Carnegie Mellon University

CARNEGIE INSTITUTE OF TECHNOLOGY

THESIS

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF Doctor of Philosophy

TITLE Discrete Dynamics in Chemical Process Control and Automation

PRESENTED BY Blake Rawlings

ACCEPTED BY THE DEPARTMENT OF

Chemical Engineering

B. ERIK YDSTIE	4/25/16
B. ERIK YDSTIE, ADVISOR	DATE
L. BIEGLER	4/25/16
LORENZ BIEGLER, DEPARTMENT HEAD	DATE

APPROVED BY THE COLLEGE COUNCIL

VIJAYAKUMAR BHAGAVATULA	4/25/16
DEAN	DATE

Discrete Dynamics in Chemical Process Control and Automation

Submitted in partial fulfillment of the requirements for

the degree of

Doctor of Philosophy

in

Chemical Engineering

Blake C. Rawlings
B.S., Chemical Engineering, The University of Texas at Austin

Carnegie Mellon University
Pittsburgh, PA

May, 2016

Acknowledgements

First, I would like to thank my advisor, Prof. Erik Ydstie, for making this project possible, and for making sure that it was interesting both intellectually and practically. I recall many stimulating conversations with Erik that guided my thoughts on a variety of topics, not limited to research. In addition, I appreciate Erik's willingness to work in an area that has largely been neglected by the academic community in chemical engineering, despite its industrial significance. I would also like to thank the members of my doctoral committee, Prof. Ignacio Grossmann, Prof. John Kitchin, Prof. Bruce Krogh, and Dr. John Wassick, for their feedback and guidance.

Funding for the project was provided by The Dow Chemical Company. Early in the project, Ben Christenson was very helpful in defining a specific problem on the industrial side to motivate the academic research. Later in the project, Joe Bucci was very helpful in applying the results of the research to actual control systems. Throughout the course of the project, John Wassick provided the oversight and coordination to make it successful within Dow.

I would also like to thank my friends in Pittsburgh for making my time here enjoyable. In particular, the grad student softball league in the summers and pick-up soccer on Sundays gave me something to look forward to when research was not enough.

Finally, I would like to thank my family for their continued support and encouragement. Nothing I've done would have been possible without them. Most importantly, thank you Soraya.

Abstract

Formal verification has previously been applied to chemical plant control and automation systems to ensure that they operate as intended. This dissertation examines the related objective of proving that a particular control system does not operate as intended. To this end, we present a set of specifications that address certain aspects of the correct operation of a general control system. Some of those specifications, which relate to invariance and reachability of states that satisfy given logical constraints, do not fall within the classes of specifications that have been addressed in previous work related to the falsification of hybrid systems. For a specification from this class, a sound falsification algorithm is presented which can guarantee that a hybrid system does not meet the specification. The algorithm involves abstraction, as a finite-state discrete system, of the infinite-state hybrid dynamical system that arises when discrete control is applied to a continuous process. The falsification result relies on new results that we present which concern the supervisory control of discrete event systems subject to specifications that involve multiple reachability requirements. The methods we present are applied to two industrial case studies, which were provided by The Dow Chemical Company.

We also present two software tools which apply the methods that we have developed. The first tool, SynthSMV, is an extension of the model checking solver NuSMV that can solve some supervisory control problems. NuSMV was chosen as the basis for our work in falsification because previous work has shown that its symbolic model checking algorithms can handle models of industrial-scale control systems in the context of verification. The second tool, st2smv, translates industrial control code to a formal model that can be solved using SynthSMV. The approach is similar to what has been done in previous work that focused on model checking and verification, with some extensions to enable the application of our work concerning supervisory control and falsification.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Control and Automation

Operation of a modern chemical plant involves a computer control system that performs control and automation tasks. The control system consists of all the logic that is required to operate the plant from startup, through operation, to shutdown. It also serves as the interface between the process and the operators, who monitor and guide the behavior of the plant to carry out tasks that cannot be automated. With modern control and automation systems, operators can complete complex tasks by providing relatively simple inputs, in a way that would otherwise be impossible. The control system then becomes a mission-critical component of the overall plant behavior, and if it fails, or its limitations are not respected, then the system will fail [Ste03]. A similar issue is discussed in a recent perspective, which advocates the viewpoint that the various elements that make up the overall system in a chemical plant are inextricably linked [LS13]. In light of these observations, any complete analysis of the operation of a chemical plant must address the interaction of the control system, the physical process, and the operators.

The physical phenomena that drive chemical processes produce systems in which the variables evolve continuously as time passes. The implementation of a control system using digital computers produces a system in which the variables change instantaneously at discrete moments in time. The closed-loop system produced by applying discrete control to a continuous process is a hybrid (continuous/discrete) dynamical system, or hybrid system for short. Analysis of the overall behavior of a chemical plant, therefore, requires the analysis of the resulting hybrid system [Eng+00].

There are many (often competing) notions of correct operation of a chemical plant, ranging from operational constraints on the acceptable ranges of variables, to economic optimality conditions, and beyond. Any statement about the required, desired, intended, or expected behavior of the plant, including the process and the control system, can be seen as a (partial) specification of the overall correct behavior. From this point of view, if the system meets the specification, then it is (at least partially) correct, and if it does not meet the specification, then it is (again, at least partially) incorrect.

The continuous process dynamics in a chemical plant are essentially never known exactly; there is always uncertainty in the model or unmodeled disturbance. The discrete dynamics of the controller, on the other hand, are known exactly; it does what it was programmed to do. Even so, the complexity of a plant-wide control system, and the fact that its behavior is determined in part by its interaction with the (uncertain) process dynamics, makes the task of analyzing these known discrete dynamics difficult. This is an important problem when it comes to determining whether or not a given chemical plant will operate correctly.

1.2 Design

The obvious goal, given a model of the process dynamics and a specification of the desired closed-loop plant behavior, is to design a control system that enforces the specification. The development of algorithms to produce such correct-by-design control systems is an important research objective [GW00].

Unfortunately, the scale of a chemical plant (let alone an integrated chemical processing site) is beyond the reach of existing hybrid control design approaches [Eng+00]. Furthermore, it is not even clear how to specify everything that the system should (and should not) do, which is a prerequisite for designing the control logic. A control system that satisfies a partial specification of the overall desired plant behavior may not meet additional requirements that were not included in the partial specification.

1.3 Analysis

Because the full hybrid control system design problem is currently intractable for a chemical processing plant, much of the previous research in this area has focused on the associated analysis problem. In this problem, given an existing control system and a (partial) specification of the desired closed-loop behavior, the objective is to determine whether or not the system meets the specification. The analysis problem can be divided further into verification and falsification. In verification, the goal is to prove that the system meets the specification. Conversely, in falsification, the goal is to prove that the system does not meet the specification.

The analysis of hybrid systems amounts to solving the hybrid systems reachability problem. Given an initial state, a target state, and a model of the hybrid dynamics, the problem is to determine whether or not the

system can reach the target state from the initial state. In general, the hybrid systems reachability problem is undecidable [Hen+98]. For certain classes of systems with restrictions placed on the form of the continuous dynamics, it is possible to either compute or approximate the set of reachable states. Some of these classes of hybrid systems (in order of increasing generality) are timed automata [LPY97; Yov97], linear hybrid automata[HHW97; Fre05], and piecewise-affine hybrid automata [Fre+11]; each class severely limits the direct application of the corresponding method to real systems. For systems with unrestricted continuous dynamics, the reachable set is approximated conservatively by solving a more restricted problem exactly [CK03; Col11]. These reachability approximations can be expensive to compute, so they are typically coupled with abstraction-based techniques. This gives rise to methods such as counterexample-guided abstraction refinement [Cla+03], in which a finite approximation of the hybrid system is iteratively refined to avoid unnecessary reachability approximations. For systems with complex discrete dynamics, such as plant control and automation systems, it is particularly important to only consider hybrid reachability problems when the discrete options have been exhausted [Seg07].

Much of the previous work in analyzing hybrid control systems in chemical plants has focused on verification [DSP97; Dim+96; Sri+98; Kow+99; KSB01; Bal+05]. The main issue with this approach is that currently it is not reasonable to require that every control system be verifiably correct, precisely because the design problem is intractable. Instead, systems are designed with the best available methods, including simulation and best practices, and put into operation. Thus, the implicit assumption is already that the system is correct, and verifying this correctness does not have any practical impact. Actionable results instead come from falsifying a specification; if the system is proven to

not meet a given specification, then the control system should be modified. This is not a formal design algorithm, but is a step toward designing correct systems; first, any behavior that can be considered incorrect should be removed from existing systems.

Approaching the analysis problem in terms of falsification, instead of verification, necessitates identifying which classes of systems and specifications can be addressed algorithmically, and how those algorithms can be applied to large-scale systems like chemical plants. At a high level, falsification can be viewed as the opposite of verification. In this way, given a system and a specification that can be verified for that system, the negated form of the specification can be falsified. For obvious reasons, previous research in verification of hybrid systems focuses on classes of specifications of the desired system behavior that can be verified algorithmically. It is not necessarily true that those classes of specifications can be falsified efficiently. Therefore, at a lower level, it is important to treat falsification not only as the opposite of verification, but as a different technique, which applies to different types of specifications. Exploring the class of specifications that can be falsified can lead to specifications which address different aspects of plant operation than in previous work.

1.4 Outline

Chapter 2 introduces a model of the dynamics of a chemical process and its control system. The model is intended to be general enough to encompass the wide range of physical phenomena, and the common implementations of control systems, that appear in the chemical processing industry. At the same time, it should be specialized enough that there is an obvious correspondence

between the model and the various components of the process and control system. The model is presented in the form of a hybrid dynamical system, and an example is used to demonstrate the modeling procedure. The model presented in Chapter 2 is the target of the analysis presented in the later chapters.

Chapter 3 presents results concerning supervisory control of discrete dynamical systems. The results relate to computing optimal supervisory control policies to enforce a class of specifications that includes multiple reachability requirements. Similar results have recently been reported in the literature for a single reachability requirement. Our results expand the previous results to multiple reachability requirements, which have also appeared in the literature in the context of supervisory control theory as multitasking supervisory control. We also show that the optimal control policy can be computed by solving a sequence of symbolic model checking problems. Chapter 4 describes the implementation of the methods presented in Chapter 3 in SynthSMV, which is an extension of the symbolic model checking solver NuSMV.

Chapter 5 describes a method for falsifying a class of specifications in hybrid systems. The systems of interest are represented by the model from Chapter 2, and the class of specifications is the same as in Chapter 3. An algorithm is presented and applied to a series of examples, including the example system from Chapter 2. The algorithm relies on the results from Chapter 3 to guarantee that the falsification results are sound.

Chapter 6 demonstrates how the results developed in the earlier chapters apply to chemical plant control and automation systems. This involves modeling the system as in Chapter 2, creating specifications of the desired system behavior that fit in the class of specifications from Chapter 3, and attempting to falsify those specifications by applying the algorithm from Chapter 5. In

addition, the application of existing verification methods is explored in the context of falsification. A series of illustrative examples is presented, followed by computational results from an industrial case study which was provided by The Dow Chemical Company. Chapter 7 gives an overview of st2smv, which is a tool to apply the methods from Chapter 6 to control systems written in the Structured Text programming language for programmable logic controllers.

Chapter 2

Modeling a Plant: Process, Control, and Automation

2.1 Comparison of Modeling Frameworks

Many frameworks exist that can be used to model hybrid dynamical systems, which combine continuous and discrete dynamics. Two notable examples are mixed logical dynamical (MLD) systems [BM99] and hybrid automata [Hen00].

MLD systems (and the equivalent classes of systems covered in [HDB01]) model discrete-time systems that involve linear dynamics and logical constraints. This modeling framework is notable because it has been successfully applied to a range of problems in the chemical processing industry, including verification [BTM01], control [DSL07], and supply chain optimization [MTA06]. In general, however, MLD systems do not capture the classes of systems that arise in the chemical processing industry. These systems usually involve continuous-time nonlinear dynamics, which are only approximated by MLD systems.

Hybrid automata provide a very general modeling framework for hybrid

systems. Informally, a hybrid automaton consists of a finite set of continuous variables and a finite set of logical modes, each of which describes the continuous dynamics of the system when it is in that mode, the conditions under which the system switches to a different mode, and the effect that switch has on the variables. One shortcoming of hybrid automata is that the logical modes are defined explicitly [BL02]. In a plant control system, the discrete mode is defined by the value assigned to each of a set of discrete variables. The discrete variables represent conditions such as whether or not a particular alarm is turned on in the control room, or which recipe is being executed in a reactor. The number of logical modes in the hybrid automaton model of such a system grows exponentially with the number of discrete variables, which is often on the order of hundreds to thousands in a chemical plant [Eng+00]; this makes it impractical to model the hybrid systems that arise in chemical plants as hybrid automata.

The modeling framework described in Section A.1 has rich enough descriptive capabilities to subsume MLD systems and hybrid automata [GST12]. In addition, it does not require the explicit enumeration of logical modes as in the case of hybrid automata, instead allowing the discrete state to be defined by a set of discrete state variables. For these reasons, we use systems that have the form (A.1) to model the dynamics of chemical plants.

In this chapter, we develop a model in the framework of (A.1) that is general enough to capture a large subset of the dynamical systems that arise in the control and automation of chemical plants. The model allows for nonlinear (and possibly uncertain) continuous process dynamics, and sample-and-hold control systems with both continuous and discrete variables.

2.2 Sample-and-Hold Control Systems

In a sample-and-hold control system (SHCS), a controller repeatedly (usually at a fixed frequency) performs the following steps:

1. The state of the plant is sampled.
2. Continuous and discrete control inputs are calculated.
3. The new inputs are applied to the plant.
4. The system evolves according to the process dynamics until the next sample.

In step 2, the continuous control inputs include values like valve positions, and discrete control inputs are logical values such as whether or not to activate a particular piece of equipment.

The system produced by an SHCS controlling a continuous chemical process is a hybrid dynamical system that can be modeled in the framework (A.1) as:

$$
\begin{aligned}
x &= \begin{pmatrix} z \\ u \\ s \\ \tau \end{pmatrix} \in \mathbb{R}^{n_z} \times \mathbb{R}^{n_u} \times \{0,1\}^{n_s} \times [0,T] =: X \\
F(x) &= \begin{pmatrix} F_z(z,u) \\ \mathbf{0} \\ \mathbf{0} \\ 1 \end{pmatrix} \\
G(x) &= \begin{pmatrix} \left\{ \begin{pmatrix} u^+ \\ s^+ \end{pmatrix} \;\middle|\; \exists r \in \rho(z) : \begin{matrix} z \\ u^+ \in G_u(z, s^+) \\ s^+ = g_s(r,s) \\ 0 \end{matrix} \right\} \end{pmatrix} \\
D &= \{x \in X \mid \tau = T\} \\
C &= X \setminus D
\end{aligned}
\tag{2.1}
$$

where:

- z is a vector of continuous process state variables.
- u is a vector of continuous control inputs.
- s is a vector of discrete variables that describe the logical state of the system.
- τ is a timer variable that tracks the amount of time that has passed since the previous sample was taken.
- $F_z : \mathbb{R}^{n_z} \times \mathbb{R}^{n_u} \rightrightarrows \mathbb{R}^{n_z}$ represents the process dynamics.
- $\dot{u}$ and $\dot{s}$ are both $\mathbf{0}$ because the control variables only change value in discrete jumps when samples are taken.
- $z^+ = z$ because the plant is continuous.
- $G_u : \mathbb{R}^{n_z} \times \{0,1\}^{n_s} \rightrightarrows \mathbb{R}^{n_u}$ is the feedback law, which may depend on the logical state of the system.
- $g_s : \{0,1\}^{n_r} \times \{0,1\}^{n_s} \to \{0,1\}^{n_s}$ is the discrete automation logic.
- $\rho : \mathbb{R}^{n_z} \rightrightarrows \{0,1\}^{n_r}$ returns discrete readings from the plant.
- $\dot{\tau} = 1$ and $\tau^+ = 0$ cause samples to occur every T time units.
- T is the sample time.

The elements in (2.1) fit into the layout of a plant as shown in Figure 2.1.

The set of possible initial states is restricted by $X_0 \subseteq \{x \in X \mid \tau = 0\}$. Because the underlying process is continuous, the jump set D is the set of points where the process state is sampled and the control inputs are updated (when $\tau = T$). The flow set C is the remainder of the state space. When

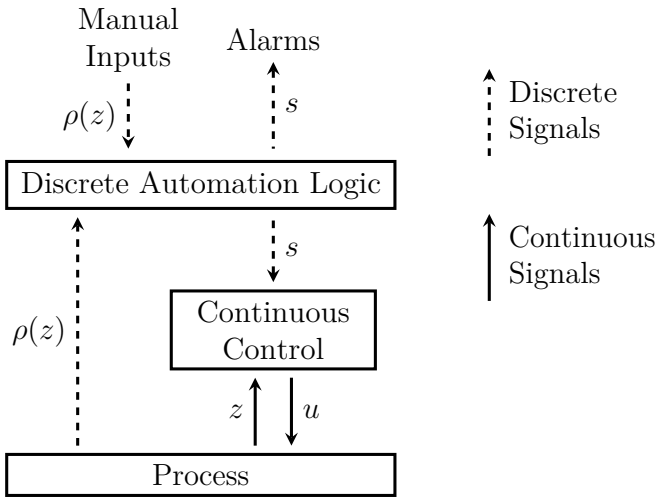

Figure 2.1: The structure of a chemical plant control and automation system.

$x \in C$, the continuous state of the plant evolves subject to $\dot{z} \in F_z(z, u)$, the timer increases subject to $\dot{\tau} = 1$, and u and s are held constant. When $x \in D$, the logical state of the control system is updated to s^+, new control inputs u^+ are computed, and the timer is reset to 0. The actions of sampling the plant, computing s^+ and u^+, applying the new values, and resetting the timer are modeled as instantaneous events, so the fact that the plant is continuous means that $z^+ = z$.

Modeling the process dynamics as a differential inclusion rather than a differential equation allows for uncertainty in the dynamics. If the process dynamics are known exactly, then the relationship reduces to the differential equation $\dot{z} = f_z(z, u)$. Similarly, modeling the continuous feedback law as a difference inclusion allows for uncertainty or indifference concerning the actual values computed by the controller. We do assume that the discrete logic is known exactly, so g_s is a function, not a set-valued map. The discrete jumps in the logical state of the plant, s, are still governed by a difference inclusion (not a difference equation), however, because there may be external inputs to the control logic that do not depend on the state. This behavior is contained in ρ.

For example, an operator may send a particular discrete signal (by pressing a button in the control room) at any time, regardless of the state of the plant, and the control system reacts accordingly.

Without loss of generality, we assume that all discrete variables are binary. To simplify notation involving discrete variables, we use "0" and "1" to represent the integer values 0 and 1 as well as the Boolean constants $false$ and $true$, respectively. That is, for $s \in \{0,1\}^2$, we treat the expressions $s_1 + s_2 \geq 1$ and $s_1 \vee s_2$ as being equivalent.

For set-valued maps and functions, we use uppercase letters to denote set-valued maps, and lowercase letters to denote functions. In the definition of a set-valued map, we will omit the bracket notation around sets that consist of a single element, i.e., the z that appears in the definition of $G(x)$ in (2.1) is the set $\{z\}$.

2.3 Example: Liquid Holding Tank

To demonstrate the model developed in this chapter, we now present an example of a simple process and control system. In the example, the process is a liquid holding tank, with inlet and outlet flows determined by upstream and downstream requirements. The tank is shown in Figure 2.2.

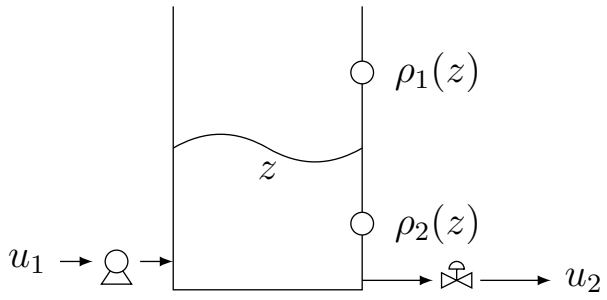

Figure 2.2: A liquid holding tank with high- and low-level indicators.

The state and dynamics are modeled by the system:

$$
\begin{aligned}
x &= (z, u_1, u_2, s_1, s_2, s_3, s_4, s_5, s_6, \tau)^{\mathsf{T}} \in \mathbb{R} \times \mathbb{R}^2 \times \{0,1\}^6 \times [0,1] =: X \\
f(x) &= \begin{pmatrix} u_1 - u_2 \\ \mathbf{0} \\ \mathbf{0} \\ 1 \end{pmatrix} \\
G(x) &= \left(\left\{ \begin{pmatrix} u^+ \\ s^+ \end{pmatrix} \;\middle|\; \exists r \in \rho(z) : \begin{array}{c} z \\ {u_1}^+ \in [0, 0.12(1-{s_1}^+)(1-{s_3}^+)] \\ {u_2}^+ \in [0, 0.10(1-{s_2}^+)(1-{s_4}^+)] \\ {s_1}^+ = \neg s_5 \wedge r_1 \\ {s_2}^+ = \neg s_6 \wedge r_2 \\ {s_3}^+ = {s_1}^+ \vee (s_3 \wedge r_1) \\ {s_4}^+ = {s_2}^+ \vee (s_4 \wedge r_2) \\ {s_5}^+ = ({s_1}^+ \wedge r_3) \vee s_5 \\ {s_6}^+ = ({s_2}^+ \wedge r_4) \vee (s_6 \wedge r_2) \\ 0 \end{array} \right\} \right) \\
\rho(z) &= \begin{pmatrix} z > 8 \\ z < 2 \\ \{0,1\} \\ \{0,1\} \end{pmatrix} \\
D &= \{x \in X \mid \tau = 1\} \\
C &= X \setminus D
\end{aligned}
$$

with the variables and parameters listed in Table 2.1. The initial state is $x_0 = \left(5, \mathbf{0}^{\mathsf{T}}, \mathbf{0}^{\mathsf{T}}, 0\right)^{\mathsf{T}}$. Note that even in this small example, the 6 discrete state variables would produce 64 discrete locations in a hybrid automaton model.

The minimum and maximum liquid levels that the tank can accommodate are 0 and 10, respectively; if the level reaches 0, then underflow occurs, and if it reaches 10, then overflow occurs. The flow rates into and out of the tank (inputs u_1 and u_2) are dictated by (unmodeled) upstream and downstream requirements, so $u^+ \in G_u$ is left as a difference inclusion. In order to prevent overflow or underflow, the controller includes alarms and logic to set the appropriate input to 0 in response to high or low level measurements. The

Table 2.1: Variables in the liquid holding tank example.

Variable	Description
z	Liquid level
u_1	Inlet flow rate
u_2	Outlet flow rate
s_1	High-level alarm
s_2	Low-level alarm
s_3	Inlet flow lock
s_4	Outlet flow lock
s_5	High-level acknowledgement
s_6	Low-level acknowledgement
ρ_1	High-level indicator
ρ_2	Low-level indicator
ρ_3	Operator: acknowledge s_1
ρ_4	Operator: acknowledge s_2

objective is that the tank can be filled and emptied as necessary, subject to the requirement that the liquid level stay within the acceptable operating range.

A simulation of the system dynamics is shown in Figure 2.3. In the simulation, the tank overflows before $t = 700$. While the simulation does indicate that the control system contains a flaw, it does not indicate what the flaw is. Furthermore, there is no guarantee that simulating a finite number of different trajectories would have uncovered this behavior. Analysis of the system dynamics is required to determine what led to overflow.

2.4 Summary

In this chapter, we have introduced a model that captures the dynamics of a continuous process interacting with a sample-and-hold control system that includes discrete logic. The modeling framework, taken from [GST12], avoids some of the shortcomings of other popular approaches, including MLD systems and hybrid automata, while still allowing for a compact representation of the dynamics. An example was included to demonstrate the class of systems that

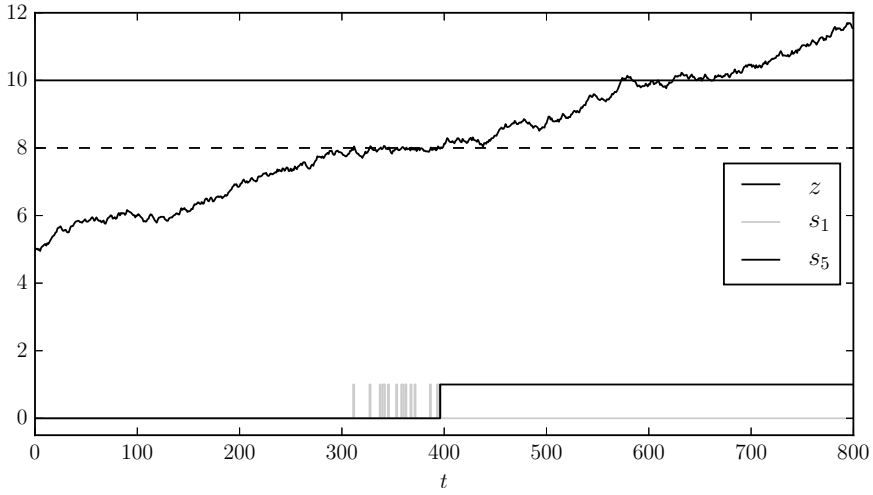

Figure 2.3: A simulation of the liquid holding tank, with overflow.

can be modeled as (2.1), and to motivate the formal analysis of such systems to uncover flaws in the control logic.

Chapter 3

Supervisory Control with Multiple Reachability Specifications

3.1 Control of Discrete Systems

The classical development of supervisory control theory for discrete event systems is based on automata and formal languages, as described in Section A.5. In particular, the system is modeled as a deterministic finite automaton, a superset of the desired system behavior is specified as a formal language, and a single set of states (the so-called marked states) is designated that should always remain reachable. One issue with this approach is that it is often more convenient to specify the behavior of a system using temporal logic such as computation tree logic (CTL), which is described in Section A.3. Another (more fundamental) issue is that only allowing a single set of marked states precludes enforcing the reachability of multiple (disjoint) sets of states simultaneously.

