA. Otherwise a VOA has occurred.

For stability-based execution monitoring we begin by noting that there is no notion of "exogenous action" in the HA framework. But, in the state transition diagrams generated above, there are "unlabeled" transitions which cannot be controlled. If we analyze the diagram in Figure 6.4, and view the idle transitions as exogenous actions, we can note that the system is only trivially maintainable (that is, that it is only maintainable w.r.t. the entire state set). This is a general feature of such systems, since every transition from every non-idle state goes to an idle state (via a controlled transition), and every transition from an idle state is unlabeled. The interesting monitoring task is, thus, to ensure that the system does not violate a given stability criterion (a set of states E). For this purpose, the algorithm in Section 5.7 can be adapted to the setting of this Chapter in a straightforward manner.

Chapter 7

The Situation Calculus/Golog framework

In this chapter we will apply most of the topics, concepts and techniques introduced in earlier chapters of this thesis to one formal framework: the *Situation Calculus* (SitCalc) and GOLOG.

7.1 Introduction

We believe that logic-based approaches to modeling systems provide many insights that can be used when applying other modeling techniques. This is especially true for the SitCalc/GOLOG framework which has been described as a *model-based programming language* in [McIlraith, 1999]. The basic idea behind model-based programming is that a programmer provides not only a control program in the language, but also a model of the system which is to be controlled. The interpretation of the programs can then exploit the model in various ways, such as for disambiguation of non-deterministic constructs in the language by predicting the outcome of the possibilities and choosing the "best"one. Clearly, model-based programming languages are interesting from an execution monitoring point of view, due to the explicit access to models and prediction mechanisms.

A reason for choosing the SitCalc/GOLOG framework is that it has a well-defined logical semantics, that is, the programming language GOLOG has a transition semantics that is described as a logical theory and the modeling language, SitCalc, is a logical language for reasoning about action and change. Thus, the analysis tasks can easily be viewed as logical reasoning tasks.

The first approach to execution monitoring in the SitCalc/GOLOG framework was presented in [De Giacomo *et al.*, 1998] (see Section 7.4 for a review). That work can be seen as a starting point for the work in this Chapter (Section 7.5) and in Appendix B, where we generalize some of their ideas and develop issues that they only sketched.

7.1.1 Overview

The purpose of this chapter is to present the SitCalc/GOLOG framework and analyze it from an execution monitoring perspective. We are interested in ensuring that we can find SitCalc implementations of the five constituting functions of execution monitoring (from section 2.3.2), that we can formulate the restrictions of ontological control in SitCalc (from Chapter 4), and that the notion "stability" (from Chapter 5) is applied to SitCalc/GOLOG.

In Section 7.2 we will present Reiter's version of SitCalc and in Section 7.3 the programming language GOLOG is presented. GOLOG uses a SitCalc theory as an explicit domain model.

Both ontological control and stability-based execution monitoring perform discrepancy detection on *states* of the system. In the Situation Calculus it is often convenient to reason with sequences of executed actions (*situations*) instead, similar to the Ramadge-Wonham theory of Discrete Event Systems (see for example [Kumar and Garg, 1995]). We address the problems of execution monitoring both with the situation view, in Section 7.5, and the state view and implement ontological control and stability-based execution monitoring in Sections 7.6, 7.7, and 7.8.

In Appendix A we develop a SitCalc formulation of "stability" (and sketch a formulation of "maintainability"). We also show how a stabilizing GOLOG program can be synthesized from an unstable SitCalc theory and prove the correctness of the synthesis.

As discussed before, SitCalc belongs to the very high end of the SCWA axis, which is problematic from our point of view. In Appendix B, we show that discrepancies in this framework causes inconsistencies (due to the strong SCWA) which makes it difficult to use for real applications. This problem is addressed in Appendix B where Pinto's SitCalc framework for concurrency and explicit situation preference relations is presented and extended to handle discrepancies.

7.2 Reiter's SitCalc

SitCalc [McCarthy and Hayes, 1969, Reiter, 1991, Levesque *et al.*, 1998] is arguably the most widespread logical formalism for reasoning about action and change today. The motivation behind the choice of using Reiter's SitCalc in this thesis over some competing formalism such as TAL [Doherty *et al.*, 1998], Action Languages [Gelfond and Lifschitz, 1998], and the Fluent calculus [Thielscher, 1998] is due to SitCalc's tight connection to GOLOG (neither of the competing formalisms has a programming language associated to it).

7.2.1 Language

The situation calculus (SitCalc) is a sorted second-order language with equality and uses at least four sorts: primitive actions, situations, fluents, and objects. A fluent is a term whose value might vary in different situations. We have a special predicate H, where $H(f(\vec{o}), s)$ denotes that the fluent $f(\vec{o})$ is true in situation s, for a vector, $\vec{o}$, of object constants. We sometimes write this as $f(\vec{o}, s)$ to denote $H(f(\vec{o}), s)$, and $\neg f(\vec{o}, s)$ to denote $\neg H(f(\vec{o}), s)$.

The set of all primitive actions for a particular theory will be denoted $\mathcal{A} = \{A_1, A_2, \ldots, A_n\}$, the set of fluents $\mathcal{F} = \{f_1, f_2, \ldots, f_m\}$, where every fluent has a particular arity, and the set of objects $\mathcal{O} = \{o_1, o_2, \ldots, o_k\}$. The set of background axioms of SitCalc is denoted Σ and we refer interested readers to the SItCalc reference document (by Levesque *et al.* [1998]) for further details about the logic. However, three of the foundational axioms are of crucial importance to the rest of this thesis. First, the second-order induction axiom:

$$\forall P.\, P(S_0) \wedge \forall a, s.\, (P(s) \wedge Poss(a, s) \supset P(do(a, s))) \supset \forall s.P(s).$$

We also require the definition of the reachability predicate $\sqsubset$, where, for two situations s, s', $s \sqsubset s'$ is intended to hold whenever the is a "legal" sequence of actions leading from s to s', i.e.

$$\begin{aligned}&\forall s.\, \neg(s \sqsubset S_0)\\ &\forall a, s, s'.\, s \sqsubset do(a, s') \equiv s \sqsubseteq s'.\end{aligned}$$

Here, $s \sqsubseteq s'$ is an abbreviation for $s \sqsubset s' \vee s = s'$.

For a particular axiomatization, the following axioms must be provided:

- **Action precondition axioms**, one for each primitive action $A(\vec{x})$, having the syntactic form

  $$Poss(A(\vec{x}), s) \equiv \Pi_A(\vec{x}, s),$$

  where $\Pi_A(\vec{x}, s)$ is a formula with free variables among $\vec{x}$, s, and whose only situation term is s. Intuitively, $Poss(A(\vec{x}), s)$ denotes the exact condition under which it is possible to execute the primitive action A.

- **Effect axioms**, two for each fluent F, having the syntactic form:

  $$\begin{aligned}&Poss(a, s) \wedge \gamma_F^+(a, s) \rightarrow H(F, do(a, s))\\ &Poss(a, s) \wedge \gamma_F^-(a, s) \rightarrow \neg H(F, do(a, s))\end{aligned}$$

  where $\gamma_F^*(a, s)$ (for $* \in \{+, -\}$) is of the form

  $$\bigvee \pi_F^*(a, s) \wedge a = A(\vec{x})$$

  with one disjunct for every action A.

To characterize the possible changes of all fluents the following *explanation closure assumption* [Schubert, 1990] is introduced.

Assumption 1 (Explanation Closure Assumption (ECA)) The formulas $\gamma_F^+(a,s)$ and $\gamma_F^-(a,s)$ characterize the positive and negative change, respectively, of the fluent F.□

By combining the ECA and the effect axioms we get the following set of axioms:

- **Successor state axioms**, one for each fluent F, having the syntactic form:

$$Poss(a,s) \supset (H(F(\vec{x}), do(a,s)) \equiv \\ \gamma_F^+(\vec{x},a,s) \vee (H(F(\vec{x}),s) \wedge \neg\gamma_F^-(\vec{x},a,s))),$$

  Intuitively, the successor state axioms for fluent F says that F is true after executing the action A in situation s iff it has changed to true (due to $\gamma_F^+(\vec{x},a,s)$) or it was true already in s and did not change to false (due to $\gamma_F^-(\vec{x},a,s)$).

- **Initial situation axioms** – what is true initially, before any actions have occurred. This is any finite set of sentences that mention only the situation term S_0, or that are situation independent.

Thus, an application axiomatization is a set $\Delta = \Sigma \cup ?_{AP} \cup ?_{SSA} \cup ?_{S_0}$, where Σ is the set of background axioms, $?_{AP}$ is a set of action precondition axioms, $?_{SSA}$ is a set of successor state axioms, and $?_{S_0}$ is a set of initial situation axioms and all situation independent axioms. We will write $a_0a_1\ldots a_l$ to denote the situation $do(a_l, do(\ldots do(a_1, do(a_0, S_0))\ldots))$.

One slight departure from Reiter's style of specification, is that we now partition the fluent sort in disjoint sub-sorts.

First, we assume a finite and fixed sort of *observable world fluents* $\mathcal{F}_w$, along with a set $F_{w1},\ldots,F_{wN}$ of constants of type $\mathcal{F}_w$; the value N is a fixed positive integer, which is defined by the application. We apply the unique name and domain closure assumption to world fluents where the constants $F_{w1},\ldots,F_{wN}$ are assumed to denote different individuals in $\mathcal{F}_w$. Furthermore, every element in $\mathcal{F}_w$ is denoted by some constant in $F_{w1},\ldots,F_{wN}$. Also, we have $\mathcal{F}_w \subseteq \mathcal{F}$. The world is fully observable when $\mathcal{F}_w = \mathcal{F}$. At any given situation, if we assume a completely known initial state, then the theory of action will predict the value of any fluent in the theory, including those in $\mathcal{F}_w$. The exact role that is played by these fluents will be clarified in Section 7.5.

Second, we introduce a set of status fluents $\mathcal{F}_s$. These fluents will be used in order to describe the execution status of high level processes that the agent might be performing. For instance, if the agent is delivering mail, then we could have the status fluent *deliveringMail* that would hold in those situations in which the agent is indeed delivering mail, and it would be false otherwise. As explained later, we assume a program structure in which these fluents have appropriate values.

Moreover, all fluents not in $\mathcal{F}_s$ nor in $\mathcal{F}_w$ are assumed to be regular domain fluents.

Similarly, we assume that we have a finite and fixed sort of *observable world actions* $\mathcal{A}_w$, along with a set $A_{w1}, \dots, A_{wM}$ of constants of type $\mathcal{A}_w$; the value M is a fixed positive integer, which is defined by the application. The constants $A_{w1}, \dots, A_{wM}$ are assumed to denote different individuals in $\mathcal{A}_w$. Furthermore, every element in $\mathcal{A}_w$ is denoted by some constant in $A_{w1}, \dots, A_{wM}$. Also, we have $\mathcal{A}_w \subseteq \mathcal{A}$. The set $\mathcal{A}_w$, corresponds to the exogenous actions of the domain (exogenous from the point of view of the agent).

We make use of the notion of *history*. A history is a sequence of actions $A_1, \dots, A_n$. If we assume theories of action without non-determinism and a complete specification of the initial situation, then a history uniquely identifies a situation. For convenience, we will use ; as a sequence operator. Thus, if h is the history $A_1, \dots, A_n$, then $h; A$ is the history $A_1, \dots, A_n, A$, Also, we say that h' is an expansion of h whenever h is a prefix of h'. Formally, h' is an expansion of h whenever $do(h, S_0) \sqsubset do(h', S_0)$.

We define an *observation in situation s*, O_s, to be the set of all *ground* observable world fluents that are true in situation s. Since there are no other function symbols besides *do*, and observations are always finite we can, for an observation O_s, write $\bigwedge O_s$ to denote the formula $(\bigwedge_{F \in O_s} H(F, s)) \wedge (\bigwedge_{\mathcal{F}_w \ni F \notin O_s} \neg H(F, s))$. By $\bigwedge O_{s/s'}$ we denote the observation obtained by replacing the situation argument, s. in $\bigwedge O_s$ by s'.

7.3 Golog

The original GOLOG language was first reported in [Levesque *et al.*, 1997]. One very important feature of the language is that its semantics was defined in terms of the *Situation Calculus*. The language has been extended in several ways: To incorporate explicit time [Reiter, 1998], interleaved accounts of concurrency [Reiter, 1998, De Giacomo *et al.*, 1997, de Giacomo *et al.*, 1999a, de Giacomo *et al.*, 1999b], true concurrency [Baier and Pinto, 1998], sensing [de Giacomo and Levesque, 1998], etc.

In this thesis, we will not propose a new variant of GOLOG. Rather, we will assume that the programs to be monitored have a specific structure, which we call *self recovery*. This structure is discussed in Section 7.5.4.

GOLOG is a high-level logic programming language used for agent-based programming. A detailed discussion of GOLOG can be found in [De Giacomo *et al.*, 1998], so we will only describe the constructs of the GOLOG language and the principles behind the interpretation process. For GOLOG programs δ, δ_1, δ_2, and ϕ a pseudo fluent[1], we have the following constructs:

[1]A pseudo fluent is a SitCalc fluent with all its situation arguments suppressed. $\phi[s]$ denotes pseudo fluent ϕ "instantiated" with situation argument s.

nil,	the empty program
a,	primitive action
$\phi?$,	test truth of condition ϕ
$(\delta_1;\delta_2)$,	sequence
$(\delta_1 \mid \delta_2)$,	nondeterministic choice between two actions
$\pi v.\delta$,	nondeterministic choice of argument to an action
δ^*,	nondeterministic iteration
proc $P(\vec{v})\,\alpha$ **endProc**	procedure with formal parameters $\vec{v}$ and body α.

For convenience we introduce conditionals and while-loops as:

$$\textbf{if } \phi \textbf{ then } \delta_1 \textbf{ else } \delta_2 \textbf{ endIf} \stackrel{\text{def}}{=} (\phi?;\delta_1)|(\neg\phi?;\delta_2)$$
$$\textbf{while } \phi \textbf{ do } \delta \textbf{ endWhile} \stackrel{\text{def}}{=} ((\phi?;\delta)^*;\neg\phi?)$$

The semantics of GOLOG is a transition semantics [De Giacomo *et al.*, 1997] based on the two predicates $Trans$ and $Final$. Here, we will only define the cases of those predicates that are necessary for our presentation. The formal definitions of $Trans$ and $Final$ can be found in [De Giacomo *et al.*, 1997] or [De Giacomo *et al.*, 1998].

$Trans(\delta, s, \delta', s')$ holds if a program δ is executed in situation s then the remainder of this program will be δ' which is up for execution in situation s'. The resulting situation s' will be different from s only if the next part in δ to be executed is a primitive action. Formally,

1. The empty program:

$$Trans(nil, s, \delta', s') \equiv False.$$

2. Primitive actions:

$$Trans(a, s, \delta', s') \equiv Poss(a, s) \wedge \delta' = nil \wedge s' = do(a, s).$$

3. Test actions:[2]

$$Trans(\phi?, s, \delta', s') \equiv \phi[s] \wedge \delta' = nil \wedge s' = s.$$

4. Sequence:

$$\begin{aligned} &Trans(\delta_1;\delta_2, s, \delta', s') \equiv \\ &\quad \exists\gamma.\, Trans(\delta_1, s, \gamma, s') \;\wedge\; \delta' = \gamma;\delta_2 \;\vee \\ &\quad Final(\delta_1, s) \;\wedge\; Trans(\delta_2, s, \delta', s'). \end{aligned}$$

[2]The formula ϕ is assumed to not contain any situation arguments. By writing $\psi[s]$ we restore all situation arguments in the formula.

