Compositional Verification of Industrial Control Systems

Methods and Case Studies

Dissertation

zur Erlangung des akademischen Grades
Doktor der Naturwissenschaften
(Dr. rer. nat.)
der Technischen Fakultät
der Christian-Albrechts-Universität zu Kiel

Ben Lukoschus

Kiel
2005

1. Gutachter	Prof. Dr. Willem-Paul de Roever
2. Gutachter	Prof. Dr.-Ing. Sebastian Engell
3. Gutachter	Prof. Dr.-Ing. Stefan Kowalewski
Datum der mündlichen Prüfung	16. Juli 2004

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The increased use of computers in our private, public, and business life has become self-evident in the recent decades, and still new fields for computer applications are being explored. Electronic controllers are present in home appliances as well as automobiles, planes, and industrial plants of all kinds.

An integral part of any computer-controlled system is its software. Its reliability is crucial for the correct functioning of the system. In practice, however, software errors are often the cause of system malfunctions. In one of his last writings, Edsger W. Dijkstra summarizes the current situation:

> "The average customer of the computing industry has been served so poorly that he expects his system to crash all the time, and we witness a massive world-wide distribution of bug-ridden software for which we should be deeply ashamed." [Dij00]

This observation calls for quality assurance measures. Apart from intensive testing and validation efforts, *formal verification* of software and hardware has become a much needed and widely accepted technique for quality assurance which is used especially in fields where undetected flaws in the final product are not acceptable because of imminent losses of reputation, money, or even human lives, e.g., in medical systems,[1] chip design,[2] avionics,[3] communication protocols,[4] space exploration,[5] or control of nuclear reactors.

[1] 1985–1987: Due to two software flaws, several Therac-25 radiation therapy machines massively overdose six people, three of them are killed [LT93, Lev95].

[2] 1994: A subtle flaw in the floating point divide unit of the Pentium processor [Int94] forces Intel to put aside $420 000 000 to cover replacement costs.

[3] 1996: The complete loss of the Ariane 501 launcher and $500 000 000 payload is due to specification and design errors in the software of the inertial reference system [LL+96].

[4] 2001: The link-layer security protocol for encryption in IEEE 802.11 wireless networks is found to be vulnerable to passive attacks [FMS01, SIR01].

[5] 2004: The Mars exploration rover "Spirit" temporarily stops communicating with Earth due to a "too many files" error in its flash memory filesystem [WS04].

1.1 The Subject of the Thesis

In this work we focus on the formal verification of industrial systems in the field of chemical process engineering which are driven by *programmable logic controllers* (PLCs), a class of automated control hardware (introduced in Section 1.2).

Chemical plants pose an interesting challenge for formal verification methods:

- Chemical plants involve a wide range of different concepts, such as discrete and continuous processes, distributed process and control structures, and scheduling as well as control tasks.
- In contrast to many other fields in which formal software verification is desired, control software for chemical plants is often not written by computer scientists, but by chemical and process control engineers. Therefore, formal verification methods are relatively new to that field.

Both *noncompositional* and *compositional* verification methods are presented. Compositional techniques reduce the verification of large systems to the *independent* verification of their parts, whereas noncompositional approaches need to verify depending parts of the system as a single connected unit. Given the PLC software, descriptions of the hardware that is controlled by the PLC, and information about the process itself, we develop a formal model on the basis of which properties of the system have to be verified. The verification methods are illustrated using two chemical laboratory batch plants which are used for teaching in the Chemical Process Control Laboratory at the University of Dortmund, Germany.

The main contributions of this thesis are:

- The translation of PLC programs (specifically, sequential function charts) into the framework of discrete condition/event systems and the input languages of the verification tools SAL and SMV.
- The implementation of a compiler which translates sets of discrete condition/event systems into the input language of the verification tool SMV. This is very useful when composing a large number of different system parts.
- Modeling strategies for the non-software parts of the system, e.g., devices like tanks, valves, and sensors.
- Deductive reasoning methods to combine local properties of modules describing the behavior of the plant hardware as well as its control into global properties of the complete chemical batch plant.
- The introduction of communicating linear hybrid automata as a compositional modeling framework for the specification of hybrid systems.

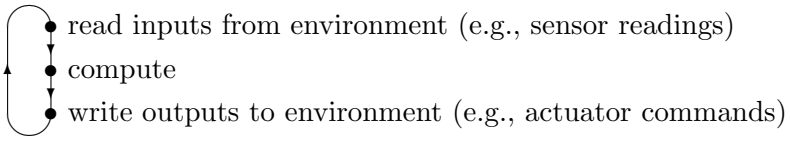


Figure 1.1: The three phases of a PLC cycle

The verification of the chemical batch plants mentioned above presents a clear example of a system of which only its components can be verified algorithmically, but because of the exessive time and memory requirements of the verification process this is not feasible for the complete system as a whole. This applies a fortiori to the verification of software for real-life chemical industrial plants. Therefore, the verification of the latter definitely requires a combination of algorithmic and deductive methods.

1.2 Programmable Logic Controllers

Programmable logic controllers, PLCs for short, have been introduced in the 1970s as an advanced replacement for electrical control circuits based on relay switches. These circuits were able to perform simple control decisions hard-wired into the relay structure.

PLCs introduced a much more flexible way of building controllers. A PLC is a piece of hardware (usually a single-board computer) equipped with a microprocessor, memory, and various kinds of input/output ports. It operates in a cyclic manner. One PLC cycle has the following three phases (see Figure 1.1) which are typical of reactive synchronous systems: First inputs are sampled at the input ports and stored into memory, then a computation takes place, and finally output values are written to the output ports. This cycle is repeated until the PLC is interrupted by external intervention, e.g., shutdown or reset commands. The cycle time is usually around a few milliseconds, depending on the timing demands of the controlled process.

The control functionality of a PLC is not hard-coded into its hardware; it is a piece of software which is usually written with the support of external programming environments and then downloaded into the PLC memory. This approach allows for easy reconfiguration during tests as well as during the operational phase of the controlled system. Several programming languages of differing flavors are used for PLCs; the most prevalent ones were standardized in the late 1990s through IEC 61131-3 [IEC98].

PLCs are mainly used to control industrial processes (or parts thereof) where there is usually little or no need for human interaction, such as chemical production processes, packaging lines, or power plants. Most PLCs work reliably in hazardous environments and can therefore be located close to the controlled process itself.

1.3 Formal Verification

Formal verification involves establishing properties of hardware and software components using methods in formal frameworks based on mathematics and logic. These properties are proven formally, and not simply argued to be correct by (often incomplete) testing or informal reasoning.

A prerequisite for the application of formal proof methods is the existence of a formal model of the system that is to be verified. In case of software, such a formal model can often be derived from the semantics of the programming languages involved.

Given a formal model M of the system, several techniques have been developed to prove that M fulfills some requirement φ. So-called *state-based* verification methods describe the behavior of M through changes of a *state,* which is a complete description of the current operational status of the system, e.g., values of variables, program counters, timer values, etc. If the set of all states is finite (or can be expressed by finite abstractions), an algorithmic method called *model checking* [CE82, QS82] can often be used to check automatically if M fulfills φ, denoted as $M \models \varphi$ ("M satisfies φ" or "M is a model of φ").

In case model checking is not possible, e.g., in case the state space is too large, and its automatic exploration takes too much space or time (see the discussion below), or because essentially properties of infinite state spaces need to be proven, *deductive verification methods* are used. The deductive analysis can often be supported by semi-automatic theorem provers.

1.3.1 Challenges in Formal Verification

The main challenge in formal verification is to keep up with the ever-growing complexity of the systems we are faced with. This applies to hardware as well as software: Gordon E. Moore estimated in 1965 that integrated circuits such as microprocessors and memory chips double their number of transistors every one to two years [Moo65]—this observation known as "Moore's Law" still holds today (though physical limits are now within sight). The first public release of the Linux operating system started with 10239 lines of code (version 0.01, September 1991), and today the Linux kernel source has over 6 million lines of code (version 2.6.11.5, March 2005).

These figures show that viable verification methods must be able to cope with very large systems, and they must be scalable to meet future demands.

A second challenge is the increased interconnectedness of systems. The behavior of many software applications depends on communication with external components like sensors, actuators, or other software-controlled systems. Since these external components often need to be taken into account in the verification of the system, we face a much bigger system than just a local one. This adds to the main complexity challenge mentioned above.

1.3.2 The State Explosion Problem

One of the drawbacks of state-based formal verification methods is their so-called *state explosion problem:* When a large system consists of several smaller components (e.g., automata) running in parallel, the number of global states increases exponentially with the number of components. For instance, consider a system of 20 automata working in parallel, each of which having 10 local states. This gives rise to 10^{20} global states. The simple task of enumerating these states on a machine that needs only one nanosecond per state (which is a rather tight estimation at the time of writing) already takes well over 3000 years. Building and searching a graph based on these states takes significantly longer and is far beyond today's memory capabilities.

The state explosion problem is inherent in any system having parallel structures and poses a major complexity barrier to any verification method based on only the exhaustive enumeration of the global states. Several techniques have been developed to minimize the impact of this problem on the consumption of time and memory during the verification process.

These techniques can be divided into two classes: optimizations in the model-checking process itself (not a subject of this thesis) and optimizations of the model before handing it over to the model checker.

1.3.3 Optimizing the Model

Our main focus of research is not the optimization of tools, but of the way these can be used efficiently by optimizing the models we feed into them. The main techniques for efficient modeling are *abstraction* and *compositionality.*

Abstraction

Abstraction is a fundamental concept used in all formal verification methods. Abstracting means replacing a concrete object with an abstract one which has a simpler structure. A well-chosen abstraction simplifies as much as possible, without losing essential (i.e., verification-relevant) information about the concrete object. Abstractions can be used in different ways during the specification and verification process:

- Building the system model: Every translation from a real-life system into a formal model is an abstraction.

- Optimizing the system model: Depending on the property that is to be checked, different abstractions of the system model can be useful, e.g., by abstracting from data, time, or continuous variables in order to obtain simpler models.

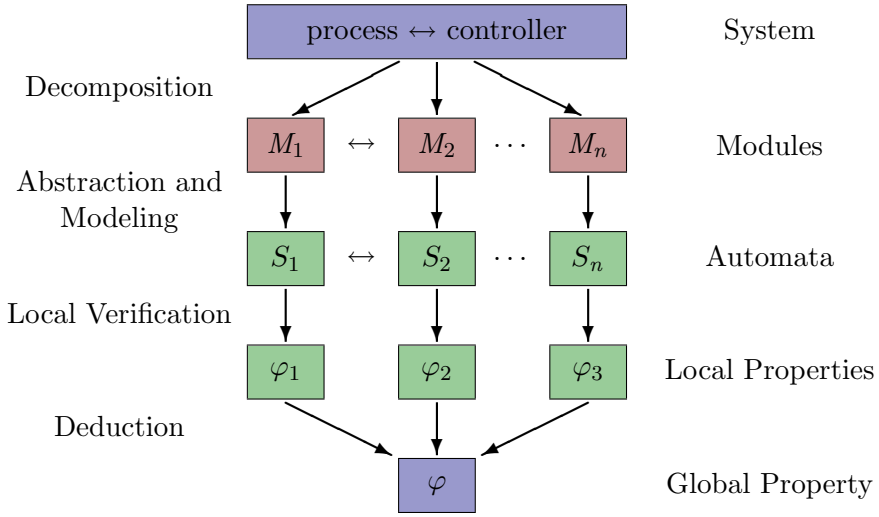


Figure 1.2: A compositional verification approach

- Reducing the complexity of model checking: Model checkers often use abstractions to minimize time and space usage, e.g., by introducing symbolic states.

When abstracting a system model, often a so-called *safe abstraction* is chosen, i.e., whenever a property holds for the abstract system, this property also holds for the concrete system. The converse, however, does not always hold, due to the over-approximation which occurs in the abstraction process. A positive model-checking result on a safe abstraction therefore means that the concrete system also fulfills the property, whereas a negative result can either mean that the concrete system is not correct or that the abstraction is too coarse.

Thus, when getting a negative result, the counterexample provided by the model checker is examined to see if the error will also occur in the concrete system. If it does not, a finer abstraction has to be chosen.

Compositionality

Compositional reasoning is a methodology in which a system can be specified by providing descriptions of its constituent parts and the ways they are put together, such that the behavior of the complete system can be inferred from these descriptions only [Fre23]. Figure 1.2 shows a compositional approach to the verification of a system. First the system is split into smaller components, called modules. Each module is then formally specified at a suitable abstraction level as a single entity, and its correct behavior is proven locally, e.g., by model checking. The specifications of all modules are then combined in a deduction step to infer a global property of the system model.

A prerequisite for a compositional approach is that the behavior of a component is completely described by its interface specification such that the behavior of the global system model can be expressed using only these interfaces and does not depend on any additional information about the internal structure of the components ("black box" principle, see [Zwi89]).

The advantage of such an approach is obvious. Recall the example introduced above (20 automata, 10 local states each). A compositional approach yields 20 applications of a model-checking algorithm, each involving only 10 states, whereas the global approach applies model checking once, but on a set of 10^{20} states. There is, however, some (often significant) overhead for the decomposition of the system and the construction and composition of the local specifications. To quote Edsger W. Dijkstra once more:

> "You see, while we all know that unmastered complexity is at the root of the misery, we do not know what degree of simplicity can be obtained, nor to what extent the intrinsic complexity of the whole design has to show up in the interfaces." [Dij00]

1.4 Case Studies

This thesis discusses two examples in detail. Both are laboratory batch plants, used for teaching at the Chemical Process Control Laboratory at the University of Dortmund, Germany. They have also been used as case studies in several research projects, e.g., the VHS project[6] [VHS].

The first plant combines a mixing and a separation step into a closed recycling process: A highly concentrated salt solution is diluted with water, resulting in in a salt solution with a lower concentration. This solution is heated until steam evaporates. The steam is cooled down in a condenser, and the condensate (distilled water) is collected. The evaporation continues until the original salt concentration is reached, and both liquids, distilled water and salt solution, can be used to restart the process.

The second plant is a multi-product batch plant. Two different colored liquids, "blue" and "green", are produced in three independent reactors by mixing three raw materials, "yellow", "red", and "white". The production is run in batches: Each of the tanks is capable of containing a maximal number of fixed volumes, called batches. The raw material storage tanks and the reactors have a maximum capacity of two batches, and the product tanks can hold up to three batches. The independent reactors allow the concurrent production of several batches.

Both plants are controlled by PLC systems. The complexity of these plants demonstrates the need for the verification strategies discussed in this thesis:

[6] VHS – Verification of Hybrid Systems. EU Esprit Long-Term Research Project 26270.

1. The need for automatic verification of single plant components. Proofs carried out by hand are not convincing and too error-prone because of the size and the resulting complexity of the components.

2. The need for deductive verification. Experiences with the plant models created in step 1 above show that the complete product of all components suffers from the state explosion problem and makes fully algorithmic verification impossible.

3. The interplay between automatic and deductive verification. The interfaces of the single plant components need to be defined in such a way that the verification results from step 1 above can easily be used as a basis for deductive reasoning in step 2.

1.5 Technical Contributions of the Thesis

This thesis develops methods for the verification of industrial control systems for chemical plants. The starting point is a description of the industrial process and its control structure. This usually includes:

- A description of the plant's physical structure, often given as a "piping and instrumentation diagram" (a schematic picture of the plant layout).

- Descriptions of the physical components of the plant, such as tanks, pumps, and valves.

- A description of the process taking place, e.g., chemical reactions.

- The software used to control the process, and the hardware platform running it, e.g., a PLC.

- Information on how the process and the control are related to each other, usually through sensors and actuators.

Furthermore, we have a list of verification goals, i.e., process states that need to be reached (e.g., "production finished") or situations that should be avoided (e.g., "tank overflow").

In the following we list the main technical contributions of this thesis that enable the verification of industrial control systems for chemical plants.

1.5.1 Analysis and Definition of PLC Software Semantics

Formal reasoning about PLC programs requires a clear understanding of their underlying semantics. In the field of computer science, semantics of programming languages is a well-researched field, and various kinds of formal

semantics for all flavors of languages have been established. PLC programming languages, however, emerged from hard-wired logical and relay circuits and are therefore based on knowledge about electrical circuits rather than mathematics.

Based on informal descriptions and observations in the behavior implemented in PLC programming tools, an unambiguous operational semantics for PLCs is defined (more precisely, for the programming language Sequential Function Charts (SFCs)). This semantics serves as the basis for the translation of SFC programs into other frameworks such as the input languages of model-checking tools.

1.5.2 Automatic Generation of SMV Code

As we have seen in Section 1.3.2, it is often not possible to model-check a complete system (modeled by the full product of all its parts) due to complexity problems. We propose a modular (though not fully compositional) method which allows to reduce the complexity of the verification process by composing only those parts of the system which are necessary to establish (or to show the violation of) the property in question. This results in a number of relatively small *open* systems, i.e., systems which have some unrestricted inputs showing arbitrary (also called "chaotic") behavior.

When using the SMV model checker, the verification of each property requires a systematic modification on the communication interfaces of the system parts to reflect the "openness" of some inputs. Since these modifications are tedious and error-prone, we have built a tool which automatically translates the models of the parts of the system into SMV code while applying these modifications.

1.5.3 Modeling of Hardware Components

Models of software components can usually be developed from (an abstraction of) the formal operational semantics of the programming language involved, e.g., sequential function charts for PLC programs. For hardware components, however, such a formal basis for a model seldom exists. We show how the hardware parts of our examples, such as valves, pumps, and tanks, can be modeled as condition/event systems, which can automatically translated into SMV code (see above). Compositional verification offers another approach: We give specifications for hardware components directly, and assume them to be correct, without proving them by model checking, since we gain no additional confidence from a specification proven by model checking if the model had been built by hand just to reflect the component's expected behavior.

1.5.4 Compositional Verification

We use the multi-product batch plant to illustrate the compositional verification approach sketched in Figure 1.2. Software and hardware components are identified, and we model a communication structure of the components which represents flows of information (sensor data and actuator commands) as well as material flows (liquids). Furthermore, local specifications are given for the components. The control programs, written in the sequential function chart language, are modeled in the input language of the SAL model checker, and the local specifications are proven algorithmically. Finally, we use deduction to derive global properties of the complete plant from the local specifications.

1.5.5 Communicating Linear Hybrid Automata

Properties which depend on the continuous behavior of a system need to be verified in a formal framework which is capable of expressing such behavior. We define *communicating linear hybrid automata* (CLHA) as a modeling framework for the specification of hybrid systems, i.e., systems which have continuous as well as discrete components. CLHA provide modular descriptions of system components and subsume most of the characteristics that are used in verification tools, e.g., discrete and continuous transitions, invariants, and communication through shared variables as well as through synchronization symbols.

In contrast to some other existing modeling paradigms, such as timed automata [AD94] or CSP [Hoa85], our model uses a directed one-to-many way of communication, which, in our experience, better suits the structure of real-life systems than models with only undirected synchronization or one-to-one communication channels.

1.6 Structure of the Thesis

Chapter 2 introduces *programmable logic controllers* (PLCs), the hardware platforms on which the software runs which we will verify. Five programming languages for PLCs, standardized in IEC 61131-3 [IEC98], are introduced. Our focus is on the graphical language *sequential function charts* (SFC).

Two industrial batch plants are described in Chapter 3. These serve as running examples illustrating the verification approaches followed in the subsequent chapters.

Chapter 4 presents a modular verification approach: The system is divided into small units, called modules. Each module is modeled locally, and for every global property that needs to be proven about the system, a minimal set of modules that is needed to fulfill that property is composed, and the validity of the global property is proven by model checking. This

approach is illustrated using the experimental batch plant introduced in Chapter 3.

The compositional verification approach illustrated in Figure 1.2 is described in Chapter 5. It is illustrated using the multi-product batch plant introduced in Chapter 3.

Chapter 6 introduces *communicating linear hybrid automata* (CLHA) as a modeling framework for the specification of hybrid systems. The syntax and a compositional semantics of CLHA are defined formally, and propositional linear temporal logic is introduced as an abstract specification language for CLHA behavior. An example illustrates the use of the presented framework, and an application of the HyTech model checker is shown.

We conclude this thesis in Chapter 7.

Appendix A describes in detail the formal framework of *discrete condition/event systems* which are used as the modeling language in Chapter 4.

The full details of the translation from discrete condition/event systems into the input language of the model checker SMV as used in Chapter 4 are explained in Appendix B.

1.7 Bibliographic Notes

Almost all parts of this thesis are based on earlier publications.

First versions of a formal syntax and semantics for sequential function charts as presented in Chapter 2 have been contributed by the author to [BHLL00b] and [BHLL00a]. Later, timing was added [BHL02]. The presented SFC semantics adds some details to previously published versions.

The experimental batch plant and the multi-product batch plant introduced in Chapter 3 have been used as case studies in several research projects, and descriptions of these plants can be found in the many publications of these projects [Kow98, KS98, Bau00, BKSL00].

The modular verification approach of Chapter 4 and its application to the experimental batch plant have been developed by the author within the VHS project and published as technical reports [Luk99b, Luk99a]. A shortened illustration of these results, along with another verification approach, appeared in [HLL01].

Descriptions of the compositional approach of Chapter 5 have been published in [FSE+01], [FSE+02], and [HLFE02]. The application to the multi-product batch plant (i.e., modeling, algorithmic verification, and deduction) has not been published before in detail.

The syntax of an earlier version of the linear hybrid automata framework presented in Chapter 6 was shown in [FSE+01] and [FSE+02]; the present version has been extended and completely reworked.

Appendices A and B have been published as part of a technical report [Luk99b].

1.8 Acknowledgments

This thesis would not be in its current state without the help and support of many individuals.

I thank Willem-Paul de Roever for many things: for providing an enjoyable research environment at the Chair of Software Technology, for widening my horizon by supporting visits abroad, especially my stay at SRI International (many thanks to John Rushby and his colleagues for six excellent months), and for supporting me in finishing this thesis.

Very enjoyable has been the work with Ralf Huuck, my colleague in all research projects I have worked in. I am grateful for his collaboration as well as many other memorable events.

I also thank all of my present and former colleagues at the University of Kiel for an excellent working atmosphere, interesting discussions, relaxing chats in the hallway and on our IRC channel, and help on many occasions. Martin Steffen, Kai Baukus, Karsten Stahl, and Marcel Kyas deserve a mention here. Sincere thanks for proof reading go to Kai, Martin, and Ralf.

Many stimulating cooperations and visits were supported by Yassine Lakhnech and his colleagues at Verimag.

The Process Control Laboratory at the Department of Biochemical and Chemical Engineering at the University of Dortmund has been our partner in several research projects.[7] [8] [9] [10] I thank Sebastian Engell and his colleagues, particularly Stefan Kowalewski, Olaf Stursberg, Nanette Bauer, Goran Frehse, and Sven Lohmann, for fruitful collaborations over the past seven years.

My brother Jan helped me programming the LaTeX-to-SMV compiler prototype. Thank you for helping out with tackling Lex and YACC.

Our secretaries Sabine Hilge and especially Änne Straßner have always been very helpful when administrative tasks threatened to slow down my scientific progress. Thanks a lot!

Finally, I owe a lot of support to my parents.

> "I don't know half of you half as well as I should like; and I like less than half of you half as well as you deserve."
>
> J.R.R. Tolkien, *The Lord of the Rings*

You do the math.

[7] DFG project on Specification and Verification of Discrete Controllers for Continuous Systems Based on Modular and Compositional Analysis (RO 1122/2-1, RO 1122/2-2)

[8] EU Esprit long-term research project 26270 on Verification of Hybrid Systems (VHS)

[9] DFG project on Integrated Algorithmic and Deductive Verification of Distributed Control Systems for Hybrid Processes (LA 1012/5-1, RO 1122/7-1)

[10] DFG project on Transformation Procedures for Sequential Function Charts and Statecharts (LA 1012/6-1, RO 1122/10-2)

Chapter 2

Programmable Logic Controllers

This chapter introduces *programmable logic controllers,* PLCs for short, a class of hardware platforms used for automated control purposes in the field of industrial process control.

2.1 What is a PLC?

A PLC is a piece of hardware (usually a single-board computer) equipped with a microprocessor, memory, and various kinds of input/output ports. It operates in a cyclic manner. One PLC cycle involves the following three phases (see Figure 2.1): first input values are sampled at the input ports and stored into memory, then a computation takes place, and finally output values are written to the output ports. This cyclic operation is repeated until the PLC is interrupted by external intervention, e.g., shutdown or reset commands. The cycle time is usually around a few milliseconds, depending on the timing restrictions of the controlled process.

The control software inside a PLC is not hard-coded into its hardware; it is usually written with the support of external programming environments (e.g., on a PC system) and then downloaded into the PLC memory. This approach allows for easy reconfiguration of the control functionality during tests as well as the operational phase.

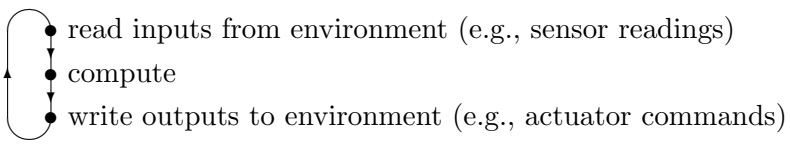


Figure 2.1: The three phases of a PLC cycle

2.2 Fields of Applications

PLCs are mainly used to control industrial processes (or parts thereof) where there is usually little or no need for human interaction, such as chemical production processes, packaging lines, or power plants. Most PLCs work reliably in hazardous environments and can therefore be located close to the controlled process itself.

PLCs are less useful if very fast response times (some microseconds or even nanoseconds) or very complex calculations are required, as, e.g., in aviation, audio/video processing, or high-speed communication equipment. For such applications, specialized hardware components, e.g., signal processors, are used.

Related to PLCs are *embedded controllers,* which are specifically designed and built for fixed control functions inside consumer products suitable for mass production, e.g., mobile phones, electronic toys, or washing machines. These controllers are often smaller and cheaper than PLCs, but usually not fully reconfigurable after production.

2.3 Programming PLCs

The software for PLCs is usually developed with the aid of programming environments which typically run on a PC system. The programming process is often supported by debugging and simulation tools which allow to test a program before it is compiled and transferred to the PLC.

2.3.1 The IEC 61131-3 Standard

The confusing variety of programming languages supported by different PLC vendors led to the IEC 61131-3 Standard [IEC98] (called "the standard" in the following) which defines a small set of the programming languages for PLC software which are mainly used in industry. After introducing common elements such as data types, two textual and two graphical programming languages are defined.

Next we briefly introduce each of these four languages. Figure 2.2 shows a small example for programs written in these languages.

Instruction List (IL)

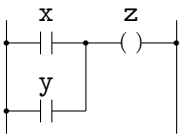
Instruction List is a textual assembler-like language for a one-register machine. The register, called "current result", can operate on the PLC variables with load and store instructions as well as algebraic and Boolean operations. The program flow can be controlled by conditional and unconditional jumps, and modularity can be achieved by using call and return instructions.

```
LD  x
OR  y
ST  z
```

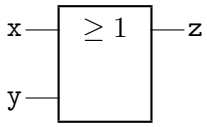
Instruction List

```
z := x OR y
```

Structured Text



(Relay) Ladder Diagram



Function Block Diagram

Figure 2.2: Implementations of the operation "x becomes $y \vee z$"

Structured Text (ST)

The second textual language is Structured Text, which is a dialect of Pascal. Its programming constructs include function calls, if–then–else, case, and for, while–do, and repeat–until loops.