The use of temporal logic specifications has been investigated as an alter-

native to formal languages [ZS05; JK06; GPT06; Ehl+16]. The most general methods address supervisory control subject to CTL* specifications, but they do not address whether or not the supervisors are maximally permissive or unique [ZS05; JK06]. The work that addresses permissiveness of the controller only applies to more restricted classes of specifications, such as invariance specifications [GPT06] or specifications with a single set of states that should be reachable [Ehl+16].

Other work has focused on the problem of specifying multiple reachability requirements. It has been shown that a system with multiple sets of marked states (modeled by a ''colored marking generator'') can be addressed by an extension of the classical theory to compute a maximally-permissive supervisor that enforces multiple reachability requirements. This is called multitasking supervisory control [dCW05]. The downside is that the results are limited to automata-based models and formal language specifications.

In addition to work concerning how the desired system behavior is specified, the way in which supervisors are implemented has also been investigated. It has been shown that instead of dynamic supervisors (as described in Section A.5), it suffices to consider control policies that only depend on the current state of the system [Len+14; Ehl+16]. Work in this area has also focused on formal language specifications.

In this chapter, we show how to compute a maximally-permissive state-based control policy for a finite transition systems to enforce a class of CTL specifications with combined invariance and reachability requirements. We address multiple reachability requirements, which is similar to the strong nonblocking requirement of multitasking supervisory control. The algorithm we develop relies on symbolic model checking for the computation.

3.2 Finite Transition Systems

The systems we consider are modeled by finite transition systems, $\mathcal{T} = (Q, \Delta)$, as in (A.2). See Table 3.1 for a description of the notation used. A system $\mathcal{T}$ begins in an initial state $q_0 \in Q_0$, where $Q_0 \subseteq Q$. The system evolves by making a sequence of transitions, where each transition has the form $(q, q^+) \in \Delta$, which indicates that the system moves from state q to state q^+. These discrete changes in the system state occur instantaneously, with the only concept of time being the order in which the transitions occur.

Table 3.1: Notation for transition systems.

Symbol	Meaning
$\mathcal{T} = (Q, \Delta)$	Finite transition system
Q	Set of states
$\Delta \subseteq Q \times Q$	Set of transitions
$\Delta_c \subseteq \Delta$	Set of controllable transitions
$\Delta_d \subseteq \Delta_c$	Set of disabled transitions
θ	CTL specification
$[\![\theta]\!]_{\mathcal{T}}$	The set of states in $\mathcal{T}$ that satisfy θ

Specifications about a transition system describe its behavior as it moves from state to state. We address specifications written in computation tree logic (CTL). These specifications involve properties such as invariance (AG) and reachability (EF) of particular sets of states in the system. The most basic properties of states are given by atomic propositions. Without loss of generality, we assume that the state is defined by a vector of state variables, and that the atomic propositions are relational expressions involving those state variables, as in Section A.3. The atomic propositions are intrinsic properties of the individual states, and therefore are not affected by the transition relation, Δ. The CTL operators listed in Table A.1 are, however, affected by the transitions and paths that exist in the system.

Analysis of transition systems using model checking is described in Sec-

tion A.4. The model checking algorithm accepts the model of a system $\mathcal{T} = (Q, \Delta)$ and a specification θ, and returns the set of states in $\mathcal{T}$ that satisfy θ, written $[\![\theta]\!]_{\mathcal{T}}$ as in Table 3.1. Given a set of initial states $Q_0 \subseteq Q$, if $Q_0 \subseteq [\![\theta]\!]_{\mathcal{T}}$, then $\mathcal{T}$ itself satisfies θ. For CTL specifications (including those we consider in this chapter), symbolic model checking algorithms exist to efficiently solve model checking problems for large systems.

3.3 Supervisor Synthesis via Model Checking

If some subset of a system's transitions, $\Delta_c \subseteq \Delta$, can be disabled, then it may be possible to alter the system's behavior (by disabling some of those transitions) so that it satisfies a specification. This gives rise to a control problem for transition systems. Given a system $\mathcal{T} = (Q, \Delta)$ with controllable transitions $\Delta_c \subseteq \Delta$ and a specification θ, the control problem is to determine which of the controllable transitions in Δ_c need to be disabled so that $\mathcal{T}_d$ satisfies θ, where $\mathcal{T}_d \coloneqq (Q, \Delta \setminus \Delta_d)$ and $\Delta_d \subseteq \Delta_c$ is the set of disabled transitions.

Certain types of solutions to the control problem are particularly interesting. First, it is usually important to maximize (in terms of set inclusion) the set of states that satisfy the specification in the controlled system, $[\![\theta]\!]_{\mathcal{T}_d}$. As a secondary objective, it is also desirable to minimize (in terms of set inclusion) the set of disabled transitions, Δ_d. In this section, we address both objectives.

3.3.1 Individual CTL Operators

For a CTL specification that consists of a single CTL operator (along with arbitrary atomic propositions and Boolean operators), the approaches listed in Table 3.2 can be used to calculate a control policy that maximizes $[\![\theta]\!]_{\mathcal{T}_d}$. No transitions should be disabled to satisfy a specification that only involves

reachability (EF). In fact, disabling transitions may remove states from $[\![\theta]\!]_{\mathcal{T}_d}$ that would otherwise be included in $[\![\theta]\!]_{\mathcal{T}}$. For specifications involving invariance (AG), some transitions may need to be disabled so that all the remaining reachable states satisfy the invariant property. These strategies are supported by Lemmas 3.1 and 3.2.

Table 3.2: Computing an optimal control policy for a single CTL operator.

Operator	Strategy
EF	Do not disable any transitions.
AG	Initially, disable all controllable transitions ($\Delta_d = \Delta_c$). After calculating $[\![\theta]\!]_{(Q,\Delta\setminus\Delta_d)}$, enable all transitions *except* $\left\{(q,q^+) \in \Delta_c \;\middle|\; q \in [\![\theta]\!]_{(Q,\Delta\setminus\Delta_d)} \wedge q^+ \notin [\![\theta]\!]_{(Q,\Delta\setminus\Delta_d)}\right\}$.

Lemma 3.1 (disabled transitions and reachability)**.** *If* $\Delta \subseteq \Delta'$*, then* $[\![\mathsf{EF}(p)]\!]_{\mathcal{T}} \subseteq [\![\mathsf{EF}(p)]\!]_{\mathcal{T}'}$*, where* $\mathcal{T} = (Q, \Delta)$ *and* $\mathcal{T}' = (Q, \Delta')$*.*

Proof. The more transitions there are available, the more paths there will be leading to a state that satisfies p. $[\![\mathsf{EF}(p)]\!]_{\mathcal{T}}$ is the least fixed point of the monotonic function $f(Z) = [\![p]\!]_{\mathcal{T}} \cup [\![\mathsf{EX}(Z)]\!]_{\mathcal{T}}$, where $[\![\mathsf{EX}(Z)]\!]_{(Q,\Delta)} \coloneqq \{q \in Q \mid \exists (q,q^+) \in \Delta : q^+ \in Z\}$. Let f' be a similar function that depends on Δ' instead of Δ, so that $[\![\mathsf{EF}(p)]\!]_{\mathcal{T}'}$ is the least fixed point of f'.

$\Delta \subseteq \Delta' \implies [\![\mathsf{EX}(Z)]\!]_{\mathcal{T}} \subseteq [\![\mathsf{EX}(Z)]\!]_{\mathcal{T}'}$, therefore $f(Z) \subseteq f'(Z)$. Both least fixed point calculations are initiated at $Z_0 = Z_0{}' = \emptyset$, so $f(Z_0) \subseteq f'(Z_0{}')$. From the monotonicity of f and f', $Z_i \subseteq Z_i{}' \implies f(Z_i) \subseteq f(Z_i{}')$, and because $f(Z_i{}') \subseteq f'(Z_i{}')$, it follows that $f(Z_i) \subseteq f'(Z_i{}')$.

In the fixed point calculations, $Z_{i+1} = f(Z_i)$ and $Z_{i+1}{}' = f'(Z_i{}')$, so $Z_i \subseteq Z_i{}' \implies Z_{i+1} \subseteq Z_{i+1}{}'$. This means $Z_i \subseteq Z_i{}'\ \forall i$, therefore $[\![\mathsf{EF}(Z)]\!]_{\mathcal{T}} \subseteq [\![\mathsf{EF}(Z)]\!]_{\mathcal{T}'}$. □

Lemma 3.2 (disabled transitions and invariance)**.** *If* $\Delta \subseteq \Delta'$*, then* $[\![\mathsf{AG}(p)]\!]_{\mathcal{T}} \supseteq [\![\mathsf{AG}(p)]\!]_{\mathcal{T}'}$*, where* $\mathcal{T} = (Q, \Delta)$ *and* $\mathcal{T}' = (Q, \Delta')$*.*

Proof. AG and EF are logical duals, i.e., $\mathsf{AG}(p) \iff \neg\mathsf{EF}(\neg p)$. Therefore, $[\![\mathsf{AG}(p)]\!]_{\mathcal{T}} \equiv Q \setminus [\![\mathsf{EF}(p)]\!]_{\mathcal{T}}$ and $[\![\mathsf{AG}(p)]\!]_{\mathcal{T}'} \equiv Q \setminus [\![\mathsf{EF}(p)]\!]_{\mathcal{T}'}$. From Lemma 3.1, $\Delta \subseteq \Delta' \implies [\![\mathsf{EF}(p)]\!]_{\mathcal{T}} \subseteq [\![\mathsf{EF}(p)]\!]_{\mathcal{T}'}$, so $[\![\mathsf{AG}(p)]\!]_{\mathcal{T}} \supseteq [\![\mathsf{AG}(p)]\!]_{\mathcal{T}'}$. □

In addition to maximizing $[\![\theta]\!]_{\mathcal{T}_d}$, the strategies in Table 3.2 return the minimal set of disabled transitions, Δ_d, that achieves this primary objective. In the case of reachability (EF) this is obviously true, because no transitions are disabled. The case of invariance (AG) is addressed in Theorem 3.3. Because the strategies meet both objectives regarding the control policy, we say they compute optimal solutions to the supervisory control problem, or optimal control policies. In both cases, the existence of an optimal control policy depends only on the form of the specification, not on the system to be controlled.

Theorem 3.3 (optimal control with invariance requirements)**.** *The strategy listed in Table 3.2 computes the minimal set of disabled transitions required to maximize the set of states that satisfy a specification of the form* $\mathsf{AG}(p)$*, where* p *does not include any additional CTL operators.*

Proof. From Lemma 3.2, initially disabling all transitions in Δ_c will maximize the set of states included in $[\![\mathsf{AG}(p)]\!]_{\mathcal{T}_d}$. Enabling any transition $(q, q^+) \in \Delta_c$, where $q \in [\![\theta]\!]_{\mathcal{T}_d}$ and $q^+ \notin [\![\theta]\!]_{\mathcal{T}_d}$, would exclude q from $[\![\mathsf{AG}(p)]\!]_{\mathcal{T}_d}$. Therefore, those transitions must remain disabled to maximize the set. All other transitions in Δ_c can be enabled without excluding any states from that satisfying set, which results in the minimal set of disabled transitions that yields the maximal set of states satisfying the specification. □

3.3.2 Multiple CTL Operators

When multiple CTL operators appear in the specification, they are linked through the set of disabled transitions, and this interaction needs to be ac-

counted for. For example, from Table 3.2, the CTL operators AG and EF require opposite control strategies to maximize the respective sets of satisfying states. The opposing strategies come from the opposite effect of disabling a transition, described in Lemmas 3.1 and 3.2.

Simply fixing Δ_d and calculating $[\![\theta]\!]_{\mathcal{T}_d}$ as an ordinary model checking problem for each of the $2^{|\Delta_c|}$ possible control policies is one way to address the interaction, but this approach is not useful for any but the smallest systems. More importantly, whether or not an optimal control policy even exists depends on the specification, θ. For example, Figure 3.1 shows a system with a single state variable $q \in \{1, 2, 3\}$ in which the specification

$$\mathsf{EF}(\mathsf{AG}((q = 1) \vee (q = 2))) \wedge \mathsf{EF}(q = 3)$$

does not have an optimal solution. Either of the states $(q = 1)$ or $(q = 2)$, but not both, can be made to satisfy the specification, depending on the control policy. Without additional information or requirements, it is impossible to determine which solution (if either) is better. In light of this difficulty, it is important to determine (for a given specification and system) whether or not there exists an optimal solution.

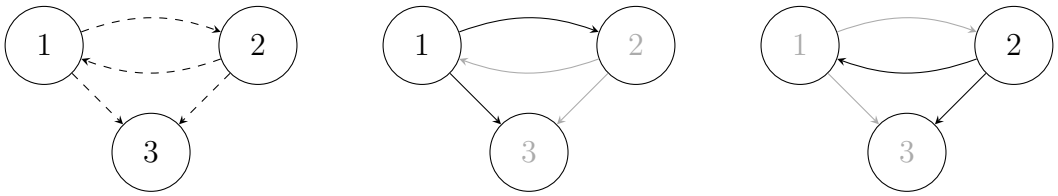

Figure 3.1: The specification $\theta = \mathsf{EF}(\mathsf{AG}((q = 1) \vee (q = 2))) \wedge \mathsf{EF}(q = 3)$ does not have an optimal solution. The first diagram shows the uncontrolled system, where dashed edges represent controllable transitions. In the two potential solutions, states shown in black satisfy the specification, while those shown in gray do not. Gray edges are disabled transitions, and black edges are enabled transitions.

A specification $\mathsf{AG}(\mathsf{EF}(p))$, where p does not contain any CTL operators, can express the controllability and nonblocking requirements of classical supervisory control theory, and therefore corresponds to an optimal control policy [Ehl+16]. In light of multitasking supervisory control [dCW05], with multiple sets that should always be reachable, we consider specifications of the form:

$$\mathsf{AG}\left(\bigwedge_I p_i \wedge \bigwedge_J \mathsf{EF}\left(p_j\right)\right) \tag{3.1}$$

where the p_i and p_j do not contain any CTL operators. We refer to specifications that can be written in the form (3.1) as *combined invariance and reachability* (CIR) specifications. Because these specifications combine multiple CTL operators, the methods in Table 3.2 do not apply directly, and it is necessary to determine whether or not such a specification admits an optimal control policy.

In a formula that involves multiple CTL operators, the interaction between those CTL operators might be managed by first disabling transitions as in Table 3.2 for each subformula, and then checking what effect that has on the other subformulas. This idea leads to the following approach:

1. For each CTL operator, starting with the innermost subformula and working outward, follow the approach in Table 3.2, and record the sets of satisfying states and disabled transitions.

2. After the outermost formula, remove from the system every transition that was disabled while processing any of the subformulas.

3. Using the updated system, return to the innermost subformula and repeat the process until no further transitions are disabled.

Algorithm 3.1 applies this approach for CIR specifications.

Algorithm 3.1: Optimal control for CIR specifications.

Input : Finite transition system $\mathcal{T} = (Q, \Delta)$ with controllable transitions $\Delta_c \subseteq \Delta$ and CIR specification $\theta = \mathsf{AG}\left(\bigwedge_I p_i \wedge \bigwedge_J \mathsf{EF}\left(p_j\right)\right)$

Output: $\Delta_d \subseteq \Delta_c$, the minimal set of disabled transitions that maximizes $[\![\theta]\!]_{(Q,\Delta\setminus\Delta_d)}$

$\theta_{in} \leftarrow \bigwedge_I p_i \wedge \bigwedge_J \mathsf{EF}\left(p_j\right)$
${\Delta_d}^0 \leftarrow \emptyset$
$\Delta^0 \leftarrow \Delta$
$\mathcal{T}^0 \leftarrow \mathcal{T}$
$k \leftarrow 0$
repeat
  $k \leftarrow k + 1$
  ${Z_{in}}^k \leftarrow [\![\theta_{in}]\!]_{\mathcal{T}^{k-1}}$
  $Z^k \leftarrow [\![\mathsf{AG}({Z_{in}}^k)]\!]_{(Q,\Delta\setminus\Delta_c)}$
  ${\Delta_d}^k \leftarrow \left\{(q, q^+) \in \Delta_c \;\middle|\; q \in Z^k \wedge q^+ \notin Z^k\right\}$
  $\Delta^k \leftarrow \Delta^{k-1} \setminus {\Delta_d}^k$
  $\mathcal{T}^k \leftarrow (Q, \Delta^k)$
until $\Delta^k = \Delta^{k-1}$
$\Delta_d \leftarrow {\Delta_d}^k$
return Δ_d

Theorem 3.4 (optimal control with invariance and reachability requirements)**.** *Algorithm 3.1 computes the minimal set of disabled transitions that maximizes the set of states that satisfy the specification if the p_i and p_j do not include any additional CTL operators.*

Proof. Let Z^* be the maximal set of states (if it exists) that can satisfy θ in the controlled system, let ${\Delta_d}^* \subseteq \Delta_c$ be the minimal set of disabled transitions (if it exists) such that $[\![\theta]\!]_{\mathcal{T}^*} = Z^*$, where $\mathcal{T}^* \coloneqq (Q, \Delta \setminus {\Delta_d}^*)$, and let ${Z_{in}}^*$ be $[\![\theta_{in}]\!]_{\mathcal{T}^*}$. Because ${\Delta_d}^*$ is the minimal set of disabled transitions, ${Z_{in}}^*$ is the maximal set of states (from Lemma 3.1) that can satisfy θ_{in}, subject to the requirement that Z^* be maximized.

First, show (by induction on k) that ${Z_{in}}^k \supseteq {Z_{in}}^*$ in Algorithm 3.1. For the base case ($k = 1$), ${Z_{in}}^1 = [\![\theta_{in}]\!]_{(Q,\Delta)} \supseteq [\![\theta_{in}]\!]_{(Q,\Delta\setminus{\Delta_d}^*)} = {Z_{in}}^*$, from Lemma 3.1. For the inductive step, assume that, in iteration k, ${Z_{in}}^k \supseteq {Z_{in}}^*$. From the monotonicity of AG, $Z^k = [\![\mathsf{AG}({Z_{in}}^k)]\!]_{(Q,\Delta\setminus\Delta_c)} \supseteq [\![\mathsf{AG}({Z_{in}}^*)]\!]_{(Q,\Delta\setminus\Delta_c)}$. From Lemma 3.2, $[\![\mathsf{AG}({Z_{in}}^*)]\!]_{(Q,\Delta\setminus\Delta_c)} \supseteq [\![\mathsf{AG}({Z_{in}}^*)]\!]_{\mathcal{T}^*} = Z^*$. Therefore, $Z^k \supseteq Z^*$, so $\forall (q, q^+) \in {\Delta_d}^k : q^+ \notin Z^*$. As a result, $({Z_{in}}^k \setminus {Z_{in}}^{k+1}) \cap {Z_{in}}^* = \emptyset$. Consider $q_{in} \in ({Z_{in}}^k \setminus {Z_{in}}^{k+1})$; q_{in} was prevented from satisfying θ_{in} because a controllable transition (q, q^+) was disabled, where $q^+ \notin Z^*$. If $q_{in} \in {Z_{in}}^*$, then q is reachable from q_{in} in $\mathcal{T}^*$, so the transition (q, q^+) must be disabled to maximize Z^* (because otherwise $q \notin Z^*$). However, disabling (q, q^+) causes q_{in} to not satisfy θ_{in}, so $q_{in} \notin {Z_{in}}^*$, a contradiction. Therefore, $\forall q_{in} \in ({Z_{in}}^k \setminus {Z_{in}}^{k+1}) : q_{in} \notin {Z_{in}}^*$. Thus, ${Z_{in}}^{k+1} \supseteq {Z_{in}}^*$. This concludes the proof by induction.

Then, from ${Z_{in}}^k \supseteq {Z_{in}}^*$, along with the monotonicity of AG and Lemma 3.2 (as before, in the inductive step), $Z^k \supseteq Z^*$ in Algorithm 3.1. In the final iteration, K, no new transitions are disabled, so $Z^K = [\![\theta]\!]_{\mathcal{T}^K}$. Therefore, the upper bound is indeed realized, and $Z^K \equiv Z^*$. From Theorem 3.3, ${\Delta_d}^K \equiv {\Delta_d}^*$. □

Corollary 3.5 (termination)**.** *Algorithm 3.1 terminates after no more than* $\min\{|Q|, |\Delta_c|\}$ *iterations.*

Proof. Both Z^k and Δ^k are finite sets that decrease monotonically from one iteration to the next, and the algorithm terminates when they stop decreasing. At most $|\Delta_c|$ transitions can be removed from Δ^k, and at most $|Q|$ states can be removed from Z^k, so Algorithm 3.1 will terminate after at most $\min\{|Q|, |\Delta_c|\}$ iterations. □

These results show that a CIR specification (3.1), that combines multiple invariance and reachability requirements, corresponds to an optimal control policy, regardless of the system. This makes it possible to check whether or not a given system can be made to satisfy such a specification by first computing the optimal control policy, and then checking whether or not $Q_0 \subseteq [\![\theta]\!]_{\mathcal{T}^*}$.

As the iterations in Algorithm 3.1 are carried out, the ''largest'' model (in terms of the number of states and transitions) that is checked is the uncontrolled system itself. After each iteration, transitions are removed from Δ, so that in future iterations model checking is applied to a system which is ''smaller'' than the uncontrolled system. These notions of system size only apply directly to the explicit representation of the system; a symbolic representation using binary decision diagrams (BDDs) may decrease or increase in size as transitions are disabled, depending on the structure of the system and the variable ordering.

3.4 Labeled Transition Systems

We now show how the results in Section 3.3 apply to discrete event systems with controllable events, modeled as deterministic finite labeled transitions systems (LTSs) as in (A.3). In [Ehl+16], the authors show that the standard supervisory control problem can be reduced to a simpler problem that only

involves reachability (a nonblocking requirement). Furthermore, they show that the simpler problem has a solution in the form of a unique, maximally-permissive, state-based supervisor (if any solution exists).

Definition 3.6 (state-based supervisor)**.** Given an LTS $\mathcal{L} = (Q, \Sigma, \Delta)$ with controllable events $\Sigma_c \subseteq \Sigma$, then a *state-based supervisor* is a set-valued map

$$\Gamma : Q \rightrightarrows \Sigma$$

where $\Gamma(q)$ is the set of events which are enabled in state q. Γ cannot disable uncontrollable events, so $\forall q \in Q : (\Sigma \setminus \Sigma_c) \subseteq \Gamma(q)$.

The closed-loop system produced by a state-based supervisor Γ controlling an LTS $\mathcal{L}$ is

$$\Gamma/\mathcal{L} = (Q, \Sigma, \Delta')$$

where

$$\Delta' \coloneqq \left\{ (q, \sigma, q^+) \in \Delta \;\mid\; \sigma \in \Gamma(q) \right\}$$

That is, the supervisor removes the transitions that are caused by disabled events, and the system is otherwise unchanged.

Definition 3.7 (permissiveness)**.** Given the state-based supervisors $\Gamma : Q \rightrightarrows \Sigma$ and $\Gamma' : Q \rightrightarrows \Sigma$, Γ is *not less permissive* than Γ' if $\forall q \in Q : \Gamma(q) \supseteq \Gamma'(q)$. If Γ is not less permissive than Γ' and, in addition, $\exists q \in Q : \Gamma(q) \supset \Gamma'(q)$, then Γ is *more permissive* than Γ'.

Definition 3.8 (maximally-permissive supervisor)**.** For an LTS $\mathcal{L}$ and a specification θ, $\Gamma : Q \rightrightarrows \Sigma$ is the *maximally-permissive supervisor* that enforces θ in $\mathcal{L}$ if and only if $\Gamma/\mathcal{L}$ satisfies θ, and there does not exist a more permissive supervisor $\Gamma' : Q \rightrightarrows \Sigma$ such that $\Gamma'/\mathcal{L}$ satisfies θ.

To convert from an LTS $\mathcal{L} = (Q, \Sigma, \Delta_{\mathcal{L}})$ with controllable events $\Sigma_c \subseteq \Sigma$ to a TS $\mathcal{T} = (Q, \Delta)$ with controllable transitions $\Delta_c \subseteq \Delta$, apply the following:

$$\begin{aligned} \Delta &= \{(q, q^+) \mid \exists \sigma \in \Sigma : (q, \sigma, q^+) \in \Delta_{\mathcal{L}}\} \\ \Delta_c &= \{(q, q^+) \mid \forall \sigma \in \Sigma : (q, \sigma, q^+) \in \Delta_{\mathcal{L}} \implies \sigma \in \Sigma_c\} \end{aligned}$$

The sets of states and initial states are the same in both systems. The set of unlabeled transitions, Δ, is the set of all labeled transitions that exist in $\mathcal{L}$, with the label removed. The set of controllable transitions, Δ_c, is the subset of Δ for which every corresponding labeled transition in $\mathcal{L}$ is labeled by a controllable event. Because the states are the same in both systems, the atomic propositions (which are relational expressions involving the state variables) are also the same.

To convert from a set of disabled transitions $\Delta_d \subseteq \Delta_c$ to a state-based supervisor $\Gamma : Q \rightrightarrows \Sigma$, apply the following:

$$\Gamma = q \mapsto \left\{ \sigma \in \Sigma \;\middle|\; \nexists q^+ \in Q : (q, q^+) \in \Delta_d \wedge (q, \sigma, q^+) \in \Delta_{\mathcal{L}} \right\}$$

The enabled events in each state are all the events such that the corresponding transition in $\mathcal{T}$ is not in the set of disabled transitions, Δ_d.