5. Nondeterministic choice:

$$Trans(\delta_1|\delta_2, s, \delta', s') \equiv Trans(\delta_1, s, \delta', s') \vee Trans(\delta_2, s, \delta', s').$$

6. Iteration:

$$Trans(\delta^*, s, \delta', s') \equiv \exists\gamma. Trans(\delta, s, \gamma, s') \wedge \delta' = \gamma; \delta^*.$$

$Final(\delta, s)$ holds if the execution of δ can be considered complete in situation s. Since GOLOG contains nondeterministic constructs, it is possible for both $Final(\delta, s)$ and $Trans(\delta, s, \delta', s)$ to hold for some programs in some situations. Formally,

1. Empty program: $Final(nil, s) \equiv True.$
2. Primitive action: $Final(a, s) \equiv False.$
3. Test action: $Final(\phi?, s) \equiv False.$
4. Sequence: $Finals(\delta_1; \delta_2, s) \equiv Final(\delta_1, s) \wedge Final(\delta_2, s).$
5. Nondeterministic choice: $Final(\delta_1|\delta_2, s) \equiv Final(\delta_1, s) \vee Final(\delta_2, s).$
6. Iteration: $Final(\delta^*, s) \equiv True.$

The predicate $TransCl(\delta, s, \delta', s')$ denotes the reflexive transitive closure of the $Trans$ predicate. Formally,

$$\begin{aligned}
&TransCl(\delta, s, \delta', s') \equiv \\
&\quad \forall T.(T(\delta, s, \delta, s) \wedge \\
&\qquad (Trans(\delta, s, \delta'', s'') \wedge T(\delta'', s'', \delta', s') \supset T(\delta, s, \delta', s'))) \supset \\
&\quad T(\delta, s, \delta', s')
\end{aligned}$$

The GOLOG transition semantics allows us to define a *step* function, which, given a program and a history, returns an action and a *continuation*. If δ is a program whose first action is A, then a continuation is what remains of δ after A. Thus, given a history h, and a continuation (a program) δ, *step*(h, δ) is a function that returns a tuple $\langle A, \delta' \rangle$, where A is a primitive action (primitive in the sense of the underlying theory of action) and δ' is a continuation, if A is $-$, then the action is failure. *step*'s implementation is the GOLOG interpreter. For a particular realization, consult [Reiter, 1999]. The *step* function uses the history h in order to determine the situation in which the next action is going to be executed. This is necessary, for instance, in order to evaluate conditional execution of actions.

7.4 The De Giacomo *et al.* approach to execution monitoring of Golog

In the original version, GOLOG was basically a planner where a user had the possibility of representing more of the domain structure than just the planning operators and the initial and goal states. The output of an interpretation of a GOLOG program was a sequence of primitive actions. In the efforts to use GOLOG for high-level control of robots, a reactive version was developed, in [De Giacomo *et al.*, 1998], where the interpreter generates one primitive action at a time, and processes "sensor readings" in between. In that paper the first attempt to add execution monitoring to the framework was presented.

Execution Monitoring is handled by the predicate $Monitor(\delta, s, \delta', s')$, where the current continuation is δ and the current situation is s. *Monitor* checks for the occurrence of an exogenous program in situation s, determines the situation s' reached by this program, and if the monitored program δ terminates off-line, the monitor returns δ, else it invokes a recovery mechanism that determines a new program δ'. The suggested recovery mechanism tries to find a sequence of primitive actions p that would move the system into a situation where the original program δ can be successfully executed, i.e. it generates a program $p;\delta$ if such a program exists.

In the current implementation of execution monitoring (EM) for GOLOG, the EM senses occurrences of "exogenous" actions. In fact the actions themselves are not exogenous, it is the *occurrences* that are, i.e. the EM senses the occurrence of a GOLOG program which is not the program GOLOG is currently executing. As argued in [De Giacomo *et al.*, 1998], this "seems dubious in practice", since it is more likely that an EM would detect *effects* of actions, rather than the occurrences. In the paper, the authors claim to reconcile this point by proposing new, fictitious actions, one for each fluent, whose effects are to alter their corresponding fluents' truth value. The authors only sketch how such a reconciliation could be made, and in Appendix B of this thesis we give a formal account of this idea, and we will show how this can be utilized as a mechanism for discrepancy detection. However, even with such fictitious actions it is not clear how measurements from sensors should be handled within their approach. In Section 7.5 we propose a more general execution monitoring framework where the interaction between the controller and the environment is handled.

7.5 General (Situation-based) execution monitoring

We want to provide a general characterization of what an execution monitor is, assuming that the execution monitor is monitoring the *situation histories* rather than the trajectory of *states* (which has been considered previously in this thesis). An execution monitor is a program interpreter that acts with a model of the world (environment), this is a logical theory of action Σ; a program δ, whose execution is monitored; a world or environment $\mathcal{W}$ (which is modeled by the theory of action), and a monitoring boolean condition γ. In the rest of this section, we describe these components. We assume all along that there is one *agent* that executes δ under the control of the execution monitor. Also, this agent has Σ and γ at its disposal.

7.5.1 The world model

Assume that Σ is a logical theory of action. It is a model of the world inhabited by the agent, possibly along with other agents. This is a *Situation Calculus* theory of action. The theory describes the world and the changes that are affected after actions are performed.

The SitCalc theories considered here are an extension of the theories in the style proposed by Reiter (see Section 7.2). The world model is a theory of action written as the union of several sets of axioms: action precondition axioms, successor state axioms, exogenous action specifications, unique names axioms, etc.

7.5.2 The world

The world $\mathcal{W}$ is the actual environment. In our model of an execution monitor, we take the world as being a *history indexed* observation structure. What this means is that given a history from some initial point, the structure tells us what is observed in the world.

Our intention is to model the interaction between the agent and the world in a manner similar to what Sandewall calls *ego-world* interaction in [Sandewall, 1995]. Thus, given a history, the world tells us what is *observed* (fluents and exogenous actions).

We assume that we have a set $\mathcal{F}_\mathcal{O}$ of faithful fluent observations[3]. Furthermore, we assume that we have exactly N fluent observations (the value of $N$is the same value mentioned in the previous subsection). Analogously, we have a set $\mathcal{A}_\mathcal{O}$ of action observations (exogenous actions). For simplicity, we assume that at any point only a single exogenous action can occur, and that there are exactly M possible exogenous actions.

We make the simplifying assumption that time is discrete. Furthermore, we assume that every situation has a unique time (as in, for instance, [Pinto, 1994]). However, we will not concern ourselves with explicit time. Instead, we will take time to be represented by histories. Thus, if h and h' are histories and h is a prefix of h', then the time of h is less than the time of h'.

The world is modeled as a tuple $\langle \omega_\mathcal{F}, \omega_\mathcal{A} \rangle$. The first element, $\omega_\mathcal{F}$, is a mapping from histories and actions in $\mathcal{A}$ to a *state*, i.e. a mapping of observations (from $\mathcal{F}_\mathcal{O}$) to $\{\top, -\}$ (true, false). The second element, $\omega_\mathcal{A}$, is a mapping from histories and actions in $\mathcal{A}$ to $\mathcal{A}_\mathcal{O} \cup \langle nil \rangle$, where *nil* is interpreted as no action.

The sets $\mathcal{F}_\mathcal{O}$ and $\mathcal{A}_\mathcal{O}$ are in a one to one correspondence with the elements of $\mathcal{F}_w$ and $\mathcal{A}_w$ (more precisely, with the names for the elements in each set). Furthermore,

[3]By faithful, we mean that what is observed is true; in other words, the sensor readings are always accurate.

we assume that we have a meta-level function `map`, that given an element $f_{\mathcal{O}} \in \mathcal{F}_{\mathcal{O}}$, $\texttt{map}(\texttt{f}_{\mathcal{O}})$ yields the constant that denotes the fluent in $\mathcal{F}_w$ corresponding to $\texttt{f}_{\mathcal{O}}$. Analogously, if $\texttt{a}_{\mathcal{O}} \in \mathcal{A}_{\mathcal{O}}$, $\texttt{map}(\texttt{a}_{\mathcal{O}})$ yields the constant that denotes the action in $\mathcal{A}_w$ corresponding to $\texttt{a}_{\mathcal{O}}$ (we are overloading the function `map`).

7.5.3 The execution monitor

The execution monitor can be seen as a computational process. As such, at any point in time one can view the execution monitor's state, which, given a world, will determine a transition to the next state. The initial state of the monitor is characterized by a tuple $\langle \Sigma, \delta_0, \mathcal{W}, \mathcal{F}_{\mathcal{O}}, \mathcal{A}_{\mathcal{O}}, \texttt{map}\rangle$, where we assume that Σ is a *Situation Calculus* theory of action that incorporates a complete specification of the initial state. δ_0 is the program to be monitored, and the rest is as before. At any point during the monitoring process, the state of the execution monitor can be characterized with:

1. h: The current history h. It corresponds to a sequence of actions (agent actions or exogenous actions), which, given an initial situation, lets us keep track of the current situation. Initially, the history is empty.

2. δ: The current continuation; i.e., what is left of the program to be executed (after successive applications of the step function). Initially, δ is the same as δ_0.

In order to decide what to do, the execution monitor uses:

1. Σ, the theory of action. Given the history, it tells the monitor what to expect.

2. A prescription on what to do next. This is the first action A prescribed by the program δ (obtained with the *step* function). If the program cannot be continued, then we assume that A is the action $-$ (which we interpret as an *abort* action). A program might not be able to continue for several reasons: it may have been completely executed, or it may have encountered a situation that does not satisfy the preconditions to carry on with the program.

3. Situation assessment. Given a situation characterized by the history h, the action A, the world view $\langle \omega_{\mathcal{F}}(h, A), \omega_{\mathcal{A}}(h, A)\rangle$, we obtain the new history h', which is an expansion of h. In the trivial case, in which everything develops according to expectations (no exogenous actions and fluent values as predicted), we'll have $h' = h; A$. Thus, h' will be the history h expanded with action A. In other cases, h' would be an expansion consistent with the exogenous actions that might have occurred and the fluents that were observed (i.e., $\langle \omega_{\mathcal{F}}(h, A), \omega_{\mathcal{A}}(h, A)\rangle$).

4. Monitoring formula: This is a logical formula whose truth value (assigned by Σ) would tell the monitor what to do. The formula will check discrepancies between what is expected (i.e., what is true after $h; A$) and what actually

happened (i.e., what is true after h'). We'll assume that the monitor will entertain two options: Stop the execution of the program and restart it. Do nothing, allowing the program to continue. The first condition is characterized by a formula $\gamma_1(h, h')$, the second condition is the default condition.

The monitoring specified in the last bullet might appear unsophisticated. Indeed, one could conceive a much more intrusive monitor. For instance, a monitor might decide to abort execution of the program on its own accord, or restart the program at some arbitrary entry point (if such were identifiable by some means). What we propose is that the programmer should write programs that would specify how to proceed from some arbitrary state obtained after interruption. For example, if our agent is delivering coffee and it drops the coffee on the floor, then the program may be restarted and the resulting situation – dirty floor and no coffee – is evaluated as a starting situation for the program (not to confuse with S_0. An actual program architecture that does this is presented below.

Given the above information structure, available to the monitor, an algorithm that implements an execution monitor is described below:

- Let $h \leftarrow$ empty. Initialize the current history as empty.
- Let $\langle A, \delta \rangle \leftarrow step(h, \delta_0)$.
- Do forever:

```
1. If $A = -$, abort.
2. Execute $A$, obtain $\langle \omega_{\mathcal{F}}(h, A), \omega_{\mathcal{A}}(h, A) \rangle$.
3. $h' \leftarrow assess(h, A, \langle \omega_{\mathcal{F}}(h, A), \omega_{\mathcal{A}}(h, A) \rangle)$.
4. If $h' = h; A$ then $\langle A, \delta \rangle \leftarrow step(h', \delta)$.
5.       else $\langle A, \delta \rangle \leftarrow recover(h, h', \delta_0, \delta)$.
6. $h \leftarrow h'$.
```

Figure 7.1: A possible monitor

Situation assessment

In the Algorithm of figure 7.1, there is a reference to the function *assess*, whose role we now explain. The function *assess* takes a history h, an action A, and an observation tuple, and returns a history h' which is an expansion of h.

The execution monitor is in a situation that results from executing history h in the initial situation. The observation tuple tells us how the world reacts. This latter information comes in the form of a tuple. The first element of the tuple is a truth assignment to all the fluent constants in $\mathcal{F}_{\mathcal{O}}$, and the second is either *nil* or an *exogenous action*[4].

Given this information, the function assesses the current situation, i.e., it determines a *true* history h', that is consistent with the action A and the observation tuple. In the most trivial case, $\omega_{\mathcal{A}}(h, A)$ would be *nil*, and the observation tuple would be exactly as predicted by Σ; that is, for every constant $\mathtt{F} \in \mathcal{F}_{\mathcal{O}}$:

$$\Sigma \models H(\mathtt{map}(\mathtt{F}), do(h; A, S_0))$$

if and only if $\omega_{\mathcal{F}}(h, A)(\mathtt{F}) = \top$. Under those circumstances $h' = h; A$.

Another simple case is obtained if $\omega_{\mathcal{A}}(h, A)$ is not $-$, and the observation tuple is such that for every constant $\mathtt{F} \in \mathcal{F}_{\mathcal{O}}$

$$\Sigma \models H(\mathtt{map}(\mathtt{F}), do(h; A; \omega_{\mathcal{A}}(h, A), S_0))$$

if and only if $\omega_{\mathcal{F}}(h, A)(\mathtt{F}) = \top$. Under those circumstances $h' = h; A; \omega_{\mathcal{A}}(h, A)$.

Any other situation corresponds to a *surprise*, in the sense that the available information (i.e., current history h and exogenous action $\omega_{\mathcal{A}}(h, A)$) is not enough to explain the fluent observations. In this case, there are several ways in which the assessment can be handled, and the two alternatives we consider are:

Alt. 1 Perform a diagnostic procedure (e.g. in the sense of McIlraith [1998]). This may lead to deriving a hypothesis h', such that for every constant $\mathtt{F} \in \mathcal{F}_{\mathcal{O}}$

$$\Sigma \models H(\mathtt{map}(\mathtt{F}), do(h', S_0))$$

if and only if $\omega_{\mathcal{F}}(h, A)(\mathtt{F}) = \top$. Recall that $\mathtt{map}(\mathtt{F})$ is not a term in the language of the situation calculus; rather, it is a term outside the logical language that has to be replaced by the fluent to which $\mathtt{F}$ is mapped.

We require h' to be an expansion of h. Thus, we need the diagnosis to be categorical (i.e., no disjunctive hypotheses). For this alternative to be viable, we need Σ to be rich enough to incorporate exogenous actions that might or might not be observable, and that would explain the *surprises.*

Alt. 2 Simply accept the new state and let h' be $h; A; \omega_{\mathcal{A}}(h, A); A_{\mathcal{F}_{\mathcal{O}}}$ (assuming that $\omega_{\mathcal{A}}(h, A)$ is not equal to $-$). Here, we assume that Σ contains one action $A_{\mathcal{F}_{\mathcal{O}}}$ for every possible assignment of truth values to elements of $\mathcal{F}_{\mathcal{O}}$. These actions (fictitious action) will simply set the observational fluents to the corresponding values. We'll call these actions *fluent setting fictitious actions.* Note that these fictitious actions carry no explanatory power, and simply help make the theory consistent. As long as we are dealing with finite object domains, this alternative can, equivalently, be phrased in a state-based setting. We explore this in Appendix B.

[4]We require a single exogenous action to avoid dealing with true concurrency. This is not a problem (see [Baier and Pinto, 1998]), however introducing true concurrency would simply distract us from our objective.

Recovery

Whenever $h; A \neq h'$ a recovery ensues. Recovery is necessary whenever there is a discrepancy between expectations (i.e., the state resulting from history $h; A$) and the actual situation (resulting from history h').

If the discrepancy between the expected situation and the actual situation is deemed to be irrelevant, then it is ignored. Otherwise, recovery will ultimately depend upon domain specific information that is made available to the system (either the actual monitor, or the monitored program). We will take the point of view that the program contains its own recovery mechanisms, as will be explained in the following section.

7.5.4 Self-recovery program structure

We consider a program structure similar to that of a production system, that is:

$$\varphi_0 \rightarrow \beta_0; \ldots ; \varphi_n \rightarrow \beta_n$$

We assume that this program, when invoked, treats the set of production rules in a fashion similar to that of a Lisp *cond* statement. That is, the conditions are evaluated in order, and the smallest i such that φ_i holds leads to the execution of β_1. We assume that φ_n is a catch all identically true condition, and that β_n is an abort statement or a *planning from scratch* procedure.

Notice that the program structure that is suggested is directly allowed by the Golog language (in all its incarnations), since it is simply a set of nested *if–then–else* statements. Some important points to keep in mind. First the φ_i conditions are all evaluated at the situation in which the program is restarted. This situation is identified by the current history, which is kept by the monitor.

In [Nilsson, 1994a], Nilsson proposed the notion of *Teleo-Reactive* (*TR*) programs. A teleo–reactive program is also similar to a set of production rules. There are several aspects of *TR*–programs that on the surface appear to make these programs more suitable for agent control. For instance, *TR* actions can be *durative*[5] rather than discrete. Also, the conditions have to be continuously evaluated, and the corresponding action is executed as long as the condition is the first one to hold (in the order defined by the production rules). This behavior can be easily simulated by a self–recovery program by constructing a suitable γ_1 condition.