Ladder Diagram (LD)

The graphical language Ladder Diagram (sometimes called Relay Ladder Diagram) uses a network of graphic symbols to describe PLC operations. These symbols are used to read and write variables and are placed between and connected to two "power rails". The "flow of current" from left to right denotes the transfer of Boolean values. Historically, this language emerged from hard-wired logical circuits using relay switches for Boolean variables.

Function Block Diagram (FBD)

The Function Block Diagram language uses a network of electrical circuit diagrams consistent with IEC 617-12 to describe logic operations on Boolean PLC variables.

Sequential Function Chart (SFC)

The standard introduces sequential function charts (SFCs) as a graphical means for structuring PLC programs written in one of the four languages described above. But since SFCs have all the characteristics of a programming language, such as case distinctions and while loops, it is justified to consider them as a fifth PLC programming language.

SFCs as given in IEC 61131-3 are based on IEC 60848 [IEC92] which defines the specification language Grafcet. The Grafcet language is based on Petri nets. For details on Grafcet and Petri nets see [DA92].

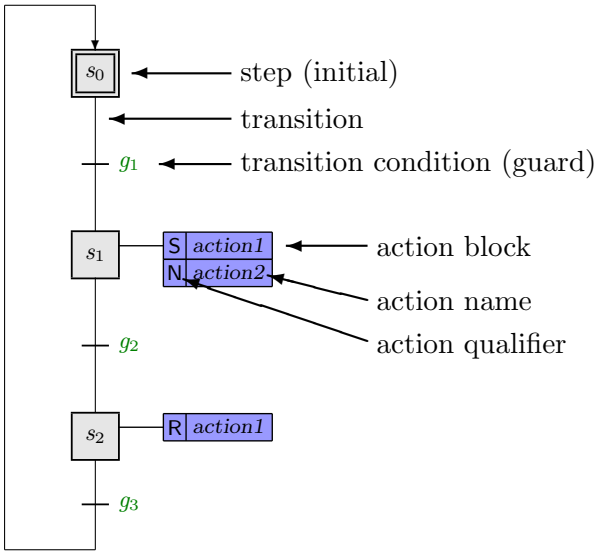


Figure 2.3: Elements of sequential function charts

An SFC is given as a transition system, cf. Figure 2.3. Its locations are called *steps,* with one step denoted (with a double-framed box) as the *initial step.* Steps can be connected by *transitions.* Each transition is labeled with a *transition condition* (also called *guard*). Guards can be written as Boolean expression in one of the four other PLC programming languages. If not denoted otherwise by an arrowhead, transitions are directed downwards. In addition to the single-sequence transitions shown in Figure 2.3, which relate exactly one step to one another, there are various possibilities for alternative and parallel branching as shown in Figure 2.4. *Sequence selections* implement alternative choice: only one of several transitions starting at the same source step can be taken. *Simultaneous sequences* implement parallelism: When the source step is active and the guard is enabled, all the target steps become active simultaneously. Furthermore, there are constructs for the convergence of parallel branches.

With each step a (possibly empty) set of *action blocks* is associated. Each action block consists of an *action qualifier* and an *action name.* An action name can be a Boolean variable, a call of another PLC program written in one of the four languages mentioned above, or the identifier of another SFC, which introduces hierarchy for SFC programs. The action qualifier associated with an action name determines when and for how long the action is executed. Table 2.1 lists all action qualifiers, which are explained below.

The execution of SFC programs happens on two levels: the *changes of step activity* and the *execution of actions.* Initially, only the initial step is active ("has a token" in the Petri net sense), and the activity of steps changes by moving tokens through transitions. The actual computation and

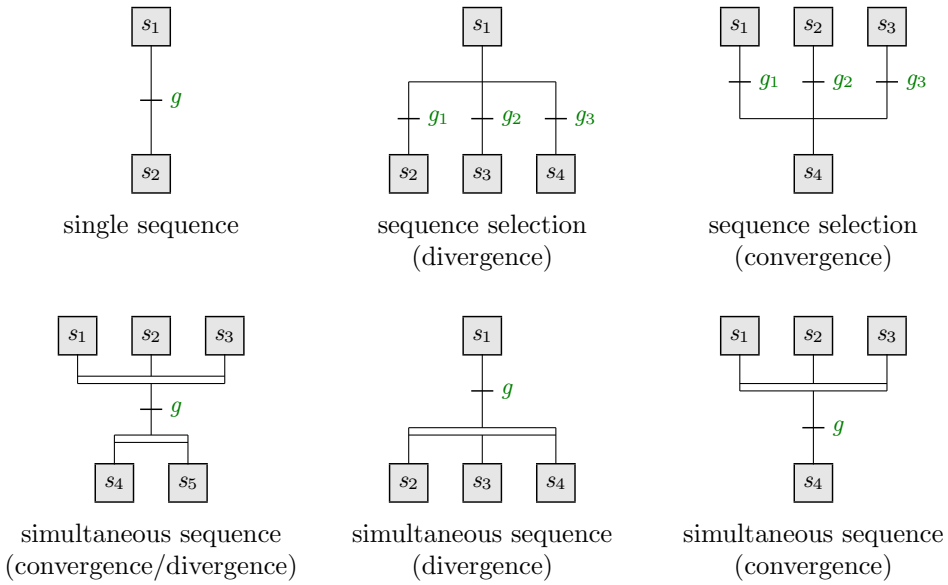


Figure 2.4: Transition types in sequential function charts

Table 2.1: Action qualifiers

untimed qualifiers		timed qualifiers	
N	non-stored	L	limited
R	reset	D	delayed
S	set (or stored)	SD	stored and delayed
P0	pulse (falling edge)	DS	delayed and stored
P1	pulse (rising edge)	SL	stored and limited

control functionality of an SFC program is given by its actions, which are executed depending on their associated action qualifiers at active steps: For any action name, the set of qualifiers associated with that name in action blocks of active steps is collected. The N qualifier executes the action as long as the step is active, e.g., in Figure 2.3 *action2* is executed as long as s_1 is active. The S qualifier marks the action as "stored"; it will be executed until a R qualifier occurs for that action, even if the step with the S qualifier is exited in the meantime. The R qualifier has priority over all other qualifiers, e.g., *action1* will never be executed whenever s_2 is active. The pulse qualifiers execute an action for one PLC cycle only when the step is entered (P1) or exited (P0). The timed qualifiers are used in conjunction with a time parameter to limit or delay actions depending on the activity of steps.

For those familiar with the concept of logical circuits and function blocks, the standard defines the semantics of the action qualifiers by a block diagram

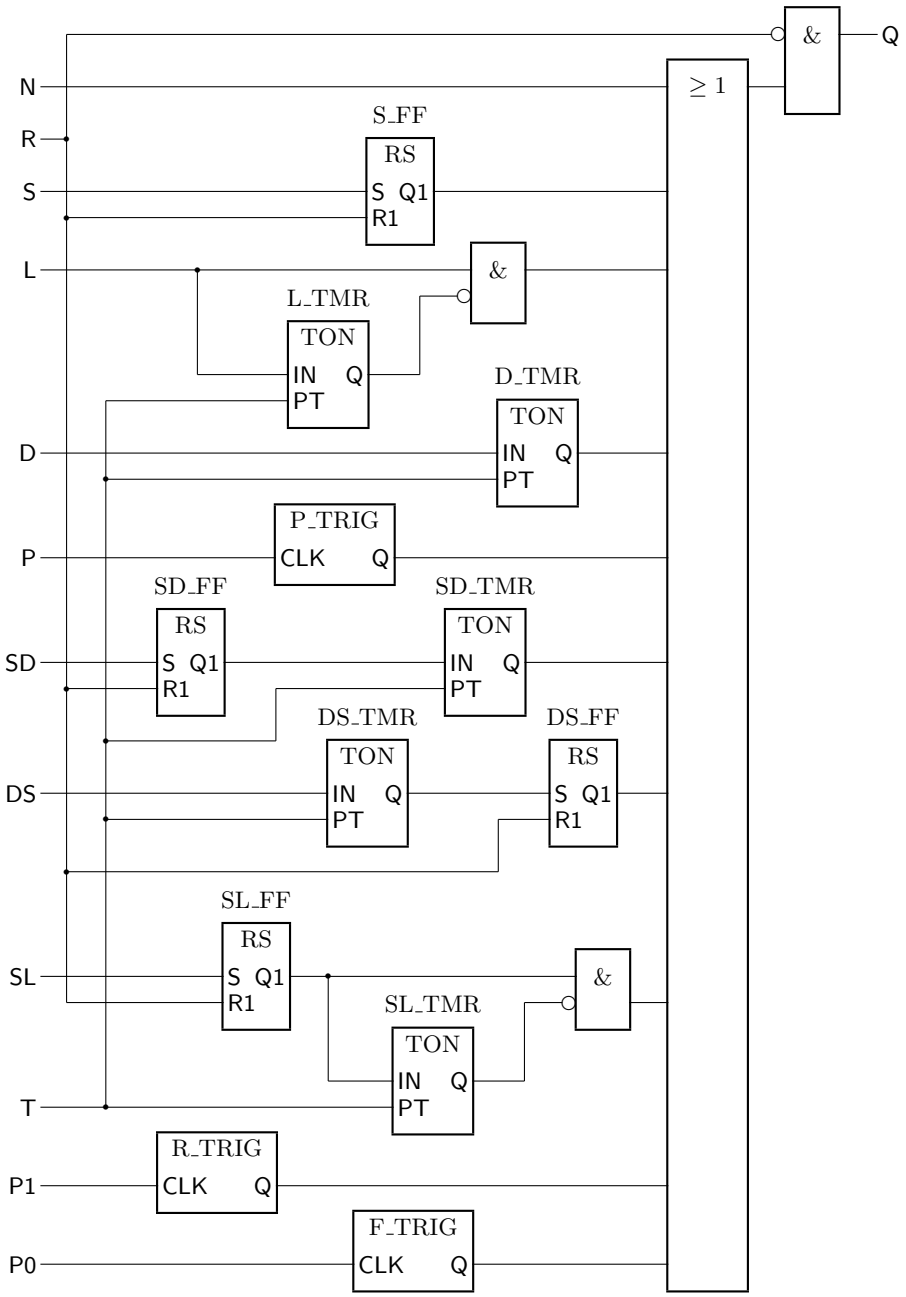


Figure 2.5: The block diagram ACTION_CONTROL given in IEC 61131-3

ACTION_CONTROL associated with each action name, see Figure 2.5. The Boolean inputs on the left are set to 1 if and only if the respective qualifier appears in an active step, the input T is the time parameter, and the Boolean output value Q on the right determines if the action is executed or not.

The next section introduces a formal operational semantics for SFCs.

2.4 PLC Semantics

Even though the standard defines the syntax for PLC programs and some aspects of their semantics (e.g., the meaning of action qualifiers through the block diagram in Figure 2.5) quite clearly, the overall semantics of these is not always obvious. Especially SFCs lack a formal execution model which is free from ambiguities. As comparisons of a wide range of PLC programming environments show [Bau03, BHLL04], there is no common semantics for the same SFC in different tools. This is a big drawback for the interoperability of PLC software, especially since the need for interoperability among different PLC manufacturers was one of the very reasons for introducing IEC 61131.

This section introduces a formal semantics for SFCs which is based on the analysis of the standard as well as observations of the SFC behavior implemented in various PLC programming environments. This semantics can be parameterized to match the particular semantics of most PLC programming environments.

2.4.1 SFC Syntax

A prerequisite for a formal semantics is a clear syntactic description of the objects it reasons about. This section introduces a syntax for SFCs. We only present a model without quantitative timing, i.e., we omit the timed action qualifiers listed in Table 2.1. See [BHL02] for a discussion of timed action qualifiers.

PLC programs operate on a finite set of *typed variables.* A type-respecting function assigning values to the variables is called *state.*

Definition 2.1 (variables, states) Let V be a finite set of *variables,* and let *type* be a function assigning a type (a set of values like $\mathbf{B} = \{true, false\}$) to each variable. A function σ assigning to each variable $v \in V$ a value $\sigma(v) \in type(v)$ is called *state.* We denote the set of all states by Σ.

As mentioned before, action names in SFCs can refer to a Boolean variable, to another SFC, or to the call of another PLC program written in one of the four programming languages IL, ST, LD, or FBD. For the sake of simplicity, we will use in the latter case a *state transformation* which represents the variable modifications taking place in the referenced PLC program.

Definition 2.2 (state transformation) A *state transformation* is a function $f : \Sigma \to \Sigma$. Let F be the set of all state transformations.

We define W as the set of Boolean variables in V which appear as action names in the SFCs.

Definition 2.3 (actions, action qualifiers) We define $A = W \cup SFC \cup F$ (with *SFC* defined below) as the set of *actions,* and $U = \{\mathsf{N}, \mathsf{R}, \mathsf{S}, \mathsf{P0}, \mathsf{P1}\}$ as the set of *action qualifiers.*

The behavior of an SFC depends on the order in which its actions are executed. Thus, an irreflexive ordering $\sqsubset \subseteq A \times A$ on actions needs to be defined, such that $a_1 \sqsubset a_2$ means that action a_1 must be executed before action a_2. Any two actions which use a common variable need to be ordered. For the sake of simplicity in the following definitions, we demand (without loss of generality) that $\sqsubset$ is linear. The ordering depends on the particular PLC programming tool that we want to represent.

A transition t is represented as a triple (Q, g, Q'), where Q is the set of source steps which have to be active before firing the transition and Q' is the set of target steps which are active after the transition has been taken. E.g., the transition at the bottom left of Figure 2.4 is represented as $(\{s_1, s_2, s_3\}, g, \{s_4, s_5\})$. Transitions which are part of sequence selections are given as several single-sequence transitions (i.e., the diverging sequence selection in Figure 2.4 is represented by three transitions: $(\{s_1\}, g_1, \{s_2\})$, $(\{s_1\}, g_2, \{s_3\})$, and $(\{s_1\}, g_3, \{s_4\})$). A guard g is a Boolean expression over the variables in V. Furthermore, the Boolean expression $s_i.X$ can be used in g, representing that step s_i is currently active.

For a set T of transitions, we define the abbreviations

$$src(T) = \{\, Q \mid (Q, g, Q') \in T\} \text{ and } tgt(T) = \{\, Q' \mid (Q, g, Q') \in T\}$$

as the unions of all source, respectively target, steps of transitions in T.

Similar to the ordering on actions, we also need a partial ordering $\prec$ on transitions to determine which of several alternative transitions in a sequence selection has priority. Any two transitions $t_1 = (Q_1, g_1, Q_1')$ and $t_2 = (Q_2, g_2, Q_2')$ have to be ordered if they share a common source step, i.e., if $Q_1 \cap Q_2 \neq \emptyset$. The term $t_1 \prec t_2$ means that transition t_1 has priority over t_2. In contrast to the global action ordering $\sqsubset$, the transition ordering will be defined locally in each SFC.

Now we can define the formal syntax for SFCs. Let *SFC* be the set of all SFCs $\mathcal{S}$ defined as follows:

Definition 2.4 (SFC) An SFC is a 5-tuple $\mathcal{S} = (S, s_0, T, a, \prec)$, where S is a finite set of *steps,* $s_0 \in S$ is the *initial step,* T is a finite set of *transitions,* $a : S \to 2^{U \times A}$ is an *action labeling function* which assigns a finite set of action blocks to each step, and $\prec \subseteq T \times T$ is an irreflexive partial order on transitions.

For any $b \in U \times A$, we use b_q to denote the first component of b (the action qualifier) and b_a to denote the second component of b (the action name).

The orders $\sqsubset$ and $\prec$ are used to adapt our model to the semantics used by the different PLC programming tools. For transitions, the IEC 61131-3 standard defines the default priorities to be assigned "from left to right", and some tools allow to add explicit priorities. For actions, many different ways to define the priorities exist, e.g., by the graphical position of action blocks, or by alphabetical or user-defined orderings of action names. See [Bau03, BHLL04] for a list of tool-specific transition priorities and action orderings.

2.4.2 Operational SFC Semantics

Now we provide a semantics for SFCs. Let $\mathcal{S} = (S, s_0, T, a, \prec)$ be an SFC, and let $\mathcal{S}_i = (S_i, s_{0,i}, T_i, a_i, \prec_i)$, $i = 1, \ldots, n$, be the SFCs nested recursively inside the action blocks of $\mathcal{S}$. For a global, flat access to the nested structure we define $\bar{S} = S \cup S_1 \cup \ldots \cup S_n$, $\bar{T} = T \cup T_1 \cup \ldots \cup T_n$, $\bar{a} = a \cup a_1 \cup \ldots \cup a_n$, and $\bar{\prec} = \prec_1 \cup \ldots \cup \prec_n$.

There are two different things we have to keep track of when observing executions of $\mathcal{S}$. First, we need information about the current state of $\mathcal{S}$, i.e., the values of its variables. And we must know in which steps of $\mathcal{S}$ and its sub-SFCs $\mathcal{S}_1, \ldots, \mathcal{S}_n$ control resides, i.e., which steps "have a token". Furthermore, we need information about which actions are currently active and which actions are "stored", i.e., have been activated by an S qualifier in a previous cycle. In Figure 2.5, the latter information is stored in the state of the flipflop S_FF. A timed semantics would also consider the other flipflops and the values of the timers.

It is crucial to notice that there is a difference between control residing in a step and steps whose actions are actually performed. The former we will call *ready* steps, and the latter *active* steps. Each active step is also a ready step, but the converse does not hold, since actions in ready steps of nested SFCs will only be performed if the nested SFC itself is active, because, e.g., it is activated by an N qualifier in a step of a top-level SFC. Since we can always deduce the active steps from the other information, they need not to be saved from one PLC cycle to the following.

We store the above information about $\mathcal{S}$ in a *configuration:*

Definition 2.5 (configuration) A *configuration* of $\mathcal{S}$ is a quadruple $(\sigma, \mathit{ready}, \mathit{activeA}, \mathit{storedA})$, where $\sigma \in \Sigma$ is a state, $\mathit{ready} \subseteq \bar{S}$ is the set of *ready* steps, $\mathit{activeA} \subseteq A$ is the set of active actions, and $\mathit{storedA} \subseteq A$ is the set of stored actions. Let C be the set of all configurations.

Such a configuration is changed in the cycles of an SFC. A cycle of an SFC can be seen as a concretization of the PLC cycle depicted in Figure 2.1 and performs the following sequence:

1. Get new input from the environment and store the information into the state σ.

2. Compute the SFC behavior:

   (a) Execute all active actions in *activeA*, in the sequence determined by $\sqsubset$, changing the state σ into σ'.

   (b) Change the set *ready* into *ready*$'$ by taking transitions. Guards are evaluated using the new state σ', and conflicts are solved using $\prec$. Furthermore, update *activeA* to *activeA*$'$ and *storedA* to *storedA*$'$.

3. Send the outputs to the environment by extracting the required information from the new state σ'.

The interaction with the environment is not discussed here in detail; we focus on items 2a and 2b.

Some PLC programming environments exchange the order of items 2a and 2b above, thus taking the enabled transitions before executing the active actions. Another possible deviation from the semantics presented here is the so-called "final scan logic" which executes each active action once more before it gets deactivated.

We define the operational semantics for SFCs by showing how a configuration changes when one SFC cycle is executed. That is, given a configuration $(\sigma, \mathit{ready}, \mathit{activeA}, \mathit{storedA})$, with σ already containing the new input information from the environment, we compute the next configuration $(\sigma', \mathit{ready}', \mathit{activeA}', \mathit{storedA}')$ by executing all active actions, computing the new set of ready steps, and by updating the sets of active and stored actions. Then the output information for the environment can be extracted from σ'. This single-cycle semantics, which is defined next, can be extended to a multi-cycle semantics which computes a sequence of outputs from a given sequence of inputs from the environment.

Definition 2.6 (SFC semantics) The transition relation $\longrightarrow \subseteq C \times C$ for the SFC $\mathcal{S}$ is defined as follows: $(\sigma, \mathit{ready}, \mathit{activeA}, \mathit{storedA}) \longrightarrow (\sigma', \mathit{ready}', \mathit{activeA}', \mathit{storedA}')$ if and only if

1. $\sigma_1 = (a_m \circ \ldots \circ a_1)(\sigma)$, where $\{a_1, \ldots, a_m\} = \mathit{activeA} \cap F$ and $a_1 \sqsubset \ldots \sqsubset a_m$,

2. $\mathit{ready}' = (\mathit{ready} \setminus \mathit{src}(\mathit{taken})) \cup \mathit{tgt}(\mathit{taken})$, where

   (a) $\mathit{taken} = \{t = (Q, g, Q') \in \mathit{enabled} \mid \neg\exists \bar{t} = (\bar{Q}, \bar{g}, \bar{Q}') \in \mathit{enabled} : Q \cap \bar{Q} \neq \emptyset \wedge \bar{t} \prec t\}$,

   (b) $\mathit{enabled} = \{(Q, g, Q') \in \bar{T} \mid Q \subseteq \mathit{active} \wedge (\sigma_1, \mathit{active}) \models g\}$, and

(c) $active = \{s \in ready \mid s \in S \vee \exists i \in \{1,\ldots,n\} : s \in S_i \wedge \mathcal{S}_i \in activeA\}$,

3. $activeA' = \{act \in A \mid aq(act) \cap \{\mathsf{N},\mathsf{S},\mathsf{P0},\mathsf{P1}\} \neq \emptyset \wedge \mathsf{R} \notin aq(act)\}$ and $storedA' = \{act \in A \mid \mathsf{S} \in aq(act) \wedge \mathsf{R} \notin aq(act)\}$, where $aq = collect(aq_0, S_0)$ with

   (a) $aq_0(act) = \begin{cases} \{\mathsf{S}\}, \text{ if } act \in A \cap storedA \\ \emptyset \quad , \text{ if } act \in A \setminus storedA \end{cases}$,

   (b) $S_0 = S \cup \{S_i \mid i \in \{1,\ldots,n\} \wedge \mathcal{S}_i \in storedA\}$,

   and the recursive function *collect* as given in Figure 2.6, and

4. for all $v \in V$, $\sigma'(v) = \begin{cases} \sigma_1(v), \text{ if } v \notin W \\ true \ , \text{ if } v \in W \cap activeA' \\ false \ , \text{ if } v \in W \setminus activeA' \end{cases}$.

The definition above and Figure 2.6 need some explanation. In step (1), all active actions which represent a state transformation are sorted using the global action ordering $\sqsubset$, and they are used to modify the current state σ in that order. The result is stored in σ_1.

Step (2) describes how transitions are taken. In (2c) the set *active* is computed which contains all active steps. These are all ready steps which are either in the top-level SFC $\mathcal{S}$ (which is always active), or are ready steps of an active nested SFC $\mathcal{S}_i$. The set *enabled* defined in (2b) contains all transitions which are enabled, i.e., transitions where all source steps are active and where the guard is satisfied. The validity of a guard, expressed by $(\sigma_1, active) \models g$, depends on the current variable evaluation σ_1 and the set *active* of active steps, since the activity of a step s can be expressed in g by $s.X$. Not all enabled transitions will be taken: In (2a) the set *taken* contains only those transitions t from *enabled* for which there exists no other conflicting transition t_1 which has priority over t (expressed by $t_1 \prec t$). Two transitions are in conflict if they share at least one common source step. Finally, all transitions in *taken* are fired, which is expressed

function $collect(aq, St)$
  for all $s \in St \cap ready', b \in \bar{a}(s) : aq(b_a) := aq(b_a) \cup (\{b_q\} \cap \{\mathsf{N},\mathsf{S},\mathsf{R}\})$;
  for all $s \in St \cap src(taken), b \in \bar{a}(s) : aq(b_a) := aq(b_a) \cup (\{b_q\} \cap \{\mathsf{P0}\})$;
  for all $s \in St \cap tgt(taken), b \in \bar{a}(s) : aq(b_a) := aq(b_a) \cup (\{b_q\} \cap \{\mathsf{P1}\})$;
  for all $i \in \{1,\ldots,n\}, s \in St, b \in \bar{a}(s)$:
    if $\mathcal{S}_i = b_a \wedge aq(\mathcal{S}_i) \neq \emptyset \wedge \{\mathsf{R}\} \notin aq(\mathcal{S}_i)$ **then** $aq := collect(aq, S_i)$
return aq

Figure 2.6: Recursive collection of action qualifiers

by removing all source steps of these transitions from *ready* and adding all target steps. The result is stored in $ready'$.

In step (3) the sets $activeA'$ (new active actions) and $storedA'$ (new stored actions) are computed recursively over the SFC nesting structure by the function *collect* (shown in Figure 2.6). This function has two parameters: aq is a function which maps actions to the set of action qualifiers that already have been found for that action in action blocks of active steps. E.g., if step s_1 in Figure 2.3 on page 16 is active, we have $aq(action1) = \{\mathsf{S}\}$ and $aq(action2) = \{\mathsf{N}\}$, and if s_2 is active, we have have $aq(action1) = \emptyset$ and $aq(action2) = \{\mathsf{R}\}$. The second parameter St is a set of steps that still need to be visited to collect more qualifiers. The return value of *collect* is aq extended with the qualifiers that have been found while visiting the steps in St.

Initially, *collect* is called with aq_0 which assigns the S qualifier to all actions in $storedA$, since these have to be memorized for the next cycle (if they are not reset by an R qualifier), and with S_0 containing all steps of the top-level SFC $\mathcal{S}$ and each nested SFC $\mathcal{S}_i$ which is active because $\mathcal{S}_i$ is a stored action.

The function *collect* adds qualifiers to aq in four for-loops. The first loop collects all N, S, and R qualifiers from the steps which are active at the beginning of the next cycle. The next two loops add the P0 (respectively, P1) qualifier if the step having that qualifier in one of its action blocks is in a source step (respectively, target step) of a transition that was taken in this cycle. The last loop recursively collects the action qualifiers from each nested SFC $\mathcal{S}_i$ which has been activated by a qualifier in one of the steps in St and which is not reset somewhere else.

After all qualifiers have been collected, $activeA'$ contains all actions for which there exists at least one activating qualifier and no reset qualifier. These actions will be executed at the beginning of the next cycle. The set $storedA'$ contains all actions for which there exists an S qualifier and no reset qualifier.

Step (4) updates all Boolean variables which appear as action names in the SFCs. Each of these variables $v \in W$ is set to *true* if they are in $activeA'$, i.e., they are active at the beginning of the next cycle, and set to *false*, otherwise. All other variables get their values from the state σ_1. The resulting state is stored in σ'.

The single-cycle semantics $\longrightarrow$ given in Definition 2.6 can be extended to a multi-cycle trace semantics. The initial configuration $(\sigma_0, ready_0, activeA_0, storedA_0)$ of the SFC $\mathcal{S}$ is the following: σ_0 contains the initial variable evaluation, assigning *false* to all Boolean variables and 0 to all numerical variables, $ready_0$ and $activeA_0$ both contain only the initial step s_0 of the top-level SFC, and $storedA_0 = \emptyset$, i.e., there are no stored actions. The initial configuration may be different if there are any action blocks associated

with the initial step s_0, which is discouraged, since this may lead to semantic ambiguities. By iteratively applying the single-cycle semantics $\longrightarrow$ to the initial configuration, a sequence of configurations is generated which describes the behavior of the SFC $\mathcal{S}$ over time. The behavior of the environment (i.e., the changes of the input variables) needs to be inserted into σ between consecutive applications of $\longrightarrow$.