The key result we take from Section 3.3 is that, given a finite deterministic LTS $\mathcal{L} = (Q, \Sigma, \Delta_{\mathcal{L}})$ with controllable events $\Sigma_c \subseteq \Sigma$ and a CIR specification $\theta = \mathsf{AG}\left(\bigwedge_I p_i \wedge \bigwedge_J \mathsf{EF}\left(p_j\right)\right)$ as in (3.1), it is possible to compute the maximally-permissive state-based supervisor that maximizes the set of states in $\mathcal{L}$ that satisfy θ. This is done by first converting $\mathcal{L}$ to the TS $\mathcal{T}$, then applying Algorithm 3.1 to compute the minimal set of disabled transitions to enforce θ in $\mathcal{T}$, and finally converting the result back to a state-based supervisor Γ. Because $\mathcal{L}$ is finite, $\mathcal{T}$ is also finite, and Algorithm 3.1 can be applied. Because

$\mathcal{L}$ is deterministic, the resulting set of disabled transitions can be implemented as a state-based supervisor. This procedure is formalized in Algorithm 3.2.

Algorithm 3.2: Optimal state-based supervisor synthesis.

Input : LTS $\mathcal{L} = (Q, \Sigma, \Delta_{\mathcal{L}})$, controllable events $\Sigma_c \subseteq \Sigma$, and CIR specification $\theta = \mathsf{AG}\left(\bigwedge_I p_i \wedge \bigwedge_J \mathsf{EF}\left(p_j\right)\right)$

Output: Γ, the maximally-permissive state-based supervisor that maximizes the set of states in $\mathcal{L}$ that satisfy θ

$\Delta \leftarrow \{(q, q^+) \mid \exists \sigma \in \Sigma : (q, \sigma, q^+) \in \Delta_{\mathcal{L}}\}$

$\mathcal{T} \leftarrow (Q, \Delta)$

$\Delta_c \leftarrow \{(q, q^+) \mid \forall \sigma \in \Sigma : (q, \sigma, q^+) \in \Delta_{\mathcal{L}} \implies \sigma \in \Sigma_c\}$

$\Delta_d \leftarrow$ Apply Algorithm 3.1 to $(\mathcal{T}, \Delta_c, \theta)$

$\Gamma \leftarrow q \mapsto \{\sigma \in \Sigma \mid \nexists q^+ \in Q : (q, q^+) \in \Delta_d \wedge (q, \sigma, q^+) \in \Delta_{\mathcal{L}}\}$

return Γ

3.5 Example

The classic cat and mouse problem [RW89] is an example of a system with a single reachability requirement. In the problem, a cat and mouse are placed in a maze, shown in Figure 3.2. The cat and mouse are initially placed in separate rooms (2 and 4, respectively). Adjacent rooms are connected by doors through which the cat or mouse can move. The cat and mouse can pass through the doors only in the directions indicated in Figure 3.2. Each door can be opened or closed depending on the current rooms occupied by the cat and mouse, except the cat's door between rooms 1 and 3, which is always open. Either the cat or the mouse may move in a given turn, but not both. The invariance requirement is that the cat and mouse should never be in the same room, and the reachability requirement is that they should always be able to return to their original rooms. The control policy is a set of disabled transitions, which corresponds to a mapping from the current rooms occupied by the cat and mouse to a set of closed doors.

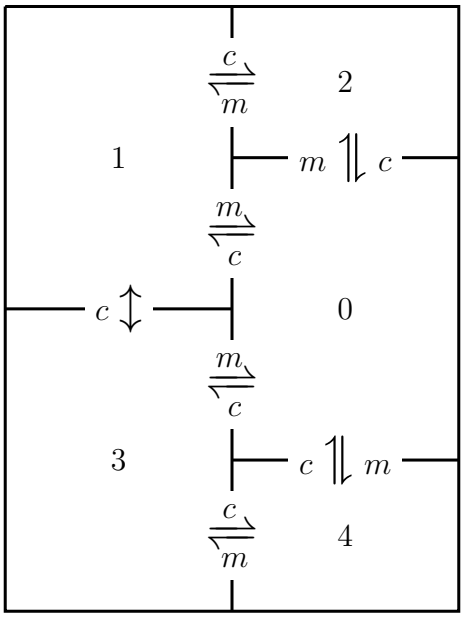

Figure 3.2: The maze from the cat and mouse problem. Arrows represent doors that the cat and mouse can pass through in the indicated direction.

The state space of the system is $Q = \{0 \ldots 4\} \times \{0 \ldots 4\}$, and each state in the system has the form $q = (c, m) \in Q$, where c and m are the rooms occupied by the cat and mouse, respectively. The set of initial states is $Q_0 = \{(2, 4)\}$, which contains the single initial state mentioned previously. The atomic propositions are numeric (equality or inequality) comparisons involving the state variables c and m; for example, $(2, 4) \in [\![(c \neq m)]\!]$. Transitions have the form $((c, m), (c^+, m^+))$, subject to the constraint $(c^+ = c) \vee (m^+ = m)$ (i.e., the cat and mouse do not both move simultaneously). The transitions are, of course, also restricted to those that are feasible given the layout of the maze in Figure 3.2. All of the transitions are controllable *except* those such that $(c \in \{1, 3\}) \wedge (c^+ \in \{1, 3\}) \wedge (c^+ \neq c)$ (the cat moves between rooms 1 and 3) or $(c^+ = c) \wedge (m^+ = m)$ (neither the cat nor mouse moves). In terms of this model, the overall specification is:

$$\mathsf{AG}((c \neq m) \wedge \mathsf{EF}((c = 2) \wedge (m = 4)))$$

Algorithm 3.1 was applied to solve the problem. The disabled transitions and the states that satisfy the specification in the controlled system are shown in Figures 3.3 and 3.4. Note that, in this example, the number of iterations (3) is much lower than the upper limit given by $\min\{|Q|, |\Delta_c|\} = |Q| = 25$.

╲	00	01	02	03	04	10	11	12	13	14	20	21	22	23	24	30	31	32	33	34	40	41	42	43	44
00	∗	·	∗	·	∗	∗	·	·	·	·	·	·	·	·	·	∗	·	·	·	·	·	·	·	·	·
01	1	∗	·	·	·	·	1	·	·	·	·	·	·	·	·	·	1	·	·	·	·	·	·	·	·
02	·	∗	∗	·	·	·	·	∗	·	·	·	·	·	·	·	·	·	∗	·	·	·	·	·	·	·
03	1	·	·	∗	·	·	·	·	1	·	·	·	·	·	·	·	·	·	1	·	·	·	·	·	·
04	·	·	·	2	∗	·	·	·	·	∗	·	·	·	·	·	·	·	·	·	∗	·	·	·	·	·
10	·	·	·	·	·	∗	·	2	·	∗	∗	·	·	·	·	∗	·	·	·	·	·	·	·	·	·
11	·	·	·	·	·	∗	∗	·	·	·	·	∗	·	·	·	·	∗	·	·	·	·	·	·	·	·
12	·	·	·	·	·	·	1	∗	·	·	·	·	1	·	·	·	·	∗	·	·	·	·	·	·	·
13	·	·	·	·	·	∗	·	·	∗	·	·	·	·	∗	·	·	·	·	∗	·	·	·	·	·	·
14	·	·	·	·	·	·	·	·	1	∗	·	·	·	·	∗	·	·	·	·	∗	·	·	·	·	·
20	1	·	·	·	·	·	·	·	·	·	∗	·	1	·	∗	·	·	·	·	·	·	·	·	·	·
21	·	2	·	·	·	·	·	·	·	·	∗	∗	·	·	·	·	·	·	·	·	·	·	·	·	·
22	·	·	∗	·	·	·	·	·	·	·	·	∗	∗	·	·	·	·	·	·	·	·	·	·	·	·
23	·	·	·	2	·	·	·	·	·	·	∗	·	·	∗	·	·	·	·	·	·	·	·	·	·	·
24	·	·	·	·	∗	·	·	·	·	·	·	·	·	∗	∗	·	·	·	·	·	·	·	·	·	·
30	·	·	·	·	·	∗	·	·	·	·	·	·	·	·	·	∗	·	2	·	∗	2	·	·	·	·
31	·	·	·	·	·	·	∗	·	·	·	·	·	·	·	·	∗	∗	·	·	·	·	∗	·	·	·
32	·	·	·	·	·	·	·	∗	·	·	·	·	·	·	·	·	1	∗	·	·	·	·	∗	·	·
33	·	·	·	·	·	·	·	·	∗	·	·	·	·	·	·	∗	·	·	∗	·	·	·	·	∗	·
34	·	·	·	·	·	·	·	·	·	∗	·	·	·	·	·	·	·	·	1	∗	·	·	·	·	1
40	1	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	∗	·	∗	·	1
41	·	∗	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	∗	∗	·	·	·
42	·	·	∗	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	∗	∗	·	·
43	·	·	·	∗	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	∗	·	·	∗	·
44	·	·	·	·	∗	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	∗	∗

Figure 3.3: Disabled transitions in the cat and mouse problem.

In Figure 3.3, each row corresponds to the initial state of a transition, and the column gives the final state. The first number is the room occupied by the cat, and the second number is the room occupied by the mouse. Transitions marked with $\cdot$ are infeasible given the system definition, those marked with a number k are disabled after the k^{th} iteration, and transitions that remain enabled are marked with $*$.

In Figure 3.4, the row and column correspond to the rooms occupied by the

╲	0	1	2	3	4
0	1	2	2	2	$*$
1	$*$	1	2	1	$*$
2	$*$	$*$	1	$*$	$*$
3	$*$	1	2	1	$*$
4	2	2	2	2	1

Figure 3.4: States that satisfy the specification in the cat and mouse problem.

cat and mouse, respectively. The states marked with $*$ satisfy the specification subject to the disabled transitions shown in Figure 3.3. The states that do not satisfy the specification are marked with a number k, where k indicates the iteration in which that state was removed from the intermediate solution.

From the state $(2, 4)$, if either the cat or mouse leaves its initial room, the other one is no longer allowed to leave its room. Whichever one left its initial room first is then allowed to travel to any room it can reach, except the other one's initial room.

Note that Figure 3.4 shows the maximal set of states that satisfy the specification, not only those that are reachable (according to the model definition) from the initial state. In particular, states $(1, 0)$, $(3, 0)$, and $(2, 1)$ are not reachable from the initial state $(2, 4)$, but they satisfy the specification subject to the solved-for optimal control policy. This demonstrates the fact that the optimal control policy does not depend on the initial state of the system; instead, it determines the set of all initial conditions that satisfy the specification. If the cat and mouse were to start in a state that is not included in this set, then no control policy can guarantee that the specification is satisfied. If they happen to move to a state in the satisfying set, then the controller could take over again and enforce the specification.

3.6 Summary

In this chapter, an algorithm was developed to compute the maximally-permissive control policy for a discrete event system that maximizes the set of states in the system that satisfy a specification that involves invariance and reachability requirements. This is called the optimal control policy, and it was proven to exist for such a specification, regardless of the particular system that is to be controlled. The class of specifications involves multiple reachability requirements, as in multitasking supervisory control. The algorithm uses CTL model checking to perform the intermediate calculations.

Chapter 4

SynthSMV v0.1.0

4.1 Related Software

Significant work in the fields of model checking and supervisory control has produced useful software tools that provide features such as efficient/high-performance solution techniques, expressive modeling languages, verification algorithms, controller synthesis algorithms, software freedom/availability of source code, compatibility with commercial use, etc. In model checking, the resulting tools include SMV [McM93], SPIN [Hol97], NuSMV [Cim+02], and LTSmin [BvW10]. SMV is the original BDD-based symbolic model checking solver, and NuSMV, which implements CTL and LTL model checking, is its successor. SPIN is a widely-used LTL model checker; it does not support CTL specifications. LTSmin is a newer tool that aims to provide a wide variety of modeling front ends and back end solvers for general model checking. NuSMV, SPIN, and LTSmin are all open source software projects under active development. In supervisory control, the resulting tools include Supremica [Åke+06], UMDES/DESUMA [RLG06], STSLib [MW08], and libFAUDES [MSP08]. Supremica and STSLib implement BDD-based symbolic

supervisor synthesis for standard supervisory control problems in terms of finite automata, formal language specifications, and marked states. UMDES and libFAUDES are libraries of routines that implement many of the algorithms from the fields of supervisory control and discrete event systems using explicit (i.e., not BDD-based) system representations.

SynthSMV[1] implements supervisor synthesis as an extension of the model checking solver NuSMV (SynthSMV v0.1.0 is based on NuSMV 2.6.0). Because of this, SynthSMV benefits from the familiar modeling language, the efficient BDD-based symbolic algorithms for building and analyzing finite-state machines (FSMs), and the model checking and verification capabilities present in NuSMV. To this, it adds the ability to define discrete event systems with controllable and uncontrollable events, and to compute a maximally-permissive state-based supervisor to enforce a specification in such a system. In this chapter, we describe how SynthSMV extends NuSMV, and present some small examples to demonstrate its use. We assume that the reader is already familiar with NuSMV.

4.2 Changes from NuSMV

4.2.1 Input Language

The first change to the input language is the addition of the `CTRBL` keyword, which is used to declare the set of state/input pairs that are controllable (i.e., that can be prevented from occurring). In model checking, all of the inputs that satisfy the model's overall state/input constraints are assumed to be enabled at all times. In supervisory control, some of the inputs (the controllable inputs) can be disabled by a supervisor in order to enforce a specification. Declarations

[1] Available at `https://bitbucket.org/blakecraw/synthsmv/`.

of the controllable inputs have the form:

```
CTRBL state_input_constraint;
```

where `state_input_constraint` is a Boolean formula that must be satisfied by the current (i.e., the `next` operator cannot appear in `state_input_constraint`) state/input pair for it to be controllable. Multiple `CTRBL` declarations are combined via conjunction, so that

```
CTRBL event in {1, 2};
CTRBL event in {2, 3};
```

and

```
CTRBL event = 2;
```

are equivalent. In the absence of `CTRBL` declarations, all state/input pairs are assumed to be controllable.

The second change to the input language is the addition of the `SYNTH` specification type. Similar to how CTL model checking is applied to `CTLSPEC` specifications, and LTL model checking is applied to `LTLSPEC` specifications, supervisor synthesis is attempted for `SYNTH` specifications. A `SYNTH` specification is assumed to be a valid CTL specification $\theta = \mathsf{AG}\left(\bigwedge_I p_i \wedge \bigwedge_J \mathsf{EF}\left(p_j\right)\right)$ as in (3.1), where none of the p_i or p_j contains any additional temporal operators. For such a specification, it is possible to compute a maximally-permissive state-based supervisor that enforces θ, if any such supervisor exists; refer to Chapter 3, where these are referred to as combined invariance and reachability (CIR) specifications.

Due to the way the `CTRBL` and `SYNTH` keywords are implemented in SynthSMV v0.1.0, they must be used carefully to obtain correct results. The `CTRBL` declaration accepts the same class of constraints as NuSMV's `TRANS` keyword.

This includes constraints that involve the `next` value of the state variables, which will not function correctly as `CTRBL` constraints. Similarly, a `SYNTH` specification only has to be valid as a `CTLSPEC` specification for SynthSMV to accept it, but the algorithm that is implemented is only intended to work correctly for specifications that have the form (3.1). In addition, all `CTRBL` and `SYNTH` declarations must be made in the main `MODULE`. For now, it is the modeler's responsibility to ensure that these requirements are met; SynthSMV will likely produce undesired or incorrect results if they are not.

4.2.2 Modeling

In general, the same types of FSMs that can be defined in NuSMV can also be defined in SynthSMV. In fact, given a model that does not contain any `CTRBL` or `SYNTH` declarations, SynthSMV should produce exactly the same results as NuSMV. For more information, refer to the NuSMV documentation.

To apply supervisor synthesis, the events that can occur in the system must be modeled as inputs to the FSM, using NuSMV's `IVAR` (input variable) declaration. The input variables are the ones whose values are restricted by the supervisor, so it is important that all events are modeled as inputs; otherwise, they cannot be disabled. For example, consider the two modules:

```
MODULE events_as_states

VAR state : boolean;
VAR event : boolean;

INIT !state;

TRANS next(state) = (state | event);
```

and

```
MODULE events_as_inputs

VAR state : boolean;
IVAR event : boolean;

INIT !state;

TRANS next(state) = (state | event);
```

where the only difference is that in the first, `event` is modeled as a free state variable, and in the second, it is modeled as a free input variable. The model

```
MODULE main

VAR eas : events_as_states;
VAR eai : events_as_inputs;

TRANS eas.event = eai.event;
CTLSPEC AG(eas.state = eai.state);

CTLSPEC AG(EF(!eas.state));
CTLSPEC AG(EF(!eai.state));
```

confirms that the two systems produce the same result in response to the same sequence of events (the first specification is satisfied). Thus, the modules are equivalent when it comes to model checking, and either modeling strategy will work; for example, the second and third specifications are both false. However, the model

```
MODULE main

VAR eas : events_as_states;
VAR eai : events_as_inputs;

SYNTH AG(EF(!eas.state));
SYNTH AG(EF(!eai.state));
```

demonstrates the difference, as the first specification is false (i.e., cannot be satisfied by any supervisor), while the second is true (with the input

`eai.event = TRUE` disabled by the supervisor). This difference in how events have to be modeled for supervisory control (where events really do need to be modeled as inputs to the FSM) compared to model checking (where an equivalent model can be produced by treating events as free state variables) will be the most visible difference between SynthSMV and NuSMV for somebody who is already familiar with modeling and analyzing discrete event systems in NuSMV.

4.3 Implementation

SynthSMV applies Algorithm 3.1 to solve supervisory control problems. Thus, it computes state-based supervisors, as in Chapter 3. For a CIR specification and a deterministic finite labeled transition system as in Section 3.4, this approach is guaranteed to produce the maximally-permissive supervisor that enforces the specification.

One of the main reasons to use SynthSMV is to take advantage of NuSMV's well-established and efficient implementation of symbolic model checking to perform the intermediate computations in the state-based supervisor synthesis algorithm. The techniques that carry over from NuSMV to SynthSMV include symbolic representation of the FSM (including the definition of the controllable events) via binary decision diagrams (BDDs), dynamic reordering of the BDD variables, and cone-of-influence (COI) reduction. For more information about how these are implemented in NuSMV, and how to best take advantage of them, refer to the NuSMV documentation.

The major limitation of SynthSMV v0.1.0 is that, while the maximally-permissive state-based supervisor is computed, it is not returned in a usable form. The simple explanation for this is that SynthSMV was developed as part

of the work in Chapter 5, where the result of interest is whether or not any supervisor can enforce the specification, not necessarily what the supervisor (if it exists) actually does. There is no technical reason that the supervisor couldn't be returned, for example, as the BDD that describes the disabled state/input pairs.

4.4 Examples

4.4.1 The Cat and Mouse Problem

Recall the cat and mouse problem from Section 3.5. The problem can be modeled in SynthSMV as shown in Figure 4.1.

Applying SynthSMV shows that while the `CTLSPEC` specification is false, the `SYNTH` specification is true. This means that there is a (non-trivial) supervisor that enforces the specification, as expected. Further analysis indicates that while all 25 states are reachable in the uncontrolled system, only 6 remain reachable in the (most permissively) controlled system; these are the same reachable states described in Section 3.5.

4.4.2 The Dining Philosophers Problem

In the well-known dining philosophers problem, a group of philosophers sits around a table, with a shared fork between each pair of philosophers. When a philosopher has no forks, it can think. Once a philosopher picks up the fork on its left, it then waits until the fork on its right is available, and picks up that fork when it wants to eat. When a philosopher has both forks, it can eat, after which it puts down both forks. The desired outcome is that each philosopher is always able to think at some point in the future, and is always able to eat at some point in the future. This is to be achieved by enforcing a set of rules

```
MODULE cat

VAR room : 0..4;
IVAR move : {01, 03, 12, 13, 20,
             31, 34, 40, wait};

ASSIGN next(room) :=
  case
    room = 0 :
      case
        move = 03 : 3;
        move = 01 : 1;
        TRUE : room;
      esac;
    room = 1 :
      case
        move = 12 : 2;
        move = 13 : 3;
        TRUE : room;
      esac;
    room = 2 :
      case
        move = 20 : 0;
        TRUE : room;
      esac;
    room = 3 :
      case
        move = 31 : 1;
        move = 34 : 4;
        TRUE : room;
      esac;
    room = 4 :
      case
        move = 40 : 0;
        TRUE : room;
      esac;
  esac;
```

```
MODULE mouse

VAR room : 0..4;
IVAR move : {02, 04, 10, 21,
             30, 43, wait};

ASSIGN  next(room) :=
  case
    room = 0 :
      case
        move = 02 : 2;
        move = 04 : 4;
        TRUE : room;
      esac;
    room = 1 :
      case
        move = 10 : 0;
        TRUE : room;
      esac;
    room = 2 :
      case
        move = 21 : 1;
        TRUE : room;
      esac;
    room = 3 :
      case
        move = 30 : 0;
        TRUE : room;
      esac;
    room = 4 :
      case
        move = 43 : 3;
        TRUE : 4;
      esac;
  esac;
```

```
MODULE main

VAR c : cat;
VAR m : mouse;

INIT c.room = 2;
INIT m.room = 4;

TRANS (c.move = wait) | (m.move = wait);
CTRBL !(c.move in {13, 31, wait}) | !(m.move in {wait});

CTLSPEC AG((c.room != m.room) & EF((c.room = 2) & (m.room = 4)));
SYNTH AG((c.room != m.room) & EF((c.room = 2) & (m.room = 4)));
```

Figure 4.1: Implementation of the cat and mouse problem in SynthSMV.

that define when each philosopher is allowed to pick up the fork on its left (the controllable events). The problem can be modeled in SynthSMV as shown in Figure 4.2. Note that this example includes multiple reachability requirements that must all be satisfied.

```
MODULE philosopher_ii()

VAR state : {think, up, eat};
IVAR event : {take, put, wait};

INIT state = think;
ASSIGN next(state) :=
  case
    state = think :
      case
        event = take : up;
        TRUE : state;
      esac;
    state = up :
      case
        event = take : eat;
        TRUE : state;
      esac;
    state = eat :
      case
        event = put : think;
        TRUE : state;
      esac;
  esac;
```

```
MODULE philosopher_oi(left)

VAR state : {think, up, eat};
IVAR event : {take, put, wait};

INIT state = think;
ASSIGN next(state) :=
  case
    state = think :
      case
        event = take : up;
        TRUE : state;
      esac;
    state = up :
      case
        event = take : eat;
        TRUE : state;
      esac;
    state = eat :
      case
        event = put : think;
        TRUE : state;
      esac;
  esac;

INVAR !((state in {eat, up})
        & (left.state = eat));
```

```
MODULE philosopher_oo(left, right)

VAR state : {think, up, eat};
IVAR event : {take, put, wait};

INIT state = think;
ASSIGN next(state) :=
  case
    state = think :
      case
        event = take : up;
        TRUE : state;
      esac;
    state = up :
      case
        event = take : eat;
        TRUE : state;
      esac;
    state = eat :
      case
        event = put : think;
        TRUE : state;
      esac;
  esac;

INVAR !((state in {eat, up})
        & (left.state = eat));
INVAR !((right.state in {eat, up})
        & (state = eat));
```

```
MODULE main

VAR p1 : philosopher_ii;
VAR p2 : philosopher_oi(p1);
VAR p3 : philosopher_oi(p2);
VAR p4 : philosopher_oo(p3, p1);

TRANS count(p1.event != wait, p2.event != wait, p3.event != wait, p4.event != wait) <= 1;
CTRBL count(p1.event = take, p2.event = take, p3.event = take, p4.event = take) > 0;

CTLSPEC AG(EF(p1.state = think) & EF(p1.state = eat) &
           EF(p2.state = think) & EF(p2.state = eat) &
           EF(p3.state = think) & EF(p3.state = eat) &
           EF(p4.state = think) & EF(p4.state = eat));

SYNTH AG(EF(p1.state = think) & EF(p1.state = eat) &
         EF(p2.state = think) & EF(p2.state = eat) &
         EF(p3.state = think) & EF(p3.state = eat) &
         EF(p4.state = think) & EF(p4.state = eat));
```

Figure 4.2: Implementation of the dining philosophers problem in SynthSMV. A circle of philosophers starts with a `philosopher_ii`, extends to the right of that philosopher with zero or more `philosopher_oi`, and terminates with a `philosopher_oo`, who sits to the left of the original `philosopher_ii`. In this case, there are 4 philosophers.

As with the cat and mouse problem, the `CTLSPEC` specification is false, and the `SYNTH` specification is true. The output from SynthSMV indicates that, of

the 81 states (4 philosophers, 3 states each), 34 are reachable in the uncontrolled system, while only 33 are reachable in the controlled system. The single state that is excluded is the deadlock state in which each of the philosophers is holding the left fork. This results in the state-based supervisor that forbids a philosopher from picking up the fork on its left if each of the other philosophers is holding a fork in its left hand.

4.5 Summary

This chapter has provided an overview of SynthSMV, and the advantages that come with using NuSMV as the basis for solving supervisory control problems. These include access to NuSMV's modeling language and efficient symbolic algorithms, along with the ability to synthesize control strategies for discrete event systems, which is not possible in NuSMV itself. The major limitation that users are likely to encounter is that the computed supervisor is not returned by SynthSMV; as such, it is currently useful primarily to check whether or not a supervisory control problem has a solution. SynthSMV is available under the same open source/free software license as NuSMV (the GNU Lesser General Public License, version 2.1 or later), ensuring that users are free to extend it further, or use it commercially.

Chapter 5

Falsification of Invariance and Reachability Specifications

5.1 Falsification of Hybrid Systems

As described in Chapter 1, existing verification techniques typically address specifications written in the temporal logic ACTL*, which is the subset of CTL* obtained by allowing only the universal path quantifier, A. This limitation includes verification of invariance (reach-avoid) properties given by the ACTL* formula $\mathsf{AG}(p)$ [Tab09]. The reason for the limitation is that a specification written in ACTL* can be verified if it holds in a finite-state abstraction of the original (infinite-state) system [Tiw07].