Example

In this example, we consider a robot that inhabits an office environment. The robot can walk through corridors to get from office to office, it can pick up and deliver mail, it can obtain coffee and deliver coffee, and it can also clean areas it finds dirty. A similar scenario was modeled in detail in [Reiter, 1998]. Thus, we exclude cluttering details here and refer to that paper. The theory of action Σ will include in its vocabulary the following fluents:

[5] I.e., they can continue indefinitely; e.g. moving forward.

1. *carryingCoffee* holds if the robot is carrying coffee. Formally,

$$\begin{aligned} &Poss(a,s) \supset [holds(carryingCoffee, do(a,s)) \equiv \\ &\quad a = pickupCoffee \vee \\ &\quad holds(carryingCoffee, s) \wedge \\ &\qquad \neg(a = putdownCoffee \vee a = droppedCoffee)] \end{aligned}$$

   The action *droppedCoffee* is assumed to be exogenous.

2. *carryingMail*(x) holds if the robot is carrying mail destined for office x.

3. *deliveringCoffee*(x) holds if the robot is delivering coffee to office x. Formally,

$$\begin{aligned} &Poss(a,s) \supset [holds(deliveringCoffee(x), do(a,s)) \equiv \\ &\quad a = goingTo(x) \wedge holds(carryingCoffee, s) \wedge \\ &\qquad holds(wantsCoffee(x), s) \vee \\ &\quad holds(deliveringCoffee(x), s) \wedge \\ &\qquad \neg(holds(At(x), s) \wedge a = giveCoffee(x)] \end{aligned}$$

4. *deliveringMail*(x) holds if the robot is delivering mail to office x.

5. *batteryLow* holds if the batteries are low. *recharging* if the robot is recharging its batteries. in this example we assume that *batteriesTurnedLow* is an exogenous action.

Aside from these fluents, the robot can execute a number of complex actions (GOLOG procedures), which are built on top of some more basic primitive actions. Informally, these procedures are: *deliverCoffee*(x), to deliver coffee to office x, and *deliverMail* to deliver mail to office x; *recharge*, which recharges the robot's batteries; *clean*, which cleans a mess caused by accidentally dropping the coffee; *Interrupt*, which interrupts the execution of whatever it is that the robot is doing.

The basic top-loop GOLOG program can have the following structure (again excluding a number of details):

```
proc TopLoop
    while $\top$ do
    $(\pi\, x)\{$
        if $hasMail(x)$ then $deliverMail(x)$
        else if $wantsCoffee(x)$ then $deliverCoffee(x)$
    $\}$
```

Note that we in this program only specify the main tasks of the robot, and leave abnormal contingencies to the monitor.

A self recovery program could be:

$$\begin{aligned} \mathit{batteryLow} &\rightarrow \mathit{recharge};\\ \mathit{messy} &\rightarrow \mathit{clean};\\ \mathit{carryingMail}(x) &\rightarrow \mathit{deliverMail}(x);\\ \mathit{carryingCoffee} \wedge \mathit{wantsCoffee}(x) &\rightarrow \mathit{deliverCoffee}(x);\\ \top &\rightarrow \mathit{recharge}; \end{aligned}$$

Now, we need to specify the conditions under which the execution monitor has to intervene. Thus, we need to specify the γ_1 condition which tells the execution monitor to restart the program. In this case, this condition might be:[6]

$$\mathit{batteryLow}(s') \vee \neg \mathit{carryingCoffee}(s') \wedge \mathit{deliveringCoffee}(s')$$

We assume that the procedure TopLoop is executing and that the current history is h. Someone, say $\mathbf{r}$, wants coffee and the robot invokes the procedure $\mathit{deliverCoffee}(\mathbf{r})$. If the coffee suddenly slips from the robot gripper before the robot finishes its task, the situation assessment function will return $h' = h; \mathit{droppedCoffee}$ which differs from the expected $h; \mathit{deliverCoffee}(\mathbf{r})$ (assuming that there is a defined action $\mathit{deliverCoffee}(x)$ in Σ). The assessment will also report that the formula $\neg \mathit{carryingCoffee}(h', S_0) \wedge \mathit{deliveringCoffee}(h', S_0)$ holds which will trigger the monitor to invoke the self-recovery program. Unless the batteries are low in this situation, the robot will note the mess it created and start to clean up. When the cleaning action has finished the monitor will give control back to the continuation of the current GOLOG program.

7.6 State-based execution monitoring, BM classification and MT recovery

Apart from the previous few sections, we have considered *state-based* execution monitoring in this thesis. This means that detection and classification of discrepancies in ontological control and stability-based execution monitoring is based on the trajectory of states, rather than on the sequence of executed actions. In this section and the following, we will examine how ontological control and stability-based execution monitoring can be handled within the SitCalc/GOLOG framework.

In this section we will describe the recovery technique employed in the two subsequent sections. In certain situations it is a feasible idea to change the underlying SitCalc model for a GOLOG program in order to, e.g., make it reflect the actual system more adequately (as in ontological control), or to ensure that certain unwanted control situations do not occur again. For GOLOG we can note that any discrepancy makes the theory inconsistent. This is somewhat unintuitive, since the real world certainly does not disappear whenever it does not behave as we expect it to.

[6]here s' is $do(h', S_0)$

Type	O_{prev}	O_{exp}	O_{mat}	Revision
A	F	F	$\neg F$	Add more information to γ_F^-
B	$\neg F$	$\neg F$	F	Add more information to γ_F^+
C	F	$\neg F$	F	Remove information from γ_F^-
D	$\neg F$	F	$\neg F$	Remove information from γ_F^+

Table 7.1: The four kinds of discrepancies.

Instead of sensing fluent values, as in de Giacomo *et al.*'s work (Section 7.4) suggest that the execution monitor should sense fluent values instead of action occurrences. The $Trans$ function makes it possible to compute the expected result of a primitive action, so it is easy to determine whether there is a discrepancy between the sensed values and the expected ones. Next, if a discrepancy is detected we know that the observation is not a consequence of the axiomatization, since the initial state is completely specified and the actions are deterministic. To proceed, we suggest that the *model* (the underlying SitCalc axiomatization) should be changed in case of a discrepancy. We can classify the possible causes of the discrepancy as either being malignant, which implies that the successor state axioms do not reflect the environment adequately, or benign, which implies that the successor state axioms are corroborated, but that the system is in a different state than what was expected.

The changes of the model that we propose are:

Benign discrepancy: Replace the initial situation axioms with the new and unexpected observation, where the situation argument of the fluents are relativized to situation S_0, and restart the execution. Invoke the recovery mechanism if necessary.

Malignant discrepancy: Let O_{prev} be the state of the previous situation, A the primitive action invoked in the previous situation, O_{exp} the expected state after invoking A in a situation with state O_{prev}, and O_{mat} the observed current state. For a particular fluent F with successor state axiom

$$\begin{aligned}&Poss(a,s) \supset (H(F, do(a,s)) \equiv \\ &\quad \gamma_F^+(a,s) \vee (H(F,s) \wedge \neg\gamma_F^-(a,s))),\end{aligned}$$

we have four types of discrepancies described in Table 7.1. Types A and B describe the cases where the fluent F is modeled to be inert (i.e. to not change its value) by the action A, but that the value is sensed to have changed. This means that, in the case of a VOA, we need to add the particular case to γ_F^+ or γ_F^-, to make sure that the observed change is considered by the model. For cases C and D, changes are modeled but do not occur. This means that the changes assumed by the model should be removed. When the changes have been performed, we replace all initial situation axioms with the materialized observation and continue (possibly with recovery).

7.6.1 Formalization of the idea

Assume that for an application axiomatization $\Delta = \Sigma \cup ?_{AP} \cup ?_{SSA} \cup ?_{S_0}$ we have an observation $O_{\mathbf{s}}$ and that we perform the action A in $\mathbf{s}$ to take us to situation $do(A, \mathbf{s})$, and that we expect to sense the observation $O_{do(A,\mathbf{s})}$, but that we instead sense an observation $O'_{do(A,\mathbf{s})}$, where $O_{do(A,\mathbf{s})} \neq O'_{do(A,\mathbf{s})}$ (i.e. we have detected a discrepancy).

Next, we need to discriminate between the four types of discrepancies described in Table 7.1. We begin by constructing the sets $S_{pos} = O_{do(A,\mathbf{s})} - O'_{do(A,\mathbf{s})}$ and $S_{neg} = O'_{do(A,\mathbf{s})} - O_{do(A,\mathbf{s})}$. S_{pos} contains all fluents that where expected to be true but where observed to be false, thus being type A or D discrepancies, and S_{neg} contains all fluents expected to be false but that where observed to be true, thus being type B or C discrepancies. Now, we check if the members of S_{pos} and S_{neg} belong to $O_{\mathbf{s}}$:

- If $F(\vec{o}) \in S_{pos}$ and $F(\vec{o}) \in O_{\mathbf{s}}$, $F(\vec{o})$ is of type A .
- If $F(\vec{o}) \in S_{pos}$ and $F(\vec{o}) \notin O_{\mathbf{s}}$, $F(\vec{o})$ is of type D.
- If $F(\vec{o}) \in S_{neg}$ and $F(\vec{o}) \in O_{\mathbf{s}}$, $F(\vec{o})$ is of type B.
- If $F(\vec{o}) \in S_{neg}$ and $F(\vec{o}) \notin O_{\mathbf{s}}$, $F(\vec{o})$ is of type C.

Given the three observations $O_{\mathbf{s}}$, $O_{do(A,\mathbf{s})}$, and $O'_{do(A,\mathbf{s})}$, we can easily classify every discrepancy as being of type A, B, C, or D.

Let $F(\vec{o})$ be a discrepancy in situation $do(A, s)$, and $H(F(\vec{x}), do(a, s)) \equiv \ldots$ the corresponding successor state axiom, where the length of $\vec{x}$ is n. To construct the formula needed for model repair, as in the fifth column in table 7.1, we need the formula describing the sensed observation in situation $\mathbf{s}$ (the *precondition* of the detected VOA), i.e. $\bigwedge O_{\mathbf{s}}$, and the *instantiation formula*, $inst^{\vec{o}}_{\vec{x}}$ of the variables in $\vec{x}$, which is $inst^{\vec{o}}_{\vec{x}} \equiv x_1 = o_1 \wedge \ldots \wedge x_n = o_n$. The instantiation formula describes the variable bindings materialized in situation $do(A, s)$. For each discrepancy $F(\vec{o})$ in situation $\mathbf{s}$ we construct the formula $\beta^{\mathbf{s}}_{F(\vec{o})} \equiv \bigwedge O_{\mathbf{s}/s} \wedge inst^{\vec{o}}_{\vec{x}}$.

We assume that the successor state axiom for F is on the form

$$\begin{aligned} & H(F(\vec{x}), do(a, s)) \equiv \\ & \quad \bigvee_i \pi_i^+(\vec{x}, s) \wedge a = A_i \qquad (7.1) \\ & \quad \vee (H(F(\vec{x}), s) \wedge \\ & \quad \neg \bigvee_i \pi_i^-(\vec{x}, s) \wedge a = A_i). \qquad (7.2) \end{aligned}$$

We formalize the repair actions mentioned in the fifth column in table 7.1 as follows:

Type A: Replace the disjunct $\pi^-(\vec{x}, s) \wedge a = A$ in (7.2) by

$$(\pi^-(\vec{x}, s) \vee \beta^s_{F(\vec{o})}) \wedge a = A.$$

Type B: Replace the disjunct $\pi^+(\vec{x}, s) \wedge a = A$ in (7.1) by

$$(\pi^+(\vec{x}, s) \vee \beta^s_{F(\vec{o})}) \wedge a = A.$$

Type C: Replace the disjunct $\pi^-(\vec{x}, s) \wedge a = A$ in (7.2) by

$$\pi^-(\vec{x}, s) \wedge \neg\beta^s_{F(\vec{o})} \wedge a = A.$$

Type D: Replace the disjunct $\pi^+(\vec{x}, s) \wedge a = A$ in (7.1) by

$$\pi^+(\vec{x}, s) \wedge \neg\beta^s_{F(\vec{o})} \wedge a = A.$$

Observe that the structure of (7.2) and (7.1) remains unchanged after the the repair actions. To make sure that the application order of the repair actions does not matter, we need the following lemma.

Lemma 7.6.1 For two discrepancies, $F(\vec{o})$ of type A and $F(\vec{o'})$ of type C, the order in which we apply the revision actions is irrelevant.
Proof: We can see that the repair actions give syntactically different resulting formulae depending on the order of the repairs, i.e. if we use the type A repair before type C, the result will be

$$(\pi^-(\vec{x}, s) \vee \beta^s_{F(\vec{o})}) \wedge \neg\beta^s_{F(\vec{o'})} \wedge a = A,$$

and if we start with type C repair we get

$$((\pi^-(\vec{x}, s) \wedge \neg\beta_{F(\vec{o'})}) \vee \beta^s_{F(\vec{o})}) \wedge a = A.$$

The only case when their respective truth values differ is when $\beta_{F(\vec{b})}$ and $\beta_{F(\vec{c})}$ are both true, which would imply that $inst^{\vec{o}}_{\vec{x}} \wedge inst^{\vec{o'}}_{\vec{x}}$ is satisfiable. This can only occur if $\vec{o} = \vec{o'}$, which means that both $F(\vec{o})$ and $\neg F(\vec{o})$ have materialized (since we had one type A and one type C discrepancy). This is obviously false, so the two formulas are equivalent.□

For discrepancies of type B and D, the proof is analogous.

When a discrepancy is detected, and it is classified as being malignant, the discrepancy type is determined. This is done for all discrepancies. Based on the type information we perform revision on the successor state axioms of the discrepancies. When all discrepancies have been dealt with, we have a new set of successor state axioms $?'_{SSA}$. We replace the previous set $?_{SSA}$ in Δ with $?'_{SSA}$ and replace the initial situation axioms in Δ with the materialized observation, i.e. the new application axiomatization is defined as

$$\Delta' = ((\Delta - ?_{S_0}) \cup O'^{S_0}_{do(A,\mathbf{s})}) \cup ((\Delta - ?_{SSA}) \cup ?'_{SSA}).$$

7.7 Stability-based execution monitoring

The first instance of BM classification for GOLOG is what we call *stability-based* execution monitoring. The basic idea is that there is a set of states, described by a property γ, that represents the desired behavior of the system. If a SitCalc theory, Δ is stable w.r.t γ, or that there exists a GOLOG program that stabilizes Δ w.r.t. γ, then we know that if no discrepancies occur the system will always return to the desired set of states in finite time. Stability-based execution monitoring means that we want to maintain stability even in the presence of discrepancies. We can do this by distinguishing between two types of discrepancies: Discrepancies that moves the system into a state described by γ and discrepancies that move the system outside of γ.

7.7.1 Discrepancy detection and classification

The general idea behind stability-based execution monitoring was introduced in section 5.7. In that framework, discrepancy detection is straightforward; if the current predicted state is different from the current actual state, then there is a discrepancy. In SitCalc the detection consists of computing whether

$$\Delta \models O_{\mathbf{s}}$$

holds or not, where Δ is the application axiomatization, $\mathbf{s}$ is the current situation, and $O_{\mathbf{s}}$ is the current actual observation.

Discrepancy classification requires somewhat more sophistication, where it is necessary to find out whether a given discrepancy belongs to the closure of the system or not. In SitCalc the "closure" notion translates to the set of *states* that can be reached by control actions from the initial situation, and this can be formalized as follows:

$$\begin{aligned} &InClosure(s) \equiv \\ &\quad s = S_0 \vee \\ &\quad (do(a, s') \sqsubseteq s \supset Poss(a, s') \wedge InClosure(s')). \end{aligned}$$

The classification process is then to compute

$$\Delta \models \exists s'.\, O_{\mathbf{s}/s'} \wedge InClosure(s'), \tag{7.3}$$

which intuitively means that the current actual observation holds in some situation reachable from the initial situation by actions. If relation 7.3 holds, we have a benign discrepancy, otherwise it is malignant.

7.8 Ontological control

From chapter 4 we recall the three restrictions sufficient for ontological control: the existence of a perfect sub-model, non-logging and energized actions. In this section we will suggest a way to encode this in a SitCalc theory. In the next Section we will then show how ontological control can be performed in the SitCalc/GOLOG framework.