2.5 Verification of SFC Programs

The formal semantic model presented in the previous section enables us to apply formal verification techniques to PLC software. The direct use of the operational semantics is however difficult, for various reasons. The calculation of active steps and actions is rather complex in the general case, and a direct implementation in a verification tool, e.g., the model checker SMV, is tedious, though doable, see [Huu03]. Given a concrete SFC program, often simplifying abstractions are obvious and can be incorporated into the verification process, whereas a direct translation of the SFC according to the operational semantics would lead to an unnecessarily large model. Furthermore, given SFCs often do not contain structures like nested SFCs or the pulse qualifiers P0 and P1 which contribute much to the complexity of the presented operational semantics.

Therefore, the model-checking approaches to the verification of SFCs followed in Chapter 4 and Chapter 5 translate SFCs intro transition systems which resemble the step/transition structure of the SFC rather than the full execution model of Definition 2.6 considering all details of the cyclic execution.

A model-checking-based verification approach which implements the full operational SFC semantics, including timed action qualifiers, can be found in [Bau03].

Chapter 3

Case Studies

This chapter introduces two chemical laboratory batch plants which serve as running examples for the remainder of this work. Both plants are used for teaching and research in the Chemical Process Control Laboratory at the University of Dortmund, Germany. Although they are of smaller size and reduced complexity compared to industrial-scale plants, they still contain the main features of real-life systems. As a consequence, both plants were used, among other industrial examples, as case studies in the VHS project[1] [VHS], where they were known as "CS1" and "CS7".

3.1 The Experimental Batch Plant

The first of the two plants combines a mixing and a separation step into a closed recycling process. Figure 3.1 shows the piping and instrumentation diagram (P/I diagram for short) of the plant. The P/I diagram describes the layout of the physical devices of the plant without the control equipment. The plant consists of seven tanks equipped with level sensors (LIS in the P/I diagram) for measuring the amount of liquid in the tank. Some tanks also have sensors to determine the salt concentration in the liquid (QI/QIS) and/or temperature (TI/TIS). There are also some sensors to measure water flow (FIS) and water pressure (PIS). Tank B3 is equipped with a mixing unit, tank B5 with a heater, and tanks B6 and B7 with cooling devices. Steam which evaporates from B5 is collected in condenser K1, which is also equipped with a cooling device. Tank B3 is called *reactor,* and tank B5 *evaporator.* Furthermore, there are two pumps (P1 and P2) for transferring liquids against gravitation. Tanks and pumps are connected with pipes which can be opened and closed through valves (V1, ..., V29).

Initially, the storage tank B1 is filled with a highly concentrated salt (sodium chloride, NaCl) solution, and B2 is filled with water. Then both liquids are mixed in reactor B3. This results in a salt solution with a lower

[1] VHS – Verification of Hybrid Systems. EU Esprit Long-Term Research Project 26270.

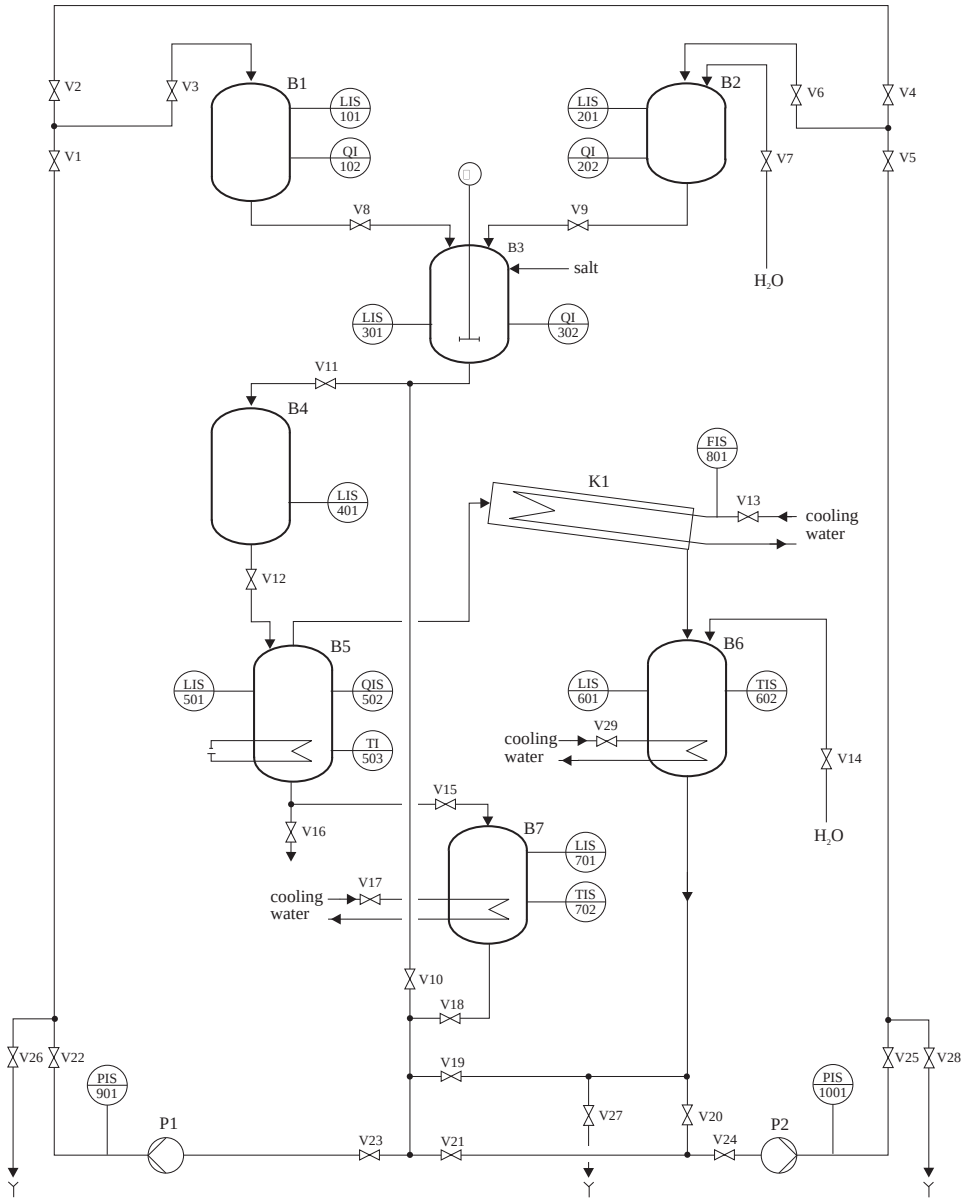


Figure 3.1: P/I diagram of the experimental batch plant

concentration, which is drained into storage tank B4. The next process step takes place in the evaporator B5. The salt solution is heated until steam evaporates. The steam is cooled down in condenser K1, and the condensate (distilled water) is collected in tank B6. The evaporation process is kept running until the salt concentration level in evaporator B3 reaches the original concentration level of the liquid in storage tank B1. Then the highly concentrated solution is drained into tank B7. The hot liquids in

Table 3.1: Control programs for the experimental batch plant

Name	Function
B2	Startup: Fill B2 with fresh water via V7
B3	Mix lower concentrated solution in B3, drain into B4
B3K	Startup: Mix highly concentrated solution in B3; salt is added manually
B3U	Startup: Pump manually produced highly concentrated solution from B3 to B1 via P1
B5	Drain solution from B4 into B5, evaporate solution in B5 until high concentration is reached, drain highly concentrated solution into B7
B6	Cool down the distilled water in B5
B6A	Shutdown: Drain B6 via V27
B6S	Pump the distilled water from B6 into B2 via P1
B6U	Pump the distilled water from B6 into B2 via P2
B7	Cool down the highly concentrated solution in B7
B7U	Pump the highly concentrated solution from B7 into B1 via P1
SP1	Maintenance: Rinse the outer ring pipe with water from B2
SP2	Maintenance: Rinse the outer ring pipe with water from B6

tanks B6 (distilled water) and B7 (highly concentrated salt solution) are cooled down to ambient temperature, and are finally pumped back into the storage tanks B2 and B1, respectively, which completes the recycling run.

The plant is controlled by a Siemens S7-300 PLC system which reads the input values from the sensors listed above and controls the valves, the pumps, and the mixing and the heating units. The software consists of 13 independent control programs implementing the production steps described above as well as startup, shutdown, and maintenance functions. Table 3.1 shows the program names along with their function. All control programs are written in the Sequential Function Charts (SFC) language.

A complete mixing and recycling run is executed by running the following programs (in that order): B3, B5, B6, B7, B6S, and B7U. Program B6U can be used to replace B6S if P1 is not working. For the initialization of the plant program B2 is used to fill tank B2 with water, and programs B3K and B3U, together with some human interaction involving a ladder and a box of salt, are used to produce the highly concentrated solution in tank B1. In regular intervals the pipes and valves need to be cleaned, otherwise the pipes get clogged and valves may get stuck. Control programs SP1 and SP2 run this cleaning procedure.

Complete descriptions of the experimental batch plant and its control software can be found in [Kow98, KS98].

Using the experimental batch plant in teaching and research revealed three disadvantages of this particular plant construction:

1. There is only one "product", and there are no redundant resources to produce it. Therefore, interesting aspects such as concurrent production, resource allocation, and scheduling are not covered.

2. The processing times (especially of the evaporation procedure) are too long to use this plant efficiently for teaching and demonstration purposes.

3. The plant cannot be used as a demonstration object at science fairs or conferences, since it is not transportable, and it is not a particular "eye-catcher", e.g., spectators cannot tell the difference between salt solutions of low and high concentration, since they both look the same.

With these aspects in mind, a second plant was constructed, which is described next.

3.2 The Multi-Product Batch Plant

The piping and instrumentation diagram of the second plant is shown in Figure 3.2. The plant's purpose is to provide two different liquid products called "blue" and "green", which are to be stored in the product tanks B31 and B32, respectively. Three independent reactors (R21, R22, and R23) equipped with mixing units (M1, M2, and M3) are used to produce these two products from three different raw materials. These raw materials, called "yellow", "red", and "white",[2] needed for the production are available in the storage tanks B11, B12, and B13. These storage tanks can be refilled from external tanks (B41, ..., B44, not shown in the diagram) via three pumps (P1, P2, and P3). The tanks and reactors are equipped with level sensors (LIS11, ..., LIS32) measuring the amount of liquid and are connected by pipes. Each connecting pipe between two vessels contains exactly one valve (V111, ..., V312) for opening and closing that pipe.

The production of "blue" and "green" is run in batches: each of the tanks is capable of containing a maximal number of fixed volumes, called batches. The raw material storage tanks and the reactors have a maximum capacity of two batches, and the product tanks can hold up to three batches.

Producing "blue" works as follows: first one of the available reactors is filled with one batch of "yellow" from B11. Then the mixing unit is turned on, and one batch of "white" from B13 is added. During the blending

[2] Actually, the liquid called "white" is transparent.

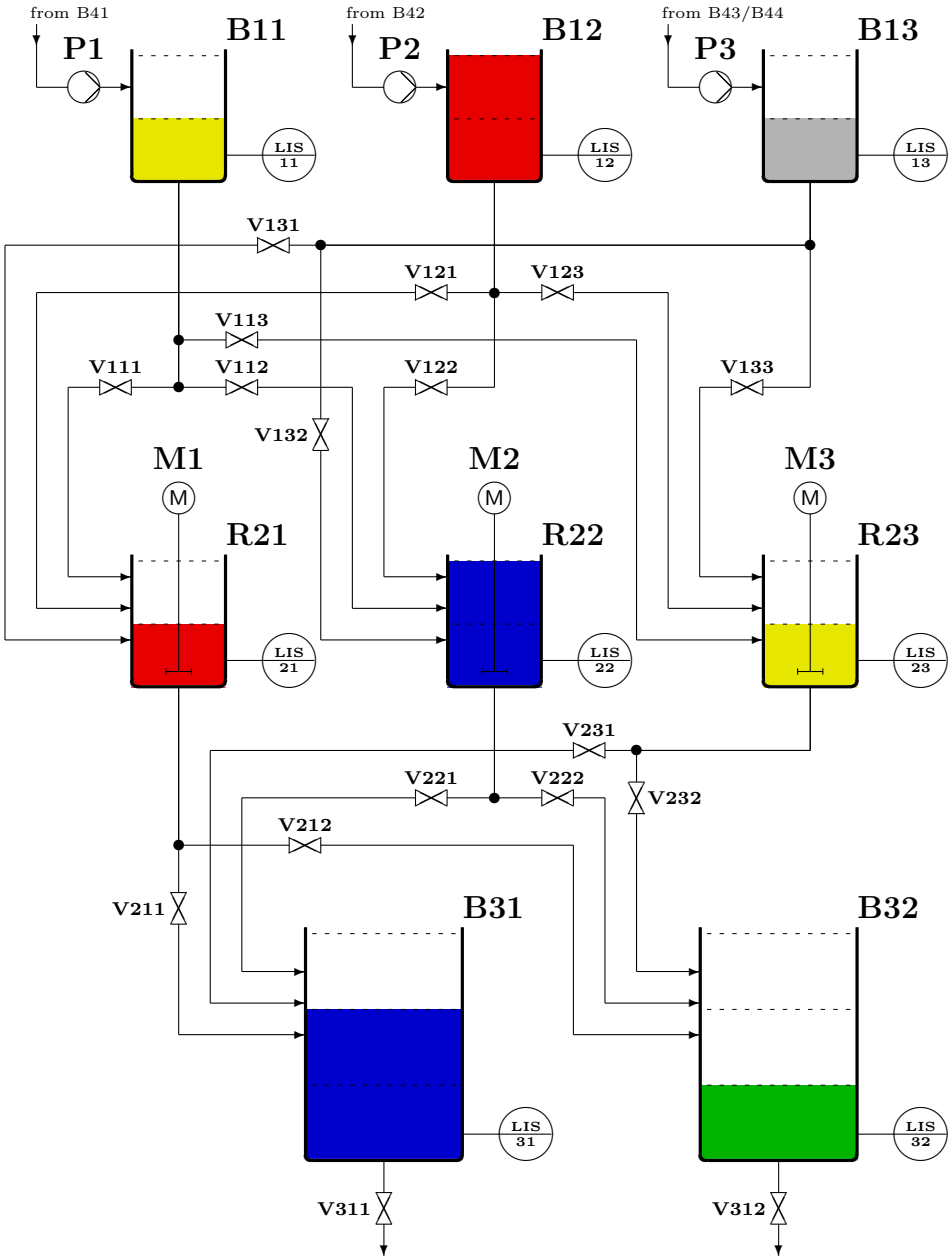


Figure 3.2: P/I diagram of the multi-product batch plant

process, the liquid turns into "blue". After the mixing unit is stopped, the reactor is drained into product tank B31. The product "green" is created analogously by mixing "red" and "white" in one of the reactors. Since one batch of a product is the mixture of two raw material batches, one batch in a product tank has twice the volume of one batch of raw material.

The change of colors is caused by a neutralization reaction: "yellow" and "red" are diluted hydrochloric acid (HCl) with different pH indicators which change their color from yellow into blue and from red into green, respectively, when neutralized. The third raw material "white" is diluted sodium hydroxide (NaOH) without any pH indicator.

Note that the existence of three independent reactors makes it possible to run the production of several batches of "blue" and "green" in a concurrent fashion. The concurrency is limited however, since any vessel should neither be filled from two different sources at the same time nor filled and emptied simultaneously.

In addition to the chemical apparatus shown in Figure 3.2, there are several layers of control software running on a PLC system. The main purpose of the software is to implement the production as described above, obeying the restrictions upon concurrent operation. Among the software tasks are:

Scheduling Based on input data like raw material delivery times and future demands for "blue" and "green", a production schedule is generated.

Raw material delivery Whenever new raw material arrives in tanks B41–B44, the pumps P1–P3 can be used to refill the raw material tanks B11–B13.

Production This software controls the production of "blue" and "green" in the reactors R21–R23 as described above.

Resource management It has to be ensured that the access to hardware resources (e.g., reactor R21) is exclusive, i.e., two control programs must not use the same resource simultaneously.

Emergency shutdown, maintenance, etc. There are several programs used for exceptional plant operations, e.g., rinsing all pipes and tanks with water.

Complete descriptions of the multi-product batch plant and its control software can be found in [Bau00, BKSL00].

Chapter 4

Modular Verification

As we have argued in Section 1.3, the main challenge in formal verification is the state explosion problem emerging from the parallel composition of many system components. This chapter introduces a modular verification approach which uses decomposition and a minimization technique for the composition of system parts to mitigate complexity issues.

This approach is demonstrated by the verification of safety properties of the first batch plant introduced in Chapter 3.

4.1 Introduction

During the verification task of proving a list of properties about a system, several decisions need to be made which influence the effectivity (i.e., the proof is possible) and efficiency (i.e., the proof can be done with small effort) of the verification process:

- The system needs to be modeled in a formal framework. This framework must be capable of modeling the behavior that is referred to in the list of properties.
- During the modeling phase, the level of abstraction must be selected in such a way that the properties can still be proven, without introducing too much details which slow down the verification process or even make it infeasible.
- The way of formal reasoning has to be chosen. Algorithmic methods like model checking are fully automatic, whereas deductive reasoning is undertaken manually, though tool support exists (e.g., theorem provers).

In this chapter we choose discrete condition/event systems as our modeling framework, since these systems can be structured modularly and have a flexible means for describing communication structures. For proving the

properties we use algorithmic verification (model checking with the SMV tool).

4.2 The Modular Verification Approach

The verification approach followed in this chapter is based on decomposition, abstraction, and model checking of (re)composed of parts of the system.

First the system is decomposed into small parts, called modules. For each module, a suitable abstraction (i.e., a simplification that does not lose essential information needed for the verification) is chosen. Each abstracted module is modeled in a state-based framework which allows formal verification by model checking.

The core of the modular approach aims at minimizing the number of modules that need to be composed for model checking. For each property that needs to be verified, only those modules are composed which directly or indirectly influence the variables of that property. In same cases, even fewer modules are sufficient to prove the property. If the set of composed modules can be kept small, state explosion only has a minor impact on the feasibility of algorithmic verification.

In the following, the details of this verification approach are worked out on a running example.

4.3 Example

We illustrate the modular verification approach by proving some properties of the first batch plant introduced in Chapter 3.

The experimental batch plant can be divided into two parts, the physical part consisting of tanks, valves, pumps, the heater, the condenser, cooling units, piping, and sensors (temperature, water level, concentration, pressure), and the control part, namely the distributed control system with the 13 control routines given as sequential function charts. The control part receives sensor readings from the physical part and sends control commands to the physical devices. This communication structure is shown in Figure 4.1.

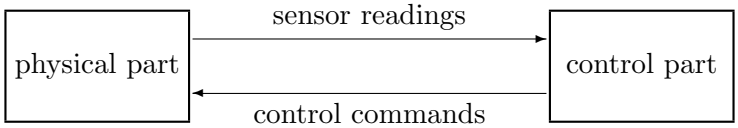


Figure 4.1: Communication structure of the experimental batch plant

The verification goal of this chapter is to establish some properties of the control programs on a formal basis. We focus on safety requirements which help to ensure that the control programs will not cause any damages to the

plant, e.g., by proving that the heater is never switched on while the tank is empty.

In order to check these requirements formally, we need a modeling framework which allows formal verification. Our approach uses discrete condition/event systems (DCESs) for both the physical and the control part. A modular structure is used; we define one DCES model for each valve, pump, tank, heater, and also one DCES for each of the 13 control programs. This enables modular reasoning in the verification process.

Since DCESs are finite-state, they can be verified algorithmically using model checking. We use the tool SMV (Symbolic Model Verifier) to perform this verification. A compiler prototype translates a set of given DCESs into an SMV input file, and each safety property is written as a formula in CTL (computation tree logic), which is then verified automatically by SMV.

4.3.1 Properties

We want to check safety properties of the control programs. The following ones have to be checked for each of the control programs:

- Whenever a control program terminates, all valves are closed.
- Whenever a control program terminates, the pumps are switched off.
- Whenever a control program terminates, the heater is switched off.

These properties can be seen as "generic", since they apply to a wide range of control programs for chemical processes without being specific about the nature of the controlled process, such as how tanks are connected or what kind of chemical reaction takes place. Such generic properties are especially interesting since they can be introduced without prior knowledge about the plant.

If these three properties are valid in our plant, we are sure that the property "all valves closed, pumps off, heater off" is not only fulfilled by the initial plant state, but also holds after the termination of a control program. Since the top-level control of the plant (run by an operator or a scheduling program) is furthermore designed in such a way that two programs never run simultaneously, we know that during normal operation of the plant the state at the start of a program always fulfills "all valves closed, pumps off, heater off". This allows us to analyze the programs independently, i.e., when checking the properties of one program, we do not have to consider the commands sent to valves, pumps, or the heater by other programs running before or simultaneously.

However, the states of the tanks are *not* preserved by the programs. We have to take this into account when verifying properties which depend on tank states.

The following properties are specific to this plant and should be satisfied to ensure a safe operation of the plant.

To prevent the heating unit in tank 5 to be damaged by overheating we need to establish the following property:

- Whenever the heater in tank 5 is turned on, the water level of tank 5 is high enough.

We demand that during the heating process steam can leave tank 5 into the condenser only:

- Whenever the heater in tank 5 is working, none of the valves 12, 15, and 16 are open.

Furthermore, the cooling device in the condenser must be provided with cold water during the heating process to prevent a dangerously high pressure inside the condenser:

- Whenever the heater in tank 5 is working, valve 13 is open.

The regular pressure of the cooling water supply is not sufficient if all three cooling units operate at the same time. The following property prevents this undesired situation:

- At most two cooling units are active simultaneously.

Although the pumps are equipped with pressure limit switches, we demand the following to prevent damages of the pumps:

- Pumps are not pumping against closed valves.

We do not want an uncontrolled flow of water through a tank; if some amount of liquid is to be transferred through a tank, it should be filled into that tank completely and drained from the tank afterwards:

- Each tank's input and output valves are not open simultaneously.

To check the properties listed above formally, we need formal models for all parts of the plant. The next section presents an abstract model of the plant hardware as well as of the control programs.

4.4 Plant Model

We use a discrete, untimed model. Although it is quite simple, many safety aspects of the interaction of the plant and its control programs can be examined. As the modeling paradigm we use *discrete condition/event systems* (DCESs), which are introduced in [SK91]. To illustrate the connections within a network of DCESs more clearly, we assign names to their input

and output components, and thus extend them to *named DCESs,* a notion which is introduced in Appendix A, Section A.3.1.

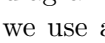
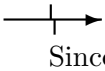
Informally, DCESs are discrete transition systems which communicate by exchanging two kinds of signals, namely *condition* and *event* signals. Condition signals represent system states (e.g., the information if a valve is currently open or closed) and can be used to enable or disable transitions, whereas event signals represent instantaneous actions (e.g., the commands "open" and "close" sent to a valve in order to change its state) and can be used to trigger transitions.

Based on the plant description in [Kow98], our model consists of two parts. The first part contains the physical devices of the plant, like valves, pumps, tanks, etc. The second part introduces one DCES for each of the control programs, which are given as sequential function charts (SFCs) in the plant description.

4.4.1 Physical Devices

We model the following physical devices of the plant:

- The 29 valves. These are either closed or open, and discrete events can be used to force them to change their state.
- The 2 pumps. They are either not pumping or pumping, and discrete events can force them to change their state.
- The 7 tanks. For each tank we define a discrete model keeping track of the quantity of water in that tank, which depends on the states of the valves connected to the tank. We choose a very abstract representation which only consists of two different states, *empty* and *full*. For tank 5 we add a third state *half*, which denotes that the water level is just below the position of the heating coils.
- The heating unit. It is either off or on, and discrete events are used to change its state.

Figure 4.2 shows a block diagram of the interconnections of all physical devices. The input event connections which can be accessed by the control programs are at the left and right hand side of the diagram. The devices are arranged in such a way that their positions approximate their actual location in the plant layout as shown in the piping and instrumentation diagram (Figure 3.1). In the P/I diagram and the following block diagrams we use an arrow like  to denote the flow of a condition signal and  for an event signal flow.

Since we do not model different concentrations or temperatures of the salt solutions or any pressure differences in pipes or tanks, and since we always assume that the condenser is working properly, the following devices

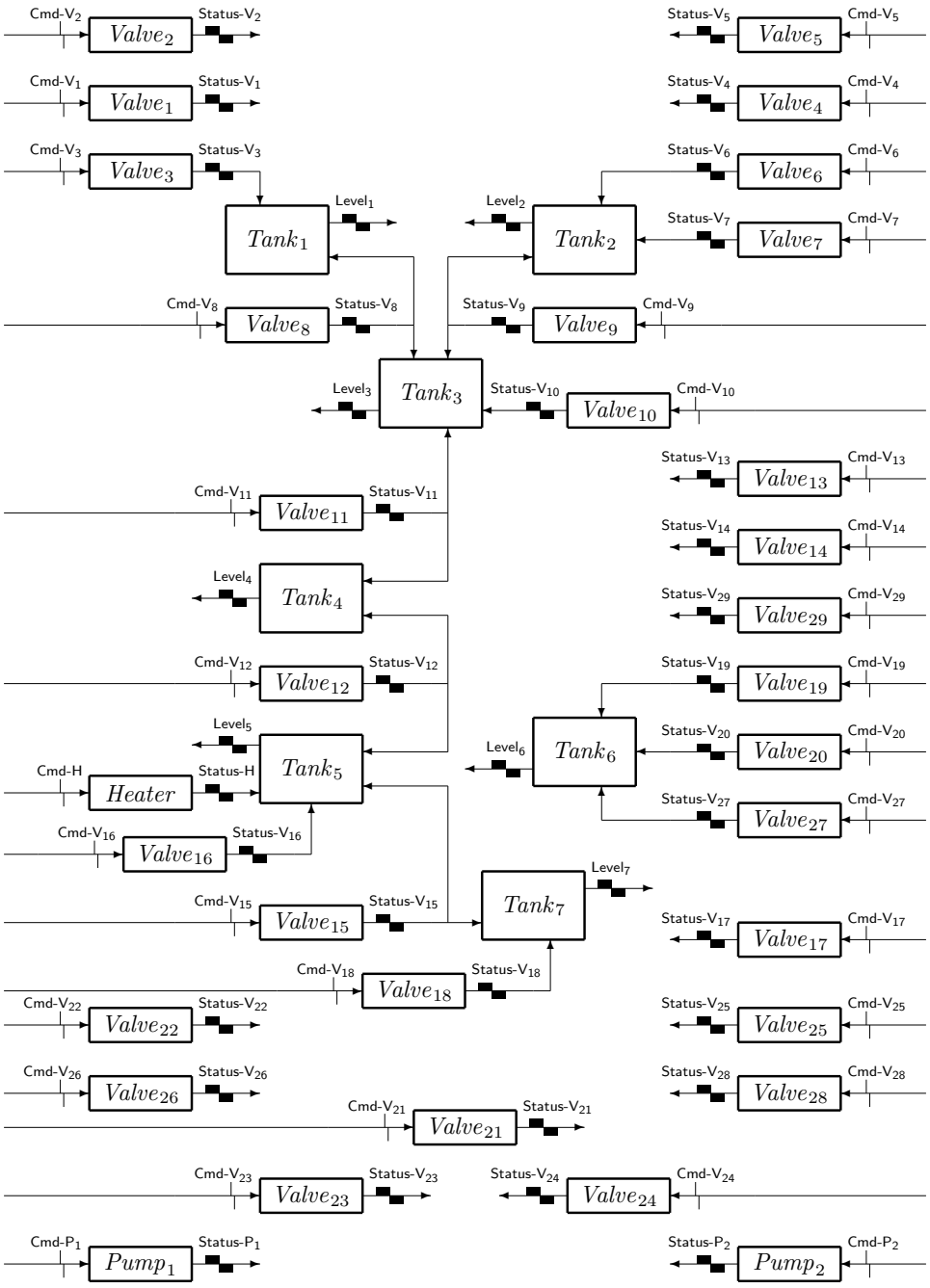


Figure 4.2: Block diagram of the physical devices modeled as DCESs

of the plant are not modeled: The condenser, the cooling units, and all kinds of sensors (pressure, concentration, temperature), except for the water level sensors, which are implicitly modeled as a condition output of each tank model. The mixing unit in tank 3 is not modeled either, since we do not have any properties concerning the use of the mixer.