Recently, there has been growing interest in falsification of temporal logic properties in hybrid systems. This has been motivated by the fact that industrial application of formal methods is often motivated by a search for errors in existing systems. Similar to how verification is mostly limited to ACTL* specifications, previous research concerning falsification of hybrid systems has focused on limited classes of specifications, including invariance [BF04;

Ler+08; PKV09; Zut+13], metric temporal logic (MTL) [Ngh+10; SF12], and signal temporal logic (STL) [Dre+15]. What these classes of specifications have in common is that they are either LTL specifications (invariance) or extensions of LTL (MTL and STL), which means that they can be falsified by showing the existence of a single violating trajectory. Thus, each of the various techniques amounts to evaluating a finite number of trajectories in search of a counterexample to the specification. One issue with this approach is that it requires either exact knowledge of the hybrid dynamics, or a conservative approximation thereof. Such a dynamic model would be difficult to obtain for a chemical plant, and even if it were available, it would be expensive to simulate. Another issue is that only LTL (and similar) specifications can be falsified in this way. This excludes specifications that combine invariance and reachability, such as the CIR specifications from Chapter 3.

In this chapter, we address the following problem: given an existing control and automation system, and a specification of the desired closed loop behavior that involves combined invariance and reachability requirements, show that the closed-loop behavior violates the specification. We develop an abstraction-based algorithm that can be applied to falsify a class of CTL specifications in sample-and-hold control systems. The key differences between this and related work are the class of specifications we consider (which combine invariance and reachability requirements), and the fact that our analysis does not depend on simulating the hybrid dynamics.

5.2 Discrete Logic in Sample-and-Hold Control Systems

In this chapter, we focus on continuous plants controlled by sample-and-hold control systems (SHCSs). Such systems, which have the form (2.1), often contain complex discrete automation logic that tracks the logical state of the plant during operation. We now describe the behavior of such systems with respect to CIR specifications.

5.2.1 Discrete Jump System

To address specifications related to the discrete state s of an SHCS $\mathcal{H}$ as in (2.1), we use the concept of a corresponding discrete jump system, which is a labeled transition system (LTS).

Definition 5.1 (discrete jump system)**.** For an SHCS $\mathcal{H}$, with initial states $X_0 \subseteq X$ in which $\tau_0 = 0$, the corresponding *discrete jump system (DJS)* is the LTS $\mathcal{J} = (Q, \Sigma, \Delta)$ with initial states Q_0, where:

- $Q \coloneqq \{x \in X \mid \tau = 0\}$
- $\Sigma \coloneqq \left\{\sigma \in \{0,1\}^{n_r} \;\middle|\; \exists (z^\mathsf{T}, u^\mathsf{T}, s^\mathsf{T}, \tau)^\mathsf{T} \in X : \sigma \in \rho(z)\right\}$
- $\Delta \subseteq Q \times \Sigma \times Q$ is the set of all transitions (q, σ, q^+) such that there

exists a solution to $\mathcal{H}$, $\phi : E \to X$, and some $(t, k) \in E$, for which:

$$\begin{aligned} q = \left(z^\mathsf{T}, u^\mathsf{T}, s^\mathsf{T}, 0\right)^\mathsf{T} &= \phi(t, k) \\ (z'^\mathsf{T}, u^\mathsf{T}, s^\mathsf{T}, T)^\mathsf{T} &= \phi(t + T, k) \\ \sigma &\in \rho(z') \\ s^+ &= g_s(\sigma, s) \\ u^+ &\in G_u(z', s^+) \\ q^+ = \left(z'^\mathsf{T}, u^{+\mathsf{T}}, s^{+\mathsf{T}}, 0\right)^\mathsf{T} &= \phi(t + T, k + 1) \end{aligned}$$

- $Q_0 \coloneqq X_0$

The trajectories $\psi = q_0, q_1, \ldots$ that can occur in $\mathcal{J}$ correspond to the sequences of states that occur along solutions to $\mathcal{H}$ immediately after jumps. Because the discrete state s only changes in discrete jumps, these trajectories ψ capture the sequences of discrete states that can occur along solutions to $\mathcal{H}$.

Definition 5.2 (implicit supervisor)**.** Given DJS $\mathcal{J} = (Q, \Sigma, \Delta)$, the *implicit supervisor* $\Gamma : Q \rightrightarrows \Sigma$ is the set-valued map:

$$\Gamma(q) \coloneqq \left\{ \sigma \in \Sigma \;\middle|\; \exists q^+ \in Q : (q, \sigma, q^+) \in \Delta \right\}$$

The implicit supervisor Γ for a DJS $\mathcal{J}$, which is a state-based supervisor as in Definition 3.6, represents which events do not occur in a given state along any solution to the hybrid system. The name refers to the idea that the events are disabled implicitly by the hybrid system's dynamics.

5.2.2 SHCSs and CIR Specifications

We now describe the relationship between a SHCS $\mathcal{H}$ and its corresponding DJS $\mathcal{J}$ with respect to satisfying specifications. Consider an SHCS $\mathcal{H}$ and a

CIR specification $\theta = \mathsf{AG}\left(\bigwedge_I p_i \wedge \bigwedge_J \mathsf{EF}\left(p_j\right)\right)$ as in (3.1), in which the p_i and p_j only involve the discrete state of the system, s. In this case, $\mathcal{H}$ satisfies θ if and only if, for every hybrid arc $\phi : E \to X$ that is a solution to $\mathcal{H}$, the following conditions hold:

1. For every $e \in E$, $\bigwedge_I p_i$ holds in $\phi(e)$.
2. For every $e \in E$, and for every $j \in J$, there exists a solution $\hat{\phi} : \hat{E} \supseteq E \to X$ such that, for all $\hat{e} \preceq e$, $\hat{\phi}(\hat{e}) = \phi(\hat{e})$, and there exists $\hat{e}' \succeq e$ such that p_j holds in $\hat{\phi}(\hat{e}')$.

Similarly, $\mathcal{J}$ satisfies θ if and only if, for every trajectory $\psi : K \to X$ that can occur in $\mathcal{J}$:

1. For every $k \in K$, $\bigwedge_I p_i$ holds in $\psi[k]$.
2. For every $k \in K$, and for every $j \in J$, there exists a solution $\hat{\psi} : \hat{K} \supseteq K \to X$ such that, for all $\hat{k} \leq k$, $\hat{\psi}[\hat{k}] = \psi[\hat{k}]$, and there exists $\hat{k}' \geq k$ such that p_j holds in $\hat{\psi}[\hat{k}']$.

Because θ only involves the discrete state of $\mathcal{H}$, the above sets of conditions are equivalent. Intuitively, whether or not $\mathcal{H}$ satisfies a CIR specification related to the discrete state of the system depends on the possible sequences of jumps that can occur along solutions to $\mathcal{H}$. As a result, $\mathcal{H}$ satisfies θ if and only if the corresponding DJS $\mathcal{J}$ satisfies θ.

5.2.3 Initial Abstraction

Consider an SHCS $\mathcal{H}$ with corresponding DJS $\mathcal{J} = (Q, \Sigma, \Delta)$, with initial states Q_0, and the following LTS:

$$\widetilde{\mathcal{J}} = \left(\widetilde{Q}, \Sigma, \widetilde{\Delta}\right) \tag{5.1}$$

with initial states $\widetilde{Q}_0$, where

$$\begin{aligned}
\widetilde{Q} &:= \left\{ \widetilde{q} \in \{0,1\}^{n_s} \;\middle|\; \exists (z^\mathsf{T}, u^\mathsf{T}, s^\mathsf{T}, \tau)^\mathsf{T} \in Q : s = \widetilde{q} \right\} \\
\widetilde{Q}_0 &:= \left\{ \widetilde{q} \in \widetilde{Q} \;\middle|\; \exists (z^\mathsf{T}, u^\mathsf{T}, s^\mathsf{T}, \tau)^\mathsf{T} \in Q_0 : s = \widetilde{q} \right\} \\
\widetilde{\Delta} &:= \left\{ (\widetilde{q}, \sigma, \widetilde{q}^+) \in \widetilde{Q} \times \Sigma \times \widetilde{Q} \;\middle|\; \widetilde{q}^+ = g_s(\sigma, \widetilde{q}) \right\}
\end{aligned}$$

and Σ is the same as in $\mathcal{J}$. Because $\widetilde{Q}$ and Σ are finite and g_s is a function, $\widetilde{\mathcal{J}}$ is a finite deterministic LTS.

Lemma 5.3 (initial abstraction)**.** *$\widetilde{\mathcal{J}}$ in* (5.1) *is an abstraction of $\mathcal{J}$, i.e., $\widetilde{\mathcal{J}} \succeq \mathcal{J}$, with the abstraction function $\alpha(q) = s$.*

Proof. Show that $\widetilde{\mathcal{J}}$ meets both of the requirements in Definition A.2 with abstraction function α. The set of initial states meets the first requirement, that $\widetilde{Q}_0 = \alpha(Q_0)$, by definition. For any transition $(q, \sigma, q^+) \in \Delta$, where $q = (z^\mathsf{T}, u^\mathsf{T}, s^\mathsf{T}, \tau)^\mathsf{T}$ and $q^+ = (z^{+\mathsf{T}}, u^{+\mathsf{T}}, s^{+\mathsf{T}}, \tau^+)^\mathsf{T}$, $s^+ = g_s(\sigma, s)$ holds (from Definition 5.1). Therefore, the corresponding abstract transition $(\alpha(q), \sigma, \alpha(q^+)) \equiv (s, \sigma, s^+)$ is an element of $\widetilde{\Delta}$, so the second requirement, that $\widetilde{\Delta} \supseteq \Delta$, is also met. Thus, $\widetilde{\mathcal{J}} \succeq \mathcal{J}$. □

5.3 Falsifying CIR Specifications

Consider an SHCS $\mathcal{H}$ with corresponding DJS $\mathcal{J}$, the abstraction $\widetilde{\mathcal{J}} \succeq \mathcal{J}$ as in (5.1), and a CIR specification θ of the form (3.1). We wish to falsify the specification, that is, to prove that $\mathcal{H}$ does not satisfy θ. To accomplish this, we seek a state-based supervisor $\widetilde{\Gamma}$ such that the closed-loop behavior of $\widetilde{\Gamma}/\widetilde{\mathcal{J}}$ satisfies the specification. Given an appropriate definition of which events are controllable in $\widetilde{\mathcal{J}}$, $\Sigma_c \subseteq \Sigma$, we can guarantee that if no such $\widetilde{\Gamma}$ exists, then $\mathcal{H}$ does not satisfy θ.

5.3.1 Computing a Restricted Abstraction

First, we note that instead of attempting to produce a supervisor that will be implemented in the plant, we view the continuous dynamics as a supervisor that disables events according to the feasible system trajectories, ϕ. This is the implicit supervisor in Definition 5.2. We now show that if $\mathcal{H}$ satisfies θ, then the implicit supervisor of $\mathcal{J}$ can act as a supervisor that enforces θ in the initial abstraction.

Definition 5.4 (abstract supervisor)**.** Consider the LTSs $\mathcal{L}$ and $\widetilde{\mathcal{L}} \succeq \mathcal{L}$ with abstraction function $\alpha : Q \to \widetilde{Q}$, and state-based supervisor Γ for $\mathcal{L}$. The *abstract supervisor* $\widetilde{\Gamma} : \widetilde{Q} \rightrightarrows \Sigma$ is the set-valued map defined by:

$$\widetilde{\Gamma}(\widetilde{q}) \coloneqq \{\sigma \in \Sigma \mid \exists q \in Q : \alpha(q) = \widetilde{q} \wedge \sigma \in \Gamma(q)\}$$

and we (further) extend the definition of an abstraction function from Section A.2 to state-based supervisors, so that $\alpha(\Gamma) \coloneqq \widetilde{\Gamma}$.

Lemma 5.5 (abstract supervisor preserves abstraction)**.** *Let $\mathcal{J}$ be a DJS with implicit supervisor Γ, let $\widetilde{\mathcal{J}}$ be an abstraction of $\mathcal{J}$ as in* (5.1) *with abstraction function α, and let $\widetilde{\Gamma} = \alpha(\Gamma)$. $\widetilde{\Gamma}/\widetilde{\mathcal{J}} \succeq \mathcal{J}$.*

Proof. The only transitions that $\widetilde{\Gamma}$ removes from $\widetilde{\mathcal{J}}$ are those that correspond to concrete transitions which do not exist in $\mathcal{J}$, so the result is still an abstraction. □

Theorem 5.6 (existence of an abstract supervisor that enforces a specification)**.** *Let $\mathcal{J}$ be a DJS with implicit supervisor Γ, let $\widetilde{\mathcal{J}}$ be an abstraction of $\mathcal{J}$ as in* (5.1) *with abstraction function α, and let $\widetilde{\Gamma} = \alpha(\Gamma)$. If $\mathcal{J}$ satisfies a* CIR *specification $\theta = \mathsf{AG}\left(\bigwedge_I p_i \wedge \bigwedge_J \mathsf{EF}\,(p_j)\right)$ in which the p_i and p_j only involve the discrete state of the system, then $\widetilde{\Gamma}/\widetilde{\mathcal{J}}$ satisfies θ.*

Proof. Consider any trajectory ψ that exists in $\mathcal{J}$, and the corresponding abstract trajectory $\widetilde{\psi} = \alpha(\psi)$. Because the p_i only involve the discrete state s, $\bigwedge_I p_i$ holds in a state $q \in Q$ if and only if $\bigwedge_I p_i$ holds in $\alpha(q)$. Therefore, for any state q on ψ that meets the invariance requirement, the corresponding $\widetilde{q}$ on $\widetilde{\psi}$ also meets the invariance requirement. Similarly, because the p_j only involve the discrete state, and $\widetilde{\Gamma}/\widetilde{\mathcal{J}} \succeq \mathcal{J}$ (from Lemma 5.5), for any state q on ψ that meets the reachability requirement, the corresponding $\widetilde{q}$ on $\widetilde{\psi}$ also meets the reachability requirement. Because $\mathcal{J}$ satisfies θ, every state q on ψ satisfies the invariance and reachability requirements. Therefore, for any trajectory ψ that exists in $\mathcal{J}$, the abstract trajectory $\widetilde{\psi}$ meets the conditions to satisfy θ, so $\widetilde{\Gamma}/\widetilde{\mathcal{J}}$ satisfies θ. □

Theorem 5.6 guarantees that if $\mathcal{H}$ satisfies θ, then there exists a state-based supervisor that enforces θ in $\widetilde{\mathcal{J}}$. Because the state space of $\mathcal{J}$ is infinite, Γ is not easy to compute, but it motivates what follows.

Theorem 5.7 (upper bound on the reachable states if a specification is met)**.** *Let $\mathcal{H}$ be an SHCS with DJS $\mathcal{J}$ and abstraction $\widetilde{\mathcal{J}} \succeq \mathcal{J}$ as in* (5.1) *with abstraction function $\alpha(q) = s$, and let $\Sigma_c = \Sigma$. For a* CIR *specification θ of the form $\mathsf{AG}\left(\bigwedge_I p_i \wedge \bigwedge_J \mathsf{EF}\left(p_j\right)\right)$, where p_i and p_j only involve the discrete state of the system, compute the maximally-permissive state-based supervisor, $\widetilde{\Gamma}$, that enforces θ in $\widetilde{\mathcal{J}}$, by applying Algorithm 3.2. If $\mathcal{H}$ satisfies θ, then $Reach(\widetilde{\Gamma}/\widetilde{\mathcal{J}}) \supseteq \alpha(Reach(\mathcal{J}))$.*

Proof. Assume that $\mathcal{H}$ satisfies θ, and let $\Gamma' : Q \rightrightarrows \Sigma$ be the implicit supervisor of $\mathcal{J}$. Then $\alpha(\Gamma') = \widetilde{\Gamma}' : \widetilde{Q} \rightrightarrows \Sigma$ is a state-based supervisor such that $\widetilde{\Gamma}'/\widetilde{\mathcal{J}}$ satisfies θ (from Theorem 5.6) and $\widetilde{\Gamma}'/\widetilde{\mathcal{J}} \succeq \mathcal{J}$ (from Lemma 5.5).

The initial state of an LTS is always reachable, so $\widetilde{Q}_0 \subseteq Reach(\widetilde{\Gamma}/\widetilde{\mathcal{J}})$. Because $\widetilde{\mathcal{J}} \succeq \mathcal{J}$, $\widetilde{Q}_0 \supseteq \alpha(Q_0)$. Therefore, any trajectory ψ in $Reach(\mathcal{J})$ that

leaves $\left\{q \in Q \;\middle|\; \alpha(q) \in Reach(\widetilde{\Gamma}/\widetilde{\mathcal{J}})\right\}$ contains a transition (q, σ, q^+) such that $\alpha(q) \in Reach(\widetilde{\Gamma}/\widetilde{\mathcal{J}})$ and $\alpha(q^+) \notin Reach(\widetilde{\Gamma}/\widetilde{\mathcal{J}})$. If $(q, \sigma, q^+) \in \Delta$, then $\sigma \in \widetilde{\Gamma}'(\alpha(q))$. However, $\sigma \notin \widetilde{\Gamma}(\alpha(q))$, because $\alpha(q^+) \notin Reach(\widetilde{\Gamma}/\widetilde{\mathcal{J}})$. This would only be possible if $\widetilde{\Gamma}$ were not the maximally-permissive supervisor (which it is), so no such trajectory ψ can exist. Thus, $\alpha(Reach(\mathcal{J})) \subseteq Reach(\widetilde{\Gamma}/\widetilde{\mathcal{J}})$. □

The supervisor $\widetilde{\Gamma}$ obtained in Theorem 5.7 represents an optimistic restriction on the events that can occur in the hybrid system $\mathcal{H}$ based on its current state. This restriction may not actually be imposed by the continuous dynamics of $\mathcal{H}$. Overapproximating the set of controllable events in $\widetilde{\mathcal{J}}$ and applying Algorithm 3.2 produces an upper bound on the set of abstract states that can be made to satisfy the θ by disabling events. If $\mathcal{H}$ satisfies θ, this in turn provides an upper bound on the set of reachable states in $\mathcal{J}$. These upper bounds lead to the following results:

Corollary 5.8 (falsification due to discrete dynamics)**.** *Consider $\mathcal{H}$, θ, $\widetilde{\mathcal{J}}$, and $\widetilde{\Gamma}$ as in Theorem 5.7. If $\widetilde{\Gamma}/\widetilde{\mathcal{J}}$ does not satisfy θ, then $\mathcal{H}$ does not satisfy θ.*

Proof. If $\widetilde{\Gamma}/\widetilde{\mathcal{J}}$ does not satisfy θ, then there are initial states $\widetilde{Q}_0$ that cannot be made to satisfy θ. Therefore, the corresponding initial states in X_0 also do not satisfy θ, so $\mathcal{H}$ does not satisfy θ. □

Corollary 5.9 (falsification due to hybrid dynamics)**.** *Consider $\mathcal{H}$, θ, $\widetilde{\mathcal{J}}$, and $\widetilde{\Gamma}$ as in Theorem 5.7. If any solution to $\mathcal{H}$ contains a jump (x, x^+) such that $\alpha(x^+) \notin Reach(\widetilde{\Gamma}/\widetilde{\mathcal{J}})$, then $\mathcal{H}$ does not satisfy θ.*

Proof. If the jump (x, x^+) occurs in a solution to $\mathcal{H}$, then $x^+ \in Reach(\mathcal{J})$. If $\alpha(x^+) \notin Reach(\widetilde{\Gamma}/\widetilde{\mathcal{J}})$, then $\alpha(Reach(\mathcal{J})) \not\subseteq Reach(\widetilde{\Gamma}/\widetilde{\mathcal{J}})$, so $\mathcal{H}$ does not satisfy θ. □

Corollaries 5.8 and 5.9 lead to slightly different conclusions regarding why $\mathcal{H}$ does not satisfy θ. Corollary 5.8 implies that there is an error in the discrete

automation logic that causes θ to fail. This is because there does not exist *any* restriction on the enabled events that the continuous dynamics might impose in order to satisfy θ. Corollary 5.9 implies that the interaction between the discrete and continuous dynamics causes $\mathcal{H}$ to violate θ. In this case, the error may lie in the continuous dynamics, the discrete dynamics, or both.

The results in this section combine to form Algorithm 5.1, which can be applied to falsify CIR specifications in SHCS. It is important to note that checking whether or not $Q_B \cap \mathit{Reach}(\mathcal{J}) \neq \emptyset$ in the algorithm may not terminate in finite time, as it involves solving the hybrid systems reachability problem. This step can be replaced with a conservative approximation (or skipped altogether) to avoid this issue, at the expense of returning ''Unknown'' in some cases when it could be shown that ''$\mathcal{H}$ does not satisfy θ''.

Algorithm 5.1: Falsification of CIR specifications.

Input : SHCS $\mathcal{H}$, DJS $\mathcal{J} = (Q, \Sigma, \Delta)$, LTS abstraction $\widetilde{\mathcal{J}} = (\widetilde{Q}, \Sigma, \widetilde{\Delta}) \succeq \mathcal{J}$ with abstraction function $\alpha(q) = s$, CIR specification $\theta = \mathsf{AG}\left(\bigwedge_I p_i \wedge \bigwedge_J \mathsf{EF}\left(p_j\right)\right)$

Output: ''$\mathcal{H}$ does not satisfy θ'' or ''Unknown''

$\Sigma_c \leftarrow \Sigma$

$\widetilde{\Gamma} \leftarrow$ Apply Algorithm 3.2 to $(\widetilde{\mathcal{J}}, \Sigma_c, \theta)$

if $\widetilde{\Gamma}/\widetilde{\mathcal{J}}$ *does not satisfy* θ **then**

  return *''$\mathcal{H}$ does not satisfy θ''*

end

$\widetilde{Q}_B \leftarrow \widetilde{Q} \setminus \mathit{Reach}(\widetilde{\Gamma}/\widetilde{\mathcal{J}})$

$Q_B \leftarrow \left\{ q \in Q \;\middle|\; \alpha(q) \in \widetilde{Q}_B \right\}$

if $Q_B \cap \mathit{Reach}(\mathcal{J}) \neq \emptyset$ **then**

  return *''$\mathcal{H}$ does not satisfy θ''*

else

  return *''Unknown''*

end

5.3.2 Refining the Initial Abstraction

If Algorithm 5.1 returns ‘‘Unknown’’, it may be possible to falsify θ by reapplying Algorithm 5.1 with a refined abstraction $\widetilde{\mathcal{J}}'$ such that $\widetilde{\mathcal{J}} \succeq \widetilde{\mathcal{J}}' \succeq \mathcal{J}$. If partial information is known about the implicit supervisor Γ for $\mathcal{J}$, i.e., that $\sigma_d \notin \Gamma(q_d)$ for some $q_d \in Q$, this can be used to refine the abstraction. Consider a supervisor Γ' which is not less permissive than Γ, meaning that Γ' represents partial information about which events are actually disabled by Γ in $\mathcal{J}$. Then $\widetilde{\mathcal{J}} \succeq \alpha(\Gamma')/\widetilde{\mathcal{J}} =: \widetilde{\mathcal{J}}' \succeq \mathcal{J}$. Applying Algorithm 5.1 with $\widetilde{\mathcal{J}}'$ (instead of $\widetilde{\mathcal{J}}$) results in $\widetilde{Q}_B{}' \supseteq \widetilde{Q}_B$, so that $Q_B{}' \supseteq Q_B$, therefore it is possible to change the conclusion from ‘‘Unknown’’ to ‘‘$\mathcal{H}$ does not satisfy θ’’.

5.4 Examples

5.4.1 Reduction to Reachability Verification

For an invariance specification $\mathsf{AG}(p)$, which lies in the intersection of CIR and ACTL*, applying the methods presented in this chapter reduces to checking for reachability of a state in which $\neg p$ holds. Consider the hybrid system:

$$\begin{aligned} x &= (z, s, \tau)^{\mathsf{T}} \in \mathbb{R} \times \{0,1\} \times [0,1] =: X \\ f(x) &= (z, 0, 1)^{\mathsf{T}} \\ g(x) &= (z, \rho(z), 0)^{\mathsf{T}} \\ \rho(z) &= z \leq 10 \\ D &= \{x \mid \tau = 1\} \\ C &= X \setminus D \end{aligned}$$

and the initial state $x_0 = (4, 0, 0)^{\mathsf{T}}$. The initial abstraction $\widetilde{\mathcal{J}} = (\widetilde{Q}, \Sigma, \widetilde{\Delta})$ has two states and two events, with:

$$\begin{aligned} \widetilde{Q} &= \{0, 1\} \\ \Sigma &= \{0, 1\} \\ \widetilde{\Delta} &= \{(\widetilde{q}, \sigma, \widetilde{q}^+) \mid \widetilde{q}^+ = \sigma\} \\ \widetilde{Q}_0 &= \{0\} \end{aligned}$$

$\mathcal{H}$ does not satisfy the specification $\mathsf{AG}(\neg s)$. Applying Algorithm 3.2 with $\Sigma_c = \Sigma$ returns the supervisor $\widetilde{\Gamma}$ such that $\sigma = 1 \notin \widetilde{\Gamma}(\widetilde{q} = 0)$, which results in $\mathit{Reach}(\widetilde{\Gamma}/\widetilde{\mathcal{J}}) = \{0\}$. From the continuous flow dynamics, z clearly increases to $+\infty$ from x_0. Therefore, there exists some jump (x, x^+) for which $z > 10$, so that $s^+ = 1$, and $\alpha(x^+) \notin \mathit{Reach}(\widetilde{\Gamma}/\widetilde{\mathcal{J}})$. From Corollary 5.9, $\mathcal{H}$ does not satisfy θ.