Perfect sub-model
Similar to the theory in chapter 4 we partition the set of fluents, $\mathcal{F}$ into *configuration fluents* and *plant fluents.* We introduce a new predicate symbol, $Conf$, for this, so that $Conf(F)$ holds iff F is a configuration fluent. Since observations are assumed to be conjunctions of ground fluents we can always partition an observation into a configuration and a plant formula part.

Non-logging
We defined a system to be non-logging iff the configurations of consecutive states are mutually exclusive. In SitCalc we will translate this to the following

$$\begin{aligned}&\Delta \models \\ &\quad \exists f.\, Conf(f) \wedge (H(f,s) \not\equiv H(f, do(a,s))).\end{aligned}$$

Intuitively this means that some configuration fluent changes value between two consecutive situations.

Energized Actions
Actions in a SitCalc theory are not necessarily energized, but the common assumption that formulas in a SitCalc theory are "simple", that is, that they only mention one particular situation term, makes the theories *Markovian.* By Markovian we mean that the entire history is encoded in the current situation and, thus, that the current state contains all information necessary to find the next action to execute. Ballistic actions in general, cannot exist in a Markovian system, since their effects may not be measurable at every sampling instance. We can safely assume that all actions are energized.

7.8.1 Discrepancy detection and classification

There is no difference between the detection process for stability-based execution monitoring and for ontological control.

For classification, there are two possible causes for the discrepancy, EA (a benign discrepancy) or VOA (a malignant discrepancy).
EA
Now, if

$$\begin{aligned}&\{\, F(\vec{o}) \mid F \in Conf \text{ and } F(\vec{o}) \in O_{\mathbf{s}} \,\} \\ &= \\ &\{\, F(\vec{o}) \mid F \in Conf \text{ and } F(\vec{o}) \in O'_{do(A,\mathbf{s})} \}, \qquad (7.4)\end{aligned}$$

we know that the actuator sensors have not changed from situation $\mathbf{s}$ to $do(A, \mathbf{s})$, and we explain the the discrepancy with EA. We then change the application axiomatization to $\Delta' = (\Delta - ?_{S_0}) \cup \{\bigwedge O'_{do(A,\mathbf{s})}/S_0\}$. That is, we replace the initial situation axioms by the latest observation.

VOA

We detect a VOA by noticing that

$$\begin{aligned} &\{\, F(\vec{o}) \mid F \in \mathit{Conf} \text{ and } F(\vec{o}) \in O_{do(A,\mathbf{s})} \,\} \\ &= \\ &\{\, F(\vec{o}) \mid F \in \mathit{Conf} \text{ and } F(\vec{o}) \in O'_{do(A,\mathbf{s})} \,\}, \end{aligned} \tag{7.5}$$

i.e. that the actuator sensors have changed exactly as expected, but other changes have not followed our expectations.

7.8.2 Trolley example

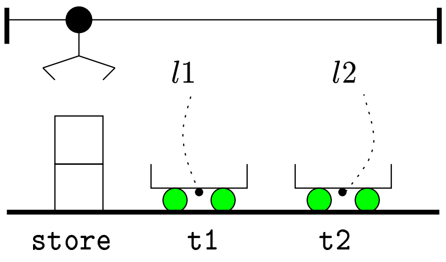

Figure 7.2: The plant.

We will now look at an example of a plant depicted in Figure 7.2, which was introduced in Chapter 4. We control a robot arm (our actuator) that is supposed to move boxes from `store` to the trolleys at positions `t1` and `t2`. The actuator output is the value of a fluent $Position(p, s)$ (for the position of the robot arm) p can take the values `store`, `t1`, or `t2`, representing the depicted possible positions. The plant signals give values to a fluent $Load(l, n)$, where l denotes the sensor ($l1$ or $l2$), and n denotes the number of boxes on the respective trolley, i.e. 0, 1, or 2. We have one control action, $Move(x, y)$, where x and y are two different positions, and we assume that when x is `store`, y is `t1` or `t2`, and there are less than 2 boxes on the trolleys, $Move(x, y)$ invokes the robot arm to grab one box at `store` and move to `t1` or `t2` and to drop the box at that position. We choose the following SitCalc axiomatization:

$$\begin{aligned} &Poss(Move(x, y), s) \equiv \\ &\quad x \neq y \wedge (y = \texttt{store} \vee (x = \texttt{store} \wedge \\ &\quad\quad \forall n, n'.\, (H(Load(l1, n), s) \wedge \\ &\quad\quad H(Load(l2, n'), s) \rightarrow n + n' < 2))), \end{aligned}$$

We define the successor state axiom for our only member of *Conf*, *Position*, as

$$\begin{aligned}
&H(Position(p), do(a,s)) \equiv \\
&\quad \exists x.a = Move(x,p) \lor \\
&\quad (H(Position(p), s) \land \neg\, a = Move(p,x)).
\end{aligned}$$

The successor state axiom for *Load* is somewhat larger:

$$\begin{aligned}
&H(Load(l,n), do(a,s)) \equiv \\
&\quad (\quad (l = l1 \land n = 0 \land \mathbf{F}) \lor \\
&\quad (\quad ((l = l1 \land n = 1 \land H(Load(l,0), s)) \lor \\
&\qquad\quad (l = l1 \land n = 2 \land H(Load(l,1), s))) \land \\
&\qquad\quad a = Move(\mathtt{store}, \mathtt{t1}) \quad) \lor \\
&\quad (l = l2 \land n = 0 \land \mathbf{F}) \lor \\
&\quad (\quad ((l = l2 \land n = 1 \land H(Load(l,0), s)) \lor \\
&\qquad\quad (l = l2 \land n = 2 \land H(Load(l,2), s))) \land \\
&\qquad\quad a = Move(\mathtt{store}, \mathtt{t2}) \quad) \\
&\quad) \quad \lor \\
&\quad (H(Load(l,n), s) \land \\
&\quad \neg\, ((l = l1 \land a = Move(\mathtt{store}, \mathtt{t1})) \lor \\
&\qquad (l = l2 \land a = Move(\mathtt{store}, \mathtt{t2})))).
\end{aligned}$$

Initially, the arm is at position `store` and there are no boxes on the trolleys, i.e.

$$\begin{aligned}
&?_{S_0} = \\
&\quad \{H(Position(\mathtt{store}), S_0), H(Load(l1,0), S_0), \\
&\quad H(Load(l2,0), S_0)\}
\end{aligned}$$

Now, assume that we execute $Move(\mathtt{store}, \mathtt{t1})$ in S_0. Let $\mathsf{s} = do(Move(\mathtt{store}, \mathtt{t1}), S_0)$. We then expect the observation

$$O_{\mathsf{s}} = \{Position(\mathtt{t1}), Load(l1,1), Load(l2,0)\}.$$

to be sensed. We will now illustrate the two causes of discrepancies:
EA
Assume that somebody moves a box from the store to the trolley with load sensor $l1$ when the robot arm has begun executing $Move(\mathtt{store}, \mathtt{t1})$. Since actions are energized, and the exogenous move will make the precondition of the action false (i.e. the state in which the action was invoked), the action will stop, and we will sense

$$O'_{\mathsf{s}} = \{Position(\mathtt{store}), Load(l1,1), Load(l2,0)\}.$$

We detect the discrepancy by noting that $O_{\mathtt{s}} \neq O'_{\mathtt{s}}$, and classify the cause of the discrepancy as EA with Equation (7.4). To continue, we replace $?_{S_0}$ by

$$\begin{aligned}&\{\bigwedge O'_{\mathtt{s}/S_0}\} = \\ &\quad \{H(\mathit{Position}(\mathtt{store}), S_0) \wedge H(\mathit{Load}(l1,1), S_0) \wedge \\ &\quad H(\mathit{Load}(l2,0), S_0)\}\end{aligned}$$

and try to recover the GOLOG program.

VOA

An unmodeled ontological assumption of the system is that position `t1` corresponds to load sensor $l1$, and that `t2` corresponds to $l2$. If this is false the model is not valid for control. Thus, we assume that the position of the trolleys has been changed, which means that the sensed observation after performing $\mathit{Move}(\mathtt{store}, \mathtt{t1})$ in S_0 will be

$$O'_{\mathtt{s}} = \{\mathit{Position}(\mathtt{t1}), \mathit{Load}(l1,0), \mathit{Load}(l2,1)\}$$

Again, we detect the discrepancy by noticing that $O_{\mathtt{s}} \neq O'_{\mathtt{s}}$, and classify it as caused by VOA with Equation (7.5). We have four discrepancies, where $\mathit{Load}(l1,1)$ is of type D, $\mathit{Load}(l2,0)$ is of type A, $\mathit{Load}(l1,0)$ is of type B, and $\mathit{Load}(l2,1)$ is of type C. The precondition of the VOA is in this case

$$\begin{aligned}&\bigwedge O_{S_0/s} \equiv \\ &\quad H(\mathit{Position}(\mathtt{store}), s) \wedge H(\mathit{Load}(l1,0), s) \wedge \\ &\quad H(\mathit{Load}(l2,0), s).\end{aligned}$$

Thus, we have

$$\beta^{S_0}_{\mathit{Load}(l1,1)} \equiv \bigwedge O_{S_0/s} \wedge \underbrace{l = l1 \wedge n = 1}_{inst^{l1,1}_{l,n}}.$$

Similarly, we can construct the β^s_F for each discrepancy F.

When we repair the successor state axiom for Load we get the following result:

$$\begin{aligned}&H(\mathit{Load}(l,n), do(a,s)) \equiv \\ &\quad (\\ &\quad (l = l1 \wedge n = 0 \wedge \mathbf{F}) \vee \\ &\quad (\quad ((l = l1 \wedge n = 1 \wedge H(\mathit{Load}(l,0), s)) \vee \\ &\qquad (l = l1 \wedge n = 2 \wedge H(\mathit{Load}(l,1), s)) \vee \beta^{S_0}_{\mathit{Load}(l1,0)}) \wedge \\ &\qquad \neg\beta^{S_0}_{\mathit{Load}(l1,1)} \wedge a = \mathit{Move}(\mathtt{store}, \mathtt{t1}) \quad) \vee \\ &\qquad\quad \ldots \\ &\quad (H(\mathit{Load}(l,n), s) \wedge \\ &\quad \neg\,(((l = l1 \vee \beta^{S_0}_{\mathit{Load}(l2,0)}) \wedge \neg\beta^{S_0}_{\mathit{Load}(l2,1)} \wedge \\ &\qquad a = \mathit{Move}(\mathtt{store}, \mathtt{t1})) \vee \\ &\qquad (l = l2 \wedge a = \mathit{Move}(\mathtt{store}, \mathtt{t2})))).\end{aligned}$$

It is easy to see that this particular VOA cannot occur again. It will be handled properly by the new successor state axiom. However, if we perform the action $Move(\texttt{store}, \texttt{t2})$ a similar VOA will occur.

Chapter 8

Conclusions and future work

In this Chapter we will conclude and summarize the work presented in this thesis, as well as present some pontentially fruitful future research paths.

8.1 Conclusions

In Chapter 1 two problems that have guided this work were posed. They were:

- How can control engineers handle the increasing demands for safety and optimality of control systems, in settings where the systems themselves or their operating environments severely restrict the possibility of precise mathematical modeling?
- How can the problem above be solved with minimal introduction cost, that is, minimal cost for introducing new technology?

The hypothesis of this dissertation is that a feasible answer to the questions is:

Use model-based execution monitors.

An execution monitor is, in this thesis, a separate architectural entity that should be mounted on a controller with access to the controllers inputs and outputs (and possibly with access to internal control structures of the controller). Then, given a model of the closed-loop system, the the execution of the controller is monitored with a particular focus on the detection of discrepancies between the actual and predicted effects of action invocations.

In arguing for the hypothesis, we have had to clarify some methodological issues such as what execution monitoring is (by specifying an appropriate working definition), the impact that a particular choice of modeling formalism has on execution monitoring, and the identification of the central research issues. This work was presented in Chapter 2. The idea of employing a model-based execution monitor is not new, nor does it belong to one specific academic discipline, so in Chapter 3 we provided an attempt to review, and compare, some representative work on execution monitoring in Control Theory, Computer Science, and AI. As a result of the discussion in Chapter 2, a set of five constituting functions of execution monitors was identified. These were *situation assessment* (mapping the current sensor measurements to a state, or situation, in the model), *expectation assessment* (computing the predicted current state, or situation), *discrepancy detection* (comparing the actual and predicted current state, and determining whether there is a discrepancy between the two), *discrepancy classification* (explaining a detected discrepancy, or assessing its potential harmfulness), and *recovery* (taking some action to make the system behave as wanted after a discrepancy, if possible).

This thesis has focused on discrepancy classification, and in Chapters 4 and 5 two classification paradigms were presented: Ontological Control and Stability-Based execution monitoring. Ontological control concerns execution monitoring of software-based industrial process controllers, and was originally (by Fodor [1995]) developed to handle infinite recovery loops. Stability-based execution monitoring concerns the insurance that a closed-loop system is stable throughout its execution, that is, that a given criterion always holds, or can be made to hold, whatever state the system currently is in.

For ontological control, we have presented an implementation of the theory, and a first set of experimental results, where the implementation was tested on a subsystem of a real industrial process controller. The results are promising (but inconclusive).

In Chapters 6 and 7 the two paradigms were applied in two different modeling formalisms: Hybrid Automata and the Situation Calculus/GOLOG framework. For both these formalisms a number of extensions were developed to cope with the requirements of execution monitoring.

Five of the most important contributions of this thesis are the following:

- Reviewing and comparing a number of previously proposed approaches to execution monitoring in Control Theory, Computer Science, and AI (in Chapter 3). A major part of this work consists in the construction of a conceptual framework that enables such comparisons (Chapter 2).

- The formalization, generalization, extension, and implementation of Ontological Control (Chapter 4). This includes our proposal as to how to (semi-) automatically generate a model of a closed-loop system by analyzing the control program.

- The development and analysis of the notion of "maintainability", and its application to an execution monitoring setting (Chapter 5).

- The attempt to bridge the gap between system verification and system execution monitoring (Chapter 6). This was demostrated by transforming closed-loop models in a formalism designed for verification to a formalism that could be utilized by an execution monitor engine.

- The analysis of the feasibility and utility of using a logical framework for the types of execution monitoring presented in this thesis (Chapter 7).

8.2 Future work

Guided by the five contributions above, we will in this section state some of the more interesting prospects for future work on model-based execution monitoring.

- We are pleased to see that steps to bridge the gap between FDI and model-based diagnosis now are occurring more commonly in the literature (see e.g. Cordier *et al.* [2000]). However, there is still a need for more work that links concepts and techniques from AI, Control Theory and Computer Science, from an execution monitoring point of view. As the importance of critical and complex software-based control systems increases in society, there are large benefits to be gained by a closer integration of the three areas.

- In its present form, ontological control places quite strict restrictions on systems in order to be applicable. An idea that has not been investigated is "partial classifiability", where even though the entire system does not satisfy the restrictions, parts of it does, and guarantees of correct classification extend only to those parts.

  The problem of automatically generating models from a control program is an interesting and complex problem with potential import. In the approach taken in Chapter 4 we only consider the logical structure of the given program, which suffices to semi-automatically generate a model. A problem that we experienced with the generated model during the experiments was that for every sensor reading the execution monitor recieved as input, there were many (20-30) states that materialized, and it was not easy to perform situation assessment even with the help of the goal paths. However, there is more information available that could be utilized to automate more of the translation process, and potentially would give us richer models. The next step we have considered is to use *temporal* information from the program, that is, the logical computations in the program are necessarily made in a partial temporal order. This information could be used to limit the number of candidate states in the situation assessment process.

- As shown in Chapter 5 the notions "stability" and "maintainability" are incomparable, while the notion "(k, l)-maintainability" generalizes both of them. This interesting notion has not yet recieved the attention it deserves, but will be analyzed more deeply in the future.

- Chapter 6 concerns the translation of hybrid automata to a state transition diagram that can be used for execution monitoring. On the other hand, how one generates an executable controller program from a hybrid automata is an open question. Is it, for instance, possible to simultaneously generate a controller program and a state transition digram of the new closed-loop system for execution monitoring, and could this be a feasible way to automatically proceed from a verified specification to a safe implemented control system?

In conclusion, the topic of execution monitoring is an exciting research area with much left to be done. As autonomous systems are becoming more common, and will continue to be, execution monitoring will be even more important. In some sense, it provides a limited form of machine introspection and a potential basis for self-repair of autonomous systems that will become increasingly more important in the future.