For each of the devices, the input and output components with their associated alphabets of the respective DCES are shown in a block diagram. The block diagrams also show the names of the signals and the sets of symbols that can be transmitted. The states and transitions of the DCES, which define its internal operation and its input/output behavior, are shown in a transition diagram. The precise semantics of such a transition diagram, i.e., the transformation from a transition diagram into the formal syntax of a DCES, is defined in Appendix A, Section A.3.2.

Valves

We have one DCES $Valve_i$ for each valve i, $i \in \{1, \ldots, 29\}$. Since all the valves are isomorphic, we only describe $Valve_1$ here.

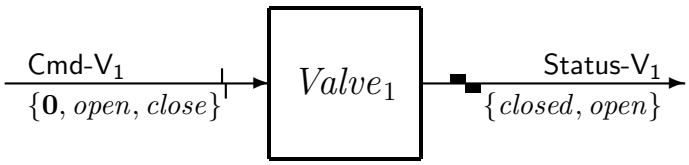


Figure 4.3: Block diagram of $Valve_1$

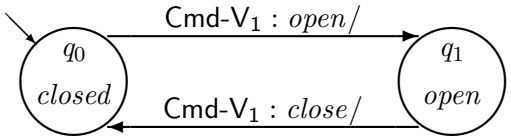


Figure 4.4: Transition diagram of $Valve_1$

The valve has one event input named Cmd-V1 with the associated alphabet $\{\mathbf{0}, open, close\}$. The special symbol $\mathbf{0}$ is an element of every event alphabet and denotes that no event is currently present. Here it means that the valve should remain in its current position. Initially, the valve is closed. The state of $Valve_1$ can be changed by the control programs by sending a Cmd-V1 : *open* or Cmd-V1 : *close* event. The current state of the valve can be observed at the condition output named Status-V1, which can take on the values *closed* or *open*.

In our abstract model, the valves have no switching delays or malfunctions.

Pumps

The two pumps are modeled by the DCESs $Pump_1$ and $Pump_2$. Since both are isomorphic, only pump 1 is shown.

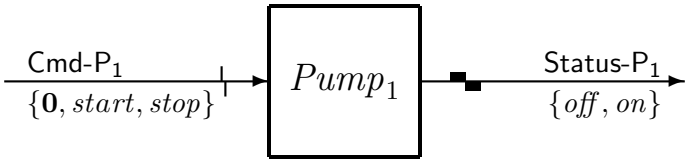


Figure 4.5: Block diagram of $Pump_1$

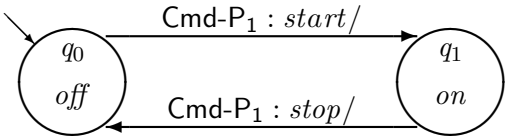


Figure 4.6: Transition diagram of $Pump_1$

The pump has one event input named Cmd-P1 with the associated alphabet $\{\mathbf{0}, start, stop\}$. Initially, the pump is off. The state of $Pump_1$ can be changed by sending a Cmd-P1 : *start* or Cmd-P1 : *stop* event. The current state of the pump can be observed at the condition output named Status-P1, which can take on the values *off* or *on*.

Our models of the pumps also don't have switching delays or malfunctions. The pressure limit switches of the pumps are not modeled; we do however verify later that the control programs never switch on a pump that would work against a closed valve, which might destroy the pump.

Tanks

Each DCES model of a tank is used to provide information about the water level in the respective tank of the plant. We use two different levels (*empty* and *full*) in our model, only tank 5 has a third level (*half*). These are output condition symbols of the DCES tank models. Whenever a control program needs to access a water level sensor, it checks the condition output of the respective tank. We do not model the water level sensors themselves.

A change of the water level of a tank depends on the amount of water flowing in and out of the tank. This again depends on pressure differences and the state of the valves. In this model a simple abstraction is chosen: Whenever a valve controlling an input pipe of a tank is open, water may flow into the tank, and the tank's level may change from *empty* to *full*. And whenever a valve controlling an output pipe of a tank is open, water may leave the tank, and the tank's level may change from *full* to *empty*. Since the transitions between the *full* and *empty* states will only be guarded by

conditions and not by events, the state is not forced to change immediately when the condition input changes; it even can remain the same forever. It is obvious that this is an abstraction of the real plant behavior.

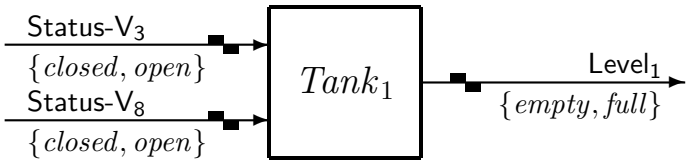


Figure 4.7: Block diagram of $Tank_1$

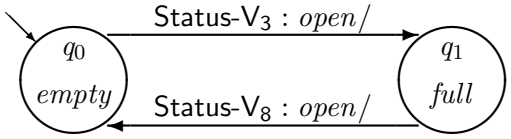


Figure 4.8: Transition diagram of $Tank_1$

Tank 1 can be filled via valve 3 and emptied via valve 8. Thus, $\mathsf{Status\text{-}V_3}$ and $\mathsf{Status\text{-}V_8}$ are condition input names of $Tank_1$. $\mathsf{Status\text{-}V_3} : open$ enables the transition from q_0 to q_1 (when taken, this transition yields the condition output change from $\mathsf{Level_1} : empty$ to $\mathsf{Level_1} : full$), and $\mathsf{Status\text{-}V_8} : open$ enables the transition from q_1 to q_0, and if this transition is taken, the condition output changes from $\mathsf{Level_1} : full$ to $\mathsf{Level_1} : empty$.

As mentioned above these transitions need not to be taken when they are enabled; in fact they may never be taken, since condition changes cannot force any transitions.

The control programs can read the condition output $\mathsf{Level_1}$ to obtain the current water level in tank 1.

Since the structure of the DCES models of tank 2, tank 3, tank 4, and tank 7 are very similar to the DCES of tank 1, we do not show their block and transition diagrams here; these can be found in [Luk99a]. The interfaces of all tanks can also be gathered from the block diagram shown in Figure 4.2.

Tank 2 can be filled via valve 6 or valve 7, and it can be emptied via valve 9. The water level is represented by the condition output $\mathsf{Level_2}$.

Tank 3 can be filled via valve 8 or valve 9, and it can be emptied via valve 10 or valve 11. The water level is represented by the condition output $\mathsf{Level_3}$. The mixing unit and the manual input of salt is not modeled.

Tank 4 can be filled via valve 11 and emptied via valve 12. The water level is represented by the condition output $\mathsf{Level_4}$.

Tank 5 can be filled via valve 12 and emptied via valve 15 or valve 16. The water level is represented by the condition output $\mathsf{Level_5}$, which can take on the values *empty*, *half*, and *full*. We use the value *half* to model that the level has fallen below 8 cm, the level at which control program B5 aborts

the evaporation process because the heating coils are no longer covered by water.

If the heater is working, steam may be produced, decreasing the water level in tank 5. This is modeled by the two transitions enabled by the Status-H : *on* condition.

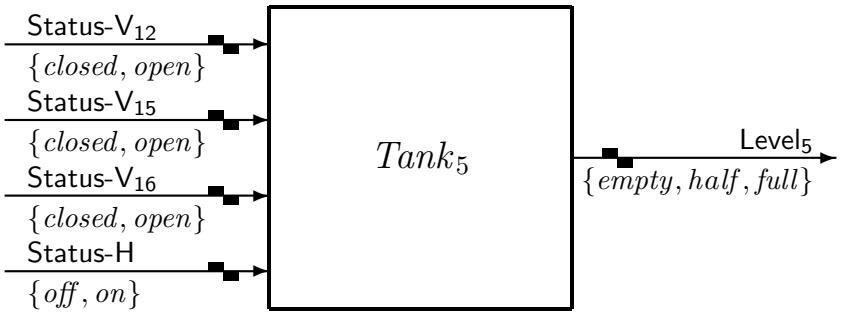


Figure 4.9: Block diagram of $Tank_5$

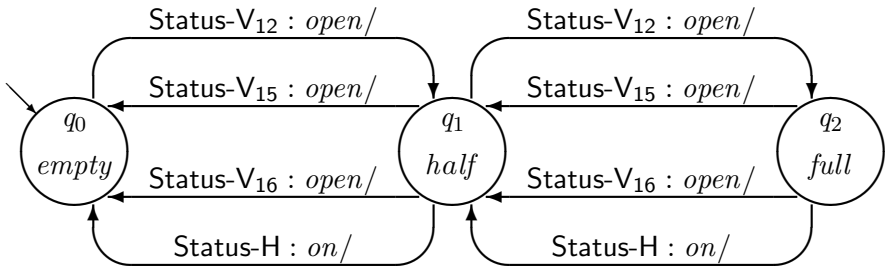


Figure 4.10: Transition diagram of $Tank_5$

Tank 6 can be filled via valve 14 or by water pouring out of the condenser, and it can be emptied via valve 19, valve 20, or valve 27. The water level is represented by the condition output Level_6. We assume that water can leave the condenser at any time, even if the heater is not working, since it takes some time to cool down the steam and since there is a delay between switching off the heater and the end of steam production in tank 5. Therefore, we have an unconditional transition from state q_0 to state q_1. As a side effect of this we do not need $\mathsf{Status\text{-}V}_{14}$ as a condition input, since an additional transition from q_0 to q_1 enabled by $\mathsf{Status\text{-}V}_{14}$: *open* would not change the semantics of the DCES $Tank_6$. The cooling unit of tank 6 is not modeled.

Tank 7 can be filled via valve 15 and emptied via valve 17. The water level is represented by the condition output Level_7. The cooling unit of tank 7 is not modeled.

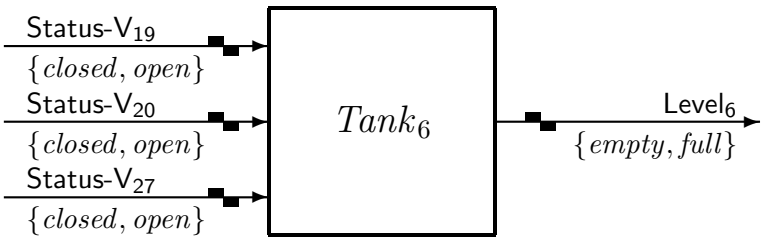


Figure 4.11: Block diagram of *Tank*$_6$

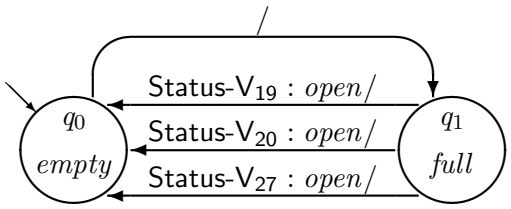


Figure 4.12: Transition diagram of *Tank*$_6$

Heater

The heater is modeled by the DCES *Heater*. Heating can be started by sending the Cmd-H : *start* event and stopped by sending the Cmd-H : *stop* event. The current state of the heater can be observed at the Status-H condition output.

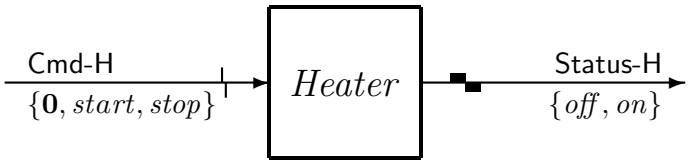


Figure 4.13: Block diagram of *Heater*

4.4.2 Control Programs

The control programs are given in [Kow98] as Sequential Function Charts (SFCs). We manually modeled these SFCs as DCESs, trying to stay as close to their original structure as we can in our abstract framework. Each numbered step i of the SFC becomes a control state q_i in the DCES. The commands executed in step i are represented in the events generated when entering q_i. Conditions which enable the change from step i to step j are the enabling conditions of the DCES transition from q_i to q_j.

The following parts of the SFCs are not modeled, since we do not model the respective physical devices: User interaction with a display and some

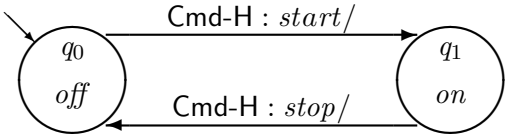


Figure 4.14: Transition diagram of *Heater*

confirmation input, concentration sensor inputs, the cooling liquid flow sensor, and temperature sensors.

All DCES control programs are started by an external event $\mathsf{Cmd\text{-}Prg_N}$: *start*, where N is the name of the program. The *start* events are either sent by an operator or a scheduling program which controls the complete recycling process. When this event occurs, the control state changes from q_1 to q_2, and the control program interacts with the physical devices until some final control state is reached. In contrast to the SFCs, the DCESs only run through once, i.e., they do not return to the initial control state when they have reached some final control state. This is just a design choice for the abstract model; the correctness of our properties is not affected, since they to do not consider repeated execution by a supervisory control program sending the *start* events.

For the sake of brevity we only show some of the block and transition diagrams of the control programs here (among others, those for which the verification reveals an error); the full set of diagrams can be found in [Luk99a].

Program Prg_{B2}

This program fills tank 2 with fresh water via valve 7 until the maximum level is reached. Note that the program initially closes valve 6 and valve 9. As a safety measure, each program sends an initial *open* or *close* event to any valve connected to a tank used by the program, and even if the valve is not used later on, a *close* event is sent. This is not necessary, since we prove later that all valves are already closed if a program is started.

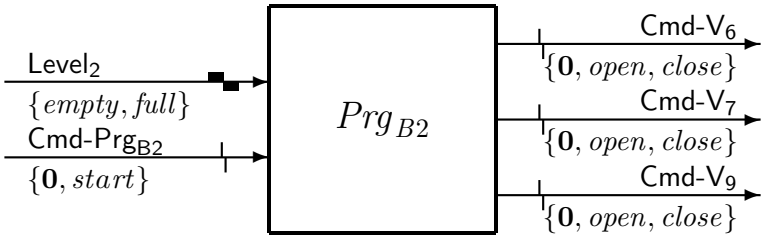


Figure 4.15: Block diagram of Prg_{B2}

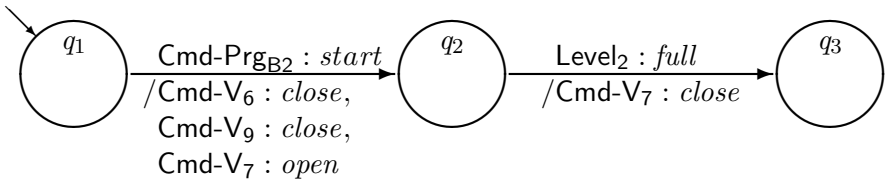


Figure 4.16: Transition diagram of Prg_{B2}

Program Prg_{B3}

This program produces the concentrated brine in tank 3 by mixing highly concentrated brine from tank 1 via valve 8 and water from tank 2 via valve 9. Finally, the solution is drained into tank 4 via valve 11. Note that the mixing unit and the concentration sensor is not modeled.

Program Prg_{B3K}

This program controls the manual production of the concentrated brine in tank 3. First tank 3 is filled with water from tank 2 via valve 9 until a desired level is reached. Then salt is inserted manually by the operator (not modeled) while the mixing unit (not modeled) is working until a certain concentration is reached (concentration sensor not modeled), which is signaled to the operator. Then again some water is added via valve 9 until the desired concentration is reached. This last step is needed because the user will normally add some more salt before she reacts to the stop signal.[1] Finally the mixing unit is stopped and valve 9 is closed.

Our model of program B3K, as well as the models of the other programs, obviously neglect some aspects like the measuring of salt concentration, but since the models are abstractions (i.e., over-approximations) of the real control programs (given as SFCs), we are still able to verify the safety properties listed in Section 4.3.1. If, however, other properties need to be verified which reason, e.g., about certain salt concentrations, a finer abstraction needs to be chosen.

Program Prg_{B3U}

This program controls pumping the manually produced concentrated brine in tank 3 into tank 1 via valves 10, 23, 22, 1, and 3, using pump 1.

[1] We believe that a malicious user could add a lot more salt after she is signaled to stop, and, as a consequence, control program B3K would produce an overflow of tank 3 while trying to reach a low concentration by adding water, since the water level is not checked during this filling step. Unfortunately, our abstract model is too coarse to find this error in program B3K by model checking without getting false warnings in other control programs.

Program Prg_{B5}

This program controls the evaporation of the concentrated brine in tank 5 until the desired concentration is reached. First some concentrated brine from tank 4 is filled into tank 5 via valve 12 until tank 5 is full. Then the cooling in the condenser is started by opening valve 13. The heater is switched on, and the evaporation starts. If the desired concentration is reached (concentration sensor not modeled), the heater is switched off, and the tank 5 is drained into tank 7 via valve 12.

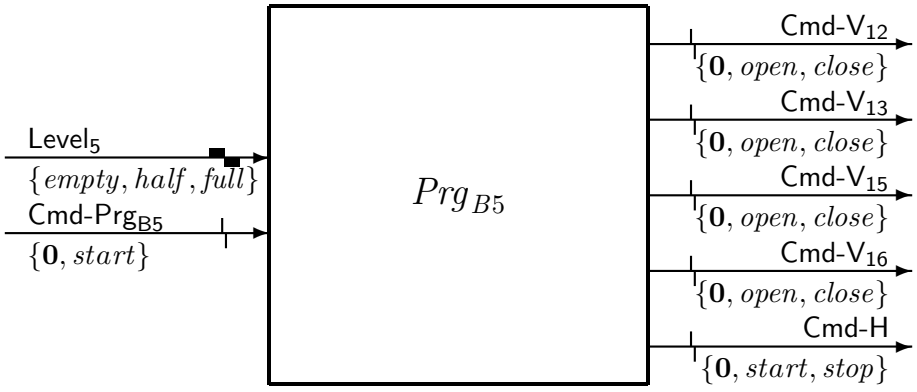


Figure 4.17: Block diagram of Prg_{B5}

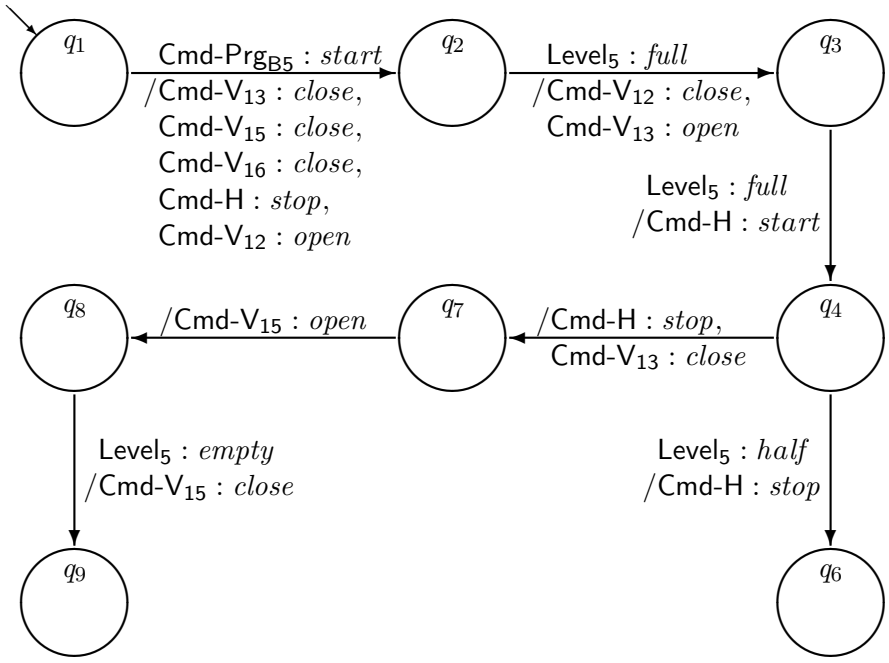


Figure 4.18: Transition diagram of Prg_{B5}

If during the evaporation process the water level in tank 5 goes below 8 cm (modeled as $\mathsf{Level}_5 : \mathit{half}$), the heater is switched of, and we enter state q_6, denoting a failure. Note that this transition from q_4 to q_6 cannot be forced by a level change in tank 5, since Level_5 is a condition, which can only enable transitions, but not trigger them. Our model-checking approach however considers *all* possible executions, which includes one that enters q_6 (see, e.g., the SMV trace on page 56). A possible extension of our plant model could include additional events generated whenever a level change occurs. These events could be used to trigger transitions.

Program Prg_{B6}

This program controls the cooling of the distilled water in tank 6. The cooling water valve 29 is opened until the temperature in tank 6 is below 25 °C (temperature sensor not modeled). If tank 6 is empty at the start of the cooling phase, we enter state q_3, denoting a failure.

Program Prg_{B6A}

This program drains the contents of tank 6 via valve 27.

Program Prg_{B6S}

This program controls pumping the cooled distilled water from tank 6 into tank 2 via valves 20, 21, 23, 22, 1, 2, 4, and 6, using pump 1.

Program Prg_{B6U}

This program controls pumping the cooled distilled water from tank 6 into tank 2 via valves 20, 24, 25, 5, and 6, using pump 2. It can be used instead of program Prg_{B6S} if pump 1 is not working.

Program Prg_{B7}

This program controls the cooling of the concentrated brine in tank 7. The cooling water valve 17 is opened until the temperature in tank 7 is below 25 °C (temperature sensor not modeled). If tank 7 is empty at the start of the cooling phase, we enter state q_3, denoting a failure.

Program Prg_{B7U}

This program controls pumping the cooled concentrated brine from tank 7 into tank 1 via valves 18, 23, 22, 1, and 3, using pump 1.

Program Prg_{SP1}

This maintenance program is used for rinsing the outer ring pipe with water from tank 2 using pump 1. First tank 2 is filled with fresh water via valve 7. If tank 2 is full, valve 7 is closed, valve 9 opened, and tank 3 is filled. If the level in tank 3 has reached some desired level, valve 10 is opened and pump 1 is started. Now water is flowing through the valves 10, 23, 22, 1, 2, 4, 5, and 28. If tank 3 is empty, the pump is stopped, and all valves are closed.

Program Prg_{SP2}

This maintenance program is used for rinsing the outer ring pipe with water from tank 6 using pump 1. First tank 6 is filled with fresh water via valve 14. If tank 6 is full, valve 14 is closed, and the valves 19, 23, 22, 1, 2, 4, 5, and 28 are opened. Then pump 1 is started, and water is flowing through these valves. If tank 6 is empty, the pump is stopped, and all valves are closed.

4.5 Transformation to SMV

We transform the DCESs into the input language of the symbolic model checker SMV [McM93, McM00]. The transformation is described in full detail in Appendix B.

Each DCES is translated into one SMV module. A straightforward global verification approach would apply a parallel composition operation to all modules (29 valves, 2 pumps, 7 tanks, one heating unit, and 13 control programs). This full product would contain the complete model of the plant including its control programs and could, in theory, be used to verify the desired plant properties listed in Section 4.3.1. There are, however, two reasons which prevent us from following this approach:

1. As we will see later, the complexity of the full product model is too large for SMV. The memory and time requirements render the model-checking process impossible.

2. Signals in the DCES framework can only communicate in a one-to-many fashion; it is not allowed that a signal can be sent by different DCES. But in our model, e.g., the $\mathsf{Cmd\text{-}V}_6$: *close* event is sent to $Valve_6$ by several DCESs, namely Prg_{B2}, Prg_{B6S}, Prg_{B6U}, Prg_{SP1}, and Prg_{SP2}. Even with using the fact that two control programs never run simultaneously, there would still be some work (e.g., SMV constructs which "merge" several events into one) to adapt our model to the one-to-many communication restriction.