5.4.2 Multiple Reachability Requirements

We now present an example in which multiple reachability requirements are to be enforced. Consider the system:

$$
\begin{aligned}
x &= (z, u, s_1, s_2, s_3, \tau)^{\mathsf{T}} \in \mathbb{R} \times \mathbb{R} \times \{0,1\}^3 \times [0,1] =: X \\
f(x) &= \begin{pmatrix} z+u \\ 0 \\ \mathbf{0} \\ 1 \end{pmatrix} \\
G(x) &= \left(\left\{ \begin{pmatrix} u^+ \\ s^+ \end{pmatrix} \;\middle|\; \exists r \in \rho(z) : \begin{array}{rcl} z & & \\ u^+ & = & {s_1}^+ - {s_2}^+ + {s_3}^+ \\ {s_1}^+ & = & r_1 \\ {s_2}^+ & = & r_2 \\ {s_3}^+ & = & r_3 \wedge \neg(r_1 \vee r_2) \\ 0 & & \end{array} \right\} \right) \\
\rho(z) &= \begin{pmatrix} z < 0 \\ z > 0 \\ \{0,1\} \end{pmatrix} \\
D &= \{x \mid \tau = 1\} \\
C &= X \setminus D
\end{aligned}
$$

with initial state $x_0 = \mathbf{0}$, and the specification $\mathsf{AG}(\mathsf{EF}(s_1) \wedge \mathsf{EF}(s_2))$. The initial abstraction $\widetilde{\mathcal{J}}$ has eight states and six possible events:

$$
\begin{aligned}
\widetilde{Q} &= \{0,1\}^3 \\
\Sigma &= \left\{ \sigma \in \{0,1\}^3 \mid \neg(\sigma_1 \wedge \sigma_2) \right\} \\
\widetilde{\Delta} &= \{ (\widetilde{q}, \sigma, \widetilde{q}^+) \mid ({\widetilde{q}_1}^+ = \sigma_1) \wedge ({\widetilde{q}_2}^+ = \sigma_2) \wedge ({\widetilde{q}_3}^+ = (\sigma_3 \wedge \neg(\sigma_1 \vee \sigma_2))) \} \\
\widetilde{Q}_0 &= \left\{ (0,0,0)^{\mathsf{T}} \right\}
\end{aligned}
$$

Applying Algorithm 3.2 with $\Sigma_c = \Sigma$ returns the supervisor $\widetilde{\Gamma}$ such that $\widetilde{\Gamma}(\widetilde{q}) = \Sigma$ for all $\widetilde{q} \in \widetilde{Q}$, for which $\mathit{Reach}(\widetilde{\Gamma}/\widetilde{\mathcal{J}}) = \widetilde{Q}$. As a result, Algorithm 5.1 returns ‘‘Unknown’’.

We now attempt to refine the abstraction as outlined in Section 5.3.2. The only way $\mathcal{H}$ can leave the set $\{q \in Q \mid z = 0\}$, and therefore satisfy either of the reachability requirements, is by an event such that $\sigma_3 = 1$. This causes a

jump to $x = (0, 1, 0, 0, 1, 0)^{\mathsf{T}}$, after which z increases to ~ 1.7 before the next sample is taken. After this point, $\dot{z} > 0$, because $u \in [-1, 2]$. Therefore, for any state x that is reachable in $\mathcal{J}$ such that $z > 0$, no event with $\sigma_1 = 1$ can occur. For any reachable state in $\mathcal{J}$, $(s_2 = 1) \implies (z > 0)$, so this partial information about the implicit supervisor yields $\Gamma' : Q \rightrightarrows \Sigma$ given by:

$$\Gamma'(q) \coloneqq \begin{cases} \{\sigma \in \Sigma \mid \neg\sigma_1\} & s_2 = 1 \text{ holds in } q \\ \Sigma & \text{otherwise} \end{cases}$$

Algorithm 3.2 fails to compute a supervisor that enforces the θ in $\alpha(\Gamma')/\widetilde{\mathcal{J}}$, so Algorithm 5.1 returns ''$\mathcal{H}$ does not satisfy θ''. Furthermore, because applying Algorithm 5.1 with the initial abstraction returned ''Unknown'', we conclude that the specification fails due to the hybrid dynamics, and not purely because of the discrete dynamics; see the discussion in Section 5.3.2.

5.4.3 Liquid Holding Tank

The intended application of Algorithm 5.1 is to industrial control and automation systems with non-trivial discrete dynamics. In this case, some errors in the discrete logic can be uncovered without having to resort to expensive hybrid system reachability approximations. Consider the tank example from Section 2.3. The initial abstraction $\widetilde{\mathcal{J}}$ has 64 states and 12 possible events:

$$
\begin{aligned}
\widetilde{Q} &= \{0,1\}^6 \\
\Sigma &= \left\{\sigma \in \{0,1\}^4 \;\middle|\; \neg(\sigma_1 \wedge \sigma_2)\right\} \\
\widetilde{\Delta} &= \left\{ (\widetilde{q}, \sigma, \widetilde{q}^+) \;\middle|\; \begin{array}{rcl} \widetilde{q}_1{}^+ &=& \neg\widetilde{q}_5 \wedge \sigma_1 \\ \widetilde{q}_2{}^+ &=& \neg\widetilde{q}_6 \wedge \sigma_2 \\ \widetilde{q}_3{}^+ &=& \widetilde{q}_1{}^+ \vee (\widetilde{q}_3 \wedge \sigma_1) \\ \widetilde{q}_4{}^+ &=& \widetilde{q}_2{}^+ \vee (\widetilde{q}_4 \wedge \sigma_2) \\ \widetilde{q}_5{}^+ &=& (\widetilde{q}_1{}^+ \wedge \sigma_3) \vee \widetilde{q}_5 \\ \widetilde{q}_6{}^+ &=& (\widetilde{q}_2{}^+ \wedge \sigma_4) \vee (\widetilde{q}_6 \wedge \sigma_2) \end{array} \right\} \\
\widetilde{Q}_0 &= \{\mathbf{0}\}
\end{aligned}
$$

The specification we wish to test is that each of the discrete variables can reach both values, 0 and 1:

$$\mathsf{AG}\left(\bigwedge_{\{1\ldots6\}} (\mathsf{EF}(\widetilde{q}_j) \wedge \mathsf{EF}(\neg\widetilde{q}_j))\right)$$

The specification is falsified, as Algorithm 3.2 fails to compute a supervisor that enforces θ when applied to $\widetilde{\mathcal{J}}$ in Algorithm 5.1; this indicates that there is an error strictly in the discrete logic, based on the discussion at the end of Section 5.3.1 regarding Corollary 5.8. In this example, the specification fails for the following reasons:

- The logic for the high-level acknowledgement is

  $$\widetilde{q}_5{}^+ = (\widetilde{q}_1{}^+ \wedge \sigma_3) \vee \widetilde{q}_5$$

  so that once the alarm is acknowledged a single time, the acknowledgement remains in place; this violates $\mathsf{AG}(\mathsf{EF}(\neg\widetilde{q}_5))$.

- The only way to prevent this is to disable all events such that $\sigma_3 = 1$ in every state such that $\widetilde{q}_1{}^+ = 1$; this clearly violates $\mathsf{AG}(\mathsf{EF}(\widetilde{q}_5))$.

As a result, even the relaxed specification $\mathsf{AG}(\mathsf{EF}(\widetilde{q}_5) \wedge \mathsf{EF}(\neg\widetilde{q}_5))$ fails. Note,

however, that each of the specifications $\mathsf{AG}(\mathsf{EF}(\widetilde{q}_j)$ and $\mathsf{AG}(\mathsf{EF}(\neg\widetilde{q}_j)$ can be satisfied on its own in $\widetilde{\mathcal{J}}$, so Algorithm 5.1 returns ''Unknown'' in those cases. This highlights the need for multiple reachability requirements.

A simulation showing the impact of the faulty logic was shown previously in Figure 2.3, in which the tank overflowed. The alarm works as intended and prevents overflow until it is acknowledged shortly before $t = 400$. After this, the level eventually increases past $z = 8$ without activating the alarm and shutting off the inlet flow, until finally overflowing the tank before $t = 700$. The specification $\mathsf{AG}(\mathsf{EF}(\widetilde{q}_6) \wedge \mathsf{EF}(\neg\widetilde{q}_6))$ (related to the low-level acknowledgement) is not proven to fail by applying Algorithm 5.1. Using the logic for ${s_6}^+$ as a template to redefine the faulty ${s_5}^+$ logic in (2.3) to

$$g_{s_5}(r, s) \coloneqq ({s_1}^+ \wedge r_3) \vee (s_5 \wedge r_1)$$

(where s_5 was changed to $s_5 \wedge r_1$) yields the simulation shown in Figure 5.1, with no overflow. After redefining the logic, Algorithm 5.1 no longer falsifies the specification $\mathsf{AG}(\mathsf{EF}(\widetilde{q}_5) \wedge \mathsf{EF}(\neg\widetilde{q}_5))$.

This example shows how some errors in SHCSs that cause CIR specifications to fail can be uncovered by applying the methods described in this chapter. In this example, the error is detected without performing any reachability approximations with the full hybrid system.

5.5 Summary

In this chapter, we have presented an algorithm that can be used to detect errors in the automation logic of sample-and-hold control systems, subject to combined invariance and reachability specifications. The algorithm relies on the application of supervisory control to a finite abstraction of an infinite-state

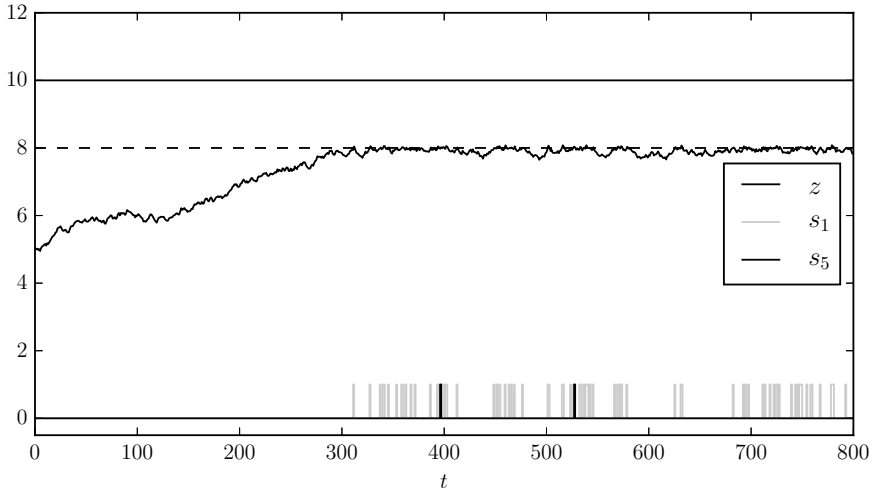

Figure 5.1: A simulation of the modified liquid holding tank, without overflow; the same random input sequence was applied as in Figure 2.3.

hybrid system to guarantee that a specification fails. The main advantage of the method is that it can detect some classes of errors without relying on costly hybrid system reachability computations. In the case that analyzing the discrete logic alone does not show an error, existing techniques for approximating reachable sets in hybrid systems can be applied as part of the algorithm.

Much of the previous work in the analysis of general hybrid systems has focused on verification rather than falsification, and the work in both directions typically focuses on invariance or reachability alone. The small examples we provided show how invariance and reachability can be treated simultaneously. They also demonstrate how it can be useful to falsify such properties when analyzing the behavior of hybrid control systems.

Chapter 6

Formal Analysis of Large-Scale Control Systems

6.1 Analysis of Logical Control Systems

A significant body of research has focused on verifying the correctness of logical control systems. This includes modeling programmable logic controllers (PLCs) so that a formal specification of the desired behavior can be verified [Moo94; RK98; Can+00; GdF08; BBK10; DBF13]. The basic approach consists of the following steps:

1. Model the (discrete) dynamical behavior of the PLC as a (finite) state transition system.

2. Specify the desired behavior as a temporal logic formula.

3. Apply model checking to determine whether or not the model fulfills the specification.

One of the main limitations of this approach is the *state-explosion problem* [CG87], which refers to the fact that the problem size increases rapidly as

the number of discrete variables increases.

To overcome the state-explosion problem, an *implicit Boolean state-space model* was proposed that describes the discrete logic [PB97]. The implicit model represents the constraints on the values of the current state of the system, the inputs, and a potential next state of the system that must be satisfied for that transition (from the current state, to the next state, in response to the input) to occur. The implicit model is converted to an integer programming problem, which is checked for feasibility. Some of the specifications to which implicit model checking is applied are invariance specifications, $\mathsf{AG}(p)$, which require that the system never leaves a particular fixed set of good states. Indeed, if there does not exist a transition from *any* state that leads to a bad state, then the bad states are all unreachable, and the specification is verified. This reasoning, however, does not capture the fact that such a transition is only problematic if it starts in a reachable state. The result is that the verification results are overly conservative, and may fail to verify invariance specifications in systems that are actually correct. More importantly, the approach is not sufficient for verifying reachability specifications (equivalently, falsifying invariance specifications). This is because checking for the existence of a transition that leads into a target set is not the same as checking for a sequence of transitions, starting in the initial state, that leads to the target set. The dramatic performance improvement that is observed when applying implicit model checking instead of symbolic model checking results from approximating the solution to the verification problem, not from the fact that an integer-programming-based formulation is somehow more efficient. For this reason, the standard approach based on symbolic model checking remains the most widely used method for analyzing logical control systems.

Bounded model checking (BMC) was developed around the same time as

implicit model checking, and has been applied as an alternative to traditional model checking to overcome the state-explosion problem [Bie+03]. The idea is to check for the existence of a path, up to a fixed (bounded) length, that violates the given property; this is done by encoding the search for such a path as a Boolean satisfiability problem, then solving that problem. The important difference between BMC and implicit model checking is that BMC checks for the existence of a finite sequence of transitions, starting in the initial state, that can verify (or falsify) a specification. This approach can be applied to falsify invariance specifications, by proving the existence of a path that violates the specifications. BMC can be very efficient for similar reasons to those described in [PB97], but the fact that it is mainly limited to falsifying invariance specifications (more generally, the same types of specifications that have been targeted for falsification in hybrid systems, as in Section 5.1) prevents it from being applied as broadly as symbolic model checking.

The standard verification approach based on model checking has previously been applied in the chemical processing industry [Moo+92; Pro+97; Bau+04; KM11]. In each case, if the verification process succeeds, then the control system is deemed to be correct, and if it fails, then the system is analyzed further (including using the counterexample trace from the model checking tool) to determine the reason for failure. However, the authors of these works do not directly account for the fact that the closed-loop system is a hybrid system, and that the discrete model is therefore only an abstraction of the system (not an exact model). The theoretical restrictions on the class of specifications that can be verified using such an abstraction of a hybrid system is described in [CK01].

While many of the results in each of these works are from the class of specifications outlined in [CK01], they also report results that are not guaranteed.

Notably, in [Moo+92], the authors claim to verify reachability requirements, in [Pro+97], the authors claim to falsify invariance requirements (and rely on manually interpreting counterexamples to support the claim), in [Bau+04], the authors claim to verify reachability requirements, and in [KM11], the authors claim to falsify invariance specifications; none of these results can be guaranteed by applying model checking to an abstract model, without analyzing the hybrid dynamics.

Similar approaches that also attempt to address the hybrid dynamics have been applied [Kow+99; Bal+05; LTS06]. In [Kow+99] and [LTS06], the authors approximate hybrid systems as timed automata and claim that the verification/falsification results hold in the actual system. For this to be true, they require that the timed automata are conservative approximations of the hybrid dynamics, but they do not provide a rigorous method to show conservativeness. In [Bal+05], the authors apply an approach described in [Cla+03] to a batch reactor system. The approach does yield correct verification results for the class of specifications outlined in [CK01], but requires detailed knowledge of the continuous process dynamics.

The main contribution of this chapter is to demonstrate the application of both verification and falsification to analyze the behavior of large-scale chemical plant automation systems. In addition, we address the limitations (imposed by the hybrid dynamical nature of the systems) on the classes of specifications that can be verified or falsified. The verification methods are the same as those that have been applied previously; the difference is the particular specifications we verify, and what the result indicates about the system. The falsification methods we apply were developed in Chapter 5, and allow a broader class of specifications to be addressed than in previous work. For both verification and falsification, we use a model-reduction technique to

mitigate the state-explosion problem; the validity of the results is maintained, and the approach allows us to address industrial-scale systems. We also provide a set of specifications that can be used with the approach we describe to analyze a general automation system. We demonstrate our results through a series of illustrative examples, and report computational results from test cases provided by The Dow Chemical Company.

In Section 6.2, we introduce discrete automation systems and some specifications that they should satisfy. In Section 6.3, we show how to model standard control and automation code (written in the PLC programming language Structured Text). In Section 6.4, we describe the formal analysis methods that are applied to determine whether or not a system meets the specifications. In Section 6.5, we introduce a way to approximate large systems to avoid the state-explosion problem. In Section 6.6, we apply the methods to an industrial case study.

6.2 Discrete Logic in Chemical Plants

6.2.1 Dynamics

We consider the hybrid systems of the form (2.1) produced by applying sample-and-hold control to continuous chemical processes, as in Chapter 2. In such a system, the discrete control and automation logic is contained in the function $g_s : \{0,1\}^{n_r} \times \{0,1\}^{n_s} \to \{0,1\}^{n_s}$, which updates logical state of the control system in response to a sequence of discrete readings during operation. The discrete readings come from the process (for example, by checking whether or not a continuous state variable is within a desired operating range) and the operators (in the form of discrete toggles on the operator's control panel). Because the discrete logic interacts with all the pieces of a chemical plant

(the process, the continuous control system, and the operators), it is a critical component of the overall control system.

The dynamics introduced by the discrete logic in a control system are fundamentally different than those introduced by the continuous control logic. A poorly-tuned PID controller will result in degraded performance of the control system, but that performance is usually qualitatively similar to the performance that would be achieved using a well-tuned controller (i.e., the system is still stable, but converges to the set point more slowly). Minor changes in the discrete logic, however, can produce qualitatively different behavior in the plant (i.e., a piece of equipment no longer turns on under the correct conditions).

6.2.2 Process-Independent Tests

When checking the discrete automation logic in an SHCS, the specifications are in terms of the discrete state variables s. The atomic propositions from Section A.3 are then relational expressions involving those variables. Using these atomic propositions, temporal logic formulas can be constructed to describe certain properties of the desired system behavior.

Specifying the entire desired behavior of the closed-loop system is difficult for the same reasons that defining the control logic correctly is difficult. This leads to the goal of automatically generating specifications that describe part of the overall requirements that a control system must meet. We refer to these as process-independent tests (PITs), because they do not relate to the underlying chemical process (and can therefore be generated without knowledge of the process). Some PITs are listed in Table 6.1.

A variable lock is the situation in which one of the discrete state variables becomes stuck in either value, 0 or 1, without the possibility of ever changing. The requirement to avoid variable locks specifies that none of the output

Table 6.1: Process-independent tests.

Property to test	Specification
Avoid variable locks.	$\mathsf{AG}(\mathsf{EF}(s_i = 0) \wedge \mathsf{EF}(s_i = 1)) \quad i \in 1 \ldots n_s$
All operating modes are reachable.	$\mathsf{AG}(\mathsf{EF}(s_j = 1)) \quad j \in J$
Operating modes are mutually exclusive.	$\mathsf{AG}(\sum_{j \in J} s_j = 1)$
Relevant logic.	$\mathsf{EF}(s_i \neq s_i') \quad i \in 1 \ldots n_s$

variables should ever become locked. This specification does not require that the variable ever changes value, only that the logic does not strictly prevent that from happening. For example, in the ideal case that a threshold alarm is never tripped, the corresponding discrete state variable is always 0. This is not a variable lock unless there is no way the alarm would *ever* turn on (even in response to the threshold being violated).

Automation logic often involves explicitly defined operating modes, such as startup, react, and shutdown. This is described in detail in [PB00]. The system must always be in one (and only one) operating mode, which is specified by the requirement that the corresponding variables sum to 1. In addition to this, the control system should always be capable of reaching each of the operating modes, which is similar to the variable lock specification. As in the case of variable locks, the requirement that the operating modes remain reachable does not mean that any of them is actually reached, only that it is always possible to reach each of them. This specification is similar to the reachability requirement in [PB00] (feasibility of a sequence).

The test for irrelevant logic is slightly different than the other specifications. The expected outcome is that removing part of the automation logic should affect the behavior of the system in some way. This is done on a per-variable basis by introducing a new variable s_i' with the same assignment logic as s_i, then removing part of the assignment logic for s_i'. The reachability specification $\mathsf{EF}(s_i \neq s_i')$ specifies that there should be some reachable state in which the

original and modified variables have different values. If the specification is satisfied, then the logic that was removed is relevant (to the behavior of the system). If the specification is not satisfied, then the logic that was removed is irrelevant, and can be removed without modifying the system's behavior. It is common practice to intentionally include redundant (irrelevant) terms to clarify the logic, but sometimes this behavior is not intended.

6.3 Modeling PLC Programs

The methods we describe in this chapter apply to any control system of the form (2.1). PLCs are often used in the chemical processing industry to implement the discrete logic of SHCSs. For this reason, we focus on PLC programs as the target of our analysis, and to give concrete examples. This topic has been studied extensively in the past; the key departure from previous work is that we account for the fact that the PLC forms part of a larger hybrid system which restricts the sequence of inputs that the PLC may receive.

PLCs operate by repeating the following steps in a non-terminating loop:

1. Input scan: inputs to the PLC program (continuous and discrete values) are read from the plant.

2. Evaluate logic: the PLC logic is executed with the new inputs to update the outputs.

3. Output scan: the new outputs are applied to the plant.

In relation to the model (2.1), step 1 corresponds to ρ, step 2 corresponds to G, and step 3 corresponds to the jump $x^+ \in G(x)$.

Standard IEC 61131-3 [IEC13] defines the programming interface to PLCs. It describes two graphical languages (Ladder Diagram and Function Block

Diagram), along with two textual languages (Instruction List, and Structured Text). Structured Text (ST) is the high-level text-based programming language defined in the standard, and is the one that we focus on.

6.3.1 Translation to a Formal Model

We address PLC programs defined using a restricted subset of the ST language, similar to previous work [RK98; GdF08]. We assume the following restrictions:

- All assignments are to elementary Boolean or numeric variables.
- Only certain function blocks are used.
- There are no loops (other than the PLC's loop over the entire program).
- There are no jumps (i.e., the program is a single routine).

That is, the ST program is a sequence of assignments, along with conditional branching.

Every variable that is assigned a value is an *output* of the program. Any variable that is not assigned a value anywhere in the program is an *input*. The output variables are the values that the control logic sets in order to influence the behavior of the plant. Variables that are not assigned values anywhere in the program are assumed to be readings from the plant, and therefore act as inputs to the PLC logic.

Consider the ST program:

```
s1 := ABS(z1 - z2) > 0;
s2 := s3 OR r1;
s3 := s2 AND r2;
```

which produces the model:

$$
\begin{aligned}
x &= (z_1, z_2, s_1, s_2, s_3, \tau)^{\mathsf{T}} \in \mathbb{R}^2 \times \{0,1\}^3 \times [0,T] =: X \\
F(x) &= \begin{pmatrix} F_z(z) \\ \mathbf{0} \\ 1 \end{pmatrix} \\
G(x) &= \begin{pmatrix} z \\ \left\{ s^+ \;\middle|\; \exists r \in \rho(z) : \begin{array}{l} {s_1}^+ = r_3 \\ {s_2}^+ = s_3 \vee r_1 \\ {s_3}^+ = {s_2}^+ \wedge r_2 \end{array} \right\} \\ 0 \end{pmatrix} \\
\rho(z) &= \begin{pmatrix} \{0,1\} \\ \{0,1\} \\ |z_1 - z_2| > 0 \end{pmatrix} \\
D &= \{x \mid \tau = T\} \\
C &= X \setminus D
\end{aligned}
$$

The expressions $\rho_1 = \{0,1\}$ and $\rho_1 = \{0,1\}$ indicate the ρ_1 and ρ_2 are external inputs to the program, and might either be 0 or 1, regardless of the continuous state z. We have not explicitly defined the continuous dynamics F_z, and there are no continuous control variables u. Note that the implicit definition of g_s that arises when the assignment of one variable depends on a previous assignment in the program (as shown above for ${s_3}^+$, which depends on ${s_2}^+$) can always be converted to an explicit definition; this is described in more detail in [RK98; PB00]. In this example, the term ${s_3}^+ = {s_2}^+ \wedge r_2$ would be replaced with ${s_3}^+ = (s_3 \vee r_1) \wedge r_2$.

6.3.2 Function Blocks

An important function block to handle is the delay timer, due to its frequent use in industrial control logic. Delay timers are used to prevent a variable from switching value unless a measurement has returned the same value for a certain number of consecutive sample intervals. We handle a delay timer

function:

```
value := DT(reading, delay, default);
```

where `delay` is the number of samples to wait (integer), `default` is the base value (Boolean), `reading` is the measurement being monitored (Boolean), and `value` is the value returned by the timer (Boolean). The timer behaves in the following way: if `(reading <> default)` (''`<>`'' is the inequality operator in Structured Text) held for the last `delay` samples (including the current sample), then the timer returns `reading`; otherwise, the timer returns `default`. If `default` is `FALSE`, then it is a ''delay-on timer'', and if `default` is `TRUE`, then it is a ''delay-off timer''.

A delay timer keeps track of an internal state, `duration`, which is the number of consecutive preceding samples in which `(reading <> default)`. Instead of modeling this behavior, we add an additional input to the model, which represents whether or not `(duration >= delay)`. This abstraction is similar to the approach in [PB97] of treating all delay timers as unit delays; we give a detailed justification in Section 6.4.1.