We have shown in this thesis that parts of the puzzle can be solved for less challenging, but still complex software systems, and we look forward to new advances in execution monitoring in the future.

Bibliography

[AAAI '00, 2000] R. Dechter and R. Sutton, editors. *Proceedings of the Seventeenth National Conference on Artificial Intelligence*, Austin, Texas, August 2000. American Association for Artificial Intelligence, AAAI Press/MIT Press.

[AAAI '87, 1987] American Association for Artificial Intelligence. *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI '87)*, Seattle, Washington, July 1987. AAAI Press/MIT Press.

[AAAI '94, 1994] American Association for Artificial Intelligence. *Proceedings of the Twelfth National Conference on Artificial Intelligence*, Seattle, Washington, August 1994. AAAI Press/MIT Press.

[AAAI '96, 1996] American Association for Artificial Intelligence. *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, Portland, Oregon, August 1996. AAAI Press/MIT Press.

[ABB, 1999] ABB Industrial Systems. *Stressometer: Application overview and and principles.*, 1999. 4/6 High Mill Application, Version 5.0.

[Abello and Dolev, 1997] J. Abello and S. Dolev. On the computational power of self-stabilizing systems. *Theoretical Computer Science*, 183:159 – 170, 1997.

[Abramson, 1991] B. Abramson. An analysis of error recovery and sensory integration for dynamic planners. In *Proceedings of the Ninth National Conference on Artificial Intelligence (AAAI'91)*, Anaheim, California, July 1991. American Association for Artificial Intelligence, AAAI Press.

[Agre and Chapman, 1987] P. Agre and D. Chapman. Pengi: An implementation of a theory of activity. In AAAI '87 [1987].

[AIPS '94, 1994] Kristian Hammond, editor. *Proceedings of the Second International Conference on Artificial Intelligence Planning Systems*, Chicago, Illinois, 1994. AAAI Press.

[Alur *et al.*, 1992] R. Alur, C. Courcoubetis, T. Henzinger, and P.-H. Ho. Hybrid automata: An algorithmic approach to the specification and and verification of hybrid systems. In *Workshop on Theory of Hybrid Systems*, volume 736 of *Lecture*

Notes in Computer Science, pages 209–229. Springer Verlag, Lyngby, Denmark, October 1992.

[Alur *et al.*, 1993] R. Alur, T. Henzinger, and P. Ho. Automatic symbolic verification of embedded systems. In *Proceedings of IEEE Real-Time Systems Symposium*, Raleigh-Durham, NC, December 1993.

[Ambros-Ingerson and Steel, 1988] J. Ambros-Ingerson and S. Steel. Intergrating planning, execution and monitoring. In *Proceedings of the Seventh National Conference on Artificial Intelligence (AAAI'88)*, Saint Paul, Minnesota, August 1988. American Association for Artificial Intelligence, AAAI Press/The MIT Press.

[Arkin, 1990] R.C. Arkin. Intergrating behavioral, perceptual and world knowledge in reactive navigation. *Robotics and Autonomous Systems*, 6:105–122, 1990.

[Baier and Pinto, 1998] Jorge Baier and Javier Pinto. Non-instantaneous Actions and Concurrency in the Situation Calculus (Extended Abstract). In Giuseppe de Giacomo and Daniele Nardi, editors, *10th European Summer School in Logic, Language and Information*, 1998.

[Baker, 1989] A. Baker. A simple solution to the Yale shooting problem. In R.J. Brachman, H.J. Levesque, and R. Reiter, editors, *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning (KR '89)*, Toronto, Canada, May 1989. Morgan Kaufmann.

[Baral *et al.*, 2000] C. Baral, S. McIlraith, and T.C. Son. Formulating diagnostic problemsolving using an action language with narratives and sensing. In F. Giunchiglia and B. Selman, editors, *Proceedings of the Seventh International Conference on Principles of Knowledge Representation and Reasoning (KR'00)*, Breckenridge, CO, US, April 2000. Morgan Kaufmann, San Francisco.

[Basseville and Nikiforov, 1993] M. Basseville and I. Nikiforov. *Detection of Abrupt Changes: Theory and Applications.* Prentice Hall, Englewood Cliffs, NJ, USA, 1993.

[Beard, 1971] R.V. Beard. *Failure accomodation in linear systems through self reorganization.* Phd thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 1971.

[Beetz and McDermott, 1994] M. Beetz and D. McDermott. Improving robot plans during their execution. In AIPS '94 [1994].

[Benson, 1996] S. Benson. *Learning Action Models for Reactive Autonomous Agents.* Phd thesis, Department of Computer Science, Stanford University, 1996.

[Bjäreland and Driankov, 1999] M. Bjäreland and D. Driankov. Synthesizing discrete controllers from hybrid automata - preliminary report. In *Working Papers of the AAAI Spring Symposium on Hybrid Systems and AI*, Stanford, CA, USA, March 1999.

[Bjäreland and Fodor, 1998] M. Bjäreland and G. Fodor. Ontological control. In *Working Papers of the Ninth International Workshop on Principles of Diagnosis (Dx'98)*, Sea Crest Resort, N. Falmouth, MA, USA, May 1998.

[Bjäreland and Fodor, 2000] M. Bjäreland and G. Fodor. Execution monitoring of industrial process controllers: An application of ontological control. In SAFEPROCESS 2000 [2000].

[Bjäreland and Haslum, 1999] M. Bjäreland and P. Haslum. Stability, stabilizability, and golog. Unpublished, August 1999.

[Bjäreland and Karlsson, 1997] M. Bjäreland and L. Karlsson. Reasoning by regression: Pre- and postdiction procedures for logics of action and change with nondeterminism. In IJCAI '97 [1997].

[Bjäreland and Pinto, 2000] M. Bjäreland and J. Pinto. Handling surprises in logics of action and change. Unpublished manuscript, 2000.

[Bjäreland, 1999a] M. Bjäreland. Execution monitor synthesis for hybrid systems – preliminary report. In *Proceedings of the Fourteenth IEEE International Symposium on Intelligent Control (ISIC'99)*, Boston, USA, September 1999.

[Bjäreland, 1999b] M. Bjäreland. Recovering from modelling faults in GOLOG. In *Proceedings of the IJCAI'99 Workshop: Scheduling and Planning meet Real-time Monitoring in a Dynamic and Uncertain World*, Stockholm, Sweden, August 1999.

[Blanke *et al.*, 2000] M. Blanke, C.W. Frei, F. Kraus, R.J. Patton, and M. Staroswiecki. What is fault-tolerant control ? In SAFEPROCESS 2000 [2000].

[Blum and Furst, 1997] A. Blum and M. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90:981, 1997.

[Brooks, 1986] R. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, page 14..23, April 1986.

[Brooks, 1991] R. Brooks. Intelligence without representation. *Artificial Intelligence*, 47(1 – 3):139 – 159, 1991.

[Broverman and Croft, 1988] C. Broverman and B. Croft. Reasoning about exceptions during plan execution monitoring. In AAAI '87 [1987].

[Carpanzano *et al.*, 1999] E. Carpanzano, L. Ferrarini, and C. Maffezzoni. An object-oriented model for hybrid control systems. In *Proceedings of the 1999 IEEE International Symposium on Computer Aided Control System Design*, Kohala Coast-Island of Hawai'i, Hawai'i, USA, August 1999.

[Cassandra *et al.*, 1994] A.R. Cassandra, L.P. Kaelbling, and M.L. Littman. Acting optimally in partially observable stochastic domains. In AAAI '94 [1994].

[Chapman, 1987] D. Chapman. Planning for conjunctive goals. *Artificial Intelligence*, 32:333 – 377, 1987.

[Chen and Patton, 1999] J. Chen and R.J. Patton. *Robust model-based fault diagnosis for dynamic systems.* Kluwer Academic Publishers, 1999.

[Chittaro *et al.*, 1993] L. Chittaro, G. Guida, C. Tasso, and E. Toppano. Functional and teleological knowledge in the multimodeling approach for reasoning about physical systems: A case study in diagnosis. *IEEE Transactions on Systems, Man and Cybernetics*, 23(6):1781–1751, November 1993.

[Coradeschi and Saffiotti, 2000] S. Coradeschi and A. Saffiotti. Anchoring symbols to sensor data: Preliminary report. In AAAI '00 [2000].

[Cordier *et al.*, 2000] M-O. Cordier, P Dague, M. Dumas, F. Lévy, J. Montmain, M. Staroswiecki, and L.nd Tr Travé-Massuyés. AI and automatic control approaches to model-based diagnosis: Links and underlying hypotheses. In SAFEPROCESS 2000 [2000].

[de Giacomo and Levesque, 1998] G. de Giacomo and H. Levesque. An Incremental Interpreter for High-Level Programs with Sensing. Technical report, University of Toronto, 1998. `URL = http://www.cs.toronto.edu/cogrobo/incr-exe.ps.Z`.

[De Giacomo and Levesque, 1999] G. De Giacomo and H. Levesque. Projection using regression and sensors. In IJCAI '99 [1999].

[De Giacomo and Vardi, 2000] G. De Giacomo and M. Vardi. Automata-theoretic approach to planning for temporally extended goals. In S. Biundo and M. Fox, editors, *Recent advances in AI Planning*, volume 1809 of *LNCS*. Springer-Verlag, 2000. Proceedings of the 5th European Conference on Planning (ECP'99) in Durham, UK, September 1999.

[De Giacomo *et al.*, 1997] G. De Giacomo, Y. Lesperance, and H.J. Levesque. Reasoning about concurrent execution, prioritized interrupts, and exogenous actions in the situation calculus. In IJCAI '97 [1997].

[De Giacomo *et al.*, 1998] G. De Giacomo, R. Reiter, and M. Soutchanski. Execution monitoring of high-level robot programs. In KR'98 [1998].

[de Giacomo *et al.*, 1999a] G. de Giacomo, Y. Lesprance, and H.J. Levesque. Congolog, a concurrent programming language based on the situation calculus: foundations. Submitted for publication, February 1999.

[de Giacomo *et al.*, 1999b] G. de Giacomo, Y. Lesprance, and H.J. Levesque. Congolog, a concurrent programming language based on the situation calculus: language and implementation. Submitted for publication, February 1999.

[de Kleer and Williams, 1987] J. de Kleer and B. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32:97–130, 1987.

[Dean and Wellman, 1991] T.L. Dean and M.P. Wellman. *Planning and Control.* Morgan Kaufmann, San Mateo, CA, USA, 1991.

[Dijkstra, 1974] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17:643 – 644, 1974.

[Dijkstra, 1986] E.W. Dijkstra. A belated proof of self-stabilization. *Distributed Computing*, 1:5 – 6, 1986.

[Doherty *et al.*, 1998] P. Doherty, J. Gustafsson, L. Karlsson, and J. Kvarnström. TAL: Temporal action logics language specification and tutorial. *Linköping Electronic Articles in Computer and Information Science*, 3(015), 1998. URL: `http://www.ep.liu.se/ea/cis/1998/015/`.

[Doyle *et al.*, 1986] R. Doyle, D. Atkinson, and R. Doshi. Generating perception requests and expectations to verify the execution of plans. In *Proceedings of the Fifth National Conference on Artificial Intelligence (AAAI'86)*, Philadelphia, Pennsylvania, August 1986. American Association for Artificial Intelligence, AAAI Press/MIT Press.

[Dvorak and Kuipers, 1989] D. Dvorak and B. Kuipers. Model-based monitoring of dynamic systems. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, Los Altos, CA, USA, 1989. Morgan Kaufmann.

[Dvorak and Kuipers, 1991] D. Dvorak and B. Kuipers. Process monitoring and diagnosis: A model-based approach. *IEEE Expert*, 5(3):67 – 74, 1991.

[Earl and Firby, 1997] C. Earl and J. Firby. Combined execution and monitoring for control of autonomous agents. In *Proceedings of the First International Conference on Autonomous Agents (AGENTS'97)*, Marina Del Rey, CA, USA, February 1997. ACM.

[Falkenroth, 2000] E. Falkenroth. *Database Technology for Control and Simulation.* Phd thesis, Linköping Studies in Science and Technology no. 637, Linköpings universitet, Sweden, 2000.

[Faurre and Depeyrot, 1977] P. Faurre and M. Depeyrot. *Elements of System Theory.* North-Holland, 1977.

[Ferguson, 1992] I. Ferguson. TouringMachines: Autonomous agents with attitudes. *IEEE Computer*, 25(5), 1992.

[Fikes and Nilsson, 1971] R. Fikes and N. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189 – 208, 1971.

[Fikes *et al.*, 1972] R. Fikes, P. Hart, and N. Nilsson. Learning and executing generalized robot plans. *Artificial Intelligence*, 3(4):251–288, 1972.

[Fodor, 1995] G. Fodor. *Ontological Control – Description, Identification, and Recovery from Problematic Control Situations.* PhD thesis, Department of Computer and Information Science, Linköpings universitet, Sweden, 1995.

[Fodor, 1998] G. Fodor. *Ontologically Controlled Autonomous Systems: Principles, Operations and Architecture.* Kluwer Academic, 1998.

[Frank, 1990] P. Frank. Fault diagnosis: A survey and some new results. *Automatica: IFAC Journal*, 26(3):459 – 474, 1990.

[Gelfond and Lifschitz, 1998] M. Gelfond and V. Lifschitz. Action languages. *Linköping Electronic Articles in Computer and Information Science*, 3(016), 1998. URL: `http://www.ep.liu.se/ea/cis/1998/016/`.

[Gil, 1992] Y. Gil. *Acquiring Domain Knowledge for Planning by Experimentation.* PhD thesis, Carnegie Mellon University, 1992.

[Gu *et al.*, 1994] W. Gu, J. Vetter, and K. Schwan. An annotated bibliography of interactive program steering. Technical Report GIT–CC–94–15, College of Computing, Georgia Institute of Technology, Atlanta, USA, 1994. Available at `http://www.cc.gatech.edu/systems/projects/FALCON/`.

[Gu *et al.*, 1997] W. Gu, G. Eisenhauer, and K. Schwan. Falcon: On-line monitoring and steering of parallel programs. In *Procfeedings of the Ninth International Conference on Parallel and Distributed Computing and Systems (PDCS'97)*, Washington D.C., USA, October 1997.

[Hammond, 1990] K. Hammond. Explaining and repairing plans that fail. *Artificial Intelligence*, 45:173–228, 1990.

[Hamscher *et al.*, 1992] W. Hamscher, L. Console, and J. de Kleer, editors. *Readings in Model-Based Diagnosis.* Morgan Kaufmann, 1992.

[Henzinger and Kopke, 1997] T. Henzinger and P. Kopke. Discrete-time control for rectangular hybrid automata. In *Proceedings of the Twentyfourth International Colloquium on Automata, Languages, and Programming (ICALP'97)*, volume 1256 of *Lecture Notes in Computer Science*, pages 582–593. Springer-Verlag, 1997.

[Henzinger *et al.*, 1997] T. Henzinger, P.-H. Ho, and H. Wong-Toi. HyTech: A model checker for hybrid systems. In *Proceedings of the Ninth International Conference on Computer-Aided Verification (CAV'97)*, volume 1254 of *Lecture Notes in Computer Science*, pages 460–463. Springer-Verlag, 1997.

[Herman, 1999] T. Herman. Self-stabilization bibliography: Access guide. `http://www.cs.uiowa.edu/ftp/selfstab/bibliography/access.html`, September 1999. Available only via www.

[IJCAI '97, 1997] M.E. Pollack, editor. *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI '97)*, Nagoya, Japan, August 1997. Morgan Kaufmann.

[IJCAI '99, 1999] T. Dean, editor. *Prooceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, Stockholm, Sweden, August 1999. Morgan Kaufmann.

[Jacobson and Nett, 1991] C.A. Jacobson and C.N. Nett. An integrated approach to controls and diagnostics using the four parameter control. *IEEE Control Systems Magazine*, 11(6):22–29, 1991.

[Kaelbling and Rosenschein, 1991] L. Kaelbling and S. Rosenschein. Action and planning in embedded agents. In P. Maes, editor, *Designing Autonomous Agents*, pages 35–48. MIT Press, 1991.

[Kalman, 1960] R.E. Kalman. A new approach to linear filtering and prediction problems. *Journal of Basic Engineering*, pages 35 – 46, 1960.

[Knoblock, 1995] C. Knoblock. Planning, executing, sensing, and replanning for information gathering. In C. Mellish, editor, *Proceedings of the Fourteenth International joint Conference on Artificial Intelligence (IJCAI'95)*, Montreal, Canada, August 1995. Morgan Kaufmann.