These two reasons and the observation that not all parts of the plant depend on each other's behavior to lead the following approach: For every

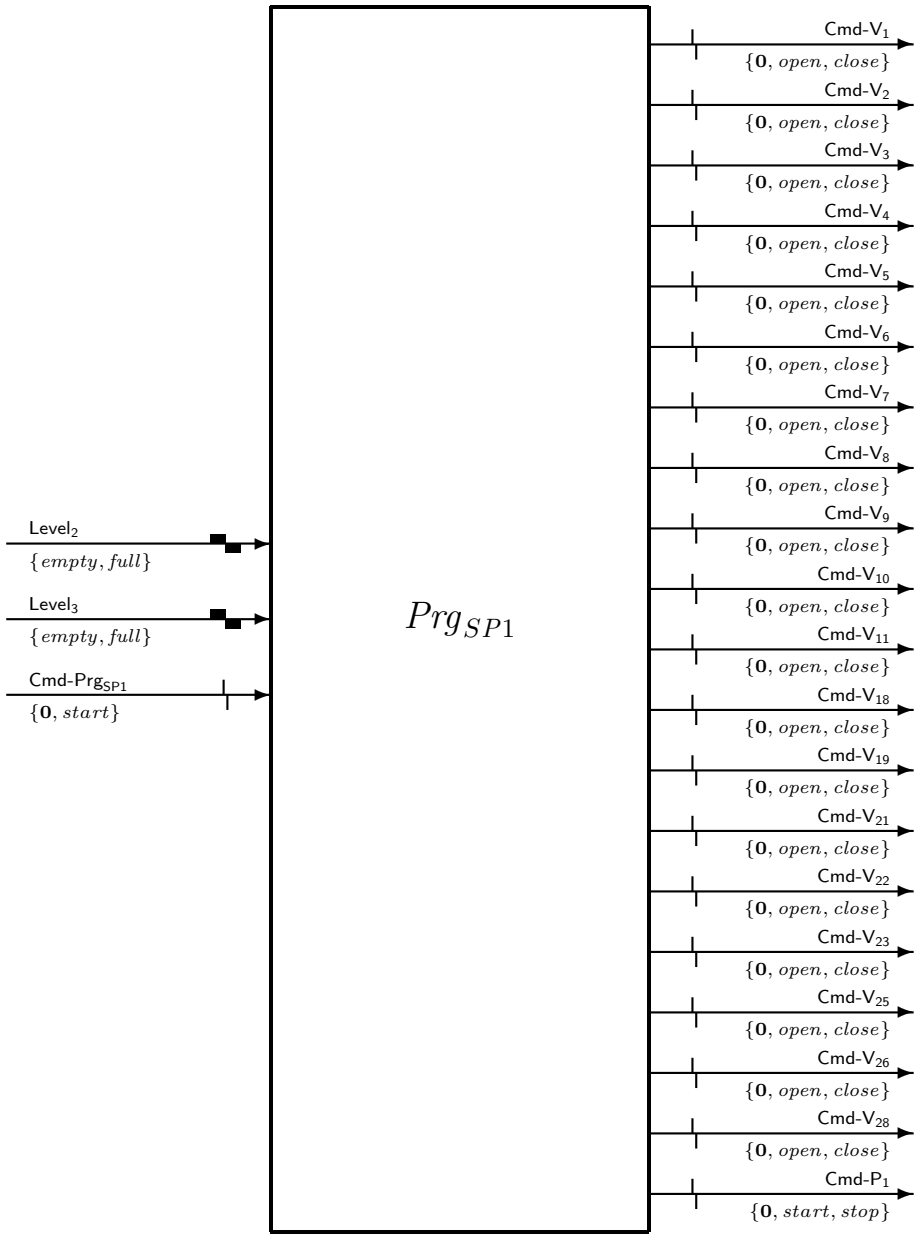


Figure 4.19: Block diagram of Prg_{SP1}

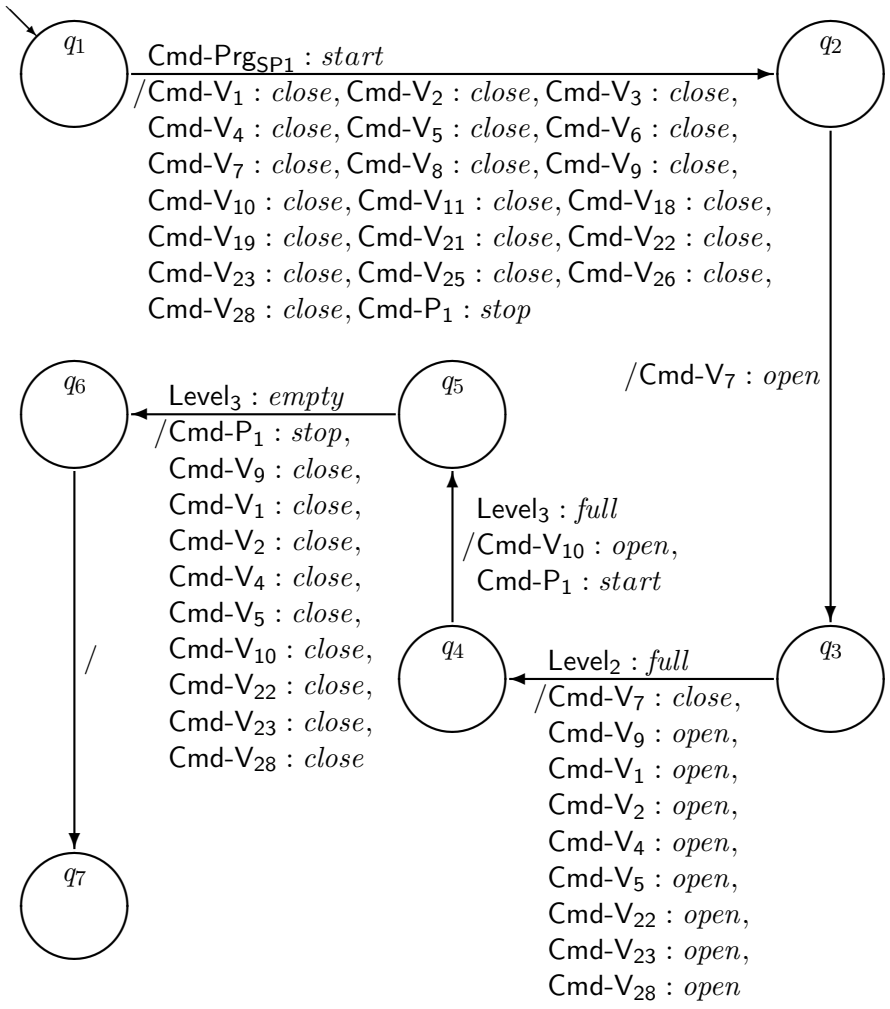


Figure 4.20: Transition diagram of Prg_{SP1}

property that is to be verified, we compose only a minimal set of DCESs that is needed to satisfy (or disprove) this property. Furthermore, if there are several control programs which need to be checked for a property, we compose several such sets, one for each of the control programs.

In general it is difficult to decide which set of DCESs is minimal. But a safe and in most cases also optimal approximation of such a set is to take only into account the DCESs which directly or indirectly modify the signals mentioned in the property in question. The block diagram shown in Figure 4.2 can be used to find these DCESs. E.g., if we need to verify a property over Level$_4$, we know from the communication structure shown in Figure 4.2 that $Tank_4$ influences Level$_4$, and that $Tank_4$ itself is influenced by $Valve_{11}$ and $Valve_{12}$. Therefore, it is sufficient for the verification of that property to analyze the DCESs for $Tank_4$, $Valve_{11}$, $Valve_{12}$, and any

control program that sends Cmd-V_{11} and/or Cmd-V_{12} events. If such a control program is influenced by other DCESs, these may have to be added as well.

Once we have fixed a set of DCESs, each DCES is translated into an SMV module. The translation works as follows: The discrete control state of the DCES is stored in a local variable `state`. While communication between condition/event systems is achieved by using condition and event signals, SMV modules communicate via shared variables. Therefore, we introduce a variable for each event output occurring in one of the DCESs. This variable is local to and written by the module of the DCES generating that event, and it can be read by the modules of other DCESs having this event as input.

Conditions are handled differently. Since a condition output symbol only depends on a DCES's current state and its current condition input symbols, we do not need to introduce a variable, but use SMV's `DEFINE` declaration to describe the mapping of the state and the input symbols to the output symbol in a direct way. As an advantage of this construction, no variable is needed to store the condition output symbol, which can significantly decrease memory and time consumption during the model checking process. The `DEFINE` declaration will be given in the module of the DCES generating the respective condition, and the modules of DCESs having this condition as input can access it.

Once every DCES is translated into an SMV module, a global declaration defines the connection of each module's outputs to other modules' inputs according to Figure 4.2. If there is an input for which there is no matching output in one of the modules, a global unrestricted variable for that input is also added to the SMV code. This is, e.g., the case for all Cmd-Prg_N events, since these are sent by an operator or a superior control program, both of which are not part of our model.

Since the transformation from a set of DCESs to an SMV input file has to be done repeatedly (once for each set we need to compose for a property), and since a transformation by hand can be the source of many errors, we use a compiler prototype to perform this step. This compiler takes the LaTeX [Lam94] source of the DCESs' graphical description (even the source code of this chapter can be used) consisting of a block diagram and a transition diagram, and a list of module names (e.g., Prg_{B6A}, $Valve_{19}$, $Valve_{20}$, $Valve_{27}$) as input, and produces the respective SMV code of the parallel interconnection of these DCESs as output.

Figure 4.21 shows the LaTeX source for the DCES $Tank_5$. For the sake of readability, optional parameters controlling the graphical position of states and transitions have been removed (these are ignored by the compiler). The syntax is self-explanatory, given the block and transition diagrams shown in Section 4.4.1. A part of the resulting SMV module is shown in Figure 4.22.

Since SMV specifications are written in the temporal logic CTL (compu-

```
\begin{CES}
 {\Name{\Tank5}
  \Connections{\inputC{\StatusV{12}}{closed,open}
               \inputC{\StatusV{15}}{closed,open}
               \inputC{\StatusV{16}}{closed,open}
               \inputC{\StatusH}{off,on}
               \outputC{\Level5}{empty,half,full}}
 }
 \State{q_0}{\Initial\Coutconst{empty}
             \Trans{q_1}{\Label{ {\StatusV{12}:open} / }}
 }
 \State{q_1}{\Coutconst{half}
             \Trans{q_2}{\Label{ {\StatusV{12}:open} / }}
             \Trans{q_0}{\Label{ {\StatusV{15}:open} / }}
             \Trans{q_0}{\Label{ {\StatusV{16}:open} / }}
             \Trans{q_0}{\Label{ {\StatusH:on} / }}
 }
 \State{q_2}{\Coutconst{full}
             \Trans{q_1}{\Label{ {\StatusV{15}:open} / }}
             \Trans{q_1}{\Label{ {\StatusV{16}:open} / }}
             \Trans{q_1}{\Label{ {\StatusH:on} / }}
 }
\end{CES}
```

Figure 4.21: The LaTeX source for $Tank_5$

tation tree logic) [CE82] we reformulate the properties in CTL. Now SMV can check the DCESs against the desired properties.

4.6 Verification

For each of the desired properties and each control program involved in the validity of this property we perform the following steps to verify it:

1. Use the LaTeX-to-SMV compiler to generate the SMV code for the DCESs we need to model check the property.
2. Write the property as a CTL formula and insert it into the SMV code using a `SPEC` declaration.
3. Use SMV to verify the property. If the verification fails, SMV produces an execution trace leading to a state showing the error. This trace can be used to track down the problem in the control software.

4.6.1 Example

We illustrate the three steps above in detail with the property "Whenever the heater in tank 5 is turned on, the water level of tank 5 is high enough":

The DCESs we have to inspect are Prg_{B5} (since this is the only control program sending commands to the heater) and $Tank_5$ (because we must take a look at its condition output Level_5). Thus, we use the LaTeX-to-SMV compiler to generate the parallel composition of Prg_{B5} and $Tank_5$ as an SMV file. The LaTeX source for the DCES $Tank_5$ is shown in Figure 4.21, the resulting SMV file in Figure 4.22.

We could have put more than just Prg_{B5} and $Tank_5$ into the parallel composition to verify the property, like the valves 12, 15, 16, or the heater. This is not necessary, since in this case these additional DCESs are not needed: The outputs of these DCESs are simply defined as global variables (`StatusV12`, `StatusV15`, `StatusV16`, and `StatusH` in the first `VAR` declaration in Figure 4.22), and SMV chooses arbitrary values for these variables in each computation step. In other words, we have "chaotic" behavior of the physical devices not contained in the parallel composition. So any behavior of the parallel composition of Prg_{B5}, $Tank_5$, and some more DCESs is also a behavior of the parallel composition of just these two DCESs.

We are only investigating safety properties that describe all possible behaviors, i.e., CTL formulae of the form $\forall\Box(\varphi)$. Thus, we can always try to compose a small set of DCESs, and if the verification fails due to a behavior that is not a behavior of the complete system, we can add more DCESs to the parallel composition until we succeed (or encounter an error in the system).

We formulate the property as a CTL formula:

$$\forall\Box(\mathsf{Cmd\text{-}H} = \mathit{start} \Rightarrow \mathsf{Level}_5 \neq \mathit{empty})$$

and insert it into the SMV file using the following `SPEC` declaration (note that variable names must be preceded by the module that is writing the variable):

```
SPEC
  AG (PrgB5.CmdH=start -> !Tank5.Level5=empty)
```

The resulting SMV input file is shown in Figure 4.22. For the sake of brevity we left out the `TRANS` declarations which define the output events and the discrete transitions for the `state` variables.

Now we use SMV to verify the property. The output of SMV is:

```
-- specification AG (PrgB5.CmdH = start ->
                     !Tank5.Level5 ...  is true
```

Thus, control program Prg_{B5} satisfies the property.

4.6.2 Verification Results

Now we start to verify if the control programs satisfy our list of properties. In the following we will list for each property which DCESs have to be

```
MODULE main

VAR  StatusV12: {closed,open};  StatusV15: {closed,open};
     StatusV16: {closed,open};  StatusH: {off,on};
     CmdPrgB5: {0,start};

VAR  conditionsymbols: {closed,open,off,on,empty,half,full};

VAR  Tank5: Tank5_module(StatusV12,StatusV15,StatusV16,StatusH);
     PrgB5: PrgB5_module(Tank5.Level5,CmdPrgB5);

INIT  CmdPrgB5=0

SPEC  AG (PrgB5.CmdH=start -> !Tank5.Level5=empty)


MODULE Tank5_module(StatusV12,StatusV15,StatusV16,StatusH)

VAR  state : {q_0,q_1,q_2};

INIT  state in {q_0}

DEFINE  Level5 := case
                    state=q_0: empty;
                    state=q_1: half;
                    state=q_2: full;
                  esac;

TRANS  [...]


MODULE PrgB5_module(Level5,CmdPrgB5)

VAR  state : {q_1,q_2,q_3,q_4,q_6,q_7,q_8,q_9};

VAR  CmdV12: {0,open,close};  CmdV13: {0,open,close};
     CmdV15: {0,open,close};  CmdV16: {0,open,close};
     CmdH: {0,start,stop};

INIT  state in {q_1}

INIT  CmdV12=0 & CmdV13=0 & CmdV15=0 & CmdV16=0 & CmdH=0

TRANS  [...]
```

Figure 4.22: SMV input file for the parallel composition of Prg_{B5} and $Tank_5$

composed, the respective CTL formula (in SMV syntax), and the verification result obtained by SMV. We also note why a certain verification fails and what the consequences are.

Whenever a control program terminates, all valves are closed. All control programs open at least one valve, so we compose each control program with the valves it sends commands to. We check if the status variable of each valve has the value `closed` when the control program is at a terminating state. Since we know that all valves are closed when a program is started and two programs do not run simultaneously, we do not need to consider valves that are not switched by the program. Programs Prg_{B5}, Prg_{B6}, Prg_{B7} have two terminating states which we need to take into consideration; all the other programs have one terminating state.

- Prg_{B2}, $Valve_6$, $Valve_7$, $Valve_9$

  ```
  SPEC
    AG (PrgB2.state=q_3 ->
        Valve6.StatusV6=closed & Valve7.StatusV7=closed &
        Valve9.StatusV9=closed)
  ```

  Verification result: `true`

- Prg_{B3}, $Valve_8$, $Valve_9$, $Valve_{10}$, $Valve_{11}$

  ```
  SPEC
    AG (PrgB3.state=q_6 ->
        Valve8.StatusV8=closed & Valve9.StatusV9=closed &
        Valve10.StatusV10=closed & Valve11.StatusV11=closed)
  ```

  Verification result: `true`

- Prg_{B3K}, $Valve_8$, $Valve_9$, $Valve_{10}$, $Valve_{11}$

  ```
  SPEC
    AG (PrgB3K.state=q_9 ->
        Valve8.StatusV8=closed & Valve9.StatusV9=closed &
        Valve10.StatusV10=closed & Valve11.StatusV11=closed)
  ```

  Verification result: `true`

- Prg_{B3U}, $Valve_1$, $Valve_2$, $Valve_3$, $Valve_8$, $Valve_{10}$, $Valve_{11}$, $Valve_{19}$, $Valve_{21}$, $Valve_{22}$, $Valve_{23}$

  ```
  SPEC
    AG (PrgB3U.state=q_6 ->
        Valve1.StatusV1=closed & Valve2.StatusV2=closed &
        Valve3.StatusV3=closed & Valve8.StatusV8=closed &
        Valve10.StatusV10=closed) & Valve11.StatusV11=closed) &
        Valve19.StatusV19=closed) & Valve21.StatusV21=closed) &
        Valve22.StatusV22=closed) & Valve23.StatusV23=closed)
  ```

  Verification result: `true`

- Prg_{B5}, $Valve_{12}$, $Valve_{13}$, $Valve_{15}$, $Valve_{16}$

```
SPEC
  AG (PrgB5.state=q_9 ->
      Valve12.StatusV12=closed & Valve13.StatusV13=closed &
      Valve15.StatusV15=closed & Valve16.StatusV16=closed)
```

Verification result: `true`

```
SPEC
  AG (PrgB5.state=q_6 ->
      Valve12.StatusV12=closed & Valve13.StatusV13=closed &
      Valve15.StatusV15=closed & Valve16.StatusV16=closed)
```

Verification result: `false`

If control program Prg_{B5} enters state q_6 this indicates that during the evaporation process the water level in tank 5 has fallen too low. Therefore the heater is switched off, and the program stops. The verification result however shows that valve 13, which was opened earlier when entering state q_3, is still open. This is indicated by the following trace generated by SMV (we omit some less interesting output):

```
state 1.1:
Valve12.StatusV12 = closed
Valve13.StatusV13 = closed
Valve15.StatusV15 = closed
Valve16.StatusV16 = closed
PrgB5.state = q_1

state 1.2:
CmdPrgB5 = start
Valve12.StatusV12 = open
PrgB5.state = q_2

state 1.3:
Valve12.StatusV12 = closed
Valve13.StatusV13 = open
PrgB5.state = q_3

state 1.4:
Level5 = half
PrgB5.state = q_4

state 1.5:
PrgB5.state = q_6
```

The violation of our property "whenever a control program terminates, all valves are closed" should lead to some action to ensure a safe operation of the plant. Two possible solutions are:

1. Change control program Prg_{B5} in such a way that valve 13 is closed some time after state q_6 has been reached.

2. Make sure that valve 13 is closed (e.g., by manual intervention) before another program is started.

Note that a permanent open state of valve 13 does not endanger the plant, since valve 13 only controls the cooling water supply for the condenser. If, e.g., valve 12 would remain open permanently, an overflow of tank 5 might occur.

- Prg_{B6}, $Valve_{14}$, $Valve_{29}$

```
SPEC
  AG (PrgB6.state in {q_3,q_5} ->
      Valve14.StatusV14=closed & Valve29.StatusV29=closed)
```

Verification result: `true`

- Prg_{B6A}, $Valve_{19}$, $Valve_{20}$, $Valve_{27}$

```
SPEC
  AG (PrgB6A.state=q_3 ->
      Valve19.StatusV19=closed & Valve20.StatusV20=closed &
      Valve27.StatusV27=closed)
```

Verification result: `true`

- Prg_{B6S}, $Valve_{1}$, $Valve_{2}$, $Valve_{3}$, $Valve_{4}$, $Valve_{5}$, $Valve_{6}$, $Valve_{10}$, $Valve_{18}$, $Valve_{19}$, $Valve_{20}$, $Valve_{21}$, $Valve_{22}$, $Valve_{23}$, $Valve_{24}$, $Valve_{26}$, $Valve_{27}$

```
SPEC
  AG (PrgB6S.state=q_6 ->
      Valve1.StatusV1=closed & Valve2.StatusV2=closed &
      Valve3.StatusV3=closed & Valve4.StatusV4=closed &
      Valve5.StatusV5=closed & Valve6.StatusV6=closed &
      Valve10.StatusV10=closed & Valve18.StatusV18=closed &
      Valve19.StatusV19=closed & Valve20.StatusV20=closed &
      Valve21.StatusV21=closed & Valve22.StatusV22=closed &
      Valve23.StatusV23=closed & Valve24.StatusV24=closed &
      Valve26.StatusV26=closed & Valve27.StatusV27=closed)
```

Verification result: `true`

- Prg_{B6U}, $Valve_{4}$, $Valve_{5}$, $Valve_{6}$, $Valve_{19}$, $Valve_{20}$, $Valve_{21}$, $Valve_{24}$, $Valve_{25}$, $Valve_{27}$, $Valve_{28}$

```
SPEC
  AG (PrgB6U.state=q_6 ->
      Valve4.StatusV4=closed & Valve5.StatusV5=closed &
      Valve6.StatusV6=closed & Valve19.StatusV19=closed &
      Valve20.StatusV20=closed & Valve21.StatusV21=closed &
      Valve24.StatusV24=closed & Valve25.StatusV25=closed &
      Valve28.StatusV28=closed & Valve27.StatusV27=closed)
```

Verification result: `true`

- Prg_{B7}, $Valve_{17}$

```
SPEC
  AG (PrgB7.state in {q_3,q_5} -> Valve17.StatusV17=closed)
```

Verification result: `true`

- Prg_{B7U}, $Valve_1$, $Valve_2$, $Valve_3$, $Valve_{18}$, $Valve_{19}$, $Valve_{21}$, $Valve_{22}$, $Valve_{23}$, $Valve_{26}$

```
SPEC
  AG (PrgB7U.state=q_6 ->
      Valve1.StatusV1=closed & Valve2.StatusV2=closed &
      Valve3.StatusV3=closed & Valve18.StatusV18=closed &
      Valve19.StatusV19=closed & Valve21.StatusV21=closed &
      Valve22.StatusV22=closed & Valve23.StatusV23=closed &
      Valve26.StatusV26=closed)
```

  Verification result: `true`

- Prg_{SP1}, $Valve_1$, $Valve_2$, $Valve_3$, $Valve_4$, $Valve_5$, $Valve_6$, $Valve_7$, $Valve_8$, $Valve_9$, $Valve_{10}$, $Valve_{11}$, $Valve_{18}$, $Valve_{19}$, $Valve_{21}$, $Valve_{22}$, $Valve_{23}$, $Valve_{25}$, $Valve_{26}$, $Valve_{28}$

```
SPEC
  AG (PrgSP1.state=q_7 ->
      Valve1.StatusV1=closed & Valve2.StatusV2=closed &
      Valve3.StatusV3=closed & Valve4.StatusV4=closed &
      Valve5.StatusV5=closed & Valve6.StatusV6=closed &
      Valve7.StatusV7=closed & Valve8.StatusV8=closed &
      Valve9.StatusV9=closed & Valve10.StatusV10=closed &
      Valve11.StatusV11=closed & Valve18.StatusV18=closed &
      Valve19.StatusV19=closed & Valve21.StatusV21=closed &
      Valve22.StatusV22=closed & Valve23.StatusV23=closed &
      Valve25.StatusV25=closed & Valve26.StatusV26=closed &
      Valve28.StatusV28=closed)
```

  Verification result: `true`

This result was difficult to obtain due to the state explosion problem; the verification took about 23 hours on a Sun UltraSPARC system running at 167 MHz and consumed about 340 Megabytes of main memory. This is not surprising: The full product of the control states of the 20 modules involved has $7 \cdot 2^{19} = 3670016$ elements, and we have $3^{19} = 1.2 \cdot 10^9$ different combinations for events sent to the valves. From this fact and the observation that this verification step only considered a small part of the complete plant model we can safely assume that the straightforward verification approach mentioned at the beginning of Section 4.5 which builds the full system model will be too complex to be handled by SMV, even on state-of-the-art machines.

In this special case avoiding this problem is easy. We split up the conjunction in the formula, thus yielding the task of checking the simple formula

```
AG (PrgSP1.state=q_7 -> Valvei.StatusVi=closed)
```

on the parallel composition of Prg_{SP1} and $Valve_i$ for each $i \in \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 18, 19, 21, 22, 23, 25, 26, 28\}$. Each of these 19 checks succeeds within a fraction of a second.

- Prg_{SP2}, $Valve_1$, $Valve_2$, $Valve_3$, $Valve_4$, $Valve_5$, $Valve_6$, $Valve_{10}$, $Valve_{14}$, $Valve_{18}$, $Valve_{19}$, $Valve_{20}$, $Valve_{21}$, $Valve_{22}$, $Valve_{23}$, $Valve_{25}$, $Valve_{26}$, $Valve_{27}$, $Valve_{28}$

```
SPEC
  AG (PrgSP2.state=q_7 ->
      Valve1.StatusV1=closed & Valve2.StatusV2=closed &
      Valve3.StatusV3=closed & Valve4.StatusV4=closed &
      Valve5.StatusV5=closed & Valve6.StatusV6=closed &
      Valve10.StatusV10=closed & Valve14.StatusV14=closed &
      Valve18.StatusV18=closed & Valve19.StatusV19=closed &
      Valve20.StatusV20=closed & Valve21.StatusV21=closed &
      Valve22.StatusV22=closed & Valve23.StatusV23=closed &
      Valve25.StatusV25=closed & Valve26.StatusV26=closed &
      Valve27.StatusV27=closed & Valve28.StatusV28=closed)
```

Verification result: `true`

Obtaining this result was similarly difficult as above: SMV needed 3.5 hours to complete and used 119 Megabytes of main memory.

Whenever a control program terminates, the pumps are switched off. There are five control programs which send commands to pump 1: Prg_{B3U}, Prg_{B6S}, Prg_{B7U}, Prg_{SP1}, and Prg_{SP2}. We compose each of these programs with $Pump_1$ and check if the status variable of pump 1 has the value `off` when the control program is at its terminating state.

- Prg_{B3U}, $Pump_1$

```
SPEC
  AG (PrgB3U.state=q_6 -> Pump1.StatusP1=off)
```

Verification result: `true`

- Prg_{B6S}, $Pump_1$

```
SPEC
  AG (PrgB6S.state=q_6 -> Pump1.StatusP1=off)
```

Verification result: `true`

- Prg_{B7U}, $Pump_1$

```
SPEC
  AG (PrgB7U.state=q_6 -> Pump1.StatusP1=off)
```

Verification result: `true`

- Prg_{SP1}, $Pump_1$

```
SPEC
  AG (PrgSP1.state=q_7 -> Pump1.StatusP1=off)
```

Verification result: `true`

- Prg_{SP2}, $Pump_1$

  ```
  SPEC
    AG (PrgSP2.state=q_7 -> Pump1.StatusP1=off)
  ```

  Verification result: `true`

Analogously, we check this for program Prg_{B6U}, which is the only program using pump 2.

- Prg_{B6U}, $Pump_2$

  ```
  SPEC
    AG (PrgB6U.state=q_6 -> Pump2.StatusP2=off)
  ```

  Verification result: `true`

Whenever a control program terminates, the heater is switched off. Only control program Prg_{B5} uses the heater. We compose Prg_{B5} and *Heater* and check if the heater's status variable is `off` when the program is in one of its final states.

- Prg_{B5}, *Heater*

  ```
  SPEC
    AG (PrgB5.state in {q_6,q_9} -> Heater.StatusH=off)
  ```

  Verification result: `true`

Now we know that whenever a control program terminates, all valves are closed (with one exception listed above), and the pumps and the heater are switched off, provided this was also the case when the control program was started. We further know that two programs are never run simultaneously. This allows us to verify the rest of our properties.

Whenever the heater in tank 5 is turned on, the water level of tank 5 is high enough. Only control program Prg_{B5} uses the heater. We compose Prg_{B5} and $Tank_5$ and check if the tank's water level does not have the `empty` value when the Cmd-H : *start* event occurs. We do not need to include the heater in the composition, since we are only interested in the starting command generated by the control program.

- Prg_{B5}, $Tank_5$

  ```
  SPEC
    AG (PrgB5.CmdH=start -> !Tank5.Level5=empty)
  ```

  Verification result: `true`

Whenever the heater in tank 5 is working, none of the valves 12, 15, and 16 are open. Again, only control program Prg_{B5} uses the heater. We compose Prg_{B5}, *Heater*, $Valve_{12}$, $Valve_{15}$, and $Valve_{16}$, and check the status variables of the valves if the heater status is `on`.

- Prg_{B5}, *Heater*, $Valve_{12}$, $Valve_{15}$, $Valve_{16}$

```
SPEC
  AG (Heater.StatusH=on ->
      Valve12.StatusV12=closed &
      Valve15.StatusV15=closed &
      Valve16.StatusV16=closed)
```

Verification result: `true`

Whenever the heater in tank 5 is working, valve 13 is open. We compose Prg_{B5}, *Heater*, and $Valve_{13}$, and check the status variable of valve 13 if the heater status is `on`.

- Prg_{B5}, *Heater*, $Valve_{13}$

```
SPEC
  AG (Heater.StatusH=on -> Valve13.StatusV13=open)
```

Verification result: `true`

At most two cooling units are active simultaneously. Each of the three cooling units is active when the valve through which the cooling water flows is open. These are the valves 13, 17, and 29. Since there is no control program which sends commands to all three valves and since any two programs never run simultaneously, this property is trivially fulfilled. This is an example of a property which can be established solely by manual analysis of the plant structure; tool support is not necessary.

Pumps are not pumping against closed valves. We show that whenever a pump is working, the water flowing through the pump can leave the system via valve 26 or valve 28, or is pumped into tank 1 or tank 2.

For pump 1, this leads to four possible paths:

1. From pump 1 through valves 22 and 26.
2. From pump 1 through valves 22, 1, and 3 into tank 1.
3. From pump 1 through valves 22, 1, 2, 4, and 6 into tank 2.
4. From pump 1 through valves 22, 1, 2, 4, 5, and 28.