6.4 Formal Analysis

6.4.1 Abstraction as a Labeled Transition System

In order to analyze the automation logic of an SHCS $\mathcal{H}$ as in (2.1), we rely on the finite deterministic LTS model:

$$(S, R, \Delta) \tag{6.1}$$

where:

$$\begin{aligned} S &:= \{0,1\}^{n_s} \\ R &:= \{0,1\}^{n_r} \\ \Delta &:= \{(s,r,s^+) \mid s^+ = g_s(r,s)\} \end{aligned}$$

As described in Section 5.2, the LTS (6.1) is an abstraction of the discrete jump system that corresponds to $\mathcal{H}$. The abstraction models the response of the discrete logic to any of the possible sequences of inputs. The result is that the abstraction overapproximates the behavior of the closed-loop system; in the actual system, whether or not a particular sequence of inputs can occur depends on the continuous dynamics.

The continuous dynamics and unmodeled discrete dynamics (such as the internal `duration` state of a delay timer from Section 6.3.2) impact the sequences of inputs that can occur in the same way that a supervisor disables events in supervisory control theory. An event r is disabled in state s of the LTS abstraction if there is no solution to the original SHCS that produces $r \in \rho(z)$ while the discrete part of the state is equal to s. This is the idea of an implicit supervisor in Definition 5.2. To account for this, we treat each of the input readings $r \in R$ in the LTS abstraction as a controllable event, meaning it is possible that r is prevented from occurring in a state s by some unmodeled behavior of the original SHCS. The result is that we set $R_c = R$ when analyzing the LTS in order to explore not only all possible sequences of inputs that might occur, but also all possible restrictions thereupon that the hybrid dynamics might impose. Treating the discrete values produced by unmodeled dynamics as inputs (rather than unrestricted state variables) is what allows for falsifying a broader class of specifications than in previous work, which is described in Section 6.4.3.

6.4.2 Verification

Given an SHCS and its LTS abstraction, it is possible to directly verify certain classes of specifications by analyzing the LTS. One such class of specifications is ACTL, which is described in Section A.3.1. For a specification that is not contained in this class, such as a CTL specification that includes E, verification of the LTS abstraction does not necessarily imply verification of the SHCS. This includes even simple reachability requirements such as $\mathsf{EF}(p)$.

Of the PITs defined in Table 6.1, the requirement of mutual exclusivity of the operating modes is an ACTL specification. Therefore, it is eligible for verification using the abstraction. In addition, the negation of a relevant logic specification, $\neg\mathsf{EF}(s_i \neq s_i{}') \equiv \mathsf{AG}(s_i = s_i{}')$, is an ACTL specification. Verifying $\mathsf{AG}(s_i = s_i{}')$ guarantees that the logic removed from $s_i{}'$ is irrelevant. The other specifications all include both invariance and reachability, so they cannot directly be verified by applying model checking to the LTS abstraction.

Algorithm 6.1 is a simplified version of the standard abstraction-based approach for verifying ACTL specifications in SHCSs [CK01; Cla+03]. If the verification fails, we do not attempt to refine the abstraction, as in [CK01; Cla+03], or interpret the counterexample, as in [Pro+97]. Refinement of the model requires a detailed model of the hybrid dynamics, which is often difficult to obtain and computationally costly to analyze for large industrial systems. Manual inspection of the counterexample amounts to informal abstraction refinement, which is difficult for the same reasons, and does not have the benefit of being algorithmically sound. The set of controllable events is not used because model checking alone (not supervisory control) is sufficient for verification; in the abstraction, all events are enabled, which guarantees that it is indeed an abstraction.

Algorithm 6.1: Verification of ACTL specifications.

Input : SHCS $\mathcal{H}$ and ACTL specification θ
Output: ''$\mathcal{H}$ satisfies θ'' or ''Unknown''
$(S, R, \Delta) \leftarrow$ LTS abstraction of $\mathcal{H}$ as in (6.1)
if (S, R, Δ) *satisfies* θ **then**
    return *''$\mathcal{H}$ satisfies θ''*
else
    return *''Unknown''*
end

Example: ACTL Specification Verified

Consider the automation logic depicted in Figure 6.1, which shows the desired paths through a set of operating modes. The desired behavior is enforced by

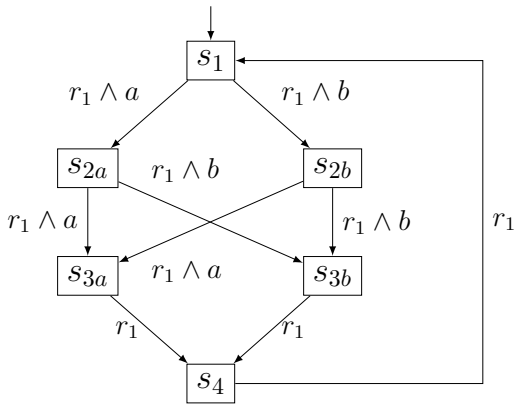

Figure 6.1: Sequence between operating modes.

the PLC program in Figure 6.2. In the initial state, $s_1 = 1$ and all of the other state variables are equal to 0. In this simple example, the inputs do not relate to the continuous state of the plant (r_1 and r_a are operator inputs to advance to the next operating mode and select ''recipe a'', respectively), so the discrete part of the automation system is decoupled from any continuous dynamics. The program corresponds to the following LTS:

```
reset := s_4 and r_1;

a := (not reset)
     and ((s_1 and r_1 and recipe_a) or a);
b := (not reset)
     and ((s_1 and r_1 and (not recipe_a)) or b);

s_4 := (s_4 and not r_1)
       or ((s_3a or s_3b) and r_1);

s_3a := (s_3a and not r_1)
        or ((s_2a or s_2b) and r_1 and a);
s_3b := (s_3b and not r_1)
        or ((s_2a or s_2b) and r_1 and b);

s_2a := (s_2a and not r_1)
        or (s_1 and r_1 and a);
s_2b := (s_2b and not r_1)
        or (s_1 and r_1 and b);

s_1 := (s_1 and not r_1)
       or reset;
```

Figure 6.2: Operating mode sequence logic.

$$
\begin{aligned}
s &= (s_1, s_{2a}, s_{2b}, s_{3a}, s_{3b}, s_4, s_a, s_b, s_r)^{\mathsf{T}} \\
r &= \begin{pmatrix} r_1 \\ r_a \end{pmatrix} \\
\Delta &= \left\{ (s, r, s^+) \;\middle|\; \begin{array}{rcl}
{s_1}^+ &=& (s_1 \wedge \neg r_1) \vee {s_r}^+ \\
{s_{2a}}^+ &=& (s_{2a} \wedge \neg r_1) \vee (s_1 \wedge r_1 \wedge {s_a}^+) \\
{s_{2b}}^+ &=& (s_{2b} \wedge \neg r_1) \vee (s_1 \wedge r_1 \wedge {s_b}^+) \\
{s_{3a}}^+ &=& (s_{3a} \wedge \neg r_1) \vee ((s_{2a} \vee s_{2b}) \wedge r_1 \wedge {s_a}^+) \\
{s_{3b}}^+ &=& (s_{3b} \wedge \neg r_1) \vee ((s_{2a} \vee s_{2b}) \wedge r_1 \wedge {s_b}^+) \\
{s_4}^+ &=& (s_4 \wedge \neg r_1) \vee ((s_{3a} \vee s_{3b}) \wedge r_1) \\
{s_a}^+ &=& (\neg {s_r}^+) \wedge ((s_1 \wedge r_1 \wedge r_a) \vee s_a) \\
{s_b}^+ &=& (\neg {s_r}^+) \wedge ((s_1 \wedge r_1 \wedge \neg r_a) \vee s_b) \\
{s_r}^+ &=& s_4 \wedge r_1
\end{array} \right\} \\
s_0 &= (1, 0, 0, 0, 0, 0, 0, 0, 0)^{\mathsf{T}}
\end{aligned}
$$

The PIT for mutually exclusive operating modes in this example is the specification $\mathsf{AG}(\sum_J(s_j) = 1)$, where $J = \{1, 2a, 2b, 3a, 3b, 4\}$, which is verified by

Algorithm 6.1.

The states described by $((s_{2a} \vee s_{2b}) \wedge s_a \wedge s_b)$ are stable according to the definition in [PB97], but lead to states in which $(s_{3a} \wedge s_{3b})$, which violates mutual exclusivity. As a result, implicit model checking does not verify the specification, even though it is satisfied. This is not caused by any hybrid dynamics, because the discrete system is decoupled from any continuous dynamics. The problem is that implicit model checking does not examine paths, starting in the initial state, consisting only of reachable states. The states in which $((s_{2a} \vee s_{2b}) \wedge s_a \wedge s_b)$ holds are not reachable, so the fact that they lead to bad states does not actually impact the specification. Note that if s_b is replaced by $\neg s_a$ in the assignment logic for variables s_{2b} and s_{3b} in the control program, then the system's behavior is unchanged, but implicit model checking correctly verifies the specification. This highlights the over-conservative nature of implicit model checking, which is what allows for the reduction in computational effort required.

Example: ACTL Specification Not Verified

Consider a batch reaction, $A + B \rightarrow C$. The PLC program

```
s_1 := N_c >= 0.99 * N_a0;
s_2 := N_a <= 0.05 * N_a0;
s_3 := s_1 and s_2;
s_3p := s_1;
```

monitors the extent of the reaction, and signals completion via discrete state variable s_3. The program corresponds to the following SHCS:

$$
\begin{aligned}
x &= \begin{pmatrix} N_A \\ N_B \\ N_C \\ s_1 \\ s_2 \\ s_3 \\ {s_3}' \\ \tau \end{pmatrix} \in \mathbb{R}^3 \times \{0,1\}^4 \times [0,T] =: X \\
f(x) &= \begin{pmatrix} -\frac{k}{V} N_A N_B \\ -\frac{k}{V} N_A N_B \\ \frac{k}{V} N_A N_B \\ \mathbf{0} \\ 1 \end{pmatrix} \\
g(x) &= \begin{pmatrix} N_A \\ N_B \\ N_C \\ \rho_1(z) \\ \rho_2(z) \\ {s_1}^+ \wedge {s_2}^+ \equiv \rho_1(z) \wedge \rho_2(z) \\ {s_1}^+ \equiv \rho_1(z) \\ 0 \end{pmatrix} \\
\rho &= \begin{pmatrix} N_C \geq 0.99 N_A(0) \\ N_A \leq 0.05 N_A(0) \end{pmatrix} \\
D &= \{ x \in X \mid \tau = T \} \\
C &= X \setminus D
\end{aligned}
$$

where $f(x)$ and $g(x)$ are functions (not set-valued maps), T, k, and V are fixed parameters, and the initial state is $x_0 = \Big(N_A(0), N_B(0), N_C(0), \mathbf{0}^\mathsf{T}, 0\Big)^\mathsf{T}$. The SHCS has the LTS abstraction:

$$
\begin{aligned}
s &= \begin{pmatrix} s_1 \\ s_2 \\ s_3 \\ {s_3}' \end{pmatrix} \\
r &= \begin{pmatrix} r_1 \\ r_2 \end{pmatrix} \\
\Delta &= \left\{ (s, r, s^+) \;\middle|\; \begin{array}{lcl} {s_1}^+ & = & r_1 \\ {s_2}^+ & = & r_2 \\ {s_3}^+ & = & {s_1}^+ \wedge {s_2}^+ \\ {s_3}'^+ & = & {s_1}^+ \end{array} \right\} \\
s_0 &= \mathbf{0}
\end{aligned}
$$

The variable ${s_3}'$ has been added to the program to test whether or not s_2 is relevant when it comes to checking for completion of the reaction. If the specification $\mathsf{AG}(s_3 = {s_3}')$ is satisfied by the abstraction (i.e., the ACTL specification is verified in the hybrid system), then s_2 is irrelevant. The specification is not satisfied, which means s_2 is not proven to be irrelevant by examining the abstraction. However, this does not guarantee that the hybrid system violates the specification, i.e., that s_2 is relevant. In fact, given the initial state $N_A(0) = 1$, $N_B(0) = 1$, $N_C(0) = 0$, a simple mass balance shows that $\forall t > 0 : (N_C(t) \geq 0.99 N_A(0)) \implies (N_A(t) \leq 0.05 N_A(0))$, so $\rho_1 \implies \rho_2$, which means s_2 is irrelevant in the assignment logic for ${s_3}^+$. There are other initial conditions for which s_2 is relevant. This demonstrates that while ACTL specifications can be verified in hybrid control systems, they cannot be falsified directly by analyzing the LTS abstraction.

6.4.3 Falsification

Falsification of combined invariance and reachability (CIR) specifications in SHCSs was described in Chapter 5. In addition to CIR specifications, any specification that is the negation of an ACTL specification can be falsified by

verifying the ACTL specification. Consider the specification $\mathsf{EF}(p)$, which is equivalent to $\neg\mathsf{AG}(\neg p)$; falsifying $\mathsf{EF}(p)$ is equivalent to verifying $\mathsf{AG}(\neg p)$ using Algorithm 6.1.

The PITs in Table 6.1 that are CIR specifications are the requirement to avoid variable locks, and the requirement that all operating modes remain reachable. In addition, negating the relevant logic specification results in $\mathsf{AG}(s_i = s_i')$, which is also a CIR specification. Falsifying $\mathsf{AG}(s_i = s_i')$ amounts to verifying the original relevant logic specification. Each of these CIR specifications can be falsified by applying the results from Chapter 5.

Algorithm 6.2 is a simplified version of Algorithm 5.1 that does not include any reachability search in the hybrid system. As with Algorithm 6.1, we omit the further analysis that involves the continuous dynamics, which was described in Chapter 5.

Algorithm 6.2: Falsification of CIR specifications (simplified).

Input : SHCS $\mathcal{H}$ and CIR specification θ
Output: "$\mathcal{H}$ does not satisfy θ" or "Unknown"
$(S, R, \Delta) \leftarrow$ LTS abstraction of $\mathcal{H}$ as in (6.1)
$R_c \leftarrow R$
$\Gamma \leftarrow$ Apply Algorithm 3.2 to $((S, R, \Delta), R_c, \theta)$
if $\Gamma/(S, R, \Delta)$ *does not satisfy* θ **then**
    return *"$\mathcal{H}$ does not satisfy θ"*
else
    return *"Unknown"*
end

Example: CIR Specification Falsified

Consider a batch reactor with a simple sequence of four operating modes shown in Figure 6.3:

1. Reset:
    - Product is removed from the reactor.

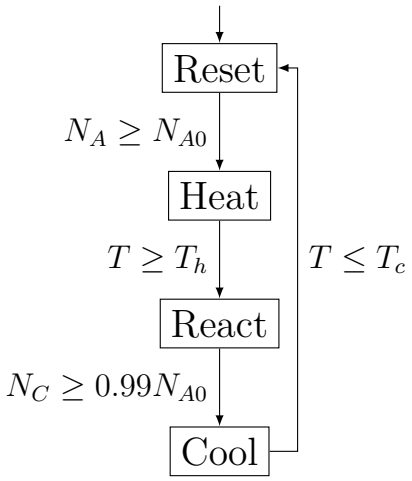

Figure 6.3: Operating mode sequence diagram for a batch reaction.

- New reactant is charged to the reactor.

2. Heat:

   - The reactor contents are heated to the required reaction temperature.

3. React:

   - The temperature is maintained at the required reaction temperature while the reaction takes place.

4. Cool:

   - The product is cooled for transfer.

and the same reaction as in Section 6.4.2, $A + B \rightarrow C$. The PLC program:

```
condition_cooled := mode_cool
                    and DT(T_reactor <= T_cold, 10, false);

condition_reacted := mode_react
                     and DT(N_C >= 0.99 * N_A0, 10, false);

condition_heated := mode_heat
                    and DT(T_reactor >= T_hot, 10, false);

condition_reset := mode_reset
                   and DT(N_A >= N_A0, 10, false);

mode_cool := (mode_cool and not (input_proceed and condition_cooled))
             or (mode_react and condition_reacted and input_proceed);

mode_react := (mode_react and not (input_proceed and condition_reacted))
              or (mode_heat and condition_heated and input_proceed);

mode_heat := (mode_heat and not (input_proceed and condition_heated))
             or (mode_reset and condition_reset and input_proceed);

mode_reset := (mode_reset and not (input_proceed and condition_reset))
              or (mode_react
                  and condition_cooled
                  and input_proceed and condition_reacted);
```

is designed to implement the sequence. The program corresponds to an SHCS with LTS abstraction:

$$
\begin{aligned}
s &= (s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8)^{\mathsf{T}} \\
r &= (r_1, r_2, r_3, r_4, r_5)^{\mathsf{T}} \\
\Delta &= \left\{ (s, r, s^+) \;\middle|\; \begin{array}{rcl}
{s_1}^+ &=& s_5 \wedge (r_6 \wedge r_1) \\
{s_2}^+ &=& s_2 \wedge (r_7 \wedge r_2) \\
{s_3}^+ &=& s_3 \wedge (r_8 \wedge r_3) \\
{s_4}^+ &=& s_4 \wedge (r_9 \wedge r_4) \\
{s_5}^+ &=& (s_5 \wedge \neg(r_5 \wedge {s_1}^+)) \vee ({s_2}^+ \wedge r_5) \\
{s_6}^+ &=& (s_2 \wedge \neg(r_5 \wedge {s_2}^+)) \vee ({s_3}^+ \wedge r_5) \\
{s_7}^+ &=& (s_3 \wedge \neg(r_5 \wedge {s_3}^+)) \vee ({s_4}^+ \wedge r_5) \\
{s_8}^+ &=& (s_4 \wedge \neg(r_5 \wedge {s_4}^+)) \vee ({s_1}^+ \wedge r_5 \wedge {s_2}^+)
\end{array} \right\} \\
s_0 &= (0, 0, 0, 0, 0, 0, 0, 1)^{\mathsf{T}}
\end{aligned}
$$

where the LTS variables relate to the ST variables according to Table 6.2.

The specification $\mathsf{AG}(\mathsf{EF}(s_1) \wedge \mathsf{EF}(s_2) \wedge \mathsf{EF}(s_3) \wedge \mathsf{EF}(s_4))$ comes from the PIT

Table 6.2: Mapping between model and Structured Text variable names.

DES	ST
s_1	`condition_cooled`
s_2	`condition_reacted`
s_3	`condition_heated`
s_4	`condition_reset`
s_5	`mode_cool`
s_6	`mode_react`
s_7	`mode_heat`
s_8	`mode_reset`
r_1	`(T_reactor <= T_cold)`
r_2	`(N_C >= 0.99 * N_A0)`
r_3	`(T_reactor >= T_hot)`
r_4	`(N_A >= N_A0)`
r_5	`proceed`
r_6	`duration_1 >= 10`
r_7	`duration_2 >= 10`
r_8	`duration_3 >= 10`
r_9	`duration_4 >= 10`

that each of the operating modes should always be reachable. The specification is a CIR specification, so Algorithm 6.2 can be applied to check if the SHCS violates it. Applying Algorithm 6.2 indicates that the specification is violated. This is explained by the condition to move from s_4 to s_1, which requires that $s_7 = 1$ (i.e., that the reactant is fully reacted). The variable s_7 is only set in the ‘‘React’’ mode, so it is always 0 in the ‘‘Cool’’ mode; this prevent the system from returning to the ‘‘Reset’’ mode after completing the reaction. Note that all the operating modes are reachable from the initial state, s_0, and the problem is that they do not always *remain* reachable. Thus, detecting this behavior requires analyzing a CIR specification that combines invariance and reachability. Finally, the result from Algorithm 6.2 guarantees that the SHCS does not meet the specification, regardless of the continuous dynamics.

6.5 Mitigating the State-Explosion Problem

To gain the performance benefit of cone-of-influence (COI) reduction (described in Section A.4) when analyzing systems in which all of the model variables appear in the COI, abstraction can be applied. Removing the transition logic of a variable, and replacing it with a free input to the model, results in an abstraction. More importantly, the variable influences that appeared in the removed transition logic are also removed from the model, potentially allowing for COI reduction. Because the reduced model is an abstraction, the same theoretical results apply as discussed in Section 6.4: ACTL specifications can be verified, and CIR specifications can be falsified, assuming that the abstracted-away variables are replaced with controllable inputs, as in Section 6.4.1.

There are many approaches for computing efficient abstractions of both discrete and hybrid systems, which are beyond the scope of this work. We apply the following abstraction procedure:

1. Set a limit on the number of state variables to include in the model.
2. Starting at 0, increment the COI search depth until the number of variables included in the COI exceeds the limit.
3. Replace each state variable that entered the COI in the previous level with an input variable.

In this way, the size of the abstract model to evaluate for each specification can be approximately set to the desired limit. The COI limit may not be enforced exactly, for two main reasons: step 3 may remove more variables than necessary, resulting in a COI that is strictly smaller than the chosen limit, or there may be too many input variables in the COI (which cannot be abstracted away) to achieve the limit.

6.5.1 Example: CIR Falsified after Simplification

Consider the system from Section 6.4.3, and the simplified PLC program (lines that begin with "`//`" are comments):

```
// condition_cooled := mode_cool
//                     and DT(T_reactor <= T_cold, 10, false);

condition_reacted := mode_react
                    and DT(N_C >= 0.99 * N_A0, 10, false);

// condition_heated := mode_heat
//                     and DT(T_reactor >= T_hot, 10, false);

condition_reset := mode_reset
                   and DT(N_A >= N_A0, 10, false);

mode_cool := (mode_cool and not (input_proceed and condition_cooled))
             or (mode_react and condition_reacted and input_proceed);

mode_react := (mode_react and not (input_proceed and condition_reacted))
              or (mode_heat and condition_heated and input_proceed);

mode_heat := (mode_heat and not (input_proceed and condition_heated))
             or (mode_reset and condition_reset and input_proceed);

mode_reset := (mode_reset and not (input_proceed and condition_reset))
              or (mode_react
                  and condition_cooled
                  and input_proceed and condition_reacted);
```

In the simplified program, `condition_cooled` and `condition_heated` are no longer set, and are instead treated as external values that act as readings. When this program is converted to a model, input variables are introduced in place of the assignment logic for `condition_cooled` and `condition_heated`, and are allowed to take any value; the result is an abstraction of the model in Section 6.4.3. This further abstraction of the model reduces the computational effort required for analysis. More importantly, the same result (that the specification $\mathsf{AG}(\mathsf{EF}(s_1) \wedge \mathsf{EF}(s_2) \wedge \mathsf{EF}(s_3) \wedge \mathsf{EF}(s_4))$ is violated) is still proven by applying Algorithm 6.2 to the abstraction.

6.6 Case Study

We now apply the methods that were developed in this chapter to two control systems provided by The Dow Chemical Company. Table 6.3 provides some basic details related to the size and complexity of the two systems. The first system, ‘‘Unit A’’, is a batch wash tank, and the second, ‘‘Unit B’’, is a batch reactor. The large number of discrete variables, in comparison to the relatively simple continuous control design (exhibited by the small number of PID loops), demonstrates the complexity of the discrete logic in a typical industrial plant control system.

Table 6.3: Overview of the case study problem size. The columns list the number of PID loops in the process, the number of discrete state variables in the control system, the number of delay timers, the number of variables that represent operating modes, and the average size of the cone of influence of the variables.

Name	PIDs	Variables	Timers	Modes	COI
Unit A	5	236	46	22	570
Unit B	8	752	162	53	2462

The variable lock and relevant logic PITs were applied for each of the discrete control variables in each system for various COI size targets. For each variable, a set of simplifications (abstractions) is computed as in Section 6.5 to enforce the desired COI limit. Table 6.4 shows the number of variables which were removed to enforce the COI limit, the COI size in the resulting abstract model, and the time taken to analyze the abstract model. The state-explosion problem appears as the rapid increase in solution time as the COI size is allowed to increase. For each of the two systems, the largest target COI size listed represents roughly the largest value for which the methods in this chapter could be applied. Comparing this limit (150) to the average original COI size in the two systems in Table 6.3, it is clear that COI reduction, made possible by analyzing abstract models, is critically important when it comes to analyzing

the closed-loop behavior.

Table 6.4: Abstraction and runtime information when applying the PITs to the case study. Each column lists the average value computed over the system variables. The columns list the target COI size when computing simplifying abstractions, the number of variables that were abstracted away, the resulting COI size after abstraction, and the time taken to analyze the PITs for each variable.

Name	COI Target	Abstr. var.	COI	Time/var. (s)
Unit A	25	8.7	16.2	5.9
Unit A	50	16.4	37.0	7.8
Unit A	100	22.3	83.5	17.2
Unit A	150	27.1	125.9	43.8
Unit B	25	16.3	27.1	16.4
Unit B	50	16.4	28.7	15.7
Unit B	100	36.2	69.4	21.3
Unit B	150	42.2	111.5	56.0

The result of applying the PITs to the sample systems are shown in Table 6.5. As expected, increasing the allowed COI size produces more conclusive results, at the expense of the increased execution time shown in Table 6.4. In both systems, each type of specification successfully detects the corresponding behavior, which supports the claim that the PITs described in Section 6.2.2 are relevant to industrial chemical plant automation systems. In addition, because the tests were produced automatically from the control logic itself, this method requires minimal effort or expert knowledge to analyze a system. The trend of being able to prove more results about the system behavior as the COI limit increases is expected to continue past the current COI limit of 150. Simply increasing the COI limit leads to increased computational effort due to the state-explosion problem; on the other hand, applying more sophisticated abstraction techniques than the one described in Section 6.5 might allow for capturing more of the important dynamics while meeting the same COI target.

The 5 variable locks detected in Unit A with the COI target set to 25 and

Table 6.5: PIT results from the case study. The columns list the number of results detected of each type.