[KR'98, 1998] A. Cohn, L. Schubert, and S. Shapiro, editors. *Principles of Knowledge Representation and Reasoning: Proceedings of the Sixth International Conference (KR'98)*, Trento, Italy, June 1998. Morgan Kaufmann, San Francisco.

[Kumar and Garg, 1995] R. Kumar and V. Garg. *Modeling and Control of Logical Discrete Event Systems.* Kluwer Academic Publishers, Boston, MA, USA, 1995.

[Lee and Durfee, 1994] J. Lee and E. Durfee. Structured circuit semantics for reactive plan execution systems. In AAAI '94 [1994].

[Lennartsson *et al.*, 1996] B. Lennartsson, M. Tittus, B. Egardt, and S. Pettersson. Hybrid systems in process control. *IEEE Control Magazine*, October 1996.

[Levesque *et al.*, 1997] Hector J. Levesque, Raymond Reiter, Yves Lespérance, Fangzhen Lin, and Richard B. Scherl. GOLOG: A Logic Programming Language for Dynamic Domains. *The Journal of Logic Programming*, 31:59–84, 1997.

[Levesque *et al.*, 1998] H. Levesque, F. Pirri, and R. Reiter. Foundations for the situation calculus. *Linköping Electronic Articles in Computer and Information Science*, 3(018), 1998. URL: `http://www.ep.liu.se/ea/cis/1998/018/`.

[Lewau, 1999] P. Lewau. A prototype of an ontological controller. Master's thesis, Linköping Studies in Science and Technology, Linköpings universitet, April 1999. No. LiTH–IDA–Ex–9949.

[Lewis, 1997] R.W. Lewis. *Programming industrial control systems using IEC 1131-3.* Number 50 in IEE Control Engineering Series. The Institution of Electrical Engineers, London, United Kingdom, 1997.

[Lin, 1996] F. Lin. Embracing Causality in Specifying the Indeterminate Effects of Actions. In AAAI '96 [1996].

[Łukaszewicz, 1990] W. Łukaszewicz. *Non-Monotonic reasoning: formalization of commonsense reasoning.* Ellis Horwood, 1990.

[Lyons and Hendriks, 1995] D. Lyons and A. Hendriks. Planning as incremental adaption of a reactive system. *Robotics and Autonomous Systems*, 14:255–288, 1995.

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

[McIlraith, 1997] S. McIlraith. *Towards a formal account diagnostic problem solving.* Phd thesis, University of Toronto, 1997.

[McIlraith, 1998] S. McIlraith. Explanatory diagnosis: Conjecturing actions to explain observations. In KR'98 [1998].

[McIlraith, 1999] S. McIlraith. Model-based programming using golog and the situation calculus. In *Working Papers of the Tenth International Workshop on Principles of Diagnosis (Dx99)*, Loch Awe, Scotland, June 1999.

[Misawa and Hedrick, 1989] E.A. Misawa and J.K. Hedrick. Nonlinear observers – A state-of-the-art survey. *Journal of Dynamic Systems, Measurement, and Control*, 111:344 – 352, 1989.

[Munson, 1971] J. Munson. Robot planning, execution, and monitoring in an uncertain environment. In *Proceedings of the Second International Joint Conference on Artificial Intelligence (IJCAI'71)*, London, England, 1971. Morgan Kaufmann.

[Muscettola *et al.*, 2000] N. Muscettola, P. Nayak, B. Pell, and B. Williams. Remote agent: To boldly go where no AI system has gone before. *Artificial Intelligence*, 2000. To Appear.

[Musliner *et al.*, 1995] D.J. Musliner, E.H. Durfee, and K.G. Shin. CIRCA: A cooperative intelligent real-time control architecture. *IEEE Transactions on Systems, Man and Cybernetics*, 23(6):1561 – 1574, 1995.

[Nakamura *et al.*, 2000] M. Nakamura, C. Baral, and M. Bjäreland. Maintainability: a weaker stabilizability-like notion for high-level control agents. In AAAI '00 [2000].

[Nilsson, 1982] N. Nilsson. *Principles of Artificial Intelligence.* Springer-Verlag, 1982.

[Nilsson, 1994a] Nils J. Nilsson. Teleo-Reactive Programs for Agent Control. *Journal of Artificial Intelligence Research*, 1:139–158, 1994.

[Nilsson, 1994b] N.J. Nilsson. Teleo-reactive programs for agent control. *Journal of Artificial Intelligence Research*, 1:139–158, 1994.

[O'Reilly, 1983] J. O'Reilly. *Observers for linear systems.* Academic Press, London, 1983.

[Özveren *et al.*, 1991] C. Özveren, A. Willsky, and P. Antsaklis. Stability and stabilizability of discrete event dynamic systems. *Journal of the ACM*, 38(3):730–752, July 1991.

[Passino and Burgess, 1998] K. Passino and K. Burgess. *Stability Analysis of Discrete Event Systems.* Adaptive and Learning Systems for Signal Processing, Communications, and Control. John Wiley and Sons, Inc., New York, 1998.

[Pinto and Bjäreland, 2001] J. Pinto and M. Bjäreland. An architecture for execution monitoring. Submitted to the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI '01), August 2001.

[Pinto, 1994] Javier Pinto. *Temporal Reasoning in the Situation Calculus.* PhD thesis, Department of Computer Science, University of Toronto, Toronto, Ontario, Canada, February 1994. URL = ftp://ftp.cs.toronto.edu/~cogrobo/jpThesis.ps.Z.

[Pinto, 1998a] J. Pinto. Concurrent actions and interacting effects. In KR'98 [1998].

[Pinto, 1998b] J. Pinto. Occurrences and narratives as constraints in the branching structure of the situation calculus. *Journal of Logic and Computation*, 8(6):777–808, December 1998.

[Ramadge and Wonham, 1989] P. Ramadge and W. Wonham. The control of discrete event systems. *Proceedings of the IEEE: Special Issue on Discrete Event Systems.*, 77:81–98, 1989.

[Reiter, 1978] R. Reiter. On closed-world data bases. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 55–76. Plenum Press, New York, 1978.

[Reiter, 1987] R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57 – 95, 1987.

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

[Reiter, 1998] Ray Reiter. Sequential, Temporal GOLOG. In A. Cohn, L. Schubert, and S. Shapiro, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Sixth International Conference (KR'98)*, pages 547–556. Morgan Kaufmann, June 1998.

[Reiter, 1999] Raymond Reiter. KNOWLEDGE IN ACTION: Logical Foundations for Describing and Implementing Dynamical Systems. Book Draft, available from `http://www.cs.utoronto.ca/~cogrobo`, 1999.

[Rich and Knight, 1991] E. Rich and K. Knight. *Artficial Intelligence.* McGraw-Hill, Inc., 2nd edition, 1991.

[Rinner and Kuipers, 1999] B. Rinner and B. Kuipers. Monitoring piecewise continuous behaviours by refining semi-quantitative trackers. In IJCAI '99 [1999].

[Sacerdoti, 1977] E. Sacerdoti. *A Structure for Plans and Behaviour.* Artificial Intelligence series. Elsevier North-Holland, New York, 1977.

[SAFEPROCESS 2000, 2000] A.M. Edelmayer, editor. *Proceedings of the Fourth IFAC Symposium on Fault Detection, Supervision, and Safety for Technical Processes (SAFEPROCESS 2000)*, Budapest, Hungary, June 2000. IFAC.

[Saffiotti, 1998] A. Saffiotti. *Autonomous Robot Navigation: A Fuzzy Logic Approach.* PhD Thesis, Faculté de Science Appliquées, IRIDIA, Université Libre de Bruxelles, 1998.

[Sandewall and Shoham, 1994] E. Sandewall and Y. Shoham. Nonmonotonic temporal reasoning. In D. Gabbay, editor, *Handbook of logic in artificial intelligence and logic programming*, volume 2. Oxford University Press, 1994.

[Sandewall, 1994] E. Sandewall. *Features and Fluents. The Representation of Knowledge about Dynamical Systems, volume I.* Oxford University Press, 1994. ISBN 0-19-853845-6.

[Sandewall, 1995] Erik Sandewall. *Features and Fluents, A Systematic Approach to the Representation of Knowledge about Dynamical Systems.* Oxford University Press, 1995.

[Schroeder, 1995] B. Schroeder. On-line monitoring: A tutorial. *Computer*, pages 72–78, June 1995.

[Schubert, 1990] L.K Schubert. Monotonic solution to the frame problem in the situation calculus: an efficient method for worlds with fully specified actions. In *Knowledge Representation and Defeasible Reasoning*, pages 23–67. Kluwer Academic Press, 1990.

[Shen, 1989] W.-M. Shen. *Learning from the Environment Based on Actions and Percepts.* PhD thesis, Carnegie Mellon University, 1989.

[Simmons *et al.*, 1997] R. Simmons, R. Goodwin, K. Haigh, S. Koenig, J. O'Sullivan, and M. Veloso. XAVIER: Experience with a layered robot architecture. *SIGART Bulletin*, 8(1-4):22–33, 1997.

[Sorensen, 1985] H.W. Sorensen, editor. *Kalman Filtering: Theory and Applications.* IEEE Press, New York, 1985.

[Struss, 1997] P. Struss. Fundamentals of model-based diagnosis. In IJCAI '97 [1997].

[Thielscher, 1997] M. Thielscher. A theory of dynamic diagnosis. *Electronic Transactions of AI*, 1997. Available at `http://www.ep.liu.se/ea/cis/1997/011`.

[Thielscher, 1998] M. Thielscher. Introduction to the fluent calculus. *Linköping Electronic Articles in Computer and Information Science*, 3(014), 1998. URL: `http://www.ep.liu.se/ea/cis/1998/014/`.

[Wang, 1994] X. Wang. Learning planning operators by observation and practice. In AIPS '94 [1994].

[Williams and Nayak, 1996] B.C. Williams and P.P. Nayak. A model-based approach to reactive self-configuring systems. In AAAI '96 [1996].

[Zhang and Mackworth, 1995] Y. Zhang and A. Mackworth. Synthesis of hybrid constraint-based controllers. In *Hybrid Systems II*, number 999 in Lecture Notes in Computer Science. Springer-Verlag, 1995.

Appendix A

Stability and stabilizing Golog programs

In this appendix we will translate "stability" from chapter 5 to SitCalc. Stability is not a first-order property, since it requires quantification over infinite situation trajectories. Therefore, we represent an infinite situation trajectories with a partial *trajectory function* $\mathcal{T} : \mathcal{S} \to \mathcal{A}$. We intend $\mathcal{T}(s) = a$ to mean that action a is executed in situation s, if the specific situation trajectory defined by $\mathcal{T}$ is followed. Formally,

Definition A.0.1 (Trajectory Function)
For a SitCalc theory Δ, $\mathcal{T}$ is a *trajectory function* if

$$\Delta \models \mathcal{T}(s) = a \supset Poss(a, s).$$

□

We need our trajectory functions to be somewhat more restricted.

Definition A.0.2 (Trajectory Prefix)
Let s be a situation, and $\mathcal{T}$ be a trajectory function. We then say that s is a *prefix* of $\mathcal{T}$, denoted $Prefix(s, \mathcal{T})$ iff, $s = S_0$ or $s = do(a, s')$ such that $\mathcal{T}(s') = a$ and $Prefix(s', \mathcal{T})$ holds, for some s' and a. Formally,

$$\begin{aligned} &Prefix(s, \mathcal{T}) \equiv \\ &\quad s = S_0 \vee \exists s', a.\, s = do(a, s') \wedge \mathcal{T}(s') = a \wedge Prefix(s', \mathcal{T}). \end{aligned}$$

□

Intuitively, $Prefix(s, \mathcal{T})$, where s is a situation and $\mathcal{T}$ a trajectory function, means that either $s = S_0$ or s is constructed exactly according to $\mathcal{T}$.

Definition A.0.3 (Alive Trajectory)
Let $\mathcal{T}$ be a trajectory function. We say that $\mathcal{T}$ is *alive*, denoted $Alive(\mathcal{T})$, iff for every prefix s of $\mathcal{T}$, there exists an action a such that $\mathcal{T}(s) = a$ holds, and for every such action a it is possible to execute a in s. Formally,

$$\begin{aligned}&Alive(\mathcal{T}) \equiv \\ &\qquad Prefix(s, \mathcal{T}) \supset \exists a.\, \mathcal{T}(s) = a\end{aligned}$$

□

Intuitively, a trajectory function $\mathcal{T}$ is alive iff there exists a continuation to every prefix of $\mathcal{T}$. Clearly, an alive trajectory function defines an infinite sequence of situations. The fact that every continuation starts by a possible action follows from the definition of trajectory functions. Let $\gamma(s)$ be a boolean combination of ground atoms, i.e. of expressions of the type $H(f(\vec{o}), s)$ or $\neg H(f(\vec{o}), s)$. Such a formula describes a set of situations in which a particular property holds, the set

$$S^{\gamma} = \{s \mid \gamma(s)\}$$

and we will refer to such a formula as a *property.* Clearly, stability is not a first-order property, so we will employ second-order quantification over trajectory functions.

Definition A.0.4 (Stable Theory)
Let Δ be a SitCalc theory and $\gamma(s)$ a property. We say that Δ is *stable w.r.t.* γ iff

$$\begin{aligned}&\Delta \models \\ &\quad \forall s.\, \neg\gamma(s) \supset \\ &\qquad \forall \mathcal{T}.\, Prefix(s, \mathcal{T}) \wedge Alive(\mathcal{T}) \supset \\ &\qquad\quad \exists s'.\, s \sqsubseteq s' \wedge Prefix(s', \mathcal{T}) \wedge \gamma(s').\end{aligned}$$

□

In general, it is a distinct possibility that a system is not stable, and that we want to construct a *stabilizing* controller. In the next section we will construct a GOLOG program that does exactly this.

A.1 Stabilizing Golog programs

The main goal of this section is to define the notion of *stabilizing* GOLOG *programs*, i.e. GOLOG programs that make an unstable SitCalc theory stable, and show how such programs can be synthesized. Our main tool for this will be trajectory functions.

We begin by connecting the semantics of a GOLOG program to trajectory functions by defining the notion *program trace.*

Definition A.1.1 (Program Trace)
Let $\mathcal{T}$ be a trajectory function and δ a GOLOG program. We say that $\mathcal{T}$ is a *trace of program* δ, denoted $Trace(\mathcal{T},\delta)$, iff

$$\begin{aligned}&\forall s.\, Prefix(s,\mathcal{T}) \supset\\ &\qquad (\exists \delta'.\, TransCl(\delta, S_0, \delta', s)) \vee\\ &\quad(\exists \delta' \exists s'.\, s' \sqsubseteq s \wedge TransCl(\delta, S_0, \delta', s') \wedge Final(\delta', s'))\end{aligned}$$

□

Intuitively, $\mathcal{T}$ being a trace of δ means that every prefix of $\mathcal{T}$ is a situation that could be arrived at by executing program δ in the initial situation. The second disjunct handles the case where δ has halted in an earlier situation s'.

Clearly, if every alive trajectory function for a theory that is a trace of the same program stabilizes the theory, we can say that the program stabilizes the theory. Formally, we have

Definition A.1.2 (Stable Under Control)
Let Δ be a SitCalc theory, $\gamma(s)$ a property, and let δ be a GOLOG program. We say that Δ is *stable w.r.t.* γ *under control of* δ, or that δ *stabilizes* Δ *w.r.t.* γ , iff

$$\begin{aligned}&\Delta \models\\ &\quad \forall s.\, \neg\gamma(s) \supset\\ &\qquad \forall \mathcal{T}.\, Prefix(s,\mathcal{T}) \wedge Alive(\mathcal{T}) \wedge Trace(\mathcal{T},\delta) \supset\\ &\qquad\quad \exists s'.\, s \sqsubseteq s' \wedge Prefix(s',\mathcal{T}) \wedge \gamma(s').\end{aligned}$$

□

The controller stabilizes Δ by limiting the set of possible trajectories to only those that are execution traces of δ, and by ensuring those trajectories satisfy the stability criterion.

A.2 Synthesis of stabilizing controllers

A necessary condition for the existence of a stabilizing controller is the existence of at least one trajectory function $\mathcal{T}$ that satisfies the stability criterion w.r.t. the desired property γ. However, the function $\mathcal{T}$ is an infinite object. In order to express it as a finite GOLOG program, some further restriction is necessary.