This yields the SMV specification show below. Pump 1 is used by the control programs Prg_{B3U}, Prg_{B6S}, Prg_{B7U}, Prg_{SP1}, and Prg_{SP2}.

- Prg_{B3U}, $Pump_1$, $Valve_1$, $Valve_2$, $Valve_3$, $Valve_4$, $Valve_5$, $Valve_6$, $Valve_{22}$, $Valve_{26}$, $Valve_{28}$

  SPEC
  ```
  AG (Pump1.StatusP1=on -> Valve22.StatusV22=open &
      (Valve26.StatusV26=open | Valve1.StatusV1=open &
       (Valve3.StatusV3=open |
        Valve2.StatusV2=open & Valve4.StatusV4=open &
         (Valve6.StatusV6=open |
          Valve5.StatusV5=open & Valve28.StatusV28=open))))
  ```

  Verification result: `true`

- Prg_{B6S}, $Pump_1$, $Valve_1$, $Valve_2$, $Valve_3$, $Valve_4$, $Valve_5$, $Valve_6$, $Valve_{22}$, $Valve_{26}$, $Valve_{28}$

  SPEC
  (same as above)

  Verification result: `true`

- Prg_{B7U}, $Pump_1$, $Valve_1$, $Valve_2$, $Valve_3$, $Valve_4$, $Valve_5$, $Valve_6$, $Valve_{22}$, $Valve_{26}$, $Valve_{28}$

  SPEC
  (same as above)

  Verification result: `true`

- Prg_{SP1}, $Pump_1$, $Valve_1$, $Valve_2$, $Valve_3$, $Valve_4$, $Valve_5$, $Valve_6$, $Valve_{22}$, $Valve_{26}$, $Valve_{28}$

  SPEC
  (same as above)

  Verification result: `true`

- Prg_{SP2}, $Pump_1$, $Valve_1$, $Valve_2$, $Valve_3$, $Valve_4$, $Valve_5$, $Valve_6$, $Valve_{22}$, $Valve_{26}$, $Valve_{28}$

  SPEC
  (same as above)

  Verification result: `true`

For pump 2, this also leads to four possible paths:

1. From pump 2 through valves 25 and 28.
2. From pump 2 through valves 25, 5, and 6 into tank 2.
3. From pump 2 through valves 25, 5, 4, 2, and 3 into tank 1.
4. From pump 2 through valves 25, 5, 4, 2, 1, and 26.

This yields the SMV specification show below. Pump 2 is only used by control program Prg_{B6U}.

- Prg_{B6U}, $Pump_2$, $Valve_1$, $Valve_2$, $Valve_3$, $Valve_4$, $Valve_5$, $Valve_6$, $Valve_{25}$, $Valve_{26}$, $Valve_{28}$

```
SPEC
  AG (Pump2.StatusP2=on -> Valve25.StatusV25=open &
      (Valve28.StatusV28=open | Valve5.StatusV5=open &
       (Valve6.StatusV6=open |
        Valve4.StatusV4=open & Valve2.StatusV2=open &
         (Valve3.StatusV3=open |
          Valve1.StatusV1=open & Valve26.StatusV26=open))))
```

Verification result: `true`

Each tank's input and output valves are not open simultaneously. This property ensures that we have a controlled flow of liquids in the plant, i.e., no program lets water flow directly through a tank without storing it first in that tank. For all tanks, we check this property on each program that is controlling at least one of the tank's input valves and one of the tank's output valves.

Tank 1

- Prg_{B3U}, $Valve_3$, $Valve_8$

```
SPEC
  AG !(Valve3.StatusV3=open & Valve8.StatusV8=open)
```

Verification result: `true`

- Prg_{SP1}, $Valve_3$, $Valve_8$

```
SPEC
  AG !(Valve3.StatusV3=open & Valve8.StatusV8=open)
```

Verification result: `true`

Tank 2

- Prg_{B2}, $Valve_6$, $Valve_7$, $Valve_9$

```
SPEC
  AG !((Valve6.StatusV6=open | Valve7.StatusV7=open) &
       Valve9.StatusV9=open)
```

Verification result: `true`

- Prg_{SP1}, $Valve_6$, $Valve_7$, $Valve_9$

```
SPEC
  AG !((Valve6.StatusV6=open | Valve7.StatusV7=open) &
       Valve9.StatusV9=open)
```

Verification result: `true`

Tank 3

- Prg_{B3}, $Valve_8$, $Valve_9$, $Valve_{10}$, $Valve_{11}$

```
SPEC
  AG !((Valve8.StatusV8=open | Valve9.StatusV9=open) &
       (Valve10.StatusV10=open | Valve11.StatusV11=open))
```

  Verification result: `true`

- Prg_{B3K}, $Valve_8$, $Valve_9$, $Valve_{10}$, $Valve_{11}$

```
SPEC
  AG !((Valve8.StatusV8=open | Valve9.StatusV9=open) &
       (Valve10.StatusV10=open | Valve11.StatusV11=open))
```

  Verification result: `true`

- Prg_{B3U}, $Valve_8$, $Valve_{10}$, $Valve_{11}$

```
SPEC
  AG !(Valve8.StatusV8=open &
       (Valve10.StatusV10=open | Valve11.StatusV11=open))
```

  Verification result: `true`

- Prg_{SP1}, $Valve_8$, $Valve_9$, $Valve_{10}$, $Valve_{11}$

```
SPEC
  AG !((Valve8.StatusV8=open | Valve9.StatusV9=open) &
       (Valve10.StatusV10=open | Valve11.StatusV11=open))
```

  Verification result: `false`

The trace generated by SMV shows that valve 9 is opened in the step from state q_3 to state q_4, and valve 10 is opened in the following step to state q_5. This violates our property, since water is now flowing through tank 3. Since programs Prg_{SP1} and Prg_{SP2} are used for maintenance purposes only and are not part of the normal production cycle, Prg_{SP1} and Prg_{SP2} should be allowed to let water flow through tanks and can be excluded from checking the property "each tank's input and output valves are not open simultaneously".

Tank 4 There is no control program that uses valve 11 *and* valve 12, so nothing has to be checked, and the property is trivially fulfilled.

Tank 5

- Prg_{B5}, $Valve_{12}$, $Valve_{15}$, $Valve_{16}$

```
SPEC
  AG !(Valve12.StatusV12=open &
       (Valve15.StatusV15=open | Valve16.StatusV16=open))
```

  Verification result: `true`

Tank 6

- Prg_{SP2}, $Valve_{14}$, $Valve_{19}$, $Valve_{20}$, $Valve_{27}$

```
SPEC
  AG !(Valve14.StatusV14=open &
       (Valve19.StatusV19=open | Valve20.StatusV20=open |
        Valve27.StatusV27=open))
```

Verification result: `true`

Tank 7 There is no control program that uses valve 15 *and* valve 18, so nothing has to be checked, and the property is trivially fulfilled.

4.7 Discussion

A discrete, untimed model of the batch plant and its control programs has been used to verify some safety properties. These properties help to ensure that the control programs do not cause damages to the plant's devices.

We used discrete condition/event systems (DCESs) to model the valves, the tanks, the pumps, the heater, and the 13 control programs. For each property, we used a compiler prototype to transform the DCESs required to check the property into SMV input, and provided SMV with the property in CTL syntax.

The formal verification reveals that all properties are fulfilled, with two exceptions. In these cases SMV shows the trace of a computation prefix leading to the state where the error occurs. This trace provides a convenient way of finding the error in the control program that causes the violation of the property. However, the two violations detected will not cause any damages to the plant; they can merely be regarded as small design inaccuracies which can be fixed with little effort.

The error in control program B3K mentioned on page 45 reveals that an abstract model like the one presented in this chapter cannot find all errors that may be hidden in the control programs. A straightforward test if control programs always prevent overflows by checking the level in a tank while it is being filled will show that many programs do not always make these checks. But this does not necessarily mean that there is a problem, since, e.g., the upstream tank may be much smaller than the tank being filled, and thus, an overflow is not possible, just by construction of the plant. This means that we need a finer model which is able to make statements about quantities of liquid in a tank. A modeling framework which leads into this direction is presented in Chapter 6.

The plant model presented in this chapter is an example of a system in which there are only a few dependencies between its constituent parts. Consequently, the sets of modules that need to be composed for model checking

are relatively small, and model checking these is easy. If there are more dependencies, these sets tend to get larger, and in the worst case, the full product of all modules needs to be built and model checked, which brings us back to the very problem why we started with the modular approach—state explosion. The following chapter presents a compositional verification approach which is better suited to systems with more dependencies.

Chapter 5

Compositional Verification

The previous chapter showed that a modular approach to a verification task can significantly reduce the complexity problem of state-based verification. There still remains, however, the problem that if many modules depend on each other, the model-checking process needs to run on a large set of components and can therefore reach its limits and fail due to time and space restrictions.

This chapter presents a compositional verification method which takes the modular approach of Chapter 4 one step further. Every module of the system is model-checked on its own, and local properties gained from the model-checking results are then combined by deduction to yield global properties of the system.

5.1 Introduction

The main underlying concepts used in the verification method presented in this chapter have already been introduced in Chapter 1: *abstraction* and *compositional reasoning*. We briefly recall these two ideas:

The concept of *abstraction* replaces concrete objects with abstract ones, often by simplifying structures which are not relevant to the current objective (e.g., verification of certain properties) and which can therefore be neglected. *Safe abstractions* have the characteristic that properties of the abstract object also hold for the concrete object. The converse needs not to hold, and when a certain property cannot be established for an abstract system (e.g., by model checking), this property either does not hold for the concrete object, or the chosen abstraction was too coarse, and a finer abstraction has to be chosen ("abstraction refinement").

Compositional reasoning is a methodology in which a system is described by the behavior of its constituent parts (for which we use the term "modules") and the ways these are put together. The behavior of the complete system can be inferred from these descriptions only [Fre23]. No further in-

formation about the internal structures of the modules is needed ("black box" principle, see [Zwi89]).

Another important principle in compositional reasoning is the use of *open systems* for the specification of the modules at all levels of abstraction. That is, the behavior of a module is specified relative to the behavior of its environment. In most cases a module can only satisfy its required properties if its environment displays some expected behavior. Therefore, the specification of a module usually consists of a description of its output behavior as well as a description of the expected inputs. A module is considered to fulfill its specification if any input which meets these requirements will only lead to outputs described in the specification. Therefore, if the input differs from the expectations, the output behavior of the module is not required to meet the requirements. Thus, a specification for a module can be written as an implication (see Section 5.5.1), giving an approximate notion of weakest precondition of that module w.r.t. a given requirement upon its output behavior [dRdBH$^+$01].

When modules are combined, however, the possibility of feedback arises, i.e., some module A may depend on another module B in that an assumption upon a trace in A's behavior actually depends on output requirements upon proper prefixes w.r.t. B's behavior, and vice versa. There are several ways to break the circularity which arises from feedback, e.g., Assume/Guarantee-based reasoning [MC81, Jon83]. In the particular example considered in this chapter, no such feedback arises, and a simplified way of Assume/Guarantee-based reasoning can be applied.

5.2 The Compositional Verification Approach

Figure 5.1 illustrates the compositional verification approach that is followed in this chapter. First the system is split into smaller parts, the modules. Then each module is formally specified at a suitable abstraction level as a single entity and its correct behavior w.r.t. the specification is proven locally, e.g., by algorithmic verification tools like model checkers. The specifications of all modules are then combined by various means of logical deduction in order to derive the desired global property of the system.

In the sequel we discuss practical aspects of the work steps of this compositional verification approach.

5.2.1 Decomposition

The system we want to verify, consisting of a process (e.g., a chemical plant) and a controller (e.g., a PLC), is split into smaller components, called modules. The decision where to split the system is an optimization task depending on two main objectives:

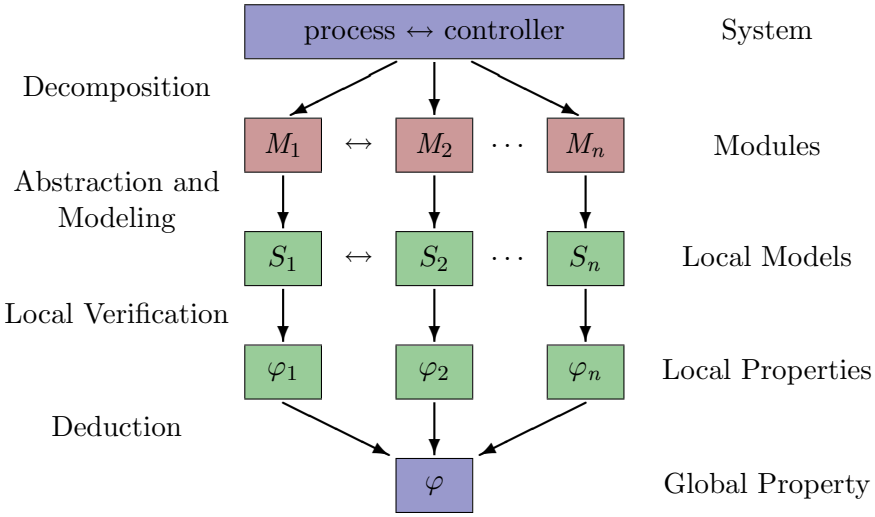


Figure 5.1: The compositional verification approach

1. The complexity of each resulting module should be small enough for a local verification process, without running into the state explosion problem. One the other hand, a large number of modules will result in an unnecessary overhead for the specification and verification of each module.

2. The interface of each resulting module should be kept simple, as complex interfaces will make the subsequent deduction step harder.

A straightforward decomposition strategy (which was also mostly followed in Chapter 4) is to define each physical component of the plant (e.g., valves, tanks, sensors, pumps) as a module, as well as each logical unit of the control software. But other decompositions are possible, such as the creation of functional groups as a module, e.g., by defining a tank, its adjacent valves, and the piece of software that controls these valves as one module.

The decomposition step does not involve any formal specification work; it is merely providing a structure for the steps to follow.

5.2.2 Abstraction and Modeling

Now each module needs to be modeled in a formal framework which allows formal verification. An important choice that has to be made is the abstraction level on which the module will be described. It is advised to make each model as abstract as possible, but without losing too much information. Furthermore, each module needs to be specified in a formal specification language. The specification should describe the part of the module behavior that is needed to conclude the global system specification.

It is not necessary that everything is modeled in the same framework. Often parts of the system can be modeled as discrete automata, whereas other parts need continuous variables.

5.2.3 Local Verification

Each model is now checked against its specification. Model checkers allow to fully mechanize this step. If the size and abstraction level of each module was chosen carefully, the state explosion problem is not an issue during the model-checking process.

If a verification fails, the model checker usually produces a counterexample which can be used to correct the error in the model or specification.

5.2.4 Deduction

Finally, the global system property is proven. The local module specifications are combined using deductive techniques. This can be supported by theorem proving tools.

If the global property cannot be proven, the most likely reason is that information is missing in one or more of the local specifications. Usually intermediate results from the proof show the missing information. After the local specifications have been corrected and verified, the proof is tried again.

5.3 Example

We illustrate the compositional approach by verifying the multi-product batch plant introduced in Chapter 3. From a list of desirable global plant properties we choose two representative cases and prove them correct. The full proof of all desirable properties would render this chapter unreadable without leading to more insights.

As before, we use the term “hardware” for any physical part of the system that is shown in Figure 3.2, and the term “software” for the programs that run on the PLC system. The actual PLC hardware and the electrical wiring between the PLC and the sensors and actuators are not discussed.

5.3.1 Desired Properties

We assume that there are no hardware failures, i.e., valves do not get stuck, pumps always have their nominal throughput, mixers function normally, pipes do not get clogged, sensors always show correct readings, etc. Since the control software was not designed to detect such failures, we can only prove its correct functioning in the absence of these disturbances.

Furthermore we assume that there is an unlimited supply of raw materials in tanks B41–B44. Therefore, these four tanks will be modeled implicitly.

Below we describe two classes of desired properties: those related to the normal operation of the plant, i.e., properties that describe the purpose of the plant, and those describing failures which should not happen, since these may lead to a damage of the plant and/or its environment.

Normal Operation

Since the purpose of the plant is to deliver the products "blue" and "green" in tanks B31 and B32, the only property desired for normal operation is:

- There is always a nonzero amount of liquid in the product tanks B31 and B32; the total amount of liquid entering B31 and B32 is unbounded.

Prevention of Failures

The following properties must hold to ensure the safe operation of the plant.

- There is no overflow in any of the tanks or reactors, i.e., they are never filled beyond their nominal capacity.
- No mixer operates when the reactor is empty.
- Each reactor is filled with "yellow" or "red" first, and then "white" is added.
- During a reaction in any of the reactors (i.e., filling in "white" after filling in "yellow" or "red") the mixer is always in operation.
- Each reactor is drained only if the reaction has finished, i.e., the reactor contains either "blue" or "green".
- B11 contains only "yellow", B12 "red", B13 "white", B31 "blue", and B32 "green".
- From one tank or reactor, the liquid is drained to only one tank or one reactor at a time.
- Inlet and outlet valves of a tank or reactor are never open simultaneously.
- Only complete batches are transferred:
  - Draining from a tank is not started while the tank is being filled.
  - Draining is started only if at least one batch is available.
  - Draining is stopped after exactly one batch has been transferred.

Once the last property about complete batches has been verified (the proof is not shown in this chapter), an abstract model which only considers transfers of complete batches can be used to establish the other properties. This model is developed next.

5.4 Plant Model

We choose a discrete abstraction as our formal model: Although the flow of liquids during the production steps is continuous, an analysis of the plant and its control software (not shown in this chapter) yields that the production process is designed in such a way that only complete batches are being transferred between vessels. It is therefore possible to use a discrete model in which the transfer of one batch is modeled as one discrete state change.

Our system model can be partitioned into three main parts: the plant hardware (tanks, reactors, mixers, valves, pumps, and pipes), the software (production programs for "blue" and "green", resource management, etc.), and transfers among and between the two parts (liquids, level sensor data, commands to valves, pumps, and mixers, etc.).

All pieces of hardware and software are modeled as single modules. Communication between modules is based on *shared variables* (for level sensor data and other state information) and *events* (e.g., for sending commands to valves and passing on information about liquid transfers). Events can be seen as Boolean variables which are *true* if and only if the event occurs in the current computation step. Communication via events is synchronous.

In the following we describe the purpose of each module and the events and variables that are used for the interaction of the modules.

5.4.1 Plant Hardware

This section lists all modules that represent some physical device of the plant, i.e., tanks, reactors, sensors, actuators, and the piping. Figure 5.2 contains the block diagrams of these modules showing their interconnections. Each arrow represents a directed data flow (either via an event or a shared variable) from one module to another. Arrows not pointing to a specific module denote outputs that are sent to modules representing the control software (not shown in Figure 5.2), and arrows not starting at a module represent commands sent from the control software to the actuators.

The modules in Figure 5.2 have been arranged to resemble the topological structure of the piping and instrumentation diagram in Figure 3.2; note that arrows in Figure 5.2 represent data flows in our abstract plant model, whereas arrows in Figure 3.2 denote the flow of liquids through pipes in the actual plant.

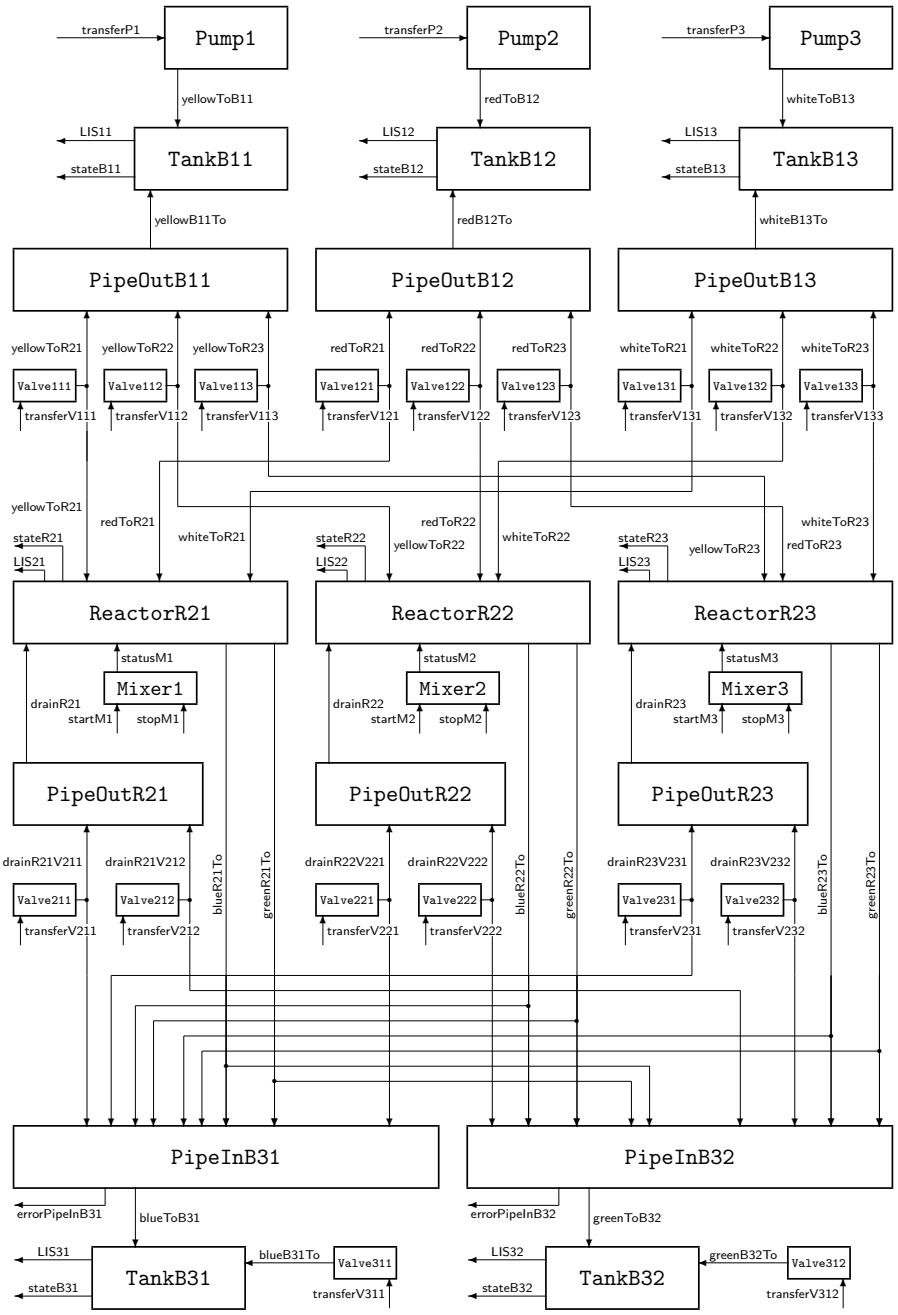


Figure 5.2: Block diagrams of the modules for the physical devices.

Pumps A pump Pj, $j \in \{1,2,3\}$, is represented by a module `Pump`j. The event transferPj sent by the control software triggers the transfer of one batch of liquid to `TankB1`j.

Raw Material Tanks A raw material tank B1j, $j \in \{1,2,3\}$, is represented by a module `TankB1`j. The event cToB1j (sent by `Pump`j) triggers the transfer of one batch of color c into B1j. The event cB1jTo (sent by `PipeOutB1`j) triggers the draining of one batch of color c from B1j into one of the reactors. The variable LIS1$j \in \{0,1,2\}$ contains the number of batches in B1j.

Reactors A reactor R2j, $j \in \{1,2,3\}$, is represented by a module `ReactorR2`j. The event cToR2j triggers the transfer of one batch of color c into R2j, and drainR2j triggers the draining of the reactor. The events blueR2jTo and greenR2jTo denote which of the two products is inside the reactor when it is drained. The variable LIS2$j \in \{0,1,2\}$ contains the number of batches in R2j.

Mixers A mixer Mj, $j \in \{1,2,3\}$, is represented by a module `Mixer`j. It is operated by the events startMj and stopMj, and its current status is passed to R2j using the statusMj variable.

Product Tanks A product tank B3j, $j \in \{1,2\}$, is represented by a module `TankB3`j. The event cToB3j triggers the transfer of one batch of color c from one reactor into B3j. The event cB3jTo triggers the draining of one batch of color c from B3j. The variable LIS3$j \in \{0,1,2,3\}$ contains the number of batches in B3j.

Valves Valve V1jk, $j,k \in \{1,2,3\}$, connects B1j with R2k. Valve V2jk, $j \in \{1,2,3\}$, $k \in \{1,2\}$, connects R2j with B3k. Valve V31j, $j \in \{1,2\}$, is used for draining B3j. The event transferVijk triggers the transfer of one batch through Vijk.

Pipes The modules `PipeOutB1`j, `PipeOutR2`j, and `PipeInB3`k are introduced to keep the communication interfaces of the tanks and reactors small. E.g., it is not important for `TankB11` into which of the three reactors "yellow" is drained. Therefore, `PipeOutB11` abstracts all three yellowToR2j events into one yellowB11To event which is sent to `TankB11`. The other pipe modules work similarly.

State Variables The variables stateB11, stateB12, stateB13, stateR21, stateR22, stateR23, stateB31, stateB32, errorPipeInB31, and errorPipeInB32 show the current state of the respective module. These variables are not

used by the control programs, but in the specifications they describe "bad" states that should be avoided.

5.4.2 Plant Software

Our abstract model of the control software consists of the following modules:

Semaphores The modules `SemaphoreB11`, `SemaphoreB12`, `SemaphoreB13`, `SemaphoreR21`, `SemaphoreR22`, `SemaphoreR23`, `SemaphoreB31`, and `SemaphoreB32` are used by the production and delivery programs to control concurrent access to the tanks and reactors. The semaphore modules have one Boolean input variable for each program to request access the resource (lock_B1j_D, lock_B11_Bj, lock_B12_Gj, lock_B13_Bj, lock_B13_Gj, lock_R2j_B, lock_R2j_G, lock_B31_Bj, lock_B32_Gj, lock_B31_C, lock_B32_C), and output variables (Bij_user, R2j_user) that show to which program access is currently granted.

Counters The modules `CounterB11`, `CounterB12`, `CounterB13`, `CounterB31`, and `CounterB32` are used to count the number of batches in the raw material and product tanks. Each module has input events for each program that needs to increase or decrease the counter (add_B1j, sub_Bij_Bk, sub_Bij_Gk, add_B31_Bk, add_B32_Gk, sub_B3j), and one output variable cij that shows the current counter value for tank Bij.

Raw Material Delivery The module `PrgDeliverToB1`j, $j \in \{1, 2, 3\}$, delivers one batch of liquid into tank B1j, using pump Pj. It requests access through `SemaphoreB1`j and notifies `CounterB1`j that one batch was added to B1j.

Production The modules `ProduceBlueInR2`j and `ProduceGreenInR2`j, $j \in \{1, 2\}$ control the production of "blue", resp., "green" in reactor R2j. They use `SemaphoreB11`, resp., `SemaphoreB12`, `SemaphoreB13`, `SemaphoreR2`j, and `SemaphoreB31`, resp., `SemaphoreB32` to request access to resources. Transfer events are sent to `Valve11`j, resp., `Valve12`j, `Valve13`j, and `Valve2`j`1`, resp., `Valve2`j`2`. Furthermore, `Mixer`j is turned on if and only if the second ingredient is drained into reactor R2j.