Name	COI Target	Lock	Irrelevant	Relevant
Unit A	25	5	2	0
Unit A	50	8	12	0
Unit A	100	15	41	1
Unit A	150	15	44	2
Unit B	25	10	6	1
Unit B	50	10	8	1
Unit B	100	10	31	3
Unit B	150	10	50	4

the 10 in Unit B were the result of variables with assignment logic consisting of a single Boolean literal (`TRUE` or `FALSE`); this behavior was intended. These results could, in principal, have been detected by simply checking for Boolean literals in each variable's assignment logic. The additional variable locks detected in Unit A for larger COI values, on the other hand, were not caused by such trivial dynamics. Instead, they resulted from the interaction of the earlier variable locks with the rest of the discrete logic. The resulting behavior did not represent an error in the system, but this does indicate that detecting certain behavior in the system requires a more detailed model of the system dynamics. The irrelevant and relevant logic results were all the result of restrictions on the reachable values of the variables, so detecting them also required analysis of the system dynamics. Similar to the variable locks, a more detailed model yields a larger set of results. Due to the proprietary nature of the control code, we have omitted the actual control logic.

6.7 Summary

In this chapter, we have demonstrated the application of formal methods to analyze chemical plant automation systems. These systems are characterized by complex discrete logic which, coupled with continuous plant dynamics,

creates a large-scale hybrid dynamical system. Such systems are currently beyond the reach of systematic design tools, so instead we settled for proving certain aspects of the closed-loop behavior. To achieve this, we relied on automatically-generated process-independent tests (PITs) that involve both invariance and reachability requirements to obtain a high-level summary of a given control system. These PITs yielded positive results when applied to a case study consisting of two industrial automation systems, each from a batch process.

We determined whether or not an automation system satisfies the various PITs by applying a combination of abstraction and cone-of-influence reduction with either symbolic model checking or supervisory control theory. In contrast to existing techniques such as bounded model checking and implicit model checking, the methods we applied yield guaranteed verification and falsification results for the PITs. This is critically important for the method to be accepted in industry, where a tool that either reports speculative results that require further investigation (such as manually inspecting counterexamples) or fails to report obvious results (by being too conservative) is quickly discarded. Finally, the methods we presented provide the same theoretical guarantees on the results (i.e., no false positives) when applied to simplified models, which allowed us to handle an industrial-scale case study.

The techniques we applied do not depend on the form of the continuous process dynamics (e.g. linear, piecewise-affine). This feature is important when it comes to applying them to the broad range of processes that exist in the chemical industry. Not only do general chemical processes fail to fit in these classes of systems, but the continuous dynamics are never known with exact certainty, so general methods are required.

Chapter 7

st2smv v0.1.0

7.1 Modeling Logical Control Systems

The class of hybrid systems that arises in industrial plant automation is an important target for formal analysis and verification. A defining feature of these systems is that the discrete part of the hybrid system is defined in the control and automation logic. This makes it possible, in principle, to obtain a perfect model of the discrete dynamics. The difficulty in analyzing these systems without a formal model comes from the complexity of a typical industrial control system, which also necessitates an automated conversion process from existing logic to model. We address SHCSs of the form (2.1), implemented using programmable logic controllers (PLCs) as described in Section 6.3.

Previous work toward modeling the hybrid dynamics of PLC-controlled plants has focused in two directions; the theoretical aspects of accurately representing the logic are covered in [Moo94; RK98; Can+00; Bau+04; GdF08; Dar+14], and software tools to achieve the task are described in [BBK12; Fer+15]. In both areas, the work has focused on analyzing properties of the infinite-state hybrid system that can be verified directly using a finite-state

discrete abstraction.

7.2 Translating Structured Text Control Code to a Model

Currently, st2smv[1] accepts PLC programs defined in the restricted subset of the Structured Text (ST) language from Section 6.3. Every variable that is assigned a value is an output of the program. Any variable that is not assigned a value anywhere in the program is an input.

The PLC program is transformed to a formal model in a static single assignment intermediate representation (IR). Each variable assignment that appears in the ST program results in a new internal variable in the model. For an output variable that is assigned a value multiple times during one loop through the program, the actual output that is visible after the PLC loop is the final internal value that was calculated. In this way, multiple assignments to the same variable produce additional internal state variables in the model. An output variable that is only assigned a value at a single location in the ST program is simply equal to the internal state value produced by that assignment. The input variables to the ST program only appear explicitly in the model on the right-hand side of assignments to internal state variables, or in the guards of conditional assignments to internal state variables. The result is a deterministic map from input values to output values, along with the necessary internal state values, that represents a single loop through the PLC program. This can be modeled as a deterministic finite labeled transition system.

The final step before analyzing the model is to translate the IR to the input

[1] Available at: `https://pypi.python.org/pypi/st2smv/`.

language of a solver. SynthSMV is chosen to allow not only model checking, but also falsification of CIR specifications as in Chapters 6. The input language of SynthSMV defines a finite state machine (FSM). To model the discrete logic of a PLC program, the state of the FSM consists of the value assigned to each of the output and internal state variables during a PLC cycle, and the (labeled) inputs to the FSM are the readings from the plant. The Boolean inputs and state/output variables in the IR model are translated directly to Boolean inputs and variables in the SynthSMV model. The numeric variables only appear in the assignment logic for Boolean variables in the form of comparisons, so each comparison (using the current internal state of the numeric variables being compared) is replaced with a Boolean (input) variable. This is an abstraction, which overapproximates the possible set of readings that the discrete logic can receive, to ensure that the output model has a finite state space.

The approach described here is similar what is done by Arcade.PLC [BBK12] and PLCverif [Fer+15]. Arcade.PLC applies more general software analysis, using its own model checking algorithm. PLCverif is more similar to st2smv, including producing a NuSMV model as output; however, it is not yet available [Fer+15], and seems to focus exclusively on model checking to verify specifications. As described in Chapter 6, using model checking alone limits the class of specifications that can be verified (or falsified).

A plugin interface to st2smv is provided so that it can be extended in the future. To demonstrate this interface, we have implemented a plugin that automatically produces the variable lock, irrelevant logic, and relevant logic specifications described in Table 6.1.

7.3 Example

To demonstrate st2smv, we apply it to the liquid holding tank example problem from Section 2.3. The ST code for the controller is shown in Figure 7.1. As

```
h_high := 8; // high level
h_low := 2; // low level


alarm_high := (NOT acknowledged_high) AND h > h_high;
lockout_high := alarm_high OR (lockout_high AND h > h_high);
acknowledged_high := (alarm_high AND acknowledge_high)
                     OR acknowledged_high;

IF (NOT (alarm_high OR lockout_high)) THEN
    flow_in := flow_in_requested;
ELSE
    flow_in := 0;
END_IF;


alarm_low := (NOT acknowledged_low) AND h < h_low;
lockout_low := alarm_low OR (lockout_low AND h < h_low);
acknowledged_low := (alarm_low AND acknowledge_low)
                    OR (acknowledged_low AND h < h_low);

IF (NOT (alarm_low OR lockout_low)) THEN
    flow_out := flow_out_requested;
ELSE
    flow_out := 0;
END_IF;
```

Figure 7.1: Structured Text program for the liquid holding tank example.

described in Section 7.2, the variables `acknowledge_high` and `acknowledge_low`, along with the inequality comparisons `h > h_high` and `h < h_low`, are converted to Boolean inputs to the FSM. The generated SynthSMV model is shown in Figure 7.2.

Applying SynthSMV to the model with the `CTLSPEC` form of the variable lock specifications (i.e., model checking, not supervisor synthesis, specifications) reports the potential variable locks in Table 7.1. This means that in the

```
MODULE main
VAR initializing : boolean;
ASSIGN init(initializing) := TRUE;
ASSIGN next(initializing) := FALSE;
IVAR ivar_acknowledge_high_0 : boolean;
VAR acknowledge_high_0 : boolean;
ASSIGN next(acknowledge_high_0) := ivar_acknowledge_high_0;
IVAR ivar_acknowledge_low_0 : boolean;
VAR acknowledge_low_0 : boolean;
ASSIGN next(acknowledge_low_0) := ivar_acknowledge_low_0;
VAR alarm_high_1 : boolean;
ASSIGN alarm_high_1 := ((! acknowledged_high_0) & h_0__gt__h_high_1);
VAR lockout_high_1 : boolean;
ASSIGN lockout_high_1 := (alarm_high_1 | (lockout_high_0 & h_0__gt__h_high_1));
VAR acknowledged_high_1 : boolean;
ASSIGN acknowledged_high_1 := ((alarm_high_1 & acknowledge_high_0) | acknowledged_high_0);
VAR alarm_low_1 : boolean;
ASSIGN alarm_low_1 := ((! acknowledged_low_0) & h_0__lt__h_low_1);
VAR lockout_low_1 : boolean;
ASSIGN lockout_low_1 := (alarm_low_1 | (lockout_low_0 & h_0__lt__h_low_1));
VAR acknowledged_low_1 : boolean;
ASSIGN acknowledged_low_1 := ((alarm_low_1 & acknowledge_low_0) | (acknowledged_low_0 & h_0__lt__h_low_1));
DEFINE acknowledged_high := acknowledged_high_1;
DEFINE acknowledged_low := acknowledged_low_1;
DEFINE alarm_high := alarm_high_1;
DEFINE alarm_low := alarm_low_1;
DEFINE lockout_high := lockout_high_1;
DEFINE lockout_low := lockout_low_1;
VAR acknowledged_high_0 : boolean;
ASSIGN next(acknowledged_high_0) := acknowledged_high;
VAR acknowledged_low_0 : boolean;
ASSIGN next(acknowledged_low_0) := acknowledged_low;
VAR lockout_high_0 : boolean;
ASSIGN next(lockout_high_0) := lockout_high;
VAR lockout_low_0 : boolean;
ASSIGN next(lockout_low_0) := lockout_low;
IVAR ivar_h_0__gt__h_high_1 : boolean;
VAR h_0__gt__h_high_1 : boolean;
TRANS next(h_0__gt__h_high_1) = ivar_h_0__gt__h_high_1;
IVAR ivar_h_0__lt__h_low_1 : boolean;
VAR h_0__lt__h_low_1 : boolean;
TRANS next(h_0__lt__h_low_1) = ivar_h_0__lt__h_low_1;
INIT acknowledge_high_0 = FALSE;
INIT acknowledged_high_0 = FALSE;
INIT lockout_high_0 = FALSE;
INIT acknowledge_low_0 = FALSE;
INIT acknowledged_low_0 = FALSE;
INIT lockout_low_0 = FALSE;
```

Figure 7.2: SynthSMV model of the Structured Text program in Figure 7.1.

abstraction, it is possible to reach a state in which one of those variables can no longer change value. This does not necessarily mean the same is true of the hybrid system; additional analysis of the dynamics is required to determine whether or not a state in the hybrid system that corresponds to one of the problematic states in the abstraction is, in fact, reachable. This follows directly from the fact that the variable lock specification is not an ACTL specification.

Applying SynthSMV to the `SYNTH` form of the specifications confirms the variable lock involving the `acknowledged_high` variable; the specifications involving `alarm_high` and `lockout_high` are not confirmed. These results are shown in Table 7.1. The confirmed variable locks indicate that no restriction on the sequence of readings received by the discrete logic exists that will make the specification pass; thus, the control system violates those specifications. The fact that the remaining variable lock was not confirmed does not indicate that it is not present in the hybrid dynamics, only that it might not be present. As in Chapter 5, further analysis of the hybrid dynamics would be required to strengthen that result.

Table 7.1: Falsification results for the liquid holding tank example. Results shown in **bold** are guaranteed to hold in the actual system.

Specification	`CTLSPEC`	`SYNTH`
AG(EF(`acknowledged_high`) ∧ EF(¬`acknowledged_high`))	False	**False**
AG(EF(`alarm_high`) ∧ EF(¬`alarm_high`))	False	True
AG(EF(`lockout_high`) ∧ EF(¬`lockout_high`))	False	True

Similar to what is proposed in Section 5.4.3, changing the alarm acknowledgement logic to:

```
// acknowledged_high := (alarm_high AND acknowledge_high)
//                      OR acknowledged_high;
acknowledged_high := (alarm_high AND acknowledge_high)
                     OR (acknowledged_high AND h > h_high);
```

causes the corresponding variable lock specification to no longer be falsified.

As described previously, this does not guarantee that it is satisfied; verifying the specification is beyond the scope of st2smv.

7.4 Summary

This chapter provided an overview of st2smv, a tool for converting PLC logic (written in a subset of the Structured Text programming language) to a formal model. In addition to the formal model, st2smv automatically generates a set of process-independent tests to analyze the general behavior of the logic. The key functionality that distinguishes st2smv from existing tools is that it correctly models inputs to the PLC logic, so that techniques for falsifying certain CTL specifications (from Chapter 5) can be applied. SynthSMV (see Chapter 4) is used to analyze the models produced by st2smv.

Chapter 8

Conclusions

8.1 Contributions

This work has addressed the problem of detecting unintended behavior in chemical plants which is caused by errors in the discrete control and automation logic. The motivation for this is that the size and complexity of existing control systems has outpaced the available design tools. In the absence of effective design tools, the analysis result that leads directly to improvements in an existing system is a guarantee that the system does not behave as intended. Given this information, the system designer can focus on a specific aspect of the overall behavior that is incorrect, and fix the error.

In Chapter 6, we presented a set of temporal logic specifications that can be applied to a chemical plant's control and automation logic to detect certain classes of unintended behavior. Some of those specifications can be checked using methods that have been proven to apply to large-scale systems in previous work that addresses verification of logical control systems. Others, which involve requirements related to both invariance and reachability, were analyzed using a new method to falsify specifications in hybrid systems.

The key falsification result presented in Chapter 5 is a sound algorithm to prove that a hybrid system violates a specification written in a subset of computation tree logic (CTL) which has not been addressed in previous work. In some cases, the falsification algorithm only requires a model of the discrete dynamics (which are defined in the control system) to prove the existence of an error; this allows the algorithm to be applied in a chemical plant setting, where the discrete dynamics are known, but the continuous dynamics are not known exactly. If the specification cannot be falsified using only the discrete dynamics, a hybrid systems reachability verification problem can be solved in an attempt to falsify the original specification. This verification problem is simpler than the original falsification problem (which involves combined invariance and reachability requirements), and can be addressed using various techniques from the literature.

The falsification algorithm in Chapter 5 relies on new results, presented in Chapter 3, concerning the supervisory control of discrete systems. In particular, we show that a specification with multiple invariance and reachability requirements corresponds to an optimal state-based supervisor. This result extends recent work that addressed a single reachability requirement. The algorithm to compute the optimal supervisor was implemented in SynthSMV, an extension of the model checking solver NuSMV, as described in Chapter 4. SynthSMV takes advantage of NuSMV's efficient symbolic model checking implementation to handle large problems.

Applying the falsification algorithm to control systems implemented using programmable logic controllers (PLCs) requires an extension of the modeling approach used in previous work, which only addresses verification. The main difference is that readings from the plant are modeled as inputs to the discrete logic, which allows for the application of supervisory control to the model.

Another issue that comes up when modeling industrial control systems (for either verification or falsification) is that the size of the resulting model is often beyond the scope of symbolic model checking. To address this, we presented an abstraction technique that yields a simplified model of an arbitrarily large control system. The trade-off is that analysis of the abstract model may fail to falsify (or verify) specifications that could be falsified (or verified) using a closer approximation of the system dynamics; the soundness of the falsification and verification algorithms is preserved when they are applied to the simplified model. The modeling and abstraction details are presented in Chapter 6, and a software tool to automate the process, st2smv, is described in Chapter 7.

The methods developed in this work were applied to a case study from The Dow Chemical Company in Chapter 6. The analysis was performed using st2smv and SynthSMV. The case study results show that the process-independent tests from Chapter 6 can detect the corresponding behavior in existing industrial control systems. The results also show that abstraction (to simplify the model) is a necessary first step before symbolic model checking and supervisor synthesis can be applied to industrial-scale systems.

8.2 Recommendations for Future Work

The success of the process-independent tests from Chapter 6 depends on their application to a large number of control systems. Checking for unintended behavior in existing systems, which have already been tested thoroughly, is not likely to produce many results. The advantage is that it can be applied to any system, with minimal effort on the part of the practitioner. This is enabled by the tool st2smv, which automates the task of modeling the control logic. Thus, the first direction for future work is to apply the existing work more

widely in industry. Similarly, st2smv (and the modeling techniques it is based on) should be expanded to a wider range of PLC programs, so that it can be applied more broadly. This includes not only expanding the accepted syntax beyond the restricted subset of Structured Text presented in this work, but also addressing different operational semantics, such as asynchronous input and output.

One of the key limiting factors in the application of this work is the size and complexity of industrial control systems. In fact, as shown in the case study from Chapter 6, the use of simplified models is necessary even when symbolic model checking (and supervisor synthesis) is applied. We applied a very simple technique to simplify the models, and more sophisticated techniques that aim to include relevant discrete dynamics (instead of simply limiting the size of the simplified model) would almost certainly produce more informative results.

Although this work addresses hybrid dynamical systems, we made no attempt to incorporate knowledge of the continuous dynamics in the models when working with plant-wide control systems. This was justified by the observation that the continuous dynamics are difficult to obtain compared to the discrete dynamics, and computationally expensive to evaluate. However, this means that the results we reported are a subset of those that we could produce by incorporating knowledge of the continuous dynamics. Future work should explore ways to incorporate the hybrid dynamics to strengthen the results.

The set of PITs we provided in Chapter 6 is by no means an exhaustive list of all the requirements that a plant automation system should meet. There remain significant opportunities both to develop more PITs, and to develop process-*dependent* tests that target a particular chemical processing unit's behavior. This includes producing specifications with the same form as those

addressed in this work, namely ACTL specifications (to be verified) and CIR specification (to be falsified), but also exploring different classes of specifications which can lead to guaranteed verification or falsification results.

The SHCS model (2.1) is limited to continuous process dynamics, and systems with a single fixed sampling frequency. It would be useful to expand the model, and the results developed in this work based on that model, to a broader class of systems. In the opposite direction, it might be beneficial to consider more restricted classes of hybrid systems which could lead to more efficient application of the falsification method in Chapter 5.

Finally, SynthSMV has been shown to be capable of solving supervisor synthesis problems with large systems, but it was not formally compared to existing supervisory control tools. It would be interesting to compare it to the other solvers that implement (BDD-based) symbolic algorithms mentioned in Chapter 4. Similarly, it would be useful to extend SynthSMV to actually return the computed supervisor, instead of simply reporting whether or not the problem had a solution.

Bibliography

[Åke+06] K. Åkesson, M. Fabian, H. Flordal, and R. Malik. ''Supremica - An integrated environment for verification, synthesis and simulation of discrete event systems''. In: *8th International Workshop on Discrete Event Systems*. 2006, pp. 384–385. DOI: 10.1109/wodes.2006.382401.

[Bal+05] A. Balluchi, L. Benvenuti, S. Engell, T. Geyer, K. H. Johansson, F. Lamnabhi-Lagarrigue, J. Lygeros, M. Morari, G. Papafotiou, A. L. Sangiovanni-Vincentelli, F. Santucci, and O. Stursberg. ''Hybrid control of networked embedded systems''. In: *European Journal of Control* 11.4 (2005), pp. 478–508. DOI: 10.1016/S0947-3580(05)71047-5.

[BL02] P. I. Barton and C. K. Lee. ''Modeling, simulation, sensitivity analysis, and optimization of hybrid systems''. In: *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 12.4 (Oct. 2002), pp. 256–289. DOI: 10.1145/643120.643122.

[Bau+04] N. Bauer, S. Engell, R. Huuck, S. Lohmann, B. Lukoschus, M. Remelhe, and O. Stursberg. ''Verification of PLC programs given as sequential function charts''. In: *Integration of Software Specification Techniques for Applications in Engineering*. Vol. 3147. Lecture Notes in Computer Science. 2004, pp. 517–540. DOI: 10.1007/978-3-540-27863-4_28.

[BTM01] A. Bemporad, F. Torrisi, and M. Morari. ''Discrete-time Hybrid Modeling and Verification of the Batch Evaporator Process Benchmark''. In: *European Journal of Control* 7.4 (Jan. 2001), pp. 382–399. DOI: 10.3166/ejc.7.382-399.

[BM99] A. Bemporad and M. Morari. ''Control of systems integrating logic, dynamics, and constraints''. In: *Automatica* 35 (1999), pp. 407–427. DOI: 10.1016/s0005-1098(98)00178-2.

[BF04] A. Bhatia and E. Frazzoli. ''Incremental Search Methods for Reachability Analysis of Continuous and Hybrid Systems''. In: *Hybrid Systems: Computation and Control*. 2004, pp. 142–156. DOI: 10.1007/978-3-540-24743-2_10.

[BBK12] S. Biallas, J. Brauer, and S. Kowalewski. ''Arcade.PLC: A verification platform for programmable logic controllers''. In: *27th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2012*. IEEE. 2012, pp. 338–341. DOI: 10.1145/2351676.2351741.

[BBK10] S. Biallas, J. Brauer, and S. Kowalewski. ''Counterexample-guided abstraction refinement for PLCs''. In: *5th International Conference on Systems Software Verification*. USENIX Association. 2010.

[Bie+03] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu. ''Bounded Model Checking''. In: *Advances in Computers* 58 (2003), pp. 118–149. DOI: 10.1016/s0065-2458(03)58003-2.

[BvW10] S. Blom, J. van de Pol, and M. Weber. ''LTSmin: Distributed and Symbolic Reachability''. In: *Computer Aided Verification*. 2010, pp. 354–359. DOI: 10.1007/978-3-642-14295-6_31.

[Bry86] R. E. Bryant. ''Graph-Based Algorithms for Boolean Function Manipulation''. In: *IEEE Transactions on Computers* C-35.8 (1986), pp. 677–691. DOI: 10.1109/tc.1986.1676819.

[Can+00] G. Canet, S. Couffin, J.-J. Lesage, A. Petit, and P. Schnoebelen. ''Towards the automatic verification of PLC programs written in Instruction List''. In: *IEEE International Conference on Systems, Man and Cybernetics* 4 (2000), pp. 2449–2454. DOI: 10.1109/icsmc.2000.884359.

[CL08] C. G. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems*. Springer, 2008. DOI: 10.1007/978-0-387-68612-7.

[CK03] A. Chutinan and B. H. Krogh. ''Computational techniques for hybrid system verification''. In: *IEEE Transactions on Automatic Control* 48.1 (2003), pp. 64–75. DOI: 10.1109/TAC.2002.806655.

[CK01] A. Chutinan and B. H. Krogh. ''Verification of infinite-state dynamic systems using approximate quotient transition systems''. In: *IEEE Transactions on Automatic Control* 46.9 (2001), pp. 1401–1410. DOI: 10.1109/9.948467.

[Cim+02] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. ''NuSMV 2: An OpenSource tool for symbolic model checking''. In: *Computer Aided Verification*. Lecture Notes in Computer Science. 2002, pp. 359–364. DOI: 10.1007/3-540-45657-0_29.

[CG87] E. M. Clarke and O. Grumberg. ''Avoiding the state explosion problem in temporal logic model checking''. In: *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*. 1987. DOI: 10.1145/41840.41865.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. ''Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications''. In: *Association for Computing Machinery Transactions on Programming Languages and Systems* 8.2 (1986), pp. 244–263. DOI: 10.1145/5397.5399.

[CGL94] E. M. Clarke, O. Grumberg, and D. E. Long. ''Model checking and abstraction''. In: *ACM transactions on Programming Languages and Systems (TOPLAS)* 16.5 (1994), pp. 1512–1542. DOI: 10.1145/186025.186051.

[CGP99] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 1999.

[Cla+03] E. M. Clarke, A. Fehnker, Z. Han, B. H. Krogh, J. Ouaknine, O. Stursberg, and M. Theobald. ''Abstraction and counterexample-guided refinement in model checking of hybrid systems''. In: *International Journal of Foundations of Computer Science* 14.4 (2003), pp. 583–604. DOI: 10.1142/S012905410300190X.

[Col11] P. Collins. ''Semantics and computability of the evolution of hybrid systems''. In: *SIAM Journal on Control and Optimization* 49.2 (2011), pp. 890–925. DOI: 10.1137/080716955.

[DBF13] D. Darvas, E. Blanco Viñuela, and B. Fernández Adiego. *Transforming PLC Programs into Formal Models for Verification Purposes*. Tech. rep. CERN, 2013. URL: http://cds.cern.ch/record/1629275/files/CERN-ACC-NOTE-2013-0040.pdf.

[Dar+14] D. Darvas, B. Fernández Adiego, A. Vörös, T. Bartha, E. Blanco Viñuela, and V. M. González Suárez. ''Formal verification of complex properties on PLC programs''. In: *Formal Techniques for Distributed Objects, Components, and Systems*. Vol. 8461. Lecture Notes in Computer Science. 2014, pp. 284–299. DOI: 10.1007/978-3-662-43613-4_18.

[dCW05] M. H. de Queiroz, J. E. R. Cury, and W. M. Wonham. ''Multitasking Supervisory Control of Discrete-Event Systems''. In: *Discrete Event Dynamic Systems* 15.4 (Oct. 2005), pp. 375–395. DOI: 10.1007/s10626-005-4058-y.

[DSP97] V. D. Dimitriadis, N. Shah, and C. C. Pantelides. ''Modeling and safety verification of discrete/continuous processing systems''. In: *AIChE Journal* 43.4 (Apr. 1997), pp. 1041–1059. DOI: 10.1002/aic.690430418.

[Dim+96] V. D. Dimitriadis, J. Hackenberg, N. Shah, and C. C. Pantelides. ''A case study in hybrid process safety verification''. In: *Computers & Chemical Engineering* 20 (1996), S503–S508. DOI: 10.1016/0098-1354(96)00093-2.

[Dre+15] T. Dreossi, T. Dang, A. Donzé, J. Kapinski, X. Jin, and J. V. Deshmukh. ''Efficient Guiding Strategies for Testing of Temporal Properties of Hybrid Systems''. In: *Lecture Notes in Computer Science*. 2015, pp. 127–142. DOI: 10.1007/978-3-319-17524-9_10.