Definition A.2.1 (Markovian Trajectory)
Let $\mathcal{T}$ be a trajectory function. We say that $\mathcal{T}$ is *Markovian* iff

$$\begin{aligned}&\Delta \models\\ &\quad \forall s \forall s'.\, (\forall f.\, H(f,s) \equiv H(f,s')) \supset \mathcal{T}(s) = \mathcal{T}(s'))\end{aligned}$$

□

Intuitively, $\mathcal{T}$ having the Markov property means that in two situations that satisfy the same set of fluents, i.e. the same *state*, the trajectory function behaves identically; it is history-independent.

Let γ be a property and suppose Δ is a SitCalc theory that is not stable w.r.t. γ. Suppose further that there exists a alive trajectory function $\mathcal{T}$ that satisfies the stability criterion, i.e.

$$\begin{aligned}\Delta \models & \\ & \forall s.\, \neg\gamma(s) \wedge Prefix(s, \mathcal{T}) \supset \\ & \quad \exists s'.\, s \sqsubseteq s' \wedge Prefix(s', \mathcal{T}) \wedge \gamma(s')\end{aligned} \tag{A.1}$$

and that $\mathcal{T}$ is Markovian. We now show how to construct a stabilizing GOLOG controller, starting from $\mathcal{T}$.

Let $\mathcal{F}_{Gr}$ be the set of ground fluents. We call a subset $\omega = \{f(\vec{o})_1, \dots, f(\vec{o})_m\}$ of $\mathcal{F}_{Gr}$ an *observation* and define an *observation test* as the formula

$$\omega(s) \equiv \bigwedge_{f(\vec{o}) \in \mathcal{F}_{Gr}} \begin{cases} f(\vec{o}) & f(\vec{o}) \in \omega \\ \neg f(\vec{o}) & f(\vec{o}) \notin \omega \end{cases}$$

For each observation ω, construct the program fragment $\omega?; a$ iff there exists some situation s such that $\omega(s)$ hold and $\mathcal{T}(s) = a$. The sought controller is then constructed as

$$\delta_{\mathcal{T}} = (\omega_1?; a_1 | \dots | \omega_m?; a_m)^*; -?,$$

where $-$ denotes any contradiction. We now set out to prove that $\delta_{\mathcal{T}}$, constructed from $\mathcal{T}$, does stabilize Δ. The basic idea is to show that every trace of $\delta_{\mathcal{T}}$ implies $\mathcal{T}$. We start with a Lemma:

Lemma A.2.2 Let $\delta_{\mathcal{T}}$ be defined as above, then, for any situation s and some program δ

$$TransCl(\delta_{\mathcal{T}}, S_0, \delta, s) \text{ implies } TransCl(\delta_{\mathcal{T}}, S_0, \delta_{\mathcal{T}}, s)$$
$$\text{and}$$
$$TransCl(\delta_{\mathcal{T}}, S_0, \delta, s) \text{ implies } Prefix(s, \mathcal{T}).$$

Proof: We use induction over s.
$TransCl(\delta_{\mathcal{T}}, S_0, \delta_{\mathcal{T}}, S_0)$ and $Prefix(S_0, \mathcal{T})$ holds by definition.
We assume that the proposition holds and prove that

$$TransCl(\delta_{\mathcal{T}}, s, \delta, do(a_i, s)) \text{ implies } TransCl(\delta_{\mathcal{T}}, s, \delta_{\mathcal{T}}, do(a_i, s)),$$

for some i. We compute the applications of $Trans$.

By definition of the *Trans* for nondeterministic iteration we have,

$$\begin{aligned} &Trans(\delta_{\mathcal{T}}, s, \delta, s') \equiv \\ &\quad (\exists \epsilon. Trans((\omega_1?; a_1| \ldots |\omega_m?; a_m), s, \epsilon, s') \wedge \delta = \epsilon; \delta_{\mathcal{T}}) \vee \\ &\quad\; (Final((\omega_1?; a_1| \ldots |\omega_m?; a_m)^*) \wedge Trans(-?, s, \epsilon, s')) \end{aligned} \tag{A.2}$$

which is the first application of *Trans*.

$Trans(-?, s, \delta, s') \equiv - \wedge \delta = nil \wedge s' = s$, which can not possibly hold, which makes the second disjunct in (A.2) false. Thus, we need to compute

$$\begin{aligned} &Trans((\omega_1?; a_1| \ldots |\omega_m?; a_m), s, \epsilon, s') \equiv \\ &\quad Trans(\omega_1?; a_1, s, \epsilon, s') \vee \ldots \vee Trans(\omega_m?; a_m, s, \epsilon, s') \end{aligned}$$

for some program ϵ, by definition of *Trans* for nondeterministic choice. For each i we have

$$Trans(\omega_i?; a_i, s, \epsilon, s') \equiv \exists \zeta. Trans(\omega_i?, s, \zeta, s') \wedge \epsilon = \zeta; a_i \tag{A.3}$$

by definition of *Trans* for sequences. Furthermore

$$Trans(\omega_i?, s, \zeta, s') \equiv \omega_i[s] \wedge \zeta = nil \wedge s' = s$$

by definition of *Trans* for conditions. This second application of *Trans* yields that $Trans((\omega_1?; a_1| \ldots |\omega_m?; a_m), s, nil; a_i, s)$ iff $\omega_i[s]$. In the third application of *Trans* we move from the program $nil; a_i$ in situation s to a new program η and a new situation s''.

$$\begin{aligned} &Trans(nil; a_i, s, \eta, s'') \equiv \\ &\qquad Final(nil, s) \wedge Trans(a_i, s, \eta, s'') \\ &\equiv \\ &\qquad Trans(a_i, s, \eta, s'') \end{aligned}$$

which by definition of *Trans* for primitive actions gives

$$\begin{aligned} &Trans(a_i, s, \eta, s'') \equiv \\ &\qquad Poss(a_i, s) \wedge \eta = nil \wedge s'' = do(a_i, s) \end{aligned}$$

By means of *Trans* we went from $\delta_{\mathcal{T}}$ in situation s, to $nil; a_i$ in s iff $\omega_i[s]$, and from that to nil in $do(a_i, s)$. Thus, we have $TransCl(\delta_{\mathcal{T}}, s, \delta, do(a_i, s))$ iff $\omega_i[s]$, for some i, and $\delta = nil; \delta_{\mathcal{T}}$. Since the action nil has no effects whatsoever it is clear that $TransCl(\delta_{\mathcal{T}}, s, \delta_{\mathcal{T}}, do(a_i, s))$ iff $\omega_i[s]$. Moreover, the construction of $\delta_{\mathcal{T}}$ ensures that some $\omega_i[s]$ will hold and since we have a completely specified initial state and all primitive actions are deterministic, $\omega_i[s]$ is unique. From the hypothesis and by transitivity we have that $TransCl(\delta_{\mathcal{T}}, S_0, \delta_{\mathcal{T}}, do(a_i, s))$
By the definition of *Prefix* (definition A.0.2) we know that:

$$Prefix(do(a_i, s), \mathcal{T}) \equiv \mathcal{T}(s) = a_i \wedge Prefix(s, \mathcal{T}),$$

for some i. Now, since $\omega_i[s]$ determines the value of every fluent, and $\mathcal{T}$ is Markovian, $\mathcal{T}(s) = a_i$, which implies $Prefix(do(a_i, s), \mathcal{T})$ (as $Prefix(s, \mathcal{T})$ holds by the hypothesis).□

Theorem A.2.3 (Correctness of $\delta_{\mathcal{T}}$) *Let Δ be a SitCalc theory, γ a property, $\mathcal{T}$ a Markovian trajectory function satisfying condition (A.1), and $\delta_{\mathcal{T}}$ defined as above. Then Δ is stable w.r.t. γ under control of $\delta_{\mathcal{T}}$.*
Proof: We show that for any trajectory function $\mathcal{T}'$ which is a trace of $\delta_{\mathcal{T}}$,

$$\Delta \models \forall s.\, Prefix(s, \mathcal{T}') \wedge \mathcal{T}'(s) = a \supset \mathcal{T}(s) = a$$

Stability then follows from the fact that $\mathcal{T}$ satisfies condition (A.1).

Let s be an arbitrary situation such that $Prefix(s, \mathcal{T}')$ holds and $\mathcal{T}'(s) = a$. Then $do(a, s)$ is also a prefix of $\mathcal{T}'$ by the definition of prefix, and since $\mathcal{T}'$ is a trace of $\delta_{\mathcal{T}}$, there exists, by the definition of program trace, a program δ such that $TransCl(\delta_{\mathcal{T}}, S_0, \delta, do(a, s))$ (and since we know from Lemma A.2.2 that $\delta_{\mathcal{T}}$ never halts, we ignore the second disjunct in the definition of program trace). By Lemma A.2.2, we have $Prefix(do(a, s), t)$ which implies that $\mathcal{T}(s) = a$.□

It should be noted that the Markov property is a *sufficient* condition for translation of a trajectory function into a GOLOG program. It may very well be the case that there are other, less restrictive, conditions that will enable such a translation too.

Appendix B

Handling discrepancies in Pinto's concurrent SitCalc

In this appendix we will show how *surprises* may be handled in SitCalc.

B.1 Surprises in Reiter's SitCalc

With a state-based view on execution monitoring, a natural definition of a discrepancy in SitCalc could be the following:

Definition B.1.1 (Discrepancy (Surprise))
Let Δ be an application axiomatization and $O_{do(A,\mathbf{s})}$ an observation with $\mathbf{s}$ being a ground situation term. If

$$\Delta \not\models O_{do(A,\mathbf{s})}$$

we say that $O_{do(A,\mathbf{s})}$ is a *discrepancy* (or, *surprise*[1]).$\Box$

In Chapter 2 the notion "Strength of Closed World Assumption" of a modeling formalism was discussed. We claimed that SitCalc belonged to the far right end of the scale, since it enforces precise predictions on the values of state variables (or, *fluents* as they will be called in this chapter and the next) over time, where changes are due to control actions. To solve the *frame problem*[2] SitCalc relies on *action-based inertia*, that is, fluents changes exactly when an action makes them change. This is the mechanism that enforces precision. Action-based inertia is a fairly strong assumption and from our perspective, it is too strong. In fact, adding an observation

[1]The terminology of the Reasoning about Action and Change research areas is partial to the term "surprise".

[2]The frame problem is a representational problem for declarative formalisms for knowledge representation. The problem is that it is desirable to only have to explicitly represent things that *change* in dynamic environment, without having to bother with all the things that do *not* change.

to the theory that does not satisfy an expected observation yields an inconsistent theory.

Example B.1.2 As a simple example of this, we can reuse an example that was popular in the 80's: the Stolen Car Scenario [Baker, 1989]. In the scenario I leave my car in a parking lot and return after three days. I expect to find the car where I left it, but instead it is stolen. We can model this scenario in a simple fashion in SitCalc as follows: there is only one action *wait*, and only one fluent *parked*, where $H(parked, s)$ denotes that the car is parked in situation s. the action *wait* does not change the truth value of the fluent. As we assume that it always is possible to wait, *Poss* is always true. We have the following successor state axiom[3]:

$$\begin{aligned}&Poss(a, s) \supset \\ &\quad (H(parked, do(a, s)) \equiv H(parked, s)),\end{aligned}$$

that is, I assume that that nothing may change the fact that the car is parked. We also have the initial situation axiom

$$H(parked, S_0).$$

After three days observe that the car is stolen, that is, that

$$\neg H(parked, do(wait, do(wait, do(wait, S_0))))$$

holds. With the successor state axiom we see that this observation is equivalent with $\neg H(parked, S_0)$ which contradicts the initial situation axiom, thus, the observation is a discrepancy. In the terminology of [Sandewall and Shoham, 1994], Reiter's SitCalc is not *consistency preserving.* The problem from an execution monitoring perspective is that we cannot reason about the discrepancy as it contradicts the application axiomatization.□

In section B.2 we will solve the problem of Example B.1.2 by relaxing ECA to allow for fictitious actions to occur concurrently with the primitive actions. The occurrences of such fictitious actions is then minimized.

Above we showed that the strong inertia assumption made theories with discrepancies inconsistent. In that case we cannot reason about discrepancies, nor distinguish between discrepancies and an inconsistent original application axiomatization.

B.2 Surprises with fictitious actions in Pinto's SitCalc

In this section we will address these problems by introducing *fictitious actions*, that is, two actions for every fluent that change the fluent value to true or false respectively. When we encounter a discrepancy, the theory will no longer be inconsistent, instead it will be possible to show that at least one fictitious action was executed.

[3]The γ-formulas are empty in this example, and since they are disjunctions they have the value **F**.

The most natural way to model this, in our mind, is to allow for fictitious actions to be executed concurrently with the primitive actions. Moreover, since fictitious could explain *any* change we need to ensure that fictitious actions only occur when it is absolutely necessary. Thus, our starting point will be Pinto's concurrent flavor of SitCalc [Pinto, 1998a]. We will also exploit his work on explicit situation preference relations to minimize the occurrences of fictitious actions [Pinto, 1998b].

B.2.1 Language

The concurrent SitCalc (CSitCalc) is a second-order sorted language with sorts $\mathcal{F}$, $\mathcal{A}$, $\mathcal{A}_{fict}$, $\mathcal{C}$, $\mathcal{S}$, and $\mathcal{S}$ for fluents, primitive actions, fictitious actions, concurrent actions, situations and domain objects. There are two fictitious actions, A_F^+ and A_F^-, for every $F \in \mathcal{F}$, and the concurrent actions are sets of primitive and fictitious actions, that is $\mathcal{C} = 2^{\mathcal{A} \cup \mathcal{A}_{fict}}$. Fluents may take arguments from the set of domain objects.

We have the following function and predicate symbols:

> $do : \mathcal{C} \times \mathcal{S} \to \mathcal{S}$. The term $do(c, s)$ denotes situation resulting from the concurrent execution of all actions in c in situation s.
>
> $Poss \subseteq \mathcal{C} \times \mathcal{S}$. The atomic formula $Poss(c, s)$ denotes that it is possible to execute the actions in c concurrently in situation s.
>
> $H \subseteq \mathcal{F} \times \mathcal{S}$. For a fluent F and situation s, $H(F((\vec{d})), s)$ denotes that the fluent $F(\vec{d})$, for a sequence of domain objects $\vec{d} = d_1, \ldots, d_n$, holds in situation s.
>
> $Prim = \mathcal{A}$, $Fict = \mathcal{A}_{fict}$. These predicates are used when we explicitly need to distinguish between the two types of actions.

B.2.2 Basic axiomatization

A number of basic axioms for CSitCalc builds the structure of the models and are very similar to the basic axioms of SitCalc (section 7.2.1). For a complete presentation, see e.g. [Pinto, 1998b]. For concurrent actions, we define $Poss$ as follows[4]:

$$Poss(c, s) \equiv \forall a.\, a \in c \supset Poss(\{a\}, s) \tag{B.1}$$

Axiom (B.1) states that a concurrent action is possible to execute in a situation iff every member of the concurrent action can be executed on its own.

B.2.3 Domain axiomatization

A domain axiomatization is divided into several sets of axioms. We say that a formula is *simple on a situation term s* if it does not mention any situation terms other

[4]Note that free occurrences of variables are assumed to be universally quantified.

than s, and does not quantify over s.

Action Precondition Axioms
The set T_{prec} of action precondition axioms for primitive actions, each on the form:

$$Poss(\{a(\vec{x})\}, s) \supset \Psi_a(s) \tag{B.2}$$

where $\Psi_a(s)$ is a formula simple on s. In principle, it should be possible to restrict the possibility of executing fictitious actions in the same way as for any action, i.e. by defining a non-trivial $\Psi_a(s)$ for fictitious actions a. This could, for example, be used to prefer one fictitious action over another in cases of non-determinism. However, in this thesis make the simplifying assumption that fictitious actions always can be executed. Formally,

$$Fict(a) \supset Poss(\{a\}, s).$$

If we assume that $\Psi_a(s)$ characterizes the set of states in which a is possible to execute, we can apply Clark's completion to all formulas of form (B.2), and thus get a characterization of the predicate $Poss$.

Effect Axioms
The set T_{eff} of effect axioms. For every fluent F and primitive action A, we can have the following two axioms:

$$Poss(c, s) \wedge A \in c \wedge G_F^+(A, s) \supset H(F, do(c, s)) \tag{B.3}$$

$$Poss(c, s) \wedge A \in c \wedge G_F^-(A, s) \supset \neg H(F, do(c, s)) \tag{B.4}$$

Intuitively, formula (B.3) ((B.4)) says that if c is possible to execute in situation s, where the action A is a member of c and the qualification $G_F^+(A, s)$ ($G_F^-(A, s)$) for the effects of actions on F holds, then F will be true (false) after the execution. The close relation between this formulation and Reiter's (from section 7.2.1) should be obvious. The only syntactical difference is that we have substitutes the term equivalence for term inclusion.