The model presented above is adequate for the verification of the properties listed in Section 5.3.1. If we are interested in other classes of desired plant properties, e.g., information about production times based on the durations of the basic production steps (like filling a reactor) given in the plant description [BKSL00], different models are required.

5.5 Compositional Verification

This section illustrates the verification step in our approach for proving global properties of the batch plant. We apply the approach to the following two global properties:

1. There is no overflow in tank B11, i.e., at any time there are no more than two batches of "yellow" in B11.

2. There is no "underflow" in tank B11, i.e., one never tries to drain a batch of "yellow" from B11 when it is already empty.

These properties are illustrative for showing our verification concept, for the following reasons: The non-existence of overflows and underflows in the other vessels is symmetrical to the situation with tank B11. The properties referring to the reactors can be proven by analysis of the production programs. The property that a tank only contains liquids of a certain color is trivial for B11, B12, and B13; for B31 and B32 this can also be shown from properties of the production programs. The remaining properties (draining only to one destination at a time, no simultaneous opening of inlet and outlet valves) can be proven from properties of the production and semaphore programs.

These proofs make no qualitative difference to the proofs of the two properties of B11.

5.5.1 Establishing Local Specifications

Since we are interested in formally proving global properties of the plant, it is necessary to establish local specifications for each module which entail global properties. The abstract model presented in Section 5.4 already introduced all modules and the communication structure, along with an informal description of the modules' behavior. Now we define their local specifications formally.

We specify each module as an *open system,* i.e., the outputs of a module depend on inputs from its environment. Most modules cannot guarantee a certain behavior without assuming some behavior of other modules. E.g., `TankB11` can only guarantee to avoid overflows if `Pump1` does not fill another batch into B11 when it is already full. Specifications of such dependent behavior are therefore given as implications: "*if* the environment behaves as expected, *then* we guarantee something".

The specification language we use is LTL (linear time temporal logic) [Pnu77] which is interpreted over sequences of states displaying the activities of the modules in a discrete time framework. Each state contains the current evaluation of the variables as well as the existence of events (encoded as a Boolean variable) at the beginning of the current computation

step. Changes between steps can be logically related by using the *next operator,* denoted by the symbol $\bigcirc$, which refers to the next state in the computation. Propositions which hold at all states in the computation are preceded by the *always operator,* denoted by the symbol $\Box$.

In the case of the modules for the plant hardware, the local specifications are formalizations of the physical behavior of their real-life counterparts, like "if one never fills another batch into a tank which already contains a maximal number of batches, it will never overflow". These specifications are assumed to be correct, based on the knowledge about physical facts. Similarly, we accept the chemical fact that mixing "yellow" and "white" results in "blue". Local properties for the modules `TankB11`, `Pump1`, `Valve111`, `Valve112`, `Valve113`, and `PipeOutB11` are given in Section 5.5.3.

In the case of the control software, things are different. Properties of finite-state programs can be proven algorithmically. The control software for the batch plant runs on a PLC. The PLC programming language used for the implementation is sequential function charts (SFC), and the execution model of an SFC is finite-state. Therefore, we translate each SFC program into the modeling language of a model checker, and prove its correctness with respect to local properties, such as (5.12)–(5.23) listed in Section 5.5.3. The translation is discussed in Section 5.6. For more background on model checking see [CGP00].

Local properties for the modules `Counter11`, `PrgDeliverToB11`, and `PrgProduceBlueInR2`j, $j \in \{1, 2, 3\}$, are given in Section 5.5.3.

It is of course possible to give algorithmically checkable models for the plant hardware as well, but these would not give us any additional knowledge about the real plant. Such models are simply based on the physical and chemical facts mentioned above, and will trivially fulfill the requirements posed upon them, without adding any confidence about the correctness of the plant. When using a global model-checking approach, such models are required, since the full automata product of the model includes the software as well as the hardware parts.

5.5.2 Desired Properties

Now we show that that there is no overflow in tank B11, i.e., that at any time there are no more than two batches of "yellow" in B11. Furthermore, we also want to prevent an "underflow", i.e., one should never try to drain a batch from B11 if B11 is empty.

In our model the variable stateB11 contains the current state of `TankB11`. The two values describing "bad" states are *over* and *under*. Therefore, the requirements above can be formalized in LTL as

$$\Box(\mathsf{stateB11} \neq \mathit{over}) \tag{5.1}$$

and

$$\Box(\mathsf{stateB11} \neq \mathit{under}), \tag{5.2}$$

i.e., it is always the case that $\mathsf{stateB11} \notin \{\mathit{over}, \mathit{under}\}$.

5.5.3 Plant Specifications

We list the local specifications that are needed to prove the global properties (5.1) and (5.2). Note that there are more specifications for the modules than those mentioned below, but we restrict ourselves to those necessary for proving these global properties.

Recall that the symbols used for events and variables used in the following specifications have been introduced in Section 5.4.

Plant Hardware

As argued in Section 5.5.1, the following local specifications for the hardware components are derived from their physical characteristics and are assumed to be correct, without further proof.

Raw Material Tank B11 Recall that the module `TankB11` maintains the variable $\mathsf{LIS11}$ which counts the number of batches in B11. A batch of "yellow" is added if the $\mathsf{yellowToB11}$ event is received, and a batch of "yellow" is drained from B11 if the $\mathsf{yellowB11To}$ event is received. Furthermore, $\mathsf{stateB11}$ is used to indicate if B11 is in a "bad" state.

Initially, the tank is empty:

$$\mathsf{LIS11} = 0 \tag{5.3}$$

The level stays the same if there is no filling or draining:

$$\Box(\neg \bigcirc(\mathsf{yellowToB11} \vee \mathsf{yellowB11To}) \Rightarrow \bigcirc\, \mathsf{LIS11} = \mathsf{LIS11}) \tag{5.4}$$

The following two specifications contain the requirement that the control programs never send the filling and draining events simultaneously.

Filling the tank when not full increases the level:

$$\begin{array}{l} \Box\, \neg(\mathsf{yellowToB11} \wedge \mathsf{yellowB11To}) \\ \Rightarrow \Box(\bigcirc\, \mathsf{yellowToB11} \wedge \mathsf{LIS11} \neq 2 \Rightarrow \bigcirc\, \mathsf{LIS11} = \mathsf{LIS11} + 1) \end{array} \tag{5.5}$$

Draining the tank when not empty decreases the level:

$$\begin{array}{l} \Box\, \neg(\mathsf{yellowToB11} \wedge \mathsf{yellowB11To}) \\ \Rightarrow \Box(\bigcirc\, \mathsf{yellowToB11} \wedge \mathsf{LIS11} \neq 0 \Rightarrow \bigcirc\, \mathsf{LIS11} = \mathsf{LIS11} - 1) \end{array} \tag{5.6}$$

If the tank is not drained while empty, there is no underflow:

$$\Box(\mathsf{LIS11} = 0 \Rightarrow \neg \bigcirc\, \mathsf{yellowB11To}) \Rightarrow \Box(\mathsf{stateB11} \neq \mathit{under}) \tag{5.7}$$

If the tank is not filled while full, there is no overflow:

$$\Box(\mathsf{LIS11} = 2 \Rightarrow \neg \bigcirc\, \mathsf{yellowToB11}) \Rightarrow \Box(\mathsf{stateB11} \neq \mathit{over}) \tag{5.8}$$

Pump P1 The module `Pump1` converts the transfer event from the control software into a filling event for `TankB11`: A batch of "yellow" is filled into B11 whenever transferP1 is sent:

$$\Box(\mathsf{transferP1} \Leftrightarrow \mathsf{yellowToB11}) \tag{5.9}$$

Valve V11j**,** $j \in \{1,2,3\}$ The module `Valve11`j converts the transfer event from the control software into a filling event for `ReactorR2`j: A batch of "yellow" is filled into R2j whenever transferV11j is sent:

$$\Box(\mathsf{transferV11}j \Leftrightarrow \mathsf{yellowToR2}j) \tag{5.10}$$

Pipes Connecting B11 **with the Reactors** The module `PipeOutB11` provides the yellowB11To event for `TankB11`. Filling one batch of "yellow" into one of the reactors is equivalent to draining one batch of "yellow" from B11. The antecedent of the following implication ensures that two reactors are never filled simultaneously from B11:

$$\begin{aligned}&\Box\,\neg(\mathsf{yellowToR21} \wedge \mathsf{yellowToR22} \vee \mathsf{yellowToR21} \wedge \mathsf{yellowToR23} \vee \\ &\quad \mathsf{yellowToR22} \wedge \mathsf{yellowToR23}) \\ &\Rightarrow \Box(\mathsf{yellowToR21} \vee \mathsf{yellowToR22} \vee \mathsf{yellowToR23} \Leftrightarrow \mathsf{yellowB11To})\end{aligned} \tag{5.11}$$

Plant Software

In the following we list local specifications of the control software. In contrast to the specifications of the hardware parts given above, these have been proven correct by model checking, as mentioned in Section 5.5.1.

The Counter for Tank B11 The module `Counter11` provides a variable c11 which is used by the control programs to keep track of the number of batches in B11, which is not allowed to be larger than two. Note that by using c11 the control programs do not rely on LIS11 provided by B11, but have their own information about the number of batches in B11.

Initially, the counter is zero:

$$\mathsf{c11} = 0 \tag{5.12}$$

The counter stays the same if there is no increasing or decreasing event:

$$\begin{aligned}&\Box(\neg \bigcirc(\mathsf{add\_B11} \vee \mathsf{sub\_B11\_B1} \vee \mathsf{sub\_B11\_B2} \vee \mathsf{sub\_B11\_B3}) \\ &\quad \Rightarrow \bigcirc\,\mathsf{c11} = \mathsf{c11})\end{aligned} \tag{5.13}$$

The following two specifications require that there is at most one increasing or decreasing event at a time.

Sending an increasing event while $\mathsf{c11} \neq 2$ increases the counter by one:

$$\begin{aligned}&\Box\,\neg(\mathsf{add\_B11} \wedge \mathsf{sub\_B11\_B1} \vee \mathsf{add\_B11} \wedge \mathsf{sub\_B11\_B2} \vee \\ &\quad \mathsf{add\_B11} \wedge \mathsf{sub\_B11\_B3} \vee \mathsf{sub\_B11\_B1} \wedge \mathsf{sub\_B11\_B2} \vee \\ &\quad \mathsf{sub\_B11\_B1} \wedge \mathsf{sub\_B11\_B3} \vee \mathsf{sub\_B11\_B2} \wedge \mathsf{sub\_B11\_B3}) \\ &\Rightarrow \Box(\bigcirc\,\mathsf{add\_B11} \wedge \mathsf{c11} \neq 2 \Rightarrow \bigcirc\,\mathsf{c11} = \mathsf{c11} + 1)\end{aligned} \tag{5.14}$$

Sending a decreasing event while $\mathsf{c11} \neq 0$ decreases the counter by one:

$$\begin{aligned}&\Box\,\neg(\mathsf{add\_B11} \wedge \mathsf{sub\_B11\_B1} \vee \mathsf{add\_B11} \wedge \mathsf{sub\_B11\_B2} \vee \\ &\quad \mathsf{add\_B11} \wedge \mathsf{sub\_B11\_B3} \vee \mathsf{sub\_B11\_B1} \wedge \mathsf{sub\_B11\_B2} \vee \\ &\quad \mathsf{sub\_B11\_B1} \wedge \mathsf{sub\_B11\_B3} \vee \mathsf{sub\_B11\_B2} \wedge \mathsf{sub\_B11\_B3}) \\ &\Rightarrow \Box(\bigcirc(\mathsf{sub\_B11\_B1} \vee \mathsf{sub\_B11\_B2} \vee \mathsf{sub\_B11\_B3}) \wedge \mathsf{c11} \neq 0 \\ &\qquad \Rightarrow \bigcirc\,\mathsf{c11} = \mathsf{c11} - 1)\end{aligned} \tag{5.15}$$

Production of "blue" in R2j**,** $j \in \{1, 2, 3\}$ The module `PrgProduceBlueInR2`j controls the production of "blue" in reactor R2j. It sends the transferV11j event to `Valve11`j, decreases the counter c11, and it regards the B11_user semaphore.

Initially, no events are sent by the program:

$$\neg\mathsf{transferV11}j \wedge \neg\mathsf{sub\_B11\_B}j \tag{5.16}$$

If the counter equals zero, B11 is not drained via V11j:

$$\Box(\mathsf{c11} = 0 \Rightarrow \neg\bigcirc\,\mathsf{transferV11}j) \tag{5.17}$$

Whenever V11j is used to transfer one batch, the counter is notified to decrease:

$$\Box(\mathsf{transferV11}j \Leftrightarrow \mathsf{sub\_B11\_B}j) \tag{5.18}$$

Whenever an event related to B11 is sent, the semaphore for B11 is reserved:

$$\Box(\bigcirc(\mathsf{transferV11}j \vee \mathsf{sub\_B11\_B}j) \Rightarrow \mathsf{B11\_user} = \mathit{PrgB}j) \tag{5.19}$$

Raw Material Delivery for Tank B11 The module `PrgDeliverToB11` controls the delivery of "yellow" into tank B11. It sends the transferP1 event to `Pump1`, increases the counter c11, and it regards the B11_user semaphore.

Initially, no events are sent by the program:

$$\neg\mathsf{transferP1} \wedge \neg\mathsf{add\_B11} \tag{5.20}$$

If the counter equals 2, B11 is not filled via P1:

$$\Box(\mathsf{c11} = 2 \Rightarrow \neg\bigcirc\,\mathsf{transferP1}) \tag{5.21}$$

Whenever P1 is activated to pump one batch, the counter is notified to increase:

$$\Box(\mathsf{transferP1} \Leftrightarrow \mathsf{add\_B11}) \tag{5.22}$$

Whenever an event related to B11 is sent, the semaphore for B11 is reserved:

$$\Box(\bigcirc(\mathsf{transferP1} \vee \mathsf{add\_B11}) \Rightarrow \mathsf{B11\_user} = \mathit{PrgD}) \tag{5.23}$$

5.5.4 Deduction

Now we come to the crucial step in this verification methodology: From the correct (proven by model checking for the software part) local specifications we deduce more complex specifications for the parallel composition of all plant parts.

The local specifications given above are used to deduce our global requirements (5.1) and (5.2). We start by collecting the specifications for the software parts `PrgDeliverToB11` and `PrgProduceBlueInR2`j, $j \in \{1, 2, 3\}$.

From (5.16) and (5.20) we know that no transfer events and no counter changing events are present in the initial state:

$$\begin{aligned}&\neg\mathsf{transferV111} \wedge \neg\mathsf{transferV112} \wedge \neg\mathsf{transferV113} \wedge \neg\mathsf{transferP1} \wedge \\ &\neg\mathsf{sub\_B11\_B1} \wedge \neg\mathsf{sub\_B11\_B2} \wedge \neg\mathsf{sub\_B11\_B3} \wedge \neg\mathsf{add\_B11}\end{aligned} \tag{5.24}$$

From (5.17) we get that no draining transfer events are sent when the counter equals 0, and from (5.21) we know that no filling transfer events are sent when the counter equals 2:

$$\begin{aligned}&\Box(\mathsf{c11} = 0 \Rightarrow \neg \bigcirc(\mathsf{transferV111} \vee \mathsf{transferV112} \vee \mathsf{transferV113})) \\ &\wedge \Box(\mathsf{c11} = 2 \Rightarrow \neg \bigcirc \mathsf{transferP1})\end{aligned} \tag{5.25}$$

From (5.18) and (5.22) we get that whenever a transfer event is sent, the counter is updated accordingly:

$$\begin{aligned}&\Box(\mathsf{transferV111} \Leftrightarrow \mathsf{sub\_B11\_B1}) \wedge \\ &\Box(\mathsf{transferV112} \Leftrightarrow \mathsf{sub\_B11\_B2}) \wedge \\ &\Box(\mathsf{transferV113} \Leftrightarrow \mathsf{sub\_B11\_B3}) \wedge \\ &\Box(\mathsf{transferP1} \Leftrightarrow \mathsf{add\_B11})\end{aligned} \tag{5.26}$$

From (5.19) and (5.23) we collect the information about the use of the semaphore B11_user:

$$\begin{aligned}&\Box(\bigcirc(\mathsf{transferV111} \vee \mathsf{sub\_B11\_B1}) \Rightarrow \mathsf{B11\_user} = \mathit{PrgB1}) \wedge \\ &\Box(\bigcirc(\mathsf{transferV112} \vee \mathsf{sub\_B11\_B2}) \Rightarrow \mathsf{B11\_user} = \mathit{PrgB2}) \wedge \\ &\Box(\bigcirc(\mathsf{transferV113} \vee \mathsf{sub\_B11\_B3}) \Rightarrow \mathsf{B11\_user} = \mathit{PrgB3}) \wedge \\ &\Box(\bigcirc(\mathsf{transferP1} \vee \mathsf{add\_B11}) \Rightarrow \mathsf{B11\_user} = \mathit{PrgD})\end{aligned} \tag{5.27}$$

By using the specifications of `Valve11`j, $j \in \{1,2,3\}$, and `Pump1`, we can substitute in (5.24)–(5.27) transferV11j by yellowToR2j, for $j \in \{1,2,3\}$, and transferP1 by yellowToB11, and we get:

$$\begin{array}{l}\neg\mathsf{yellowToR21} \wedge \neg\mathsf{yellowToR22} \wedge \neg\mathsf{yellowToR23} \wedge \neg\mathsf{yellowToB11} \\ \wedge \neg\mathsf{sub\_B11\_B1} \wedge \neg\mathsf{sub\_B11\_B2} \wedge \neg\mathsf{sub\_B11\_B3} \wedge \neg\mathsf{add\_B11}\end{array} \tag{5.28}$$

$$\begin{array}{l}\Box(\mathsf{c11} = 0 \Rightarrow \neg \bigcirc(\mathsf{yellowToR21} \vee \mathsf{yellowToR22} \vee \mathsf{yellowToR23})) \\ \wedge \Box(\mathsf{c11} = 2 \Rightarrow \neg \bigcirc \mathsf{yellowToB11})\end{array} \tag{5.29}$$

$$\begin{array}{l}\Box(\mathsf{yellowToR21} \Leftrightarrow \mathsf{sub\_B11\_B1}) \wedge \\ \Box(\mathsf{yellowToR22} \Leftrightarrow \mathsf{sub\_B11\_B2}) \wedge \\ \Box(\mathsf{yellowToR23} \Leftrightarrow \mathsf{sub\_B11\_B3}) \wedge \\ \Box(\mathsf{yellowToB11} \Leftrightarrow \mathsf{add\_B11})\end{array} \tag{5.30}$$

$$\begin{array}{l}\Box(\bigcirc(\mathsf{yellowToR21} \vee \mathsf{sub\_B11\_B1}) \Rightarrow \mathsf{B11\_user} = \mathit{PrgB1}) \wedge \\ \Box(\bigcirc(\mathsf{yellowToR22} \vee \mathsf{sub\_B11\_B2}) \Rightarrow \mathsf{B11\_user} = \mathit{PrgB2}) \wedge \\ \Box(\bigcirc(\mathsf{yellowToR23} \vee \mathsf{sub\_B11\_B3}) \Rightarrow \mathsf{B11\_user} = \mathit{PrgB3}) \wedge \\ \Box(\bigcirc(\mathsf{yellowToB11} \vee \mathsf{add\_B11}) \Rightarrow \mathsf{B11\_user} = \mathit{PrgD})\end{array} \tag{5.31}$$

From (5.28) and (5.31) we can conclude that at most one of the events yellowToR2j occurs at a time; thus we get the following from the specification of `PipeOutB11`:

$$\Box(\mathsf{yellowToR21} \vee \mathsf{yellowToR22} \vee \mathsf{yellowToR23} \Leftrightarrow \mathsf{yellowB11To}) \tag{5.32}$$

Similarly, we conclude that there is at most one of the add or sub events at a time, and get from the specification of `Counter11` and (5.30):

$$\mathsf{c11} = 0 \tag{5.33}$$

$$\Box(\neg \bigcirc(\mathsf{yellowToB11} \vee \mathsf{yellowB11To}) \Rightarrow \bigcirc \mathsf{c11} = \mathsf{c11}) \tag{5.34}$$

$$\Box(\bigcirc \mathsf{yellowToB11} \wedge \mathsf{c11} \neq 2 \Rightarrow \bigcirc \mathsf{c11} = \mathsf{c11} + 1) \tag{5.35}$$

$$\Box(\bigcirc \mathsf{yellowB11To} \wedge \mathsf{c11} \neq 0 \Rightarrow \bigcirc \mathsf{c11} = \mathsf{c11} - 1) \tag{5.36}$$

From (5.29) we conclude with (5.32):

$$\Box(\mathsf{c11} = 0 \Rightarrow \neg \bigcirc \mathsf{yellowB11To}) \tag{5.37}$$

$$\Box(\mathsf{c11} = 2 \Rightarrow \neg \bigcirc \mathsf{yellowToB11}) \tag{5.38}$$

From (5.28) and (5.31) we get:

$$\Box \neg(\mathsf{yellowToB11} \wedge \mathsf{yellowB11To}) \tag{5.39}$$

Now recall the specification of module `TankB11`, where the requirement $\Box\,\neg(\mathsf{yellowToB11} \wedge \mathsf{yellowB11To})$ is already discharged using (5.39):

$$\mathsf{LIS11} = 0 \tag{5.40}$$

$$\Box(\neg \bigcirc(\mathsf{yellowToB11} \vee \mathsf{yellowB11To}) \Rightarrow \bigcirc\,\mathsf{LIS11} = \mathsf{LIS11}) \tag{5.41}$$

$$\Box(\bigcirc\,\mathsf{yellowToB11} \wedge \mathsf{LIS11} \neq 2 \Rightarrow \bigcirc\,\mathsf{LIS11} = \mathsf{LIS11} + 1) \tag{5.42}$$

$$\Box(\bigcirc\,\mathsf{yellowToB11} \wedge \mathsf{LIS11} \neq 0 \Rightarrow \bigcirc\,\mathsf{LIS11} = \mathsf{LIS11} - 1) \tag{5.43}$$

$$\Box(\mathsf{LIS11} = 0 \Rightarrow \neg \bigcirc\,\mathsf{yellowB11To}) \Rightarrow \Box(\mathsf{stateB11} \neq \mathit{under}) \tag{5.44}$$

$$\Box(\mathsf{LIS11} = 2 \Rightarrow \neg \bigcirc\,\mathsf{yellowToB11}) \Rightarrow \Box(\mathsf{stateB11} \neq \mathit{over}) \tag{5.45}$$

We can prove the global properties (5.1) and (5.2) from (5.44) and (5.45) if we can establish $\Box(\mathsf{LIS11} = 0 \Rightarrow \neg \bigcirc\,\mathsf{yellowB11To})$ and $\Box(\mathsf{LIS11} = 2 \Rightarrow \neg \bigcirc\,\mathsf{yellowToB11})$. These two conditions can be proven from (5.37) and (5.38) if we can prove $\Box\,\mathsf{LIS11} = \mathsf{c11}$, i.e., the counter variable $\mathsf{c11}$ always has the same value as the level sensor variable $\mathsf{LIS11}$.

The validity of $\Box\,\mathsf{LIS11} = \mathsf{c11}$ is proven in Section 5.5.5. This completes the deductive proof of the global properties (5.1) and (5.2).

5.5.5 Temporal Induction

As explained above, we need to prove $\Box\,\mathsf{LIS11} = \mathsf{c11}$ to complete the proof of (5.1) and (5.2). By comparing (5.33)–(5.36) and (5.40)–(5.43), it can be deduced that $\mathsf{LIS11}$ and $\mathsf{c11}$ always have the same values, since they are both initialized to 0, and both change in the same way triggered by the same events.

A formal proof of $\Box\,\mathsf{LIS11} = \mathsf{c11}$ requires a technique that is more elaborate than the implication-driven deduction used in Section 5.5.4. This proof technique is called *temporal induction* [ZdBdR84].

We prove $\Box\,\mathsf{LIS11} = \mathsf{c11}$ by temporal induction over all traces of the system. A trace t is a sequence of states that are generated by the system. The sequence t^k, for $k \in \mathbf{N}$, denotes t without its first k elements. The LTL semantics defines that we can relate a state and its successor state by using the following formula:

$$t^k \models \bigcirc\,\varphi \text{ if and only if } t^{k+1} \models \varphi,$$

for any LTL formula φ and $k \in \mathbf{N}$.

Proof. Let t be a trace of the system. In Section 5.5.4 we have already established several LTL properties of the system. These properties restrict t, and we will make use of that in this proof.

We show $t \models \Box\,\mathsf{LIS11} = \mathsf{c11}$, i.e., $\forall k \in \mathbf{N} : t^k \models \mathsf{LIS11} = \mathsf{c11}$, by induction on k:

Induction base ($k = 0$): $t^0 \models \mathsf{LIS11} = \mathsf{c11}$ follows from (5.40) and (5.33).
Induction step: Let $k \in \mathbf{N}$, and let the induction hypothesis

$$t^k \models \mathsf{LIS11} = \mathsf{c11}$$

hold. We need to prove $t^{k+1} \models \mathsf{LIS11} = \mathsf{c11}$, i.e., $t^k \models \bigcirc \mathsf{LIS11} = \bigcirc \mathsf{c11}$.

Case 1: $t^{k+1} \not\models \mathsf{yellowToB11} \vee \mathsf{yellowB11To}$. By (5.41), $t^k \models \bigcirc \mathsf{LIS11} = \mathsf{LIS11}$, and, by (5.34), $t^k \models \bigcirc \mathsf{c11} = \mathsf{c11}$. With the induction hypothesis, we get $t^k \models \bigcirc \mathsf{LIS11} = \bigcirc \mathsf{c11}$.

Case 2: $t^{k+1} \models \mathsf{yellowToB11} \vee \mathsf{yellowB11To}$. By (5.39), we know that $t^{k+1} \models \mathsf{yellowToB11} \wedge \mathsf{yellowB11To}$ does not hold, and thus, either $t^{k+1} \models \mathsf{yellowToB11}$ or $t^{k+1} \models \mathsf{yellowB11To}$ holds.

Case 2a: $t^{k+1} \models \mathsf{yellowToB11}$. By (5.38), we have $t^k \models \mathsf{c11} \neq 2$, and by the induction hypothesis, $t^k \models \mathsf{LIS11} \neq 2$. With (5.39), we get from (5.42): $t^k \models \bigcirc \mathsf{LIS11} = \mathsf{LIS11} + 1$. We get from (5.35): $t^k \models \bigcirc \mathsf{c11} = \mathsf{c11} + 1$. With the induction hypothesis, we conclude $t^k \models \bigcirc \mathsf{LIS11} = \bigcirc \mathsf{c11}$.

Case 2b: $t^{k+1} \models \mathsf{yellowB11To}$. Follows analogously from (5.37), (5.43), and (5.36).

Inductive reasoning is also needed if there is *feedback* between modules, i.e., the behavior of one part of the system depends on the behavior of another part, and vice versa. A detailed example for this proof technique is given in [ZdBdR84].

5.6 Algorithmic Verification

The previous section showed how local properties can be used to deduce global properties. This section illustrates how the correctness of local properties of the PLC programs used in the plant can be established by algorithmic verification. We show the correctness of a representative part which has been chosen since it controls valves and uses semaphores as well as counters, which constitute the main control features of the plant.

Figure 5.3 shows an excerpt of the SFC program that controls the production of "blue" in reactor R21, in particular, the part which controls the filling of one batch of "yellow" from raw material tank B11 into reactor R21 through Valve V111. We show how this part is translated into the input language of the SAL model checker [BGL+00], which we use to mechanically verify the local LTL properties.