[DSL07] J. Du, C. Song, and P. Li. ''Modeling and Control of a Continuous Stirred Tank Reactor Based on a Mixed Logical Dynamical Model''. In: *Chinese Journal of Chemical Engineering* 15.4 (Aug. 2007), pp. 533–538. DOI: 10.1016/s1004-9541(07)60120-7.

[Ehl+16] R. Ehlers, S. Lafortune, S. Tripakis, and M. Y. Vardi. ''Supervisory control and reactive synthesis: a comparative introduction''. In: *Discrete Event Dynamic Systems* (2016), pp. 1–52. ISSN: 1573-7594. DOI: 10.1007/s10626-015-0223-0.

[Eng+00] S. Engell, S. Kowalewski, C. Schulz, and O. Stursberg. ''Continuous-discrete interactions in chemical processing plants''. In: *Proceedings of the IEEE* 88.7 (2000), pp. 1050–1068. DOI: 10.1109/5.871308.

[Fer+15] B. Fernández Adiego, D. Darvas, E. Blanco Viñuela, J.-C. Tournier, S. Bliudze, J. O. Blech, and V. M. González Suárez. ''Applying model checking to industrial-sized PLC programs''. In: *IEEE Transactions on Industrial Informatics* 11.6 (2015), pp. 1400–1410. DOI: 10.1109/tii.2015.2489184.

[Fre+11] G. Frehse, C. Le Guernic, A. Donzé, S. Cotton, R. Ray, O. Lebeltel, R. Ripado, A. Girard, T. Dang, and O. Maler. ''SpaceEx: Scalable verification of hybrid systems''. In: *Computer Aided Verification*. Vol. 6806. Lecture Notes in Computer Science. 2011, pp. 379–395. DOI: 10.1007/978-3-642-22110-1_30.

[Fre05] G. Frehse. ''PHAVer: Algorithmic Verification of Hybrid Systems Past HyTech''. In: *Hybrid Systems: Computation and Control*. 2005, pp. 258–273. DOI: 10.1007/978-3-540-31954-2_17.

[GST12] R. Goebel, R. G. Sanfelice, and A. R. Teel. *Hybrid Dynamical Systems: Modeling, Stability, and Robustness*. Princeton University Press, 2012.

[GdF08] V. Gourcuff, O. de Smet, and J.-M. Faure. ''Improving large-sized PLC programs verification using abstractions''. In: *17th IFAC World Congress*. July 2008. DOI: 10.3182/20080706-5-KR-1001.00857.

[GPT06] A. Gromyko, M. Pistore, and P. Traverso. ''A tool for controller synthesis via symbolic model checking''. In: *8th International Workshop on Discrete Event Systems*. 2006, pp. 475–476. DOI: 10.1109/wodes.2006.382523.

[GW00] I. E. Grossmann and A. W. Westerberg. ''Research challenges in process systems engineering''. In: *AIChE Journal* 46.9 (2000), pp. 1700–1703. DOI: 10.1002/aic.690460902.

[HDB01] W. Heemels, B. De Schutter, and A. Bemporad. ''Equivalence of hybrid dynamical models''. In: *Automatica* 37.7 (July 2001), pp. 1085–1091. DOI: 10.1016/s0005-1098(01)00059-0.

[Hen00] T. A. Henzinger. ''The Theory of Hybrid Automata''. In: *Verification of Digital and Hybrid Systems*. 2000, pp. 265–292. DOI: 10.1007/978-3-642-59615-5_13.

[HHW97] T. A. Henzinger, P. H. Ho, and H. Wong-Toi. ''HyTech: A model checker for hybrid systems''. In: *International Journal on Software Tools for Technology Transfer* 1.1--2 (1997), pp. 110–122. DOI: 10.1007/s100090050008.

[Hen+98] T. A. Henzinger, P. W. Kopke, A. Puri, and P. Varaiya. ''What's Decidable about Hybrid Automata?'' In: *Journal of Computer and System Sciences* 57.1 (1998), pp. 94–124. DOI: 10.1006/jcss.1998.1581.

[Hol97] G. Holzmann. ''The model checker SPIN''. In: *IEEE Transactions on Software Engineering* 23.5 (May 1997), pp. 279–295. DOI: 10.1109/32.588521.

[HU79] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Publishing Company, Inc., 1979.

[HR04] M. Huth and M. Ryan. *Logic in Computer Science*. Cambridge University Press, 2004. DOI: 10.1017/cbo9780511810275.

[IEC13] IEC. ''Part 3: Programming languages''. In: *Programmable controllers*. IEC Standard 61131. 2013. URL: https://webstore.iec.ch/publication/4552.

[JK06] S. Jiang and R. Kumar. ''Supervisory Control of Discrete Event Systems with CTL* Temporal Logic Specifications''. In: *SIAM Journal on Control and Optimization* 44.6 (2006), pp. 2079–2103. DOI: 10.1137/s0363012902409982.

[KM11] J. Kim and I. Moon. ''Model Checking for Automatic Verification of Control Logics in Chemical Processes''. In: *Industrial & Engineering Chemistry Research* 50.2 (Jan. 2011), pp. 905–915. DOI: 10.1021/ie100007w.

[Kow+99] S. Kowalewski, S. Engell, J. Preußig, and O. Stursberg. ''Verification of Logic Controllers for Continuous Plants Using Timed Condition/Event-System Models''. In: *Automatica - Special Issue on Hybrid Systems* 35.3 (Mar. 1999). DOI: 10.1016/s0005-1098(98)00179-4.

[KSB01] S. Kowalewski, O. Stursberg, and N. Bauer. ''An Experimental Batch Plant as a Test Case for the Verification of Hybrid Systems''. In: *European Journal of Control* 7.4 (Jan. 2001), pp. 366–381. DOI: 10.3166/ejc.7.361-381.

[LPY97] K. G. Larsen, P. Pettersson, and W. Yi. ''Uppaal in a nutshell''. In: *International Journal on Software Tools for Technology Transfer* 1.1--2 (1997), pp. 134–152. DOI: 10.1007/s100090050010.

[Len+14] B. Lennartson, F. Basile, S. Miremadi, Z. Fei, M. N. Hosseini, M. Fabian, and K. Åkesson. ''Supervisory Control for State-Vector Transition Models — A Unified Approach''. In: *IEEE Transactions on Automation Science and Engineering* 11.1 (2014), pp. 33–47. DOI: 10.1109/TASE.2013.2291115.

[Ler+08] F. Lerda, J. Kapinski, H. Maka, E. M. Clarke, and B. H. Krogh. ''Model checking in-the-loop: Finding counterexamples by systematic simulation''. In: *American Control Conference*. 2008, pp. 2734–2740. DOI: 10.1109/acc.2008.4586906.

[LS13] N. G. Leveson and G. Stephanopoulos. ‘‘A system-theoretic, control-inspired view and approach to process safety’’. In: *AIChE Journal* 60.1 (Nov. 2013), pp. 2–14. DOI: 10.1002/aic.14278.

[LTS06] S. Lohmann, L. A. D. Thi, and O. Stursberg. ‘‘Design of verified logic control programs’’. In: *IEEE International Conference on Control Applications*. 2006, pp. 1855–1860. DOI: 10.1109/CACSD-CCA-ISIC.2006.4776923.

[MW08] C. Ma and W. M. Wonham. ‘‘STSLib and its application to two benchmarks’’. In: *9th International Workshop on Discrete Event Systems*. 2008. DOI: 10.1109/wodes.2008.4605932.

[McM92] K. L. McMillan. ‘‘Symbolic Model Checking’’. PhD thesis. Carnegie Mellon University, May 1992.

[McM93] K. L. McMillan. ‘‘The SMV System’’. In: *Symbolic Model Checking*. 1993, pp. 61–85. DOI: 10.1007/978-1-4615-3190-6_4.

[MTA06] E. Mestan, M. Türkay, and Y. Arkun. ‘‘Optimization of Operations in Supply Chain Systems Using Hybrid Systems Approach and Model Predictive Control’’. In: *Industrial & Engineering Chemistry Research* 45.19 (Sept. 2006), pp. 6493–6503. DOI: 10.1021/ie0511938.

[Moo94] I. Moon. ‘‘Modeling Programmable Logic Controllers for Logic Verification’’. In: *IEEE Control Systems Magazine* 14.2 (1994), pp. 53–59. DOI: 10.1109/37.272781.

[Moo+92] I. Moon, G. J. Powers, J. R. Burch, and E. M. Clarke. ‘‘Automatic verification of sequential control systems using temporal logic’’. In: *AIChE Journal* 38.1 (1992), pp. 67–75. DOI: 10.1002/aic.690380107.

[MSP08] T. Moor, K. Schmidt, and S. Perk. ‘‘libFAUDES — An open source C++ library for discrete event systems’’. In: *9th International Workshop on Discrete Event Systems*. 2008. DOI: 10.1109/wodes.2008.4605933.

[Ngh+10] T. Nghiem, S. Sankaranarayanan, G. Fainekos, F. Ivančić, A. Gupta, and G. J. Pappas. ‘‘Monte-Carlo techniques for falsification of temporal properties of non-linear hybrid systems’’. In: *13th ACM International Conference on Hybrid Systems: Computation and Control*. 2010, pp. 211–220. DOI: 10.1145/1755952.1755983.

[PB00] T. Park and P. I. Barton. ‘‘Formal verification of sequence controllers’’. In: *Computers & Chemical Engineering* 23.11 (2000), pp. 1783–1793. DOI: 10.1016/s0098-1354(99)00327-0.

[PB97] T. Park and P. I. Barton. ‘‘Implicit model checking of logic-based control systems’’. In: *AIChE Journal* 43.9 (1997), pp. 2246–2260. ISSN: 1547-5905. DOI: 10.1002/aic.690430911.

[PKV09] E. Plaku, L. E. Kavraki, and M. Y. Vardi. ‘‘Hybrid systems: From verification to falsification by combining motion planning and discrete search’’. In: *Formal Methods in System Design* 34.2 (2009), pp. 157–182. DOI: 10.1007/s10703-008-0058-5.

[Pro+97] S. T. Probst, G. J. Powers, D. E. Long, and I. Moon. ‘‘Verification of a Logically Controlled, Solids Transport System Using Symbolic Model Checking’’. In: *Computers & Chemical Engineering* 21.4 (1997), pp. 417–429. DOI: 10.1016/0098-1354(95)00265-0.

[RW87] P. J. Ramadge and W. M. Wonham. ‘‘Supervisory Control of a Class of Discrete Event Processes’’. In: *SIAM Journal on Control and Optimization* 25.1 (Jan. 1987), pp. 206–230. DOI: 10.1137/0325013.

[RW89] P. J. Ramadge and W. M. Wonham. ''The Control of Discrete Event Systems''. In: *Proceedings of the IEEE* 77.1 (Jan. 1989), pp. 81–98. DOI: 10.1109/5.21072.

[RK98] M. Rausch and B. H. Krogh. ''Formal verification of PLC programs''. In: *American Control Conference*. Vol. 1. June 1998, pp. 234–238. DOI: 10.1109/ACC.1998.694666.

[RLG06] L. Ricker, S. Lafortune, and S. Genc. ''DESUMA: A Tool Integrating GIDDES and UMDES''. In: *8th International Workshop on Discrete Event Systems*. 2006. DOI: 10.1109/wodes.2006.382402.

[SF12] S. Sankaranarayanan and G. Fainekos. ''Falsification of temporal properties of hybrid systems using the cross-entropy method''. In: *15th ACM International Conference on Hybrid Systems: Computation and Control*. 2012, pp. 125–134. DOI: 10.1145/2185632.2185653.

[Seg07] M. Segelken. ''Abstraction and counterexample-guided construction of ω-automata for model checking of step-discrete linear hybrid models''. In: *Computer Aided Verification*. Vol. 4590. Lecture Notes in Computer Science. 2007, pp. 433–448. DOI: 10.1007/978-3-540-73368-3_46.

[Sri+98] R. Srinivasan, V. D. Dimitriadis, N. Shah, and V. Venkatasubramanian. ''Safety Verification Using a Hybrid Knowledge-Based Mathematical Programming Framework''. In: *AIChE Journal* 44.2 (1998), pp. 361–370. DOI: 10.1002/aic.690440213.

[Ste03] G. Stein. ''Respect the Unstable''. In: *IEEE Control Systems Magazine* (Aug. 2003). DOI: 10.1109/mcs.2003.1213600.

[Tab09] P. Tabuada. *Verification and control of hybrid systems: A symbolic approach*. Springer, 2009. DOI: 10.1007/978-1-4419-0224-5.

[Tiw07] A. Tiwari. ''Abstractions for hybrid systems''. In: *Formal Methods in System Design* 32.1 (Dec. 2007), pp. 57–83. DOI: 10.1007/s10703-007-0044-3.

[Yov97] S. Yovine. ''KRONOS: A verification tool for real-time systems''. In: *International Journal on Software Tools for Technology Transfer* 1.1--2 (1997), pp. 123–133. DOI: 10.1007/s100090050009.

[ZS05] R. Ziller and K. Schneider. ''Combining supervisor synthesis and model checking''. In: *ACM Transactions on Embedded Computing Systems* 4.2 (2005), pp. 331–362. DOI: 10.1145/1067915.1067920.

[Zut+13] A. Zutshi, S. Sankaranarayanan, J. V. Deshmukh, and J. Kapinski. ''A trajectory splicing approach to concretizing counterexamples for hybrid systems''. In: *52nd IEEE Conference on Decision and Control*. Dec. 2013. DOI: 10.1109/cdc.2013.6760488.

Appendix A

Mathematical Background

A.1 Hybrid Dynamical Systems

Systems that combine continuous and discrete dynamics are called hybrid dynamical systems, or simply hybrid systems. The following model represents a general hybrid system [GST12]:

$$\begin{cases} x \in C & \dot{x} \in F(x) \\ x \in D & x^+ \in G(x) \end{cases} \tag{A.1}$$

where x is the *state*, F is the *flow map*, G is the *jump map*, C is the *flow set*, and D is the *jump set*. The state varies continuously (flows) subject to the differential inclusion $\dot{x} \in F(x)$ when $x \in C$, and changes value discretely (jumps) subject to the difference inclusion $x^+ \in G(x)$ when $x \in D$.

The solutions to hybrid systems are parameterized by $t \in \mathbb{R}_{\geq 0}$, the amount of time that has passed, and $k \in \mathbb{N}$, the number of discrete jumps that have occurred. Only certain subsets $E \subset \mathbb{R}_{\geq 0} \times \mathbb{N}$, called *hybrid time domains*, correspond to actual solutions to a hybrid system. Points in a hybrid time domain are ordered such that $(t, k) \preceq (t', k') \iff t + k \leq t' + k'$. A solution

to a hybrid system (A.1) is a hybrid arc $\phi : E \to X$ that satisfies the system dynamics and constraints (starting from an initial state x_0), where E is a hybrid time domain and X is the state space of the hybrid system. It is most often useful to consider a hybrid arc ϕ that is a solution to a hybrid system, and that therefore defines its domain, E.

A.2 Transition Systems

A transition system (TS) has the form:

$$(Q, \Delta) \tag{A.2}$$

where Q is the set of system states, and $\Delta \subseteq Q \times Q$ is the set of transitions. When the system's state is $q \in Q$, it can make a transition to a new state $q^+ \in Q$ if $(q, q^+) \in \Delta$. Without loss of generality, Q can be defined in terms of a set of state variables, so that relational expressions in terms of the state variables (and Boolean combinations thereof) describe fixed subsets of Q.

Adding a set of labels to a transition system results in a labeled transition system (LTS):

$$(Q, \Sigma, \Delta) \tag{A.3}$$

where Q is the same as in an unlabeled transition system, Σ is the set of labels, and $\Delta \subseteq Q \times \Sigma \times Q$ is the set of labeled transitions. The system behavior is similar to that of an unlabeled transition system: a transition $\delta = (q, \sigma, q^+)$, which is a transition from state q to state q^+ labeled by σ, can only occur if $\delta \in \Delta$. As with unlabeled transition systems, Q and Σ can be defined in terms of state and label variables.

A labeled transition system (Q, Σ, Δ) is finite if Q and Σ are finite, and

deterministic if, for each $(q, \sigma) \in Q \times \Sigma$, there exists a single $\delta = (q, \sigma, q^+)$ such that $\delta \in \Delta$. Finite and deterministic unlabeled transition systems are defined similarly.

A path in an LTS is a sequence of states $\psi = q_0 \dots q_n$ where $n \geq 1$ such that $\forall i \in 0 \dots n-1 : \exists \sigma_i \in \Sigma : (q_i, \sigma_i, q_{i+1}) \in \Delta$. If a set of initial states is defined, then the reachable states are those that the system can arrive at by following a path from one of the initial states.

Definition A.1 (reachable states)**.** Given a transition system $\mathcal{L}$ as in (A.3) and a set of initial states $Q_0 \subseteq Q$, the set of *reachable states*, written $Reach(\mathcal{L})$, is given by:

$$Reach(\mathcal{L}) \coloneqq \{q_n \in Q \mid \exists \psi = q_0 \dots q_n : (q_0 \in Q_0) \wedge (\psi \text{ is a path in } \mathcal{L})\} \cup Q_0$$

It is often useful to consider abstractions of LTSs.

Definition A.2 (abstraction)**.** Given the LTS $\mathcal{L} = (Q, \Sigma, \Delta)$ with initial states Q_0 and $\widetilde{\mathcal{L}} = (\widetilde{Q}, \Sigma, \widetilde{\Delta})$ with initial states $\widetilde{Q}_0$, $\widetilde{\mathcal{L}}$ is an *abstraction* of $\mathcal{L}$, written $\widetilde{\mathcal{L}} \succeq \mathcal{L}$, if there exists an abstraction function $\alpha : Q \to \widetilde{Q}$ such that:

- $\widetilde{Q}_0 = \{\widetilde{q} \mid \exists q \in Q_0 : \alpha(q) = \widetilde{q}\}$
- $\forall (q, \sigma, q^+) \in \Delta : (\alpha(q), \sigma, \alpha(q^+)) \in \widetilde{\Delta}$

Definition A.2 relies on an abstraction function $\alpha : Q \to \widetilde{Q}$ which maps states in the concrete system to states in the abstract system. To simplify notation, α can be extended to sets of states:

$$\alpha(Q) \coloneqq \{\widetilde{q} \mid \exists q \in Q : \alpha(q) = \widetilde{q}\}$$

and sets of (labeled) transitions:

$$\alpha(\Delta) \coloneqq \left\{ (\widetilde{q}, \sigma, \widetilde{q}^{+}) \;\middle|\; \exists (q, \sigma, q^{+}) \in \Delta : \alpha(q) = \widetilde{q} \wedge \alpha(q^{+}) = \widetilde{q}^{+} \right\}$$

so that the requirements in Definition A.2 can be rewritten as $\widetilde{Q_0} = \alpha(Q_0)$ and $\widetilde{\Delta} \supseteq \alpha(\Delta)$. The abstraction function also extends to paths:

$$\alpha(\psi) \coloneqq \alpha(\psi[0]), \alpha(\psi[1]), \ldots$$

where ψ is a path $q_0, q_1, \ldots$ in the concrete system.

A.3 Computation Tree Logic

Temporal logic is used to specify properties of the dynamic behavior of state transition systems. Various temporal logics, including computation tree logic (CTL), are described in [HR04] and [CGP99].

Some of the CTL operators, which are used to build CTL formulas, are described in Table A.1. A *state formula* is a formula that describes a set of states, i.e., a subset of the state space. The most basic type of state formula is an *atomic proposition*, a, that describes a fundamental (atomic) property of the system's state. A *path formula* is a formula that describes a set of paths along which the system might evolve. Path formulas are formed by applying one of the temporal operators, F or G, to a state formula. State formulas are formed by atomic propositions, or by applying one of the path quantifiers, A or E, to a path formula. State formulas can also be combined and modified using the Boolean operators $\wedge$, $\vee$, and $\neg$, with the result being another state formula. A CTL specification is a state formula, and the states that satisfy the specification are those in which the formula holds.

Table A.1: A subset of the CTL operators.

Symbol	Mnemonic	Type	Returns
$\mathsf{F}(p)$	Finally	Temporal operator	The set of paths along which state formula p holds at some point.
$\mathsf{G}(p)$	Globally	Temporal operator	The set of paths along which state formula p always holds.
$\mathsf{X}(p)$	neXt	Temporal operator	The set of paths along which state formula p holds in the next state.
$\mathsf{A}\Theta$	All	Path quantifier	The set of states, from each of which all paths satisfy path formula Θ.
$\mathsf{E}\Theta$	Exists	Path quantifier	The set of states, from each of which there exists a path that satisfies path formula Θ.
a	$a$tom	Atomic proposition	The set of states in which the atomic proposition a holds.

The specification $\mathsf{AG}(p)$ is an *invariance* specification, which requires that the system never leave the set of states in which p holds. The specification $\mathsf{EF}(p)$ is a *reachability* specification, which requires that the system can reach the set of states in which p holds. Invariance and reachability are logical duals, i.e., $\mathsf{AG}(p) \iff \neg\mathsf{EF}(\neg p)$.

For a system in which the state is defined by the value assigned to a set of state variables (as in Section A.2 for transition systems), relational expressions involving the state variables can be used as atomic propositions. For example, if the set of states is defined by $Q = \{0,1\}^2$, then the atomic proposition $q_1 = 1$ returns the set $\{(1,0),(1,1)\}$ and the atomic proposition $q_1 = q_2$ returns the set $\{(0,0),(1,1)\}$. Clearly, these atomic propositions describe fixed sets of states, and are not affected by the dynamics of the system.

A.3.1 Other Temporal Logics

The *universal fragment* of CTL, called ACTL, is obtained by excluding the existential path quantifier, E. This assumes that the formulas are in *positive normal form*, meaning that the temporal operators are not directly negated (e.g., $\neg\mathsf{AG}(p)$ is first converted to $\mathsf{EF}(\neg p)$, which is not an ACTL formula). ACTL is of practical interest primarily because if an ACTL formula is verified in an abstraction of a system, then it is guaranteed to hold in the actual system also [CGL94].

Other commonly-used temporal logics include CTL* (a superset of CTL) [CES86], ACTL* (defined similarly to ACTL by excluding E from CTL*), and linear temporal logic (LTL), which is a subset of ACTL*. To briefly motivate the use of CTL, consider the formula $\mathsf{AG}(\mathsf{EF}(p))$, which specifies that it is always possible to reach a state in which p holds. This CTL formula cannot be expressed in ACTL*, ACTL, or LTL.

A.4 Model Checking

The objective of model checking is to determine whether or not a given model meets a temporal logic specification [CGP99]. For CTL specifications, this is achieved by first computing the set of all states that satisfy the specification, then checking whether or not every initial state of the system is included in that set. If the specification holds in every initial state, then the model itself satisfies the specification; if not, then the model violates the specification.

The CTL model checking algorithm is covered in detail in [CGP99], including the correspondence between CTL formulas and fixed points of monotonic functions. The time complexity of CTL model checking for a specification f and a (finite) transition system (Q, Δ) is $O(|f| \cdot (|Q| + |\Delta|))$, where $|f|$ is the

number of subformulas of f, $|Q|$ is the number of states in the system, and $|\Delta|$ is the number of transitions in the system.

The *state-explosion problem* is the fact that the number of states in a system can grow exponentially with the number of interacting components. For example, in a typical plant automation system, the number of components is roughly the number of discrete state variables in the control system, which is typically on the order of hundreds to thousands [Eng+00]; this leads to very large models for which the discrete state space cannot reasonably be *explicitly* enumerated. One successful technique for overcoming this problem is the use of binary decision diagrams (BDDs), which allow for efficient representation of Boolean functions [Bry86], to represent the model *symbolically* [McM92]. Symbolic model checking enables the analysis of systems with hundreds of interacting components, for which explicit model checking is intractable.

Another technique to avoid the state-explosion problem is to only consider variables that influence a given specification. In this way, a reduced model is built using only the state variables that appear in the specification, and the variables that (directly or indirectly) influence the value assigned to those state variables. In systems that contain multiple disconnected groups of variables, this cone-of-influence (COI) reduction can drastically improve the performance compared to the naïve approach of always building the full model [CGP99]. However, for systems in which all the variable influence each other, COI reduction has no effect, because the reduced model is equivalent to the full model.

A.5 Supervisory Control

A system with a discrete state space that evolves by making discrete transitions in response to a sequence of events is called a *discrete event system* (DES) [CL08]. A DES can be modeled by an LTS (Q, Σ, Δ), where Q is the state space, Σ is the set of events that can occur, and Δ is the set of transitions that occur in response to events. If, for some $(q, \sigma) \in Q \times \Sigma$, there is no $q^+ \in Q$ such that $(q, \sigma, q^+) \in \Delta$, then the event σ does not occur when the system is in state q.

Supervisory control theory was introduced as a tool to control DESs [RW87; RW89]. While the objective of model checking is to determine whether or not a DES meets a requirement, the objective of supervisory control is to modify the behavior of a DES to ensure that it meets the requirement. Informally, given a DES, a set of controllable events $\Sigma_c \subseteq \Sigma$, and a specification, the supervisory control problem is to compute a strategy for disabling events in Σ_c such that the modified DES satisfies the specification. A formal definition can be found in [CL08].

In the traditional development of supervisory control theory, a specification is a formal language [HU79] over the set of events, and the controlled system's behavior should be restricted to a subset of that language. In addition, a set of marked states is defined that should always remain reachable in the controlled system. The controller that restricts the system's behavior, called the supervisor, reads the entire string of events that have occurred and maps that string to a control action (a set of disabled events). Thus, the supervisor is dynamic, in the sense that it responds to new events by updating its state and associated control action; this type of dynamic supervisor can be realized by an automaton.