By simple logical rewriting all axioms on the form of (B.3) and (B.4) can be compiled to one axiom for positive effects of actions on F and one for all negative effects:

$$Poss(c, s) \wedge \gamma_F^+(c, s) \supset H(F, do(c, s)) \tag{B.5}$$

$$Poss(c, s) \wedge \gamma_F^-(c, s) \supset \neg H(F, do(c, s)) \tag{B.6}$$

If there are no positive (or negative) effect axioms a fluent F, then the formula $\gamma_F^+(c, s)$ (or $\gamma_F^-(c, s)$) is the atom $\mathbf{F}$. Note that γ^* (for $* \in \{+, -\}$) is on the form

$$\bigvee A \in c \wedge G^*(A, s).$$

This implies that if there are no positive or negative qualifications for a fluent, the

corresponding γ is equivalent to $\mathbf{F}$[5], as it can be seen as an empty disjunction.
Successor State Axioms
The set T_{ssa} of formulas. Recall that in SitCalc the effect axioms and ECA, that is that $\gamma_F^+(c,s)$ ($\gamma_F^-(c,s)$) characterizes all positive (negative) change of F, yields the successor state axioms. To accommodate for occurrences of fictitious actions, we need to relax the explanation closure assumption to the following:

Assumption 2 (Relaxed Explanation Closure Assumption (RECA)) The formula $\gamma_F^+(c,s) \wedge A_F^- \notin c$ ($\gamma_F^-(c,s) \wedge A_F^+ \notin c$) *or* the execution of the fictitious action, that is $A_F^+ \in c$ ($A_F^- \in c$) characterizes positive (negative) change of the fluent F.□

RECA and the effect axioms combine to the successor state axioms:

$$\begin{aligned} & Poss(c,s) \supset (H(F, do(c,s)) \equiv \\ & \quad ((\gamma_F^+(c,s) \wedge A_F^- \notin c) \vee A_F^+ \in c) \vee \\ & \quad (H(F,s) \wedge \neg((\gamma_F^-(c,s) \wedge A_F^+ \notin c) \vee A_F^- \in c)) \end{aligned} \tag{B.7}$$

A transformation of Reiter's SitCalc to CSitCalc is clearly straightforward. The other direction is in the general case not possible.
Initial situation axioms
The set T_{S_0}. This is any finite set of sentences that mention only the situation term S_0, or that are situation independent. The initial situation necessarily has to completely specified, i.e. every ground atom is either true or false in S_0.

Thus, an application axiomatization is a set $\Delta = \Sigma \cup T_{prec} \cup T_{ssa} \cup T_{S_0}$, where Σ is the set of background axioms (including unique names axioms that we have omitted in the presentation above), T_{prec} is a set of action precondition axioms, T_{ssa} is a set of successor state axioms, and T_{S_0} is a set of initial situation axioms and all situation independent axioms. We have the following useful lemma:

Lemma B.2.1 Let Δ be an application axiomatization and F a fluent of the theory. Then

$$\Delta \models Poss(c,s) \supset \neg(\gamma_F^+(c,s) \wedge \gamma_F^-(c,s))$$

Proof: An immediate consequence of the construction of the γs, see formulas (B.5) and (B.6).□

We define an *observation in situation s*, O_s, completely analogous to before.

Example B.2.2 (Stolen Car Scenario Cont'd)
We revisit the Stolen Car Scenario, this time with the successor state axiom:

$$\begin{aligned} & Poss(c,s) \supset (H(parked, do(c,s)) \equiv \\ & \quad A_{parked}^+ \in c \vee \\ & \qquad H(parked,s) \wedge A_{parked}^- \notin c), \end{aligned}$$

[5]That is, the truth value *false*. The truth value *true* is denoted $\mathbf{T}$.

Again we have the initial situation axiom

$$H(parked, S_0),$$

and the observation

$$\begin{aligned}&\neg H(parked, do(\mathbf{c}'', do(\mathbf{c}', do(\mathbf{c}, S_0)))) \wedge\\ &wait \in \mathbf{c} \wedge wait \in \mathbf{c}' \wedge wait \in \mathbf{c}''.\end{aligned}$$

By applying the successor state axiom (or *performing goal regression* in Reiter's terminology) three times to the observation, we get the following equivalent formula:

$$\begin{aligned}&(A^+_{parked} \notin \mathbf{c}'' \wedge\\ &\quad((A^+_{parked} \notin \mathbf{c}' \wedge\\ &\qquad((A^+_{parked} \notin \mathbf{c} \wedge\\ &\qquad(\neg H(parked, S_0) \vee A^-_{parked} \in \mathbf{c})) \vee\\ &\quad A^-_{parked} \in \mathbf{c}')) \vee\\ &A^-_{parked} \in \mathbf{c}'')) \wedge\\ &wait \in \mathbf{c} \wedge wait \in \mathbf{c}' \wedge wait \in \mathbf{c}''\end{aligned}$$

The conjunction of the first conjunct of this formula and the initial situation axiom yields the following formula on conjunctive normal form:

$$\begin{aligned}&(A^-_{parked} \in \mathbf{c} \vee A^-_{parked} \in \mathbf{c}' \vee A^-_{parked} \in \mathbf{c}'') \wedge\\ &(A^+_{parked} \notin \mathbf{c}' \vee A^-_{parked} \in \mathbf{c}'') \wedge\\ &A^+_{parked} \notin \mathbf{c}''\end{aligned} \tag{B.8}$$

This formula is clearly satisfiable and we have therefore hedged the inconsistencies.□

However, there are a number of unintuitive models of the theory in example B.1.2. For example, in one model the fictitious action A^-_{parked} is a member of $\mathbf{c}$, $\mathbf{c}'$, and $\mathbf{c}''$. In another model both A^-_{parked} and A^+_{parked} are members of $\mathbf{c}$. To handle this we will need to define a preference policy on situations to filter out unwanted models. In the next section we will formally define the notion "surprise" and show that RECA hedges inconsistencies in the general case.

B.2.4 Reasoning about surprises

We again assume that an observation $O_{do(\mathbf{c},\mathbf{s})}$ represents some sensor inputs. A surprise is then an observation that is not entailed by the SitCalc theory if *only primitive actions have been executed.* Formally,

Definition B.2.3 (Surprise)
Let Δ be an application axiomatization and $O_{do(\mathbf{c}_{n+1},\mathbf{s})}$ and observation with $\mathbf{s} = do(\mathbf{c}_n, do(\mathbf{c}_{n-1}, \ldots do(\mathbf{c}_2, do(\mathbf{c}_1, S_0)) \ldots))$. If

$$\Delta \not\models \bigwedge O_{do(\mathbf{c}_{n+1},\mathbf{s})} \wedge \bigwedge_{i=1}^{n+1} \forall a.\, a \in \mathbf{c}_i \supset \neg Fict(a)$$

then we say that $O_{do(\mathbf{c}_{n+1},\mathbf{s})}$ is a *surprise*.$\Box$

The reasoning task can, thus, be described as, for a system

$$\langle \Delta, \mathbf{s}, O \rangle,$$

where Δ is a CSitCalc theory, $\mathbf{s}$ is a situation, and O an observation, to find an "adequate" set of fictitious actions c, such that

$$\Delta \models \bigwedge O_{do(\mathbf{c}_{n+1},\mathbf{s})}.$$

We will formally define the adequacy notion below.

We start by showing that our CSitCalc theory does handle surprises in the general case.

Proposition B.2.4 Let Δ be an application axiomatization, and assume that $O_{do(\mathbf{c},\mathbf{s})}$ is a surprise. Then,

$$\Delta \models \bigwedge O_{do(\mathbf{c},\mathbf{s})}.$$

Proof: We show that Δ entails $\bigwedge O_{do(\mathbf{c},\mathbf{s})}$ where all fictitious actions that occur in the situation $do(\mathbf{c}, \mathbf{s})$, are members of $\mathbf{c}$. That is, no fictitious actions have occurred in $\mathbf{s}$.

Since $O_{do(\mathbf{c},\mathbf{s})}$ is a surprise, there must be at least one fluent, F, too much or too little in $O_{do(\mathbf{c},\mathbf{s})}$. There are four cases to consider for every such F; When F should have changed from true to false (C1) or from false to true (C2) due to $\mathbf{c}$, but did not, and when F should have remained true (S1) or false (S2) but did not, that is, it should not have been affected by the execution of A. We will only show the proposition for cases C1 and S1. The other two cases can be proven analogously.For readability we omit the object arguments of the actions and fluents.

C1: Failed change from true to false: We assume that $H(F, \mathbf{s})$ and $Poss(\mathbf{c}, \mathbf{s})$ holds and that the execution of the primitive actions in $\mathbf{c}$ in $\mathbf{s}$ would make $\neg H(F, do(\mathbf{c}, \mathbf{s}))$ hold, that is that $\gamma_F^-(\mathbf{c}, \mathbf{s})$ holds. Moreover, we assume that $H(F, do(\mathbf{c}, \mathbf{s})$ holds, contrary to our expectations. Thus, we are interested in the formula

$$\begin{aligned}&((\gamma_F^+(\mathbf{c}, \mathbf{s}) \wedge A_F^- \notin \mathbf{c}) \vee A_F^+ \in \mathbf{c}) \vee \\ &(H(F, \mathbf{s}) \wedge \neg((\gamma_F^-(\mathbf{c}, \mathbf{s}) \wedge A_F^+ \notin \mathbf{c}) \vee A_F^- \in \mathbf{c}))\end{aligned}$$

From Lemma B.2.1 $\gamma_F^+(\mathbf{c}, \mathbf{s})$ cannot hold. This together with the other assumptions above yields

$$A_F^+ \in \mathbf{c} \vee (A_F^+ \in \mathbf{c} \wedge A_F^- \notin \mathbf{c}),$$

which is equivalent to $A_F^+ \in \mathbf{c} \wedge A_F^- \notin \mathbf{c}$.

S1: Failed persistence of a true fluent: We assume that $H(F, \mathbf{s})$ and $Poss(\mathbf{c}, \mathbf{s})$ holds and that the execution of the primitive actions in $\mathbf{c}$ in $\mathbf{s}$ would make $H(F, do(\mathbf{c}, \mathbf{s}))$ hold, that is that $\gamma_F^+(\mathbf{c}, \mathbf{s})$ holds. Moreover, we assume that $\neg H(F, do(\mathbf{c}, \mathbf{s})$ holds, contrary to our expectations. Thus, we are interested in the formula

$$\begin{gathered}((\neg\gamma_F^+(\mathbf{c}, \mathbf{s}) \vee A_F^- \in \mathbf{c}) \wedge A_F^+ \notin \mathbf{c}) \wedge \\ (\neg H(F, \mathbf{s}) \vee ((\gamma_F^-(\mathbf{c}, \mathbf{s}) \wedge A_F^+ \notin \mathbf{c}) \vee A_F^- \in \mathbf{c}))\end{gathered}$$

From Lemma B.2.1 $\gamma_F^-(\mathbf{c}, \mathbf{s})$ cannot hold. This together with the other assumptions above yields

$$(A_F^- \in \mathbf{c} \wedge A_F^+ \notin \mathbf{c}) \vee A_F^- \in \mathbf{c},$$

which is equivalent to $A_F^- \in \mathbf{c} \wedge A_F^+ \notin \mathbf{c}$.

□

By adding the fictitious actions in this way we have allowed for many models that are unwanted. Especially, when no surprises exist it is desirable that we can prove that only primitive actions are executed, since we then have exactly the type of theory utilized by a GOLOG interpreter. To achieve this we propose that the occurrences of fictitious actions are minimized, that is, that we try to explain *any* change with as few fictitious actions as possible.

Following [Pinto, 1998b] we introduce two predicates *proper* and *legal* on situations, for technical reasons: Basically, actions that are not possible do not lead to a proper situation. Formally,

$$proper(s) \supset (s = S_0 \vee \forall c, s'. do(a + c, s') \sqsubseteq s \supset Poss(c, s')).$$

A legal situation is a situation such that there exists a proper continuation of the history.[6] Formally,

$$legal(s) \equiv \exists s'. s \sqsubseteq s' \wedge proper(s'). \tag{B.9}$$

Now, we can define a preference relation on situations:

[6]Note that Pinto's legality predicate is a specialization of the Aliveness predicate in definition A.0.3.

Definition B.2.5 (Fictitious-Action Preferred Situation)
We define a preference relation, $\trianglelefteq$, on situations, as follows

$$
\begin{aligned}
& legal(s_1) \wedge legal(s_2) \supset s_1 \trianglelefteq s_2 \equiv \\
& \quad s_1 = s_2 = S_0 \vee \\
& \quad \exists s_3, s_4.\, s_1 = do(c, s_3) \wedge s_2 = do(c', s_4) \wedge \\
& \quad (H(f, s_1) \equiv H(f, s_2)) \wedge (\forall a.\, Fict(a) \wedge a \in c \supset a \in c') \wedge && \text{(B.10)} \\
& \quad s_3 \trianglelefteq s_4 && \text{(B.11)}
\end{aligned}
$$

If $s \trianglelefteq s'$ we say that s *is preferred over* s'.□

Basically, this preference relation compares situations that reach equivalent states.

Definition B.2.6 (Minimal situation)
A situation s is minimal if no situation exists that is preferred over s and that is different from s. Formally,

$$minimal(s) \equiv legal(s) \wedge \forall s'.\, legal(s') \wedge s' \trianglelefteq s \supset s = s'. \tag{B.12}$$

□

Let T_m be the set with the formulas (B.9) , (B.11) and (B.12).

We redefine the reasoning task to incorporate the minimality criterion as follows:

$$\Delta \cup T_m \models \bigwedge O_{do(\mathbf{c},\mathbf{s})} \wedge minimal(do(\mathbf{c}, \mathbf{s})).$$

Example B.2.7 (Stolen Car Scenario Cont'd)
We denote the four situation $\mathbf{s}_0$, $\mathbf{s}_1 = do(\mathbf{c}, \mathbf{s}_0)$, $\mathbf{s}_2 = do(\mathbf{c}', \mathbf{s}_1)$, and $\mathbf{s}_3 = do(\mathbf{c}'', \mathbf{s}_2)$, respectively. We know that $H(parked, \mathbf{s}_0)$ and we progress to find the minimal situations. In $\mathbf{s}_1$ we have two cases

1. $H(parked, do(\mathbf{c}, \mathbf{s}_0))$: The minimal choice of $\mathbf{c}$ is that no fictitious actions belong to $\mathbf{c}$. In the next step we get
   - **(a)** $H(parked, do(\mathbf{c}', do(\mathbf{c}, \mathbf{s}_0)))$: The minimal choice of $\mathbf{c}'$ is again that no fictitious actions have occurred. Finally, since we know that the literal $\neg H(parked, do(\mathbf{c}'', \mathbf{s}_2))$ holds we must have $A^-_{parked} \in \mathbf{c}''$. In this situation trajectory a fictitious action only occurred in the last situation.
   - **(b)** $\neg H(parked, do(\mathbf{c}', do(\mathbf{c}, \mathbf{s}_0))))$: The minimal choice of $\mathbf{c}'$ is here that the membership $A^-_{parked} \in \mathbf{c}'$ holds. In the next step the minimal choice is that no fictitious action belongs to $\mathbf{c}''$. In this situation trajectory the only occurrence of a fictitious action is in $\mathbf{c}'$.
2. $\neg H(parked, do(\mathbf{c}, \mathbf{s}_0))$. Clearly A^-_{parked} is the only member of $\mathbf{c}$. In the next step we get:
   - **(a)** $H(parked, do(\mathbf{c}', do(\mathbf{c}, \mathbf{s}_0)))$: If A^+_{parked} is member of $\mathbf{c}'$ we do not have a minimal situation, since $H(parked, \mathbf{s}_2)$ holds by case (1a) above.

(b) $\neg H(parked, do(\mathbf{c}', do(\mathbf{c}, \mathrm{s}_0))))$: The minimal choice of $\mathbf{c}'$ is that no fictitious actions have occurred, and as above, no fictitious action belongs to $\mathbf{c}''$. In this situation trajectory the only occurrence of fictitious actions is in $\mathbf{c}$.

Thus, in each of the three minimal situations the fictitious action A^-_{parked} occurs exactly once, and A^+_{parked} does not occur at all.$\Box$