The part of the SFC shown in Figure 5.3 operates as follows: Control waits in step S1 until the guard associated with T1 evaluates to *true*. The guard is written in the language Ladder Diagram with embedded Function Block Diagrams (see Section 2.3.1) describing that the Boolean semaphore variables `t_11_free` and `t_21_free` have to be *true* (meaning that the resources B11 and R21 are currently available), and the counter variable `c_11` has to be larger than zero (i.e., there is at least one batch of "yellow" in

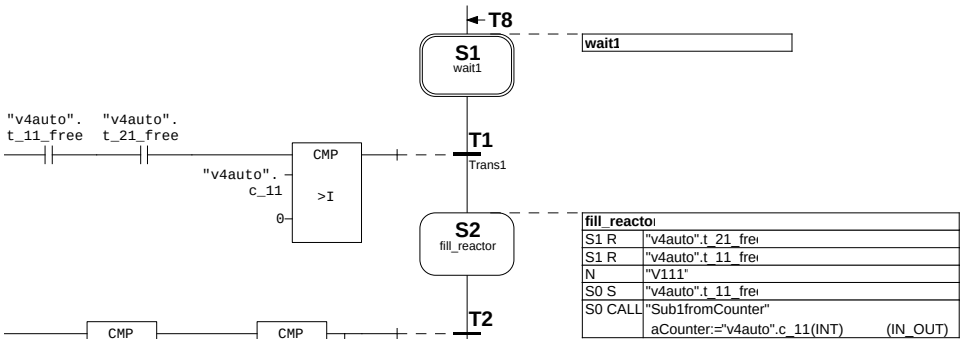


Figure 5.3: SFC program for the production of "blue" in R21 (excerpt)

B11). When S2 is entered, the variables `t_11_free` and `t_21_free` are set to *false* (meaning that B11 and R21 are no longer available for other programs), and valve V111 is opened. When S2 is left via transition T2 (which is guarded by expressions over `LIS11` stating that the level B11 has decreased by the amount of one batch), valve V111 is closed, `t_11_free` is set to *true* (the resource B11 is available again), and the counter `c_11` is decreased (by calling the PLC program `Sub1fromCounter` with parameter `c_11`).

Now we translate the SFC program into our discrete model as described in Section 5.4, which abstracts the sequence "open V111, wait for LIS111 to decrease by the amount of one batch, close V111" into one discrete step "drain one batch from B11 into R21".

For technical reasons, our model handles the semaphores differently than the SFC programs[1]; the variables lock_B11_B1 and lock_R21_B are set to *true* to request access, and the variables B11_user and R21_user contain the value *PrgB1* when the access has been granted.

Figure 5.4 shows the translation of the SFC transition and actions shown in Figure 5.3 into the modeling language of the SAL model checker. The variable `step` models the current active step of the SFC. Whenever this variable equals 2, the semaphore access is granted, and the counter `c11` is greater than zero, then in the next step (denoted by the primed variables) the event transferV111 (drain one batch from B11 into R21) is sent, the request for B11 is no longer sent (because after draining, B11 is available again), the request for R21 is kept (because the production in R21 is not yet finished), the event sub_B11_B1 is sent to `Counter11` to decrease `c11`, and the `step` variable is set to 2.

This translation is done for all software modules described in Section 5.4.2. Finally, all local LTL specifications (see Section 5.5.3) are fed into the

[1]The use of "pulse" qualifiers to change the semaphores is difficult to model in the synchronous framework of the SAL model checker.

```
TRANSITION
  [
    step=1 AND B11_user=PrgB1 AND R21_user=PrgB1 AND c11>0
      --> transferV111'=TRUE; transferV131'=FALSE; transferV211'=FALSE;
          lock_B11_B1'=FALSE; lock_B13_11'=FALSE; lock_R21_B'=TRUE;
          lock_B31_B1'=FALSE;
          sub_B11_B1'=TRUE; sub_B13_B1'=FALSE; add_B31_B1'=FALSE;
          startM1'=FALSE; stopM1'=FALSE; step'=2
```

Figure 5.4: SAL module `PrgProduceBlueInR21` (excerpt)

model checker in SAL syntax, e.g.,

```
THEOREM  PrgProduceBlueInR21
         |- ltl!G (transferV111 <=> sub_B11_B1)
```

for specification (5.18), and then model checked. The model checker either acknowledges that the LTL formula is valid, or gives a counterexample that shows a computation trace leading to the error.

Experiments show that even model checking the parallel composition of a small set of modules is not possible, since the state explosion problem forces the model checker to run out of memory. Hence, the compositional approach is absolutely necessary.

5.7 Discussion

We have presented a compositional method for the verification of chemical batch plants, illustrated by an example which is considered to be representative for this class of systems. The approach avoids the state explosion problem by decomposing the plant and its controller into smaller units which can be specified and model-checked locally. At the next stage the local specifications are composed by deduction to get global properties of the plant.

The main advantage of this method is the avoidance of the state explosion problem during model checking by splitting the task of checking one large system into several small checks. Furthermore, it is not necessary that all modules are formalized in the same modeling language, making it possible to use different tools during the local verification, e.g., model checkers for discrete and hybrid systems. The modular approach also promotes the reuse of modules from libraries containing models which have already been verified w.r.t. a standard interface specification. And if some part of the plant is changed after a successful verification, e.g., if some control program is rewritten, it is often possible to re-check the changed module locally, without invalidating the global proof.

A disadvantage of the compositional approach in comparison to a global one is the (often significant) overhead for the decomposition of the system model and the construction and the composition of the local specifications

(see Section 1.6.5 of [dRdBH$^+$01]). This can be ameliorated by choosing a coarse decomposition which creates modules that are just small enough to be locally checked by algorithmic means. The deduction process also requires more knowledge about mathematics, logic, and proof theory than a global approach. Actually this disadvantage played a significant role in the acceptance of compositional methods in several teams of engineers with whom we have collaborated.

Experiments show that given the present state of the art it is impossible to model-check our batch plant example on the basis of explicit product automata. That is, currently only (a combination of) abstraction and compositionality—as the one presented in this chapter—can lead to a machine-checkable proof.

Chapter 6

Hybrid Systems

The previous chapters used discrete frameworks for the modeling and specification of hardware and software components. Often these discrete models are sufficient to cover a wide range of behaviors and properties. However, especially when modeling hardware parts, such frameworks are not sufficient when continuous behavior needs to be studied. In that case, models for hybrid systems can be used. Hybrid systems contain discrete as well as continuous parts.

This chapter presents *communicating linear hybrid automata* (CLHA) as a modeling framework for the specification of hybrid systems. CLHA provide modular descriptions of system components and subsume most of the characteristics that are used in verification tools, e.g., discrete and continuous transitions, invariants, and communication via shared variables as well as synchronization symbols. The syntax and a compositional semantics of CLHA is defined formally, and propositional linear temporal logic is introduced as an abstract specification language for CLHA behavior. An example illustrates the use of the presented framework, and an application of the HyTech model checker is shown.

6.1 Introduction

Formal reasoning about hybrid systems often involves a lot of different modeling frameworks. Since many systems consist of a wide range of of hardware and software subsystems with varying complexities, it is necessary to individually choose models that are precise enough to describe a certain behavior, but on the other hand as coarse as possible to avoid complexity problems during the specification and verification process.

These choices lead to the use of different verification tools and their associated specification languages, e.g., the model checkers SMV (Symbolic Model Verifier) [McM00] and Uppaal [LPY97] for purely discrete systems, KRONOS [OY93] for timed automata [AD94], and HyTech [HHWT97] for

linear hybrid systems [ACHH93]. The models used in these tools employ different formalisms for state changes, communication and synchronization. All of them use discrete transition systems, some use continuous transitions. Communication can be implemented by shared variables, synchronization symbols, or both.

This heterogeneous field of models can make formal reasoning about the systems difficult, especially when interfaces between different subsystems are concerned. Therefore, we propose a formal modeling framework that subsumes a wide range of formalisms used in formal verification and that can be simplified on a case-by-case basis to match the input language of the tool one intends to use. This modeling framework is presented in the following section.

[ACH+95] presents a specification and verification framework for linear hybrid systems and points out decidability results. In [LSVW99] such a framework is generalized to hybrid I/O automata for modeling nonlinear systems.

6.2 Communicating Linear Hybrid Automata

This section introduces a formal modeling framework for the description of linear hybrid systems that is capable of subsuming a wide range of modeling paradigms used in formal verification, e.g., discrete or timed automata [AD94], discrete or timed condition/event systems [SK91, EKKP95], or linear hybrid automata [ACHH93]. We define *communicating linear hybrid automata* which have the following characteristics:

- discrete control locations,
- input and output variables,
- communication via shared variables,
- communication via synchronization symbols (directed, one-to-many), and
- continuous variables restricted by invariants and linear rates at each control location.

Other models can be embedded into this framework by syntactic restrictions. E.g., timed automata only have one kind of continuous variables called clocks. These are restricted to the fixed rate 1 and can only be set to 0 in discrete transitions.

6.2.1 Variables

Our modeling framework uses a global set V of typed variables. Each variable $v \in V$ can be read by any CLHA in the system, but at most one CLHA

is allowed to change it. We call the changeable (or controlled) variables of a CLHA its *output* variables.

Let V be a finite set of variables, and let *type* be a function assigning a type, i.e., a set of possible values like $\mathbf{B}$ (Booleans) or $\mathbf{R}$ (reals), to each variable.

Definition 6.1 (Evaluation) A function σ assigning to each variable $v \in V$ a value $\sigma(v) \in \mathit{type}(v)$ is called *evaluation* of V. We denote the set of all evaluations of V by Σ.

Notation: Given some subset of the variables (e.g., $V_y^x \subseteq V$), we use Σ with the same decorations (e.g., Σ_y^x) to denote the set of all evaluations of these variables and σ with the same decorations (e.g., σ_y^x) to denote the restriction of a given $\sigma \in \Sigma$ on these variables, if not defined otherwise. Furthermore, for any set M, let 2^M denote the set of all subsets of M.

6.2.2 Syntax

The formal syntax of a communicating linear hybrid automaton is defined as follows:

Definition 6.2 (Communicating linear hybrid automaton) A *communicating linear hybrid automaton* (CLHA)

$$\mathcal{A} = (Q, Q_0, V^{out}, \Sigma_0, R, I, L, E)$$

consists of

- a finite set Q of *locations,*
- a set $Q_0 \subseteq Q$ of *initial locations,*
- a set $V^{out} \subseteq V$ of *output variables,* constituting the set of *input variables* $V^{in} = V \setminus V^{out}$,
- a set $\Sigma_0 \subseteq \Sigma$ of *initial variable evaluations,*
- a function $R : Q \times V^{cout} \to \mathbf{R}$ assigning a *rate* at each location to each of the *continuous output variables* $V^{cout} = \{v \in V^{out} \mid \mathit{type}(v) = \mathbf{R}\}$,
- a function $I : Q \to 2^{\Sigma^{cout}}$ assigning an *invariant* for the continuous output variables to each location,
- a finite set L of *synchronization symbols,* consisting of two disjoint sets of *input symbols* L^{in} and *output symbols* L^{out}, and
- a set E of *edges,* where each edge $e = (q, l, \rho, q') \in E$ consists of a *source location* $q \in Q$, a *destination location* $q' \in Q$, a set of synchronization symbols $l \subseteq L$, and a *variable transition relation* $\rho \subseteq \Sigma \times \Sigma^{out}$.

6.2.3 Computation Semantics

The computations of a CLHA are defined by changes in three components: its (discrete) location, its variable evaluations, and the synchronization symbols that are communicated between the CLHA and its environment.

Initially, the CLHA is in one of the initial locations Q_0, and its variables are set to one of the initial variable evaluations Σ_0. These are changed during two different kinds of *computation steps:*

1. *Discrete step:* The CLHA changes its location and its variables according to one of its edges $e \in E$. The transition is instantaneous; time does not progress. The synchronization symbols of e will be used later to combine edges during parallel composition (see Section 6.2.4).

2. *Continuous step:* The CLHA remains at its current location for a finite amount of time $t \in \mathbf{R}_{>0}$, and the continuous output variables change according to their rate defined in R. The other output variables do not change. During the time period t, the invariant of the current location has to hold. No synchronization symbols are sent during continuous steps.

Definition 6.3 (Computations of a CLHA) Given a CLHA $\mathcal{A}$ as in Definition 6.2, a *computation of* $\mathcal{A}$ is a maximal or infinite sequence

$$(q_0, \sigma_0) \xrightarrow{l_1} (q_1, \sigma_1) \xrightarrow{l_2} (q_2, \sigma_2) \xrightarrow{l_3} \dots \ ,$$

where $q_0 \in Q_0$, $\sigma_0 \in \Sigma_0$, $q_i \in Q$, $\sigma_i \in \Sigma$, l_i is either a set of synchronization labels ($l_i \subseteq L$) or a time span ($l_i \in \mathbf{R}_{>0}$), and for all i, one of the following cases applies:

1. Discrete step: $l_{i+1} \subseteq L \wedge \exists \rho \subseteq \Sigma \times \Sigma^{out} : (q_i, l_{i+1}, \rho, q_{i+1}) \in E \wedge (\sigma_i, \sigma_{i+1}^{out}) \in \rho$.

2. Continuous step (evolution of continuous output variables): $l_{i+1} \in \mathbf{R}_{>0} \wedge q_{i+1} = q_i \wedge \sigma_{i+1}^{out} = \sigma_i^{out} \oplus_R^{q_i} l_{i+1} \wedge \forall 0 \leq t < l_{i+1} : \sigma_i^{cout} \oplus_R^{q_i} t \in I(q_i)$, where

$$(\sigma \oplus_R^q t)(v) = \begin{cases} \sigma(v) + t \cdot R(q, v) & \text{if } v \in V^{cout}, \\ \sigma(v) & \text{otherwise.} \end{cases}$$

We denote the set of all computations of $\mathcal{A}$ by $Comp(\mathcal{A})$.

Note that both discrete and continuous steps do not impose any restrictions on the values of the *input* variables in σ_{i+1}; they can change arbitrarily. Thus, it is possible for $\mathcal{A}$ to accept any changes of these variables by the environment of $\mathcal{A}$ during a transition. This is the basis of the parallel composition of CLHA, which is defined next.

6.2.4 Parallel Composition

The parallel composition of two CLHA $\mathcal{A}_1$ and $\mathcal{A}_2$ formally defines how these two automata interact. An obvious prerequisite for a successful composition is that $\mathcal{A}_1$ and $\mathcal{A}_2$ have no outputs in common, neither variables ($V_1^{out} \cap V_2^{out} = \emptyset$) nor synchronization symbols ($L_1^{out} \cap L_2^{out} = \emptyset$), since we want one distinct writer for each variable and one single source for each synchronization symbol.

The parallel composition of $\mathcal{A}_1$ and $\mathcal{A}_2$ has the following characteristics:

- its (initial) location set is the Cartesian product of the (initial) location sets of $\mathcal{A}_1$ and $\mathcal{A}_2$,
- its set of output variables is the union of the output variables of $\mathcal{A}_1$ and $\mathcal{A}_2$,
- its set of the initial variable evaluations is the intersection of the initial variable evaluations of $\mathcal{A}_1$ and $\mathcal{A}_2$,
- its rates for the continuous output variables are taken from $\mathcal{A}_1$ and $\mathcal{A}_2$,
- its invariants are taken from $\mathcal{A}_1$ and $\mathcal{A}_2$, and
- its set of output synchronization symbols is the union of the respective sets of $\mathcal{A}_1$ and $\mathcal{A}_2$, whereas the union of the input symbols is reduced by each other's output symbols. An effect of the output symbols staying visible for other components is a directed one-to-many communication.

Two edges of $\mathcal{A}_1$ and $\mathcal{A}_2$ can be combined if:

- the edge labeling l_2 of $\mathcal{A}_2$ contains all the synchronization symbols that the edge labeling l_1 of $\mathcal{A}_1$ needs as input from $\mathcal{A}_2$ (i.e., $l_1 \cap L_1^{in} \cap L_2^{out} \subseteq l_2$), and vice versa.

The variable transition relation of the resulting edge combines the elements of the variable transition relations of the edges of $\mathcal{A}_1$ and $\mathcal{A}_2$ which have a common pre-state σ.

In our experience this form of communication using sets of input/output symbols instead of undirected synchronization (often with only one symbol) as used in (timed) automata makes modeling of complex systems much easier.

Definition 6.4 (Parallel composition of CLHA) Given the two CLHA $\mathcal{A}_i = (Q_i, Q_0^i, V_i^{out}, \Sigma_0^i, R_i, I_i, L_i, E_i)$, $i \in \{1, 2\}$, with $V_1^{out} \cap V_2^{out} = \emptyset = L_1^{out} \cap L_2^{out}$, the *parallel composition of* $\mathcal{A}_1$ *and* $\mathcal{A}_2$, denoted by $\mathcal{A}_1 \| \mathcal{A}_2$, is defined as the CLHA $\mathcal{A} = (Q, Q_0, V^{out}, \Sigma_0, R, I, L, E)$ with

- $Q = Q_1 \times Q_2$, $Q_0 = Q_0^1 \times Q_0^2$,
- $V^{out} = V_1^{out} \cup V_2^{out}$,
- $\Sigma_0 = \Sigma_0^1 \cap \Sigma_0^2$,
- for all $q_1 \in Q_1$, $q_2 \in Q_2$, and $v \in V^{out}$,

$$R((q_1, q_2), v) = \begin{cases} R_1(q_1, v) & \text{if } v \in V_1^{out}, \\ R_2(q_2, v) & \text{if } v \in V_2^{out}, \end{cases}$$

- for all $q_1 \in Q_1$ and $q_2 \in Q_2$, $I((q_1, q_2)) = \{\sigma_1 \cup \sigma_2 \,|\, \sigma_1 \in I_1(q_1) \wedge \sigma_2 \in I_2(q_2)\}$,
- $L = L_1 \cup L_2$, $L^{in} = (L_1^{in} \setminus L_2^{out}) \cup (L_2^{in} \setminus L_1^{out})$, $L^{out} = L_1^{out} \cup L_2^{out}$,
- for all $q_1, q_1' \in Q_1$, $q_2, q_2' \in Q_2$, $\rho_1 \subseteq \Sigma \times \Sigma_1^{out}$, $\rho_2 \subseteq \Sigma \times \Sigma_2^{out}$, $l_1 \subseteq L_1$, and $l_2 \subseteq L_2$, $((q_1, q_2), l_1 \cup l_2, \rho_1 \odot \rho_2, (q_1', q_2')) \in E$ if and only if $(q_i, l_i, \rho_i, q_i') \in E_i$, for $i \in \{1, 2\}$, $l_1 \cap L_1^{in} \cap L_2^{out} \subseteq l_2$, and $l_2 \cap L_2^{in} \cap L_1^{out} \subseteq l_1$, where $\rho_1 \odot \rho_2 = \{(\sigma, \sigma^{out}) \in \Sigma \times \Sigma^{out} \,|\, (\sigma, \sigma_1^{out}) \in \rho_1 \wedge (\sigma, \sigma_2^{out}) \in \rho_2\}$.

Definition 6.4 combines two CLHA on the syntactic layer, i.e., by combining their sets of edges. Parallel composition can also be defined semantically, by composing computations. This is formalized in the following definition. As notation we use "$\|$", the same operator as in Definition 6.4; these two can be distinguished from each other by looking at the types of the operands (CLHA or sets of computations).

Definition 6.5 (Parallel composition of computations) Given the two CLHA $\mathcal{A}_1$ and $\mathcal{A}_2$ as in Definition 6.4 and given their sets of computations $c_1 = \mathit{Comp}(\mathcal{A}_1)$ and $c_2 = \mathit{Comp}(\mathcal{A}_2)$, the *parallel composition of c_1 and c_2,* denoted by $c_1 \| c_2$, is defined as follows: The computation $(q_0, \sigma_0) \xrightarrow{l_1} (q_1, \sigma_1) \xrightarrow{l_2} \ldots$ is in $c_1 \| c_2$ if and only if there exist computations $(q_0^1, \sigma_0^1) \xrightarrow{l_1^1} (q_1^1, \sigma_1^1) \xrightarrow{l_2^1} \ldots$ in c_1 and $(q_0^2, \sigma_0^2) \xrightarrow{l_1^2} (q_1^2, \sigma_1^2) \xrightarrow{l_2^2} \ldots$ in c_2, where for all indices i,

- $q_i = (q_i^1, q_i^2)$,
- $\sigma_i = \sigma_i^1 = \sigma_i^2$, and
- $l_i^1 = l_i^2 = l_i \in \mathbf{R}_{>0}$ or $l_i = l_i^1 \cup l_i^2 \wedge l_i^1 \cap L_1^{in} \cap L_2^{out} \subseteq l_i^2 \wedge l_i^2 \cap L_2^{in} \cap L_1^{out} \subseteq l_i^1$

hold.

Since we now have two different ways of defining parallel composition, it is natural to demand that both ways coincide. The next lemma shows that the composition of the computation sets of $\mathcal{A}_1$ and $\mathcal{A}_2$ equals the computation set of $\mathcal{A}_1 \| \mathcal{A}_2$. Thus, the parallel composition of computations is compositional.

Lemma 6.6 (Parallel composition) Given two CLHA $\mathcal{A}_1$ and $\mathcal{A}_2$ as in Definition 6.4, we have

$$Comp(\mathcal{A}_1) \| Comp(\mathcal{A}_2) = Comp(\mathcal{A}_1 \| \mathcal{A}_2) \ .$$

Proof. Let $\mathcal{A}_1$, $\mathcal{A}_2$, and $\mathcal{A}$ be given as in Definition 6.4. Then,

$$(q_0, \sigma_0) \xrightarrow{l_1} (q_1, \sigma_1) \xrightarrow{l_2} \ldots \in Comp(\mathcal{A}_1) \| Comp(\mathcal{A}_2)$$

if and only if (by Definition 6.5)

$$\begin{array}{l}
\exists (q_0^1, \sigma_0^1) \xrightarrow{l_1^1} (q_1^1, \sigma_1^1) \xrightarrow{l_2^1} \ldots \in Comp(\mathcal{A}_1), \\
\quad (q_0^2, \sigma_0^2) \xrightarrow{l_1^2} (q_1^2, \sigma_1^2) \xrightarrow{l_2^2} \ldots \in Comp(\mathcal{A}_2) : \\
\forall i : q_i = (q_i^1, q_i^2) \wedge \sigma_i = \sigma_i^1 = \sigma_i^2 \wedge (l_i^1 = l_i^2 = l_i \in \mathbf{R}_{>0} \vee \\
\qquad (l_i = l_i^1 \cup l_i^2 \wedge l_i^1 \cap L_1^{in} \cap L_2^{out} \subseteq l_i^2 \wedge l_i^2 \cap L_2^{in} \cap L_1^{out} \subseteq l_i^1))
\end{array}$$

if and only if (by Definition 6.3)

$$\begin{array}{l}
\exists (q_0^1, \sigma_0^1) \xrightarrow{l_1^1} (q_1^1, \sigma_1^1) \xrightarrow{l_2^1} \ldots : q_0^1 \in Q_0^1 \wedge \sigma_0^1 \in \Sigma_0^1 \wedge \\
\forall i : (l_{i+1}^1 \subseteq L_1 \wedge \exists \rho_1 \subseteq \Sigma \times \Sigma_1^{out} : (q_i^1, l_{i+1}^1, \rho_1, q_{i+1}^1) \in E_1 \wedge \\
\qquad\qquad\qquad\qquad (\sigma_i^1, \sigma_{i+1}^{1,out}) \in \rho_1) \vee \\
\qquad (l_{i+1}^1 \in \mathbf{R}_{>0} \wedge q_{i+1}^1 = q_i^1 \wedge \sigma_{i+1}^{1,out} = \sigma_i^{1,out} \oplus_{R_1}^{q_i^1} l_{i+1}^1 \wedge \\
\qquad \forall 0 \leq t < l_{i+1}^1 : \sigma_i^{1,cout} \oplus_{R_1}^{q_i^1} t \in I_1(q_i^1)) \wedge \\
\exists (q_0^2, \sigma_0^2) \xrightarrow{l_1^2} (q_1^2, \sigma_1^2) \xrightarrow{l_2^2} \ldots : q_0^2 \in Q_0^2 \wedge \sigma_0^2 \in \Sigma_0^2 \wedge \\
\forall i : (l_{i+1}^2 \subseteq L_2 \wedge \exists \rho_2 \subseteq \Sigma \times \Sigma_2^{out} : (q_i^2, l_{i+1}^2, \rho_2, q_{i+1}^2) \in E_2 \wedge \\
\qquad\qquad\qquad\qquad (\sigma_i^2, \sigma_{i+1}^{2,out}) \in \rho_2) \vee \\
\qquad (l_{i+1}^2 \in \mathbf{R}_{>0} \wedge q_{i+1}^2 = q_i^2 \wedge \sigma_{i+1}^{2,out} = \sigma_i^{2,out} \oplus_{R_2}^{q_i^2} l_{i+1}^2 \wedge \\
\qquad \forall 0 \leq t < l_{i+1}^2 : \sigma_i^{2,cout} \oplus_{R_2}^{q_i^2} t \in I_2(q_i^2)) \wedge \\
\forall i : q_i = (q_i^1, q_i^2) \wedge \sigma_i = \sigma_i^1 = \sigma_i^2 \wedge (l_i^1 = l_i^2 = l_i \in \mathbf{R}_{>0} \vee \\
\qquad (l_i = l_i^1 \cup l_i^2 \wedge l_i^1 \cap L_1^{in} \cap L_2^{out} \subseteq l_i^2 \wedge l_i^2 \cap L_2^{in} \cap L_1^{out} \subseteq l_i^1))
\end{array}$$

$\Leftrightarrow$ (rearrange quantors and variables)

$$\begin{array}{l}
\exists (q_0^1, \sigma_0^1) \xrightarrow{l_1^1} (q_1^1, \sigma_1^1) \xrightarrow{l_2^1} \ldots, (q_0^2, \sigma_0^2) \xrightarrow{l_1^2} (q_1^2, \sigma_1^2) \xrightarrow{l_2^2} \ldots : \\
q_0^1 \in Q_0^1 \wedge q_0^2 \in Q_0^2 \wedge \sigma_0^1 \in \Sigma_0^1 \wedge \sigma_0^2 \in \Sigma_0^2 \wedge \\
\forall i : q_i = (q_i^1, q_i^2) \wedge \sigma_i = \sigma_i^1 = \sigma_i^2 \wedge (l_i^1 = l_i^2 = l_i \in \mathbf{R}_{>0} \vee l_i = l_i^1 \cup l_i^2) \wedge \\
\qquad ((l_{i+1}^1 \subseteq L_1 \wedge l_{i+1}^2 \subseteq L_2 \wedge \exists \rho_1 \subseteq \Sigma \times \Sigma_1^{out}, \rho_2 \subseteq \Sigma \times \Sigma_2^{out} : \\
\qquad\quad (q_i^1, l_{i+1}^1, \rho_1, q_{i+1}^1) \in E_1 \wedge (q_i^2, l_{i+1}^2, \rho_2, q_{i+1}^2) \in E_2 \wedge \\
\qquad\quad l_i^1 \cap L_1^{in} \cap L_2^{out} \subseteq l_i^2 \wedge l_i^2 \cap L_2^{in} \cap L_1^{out} \subseteq l_i^1 \wedge (\sigma_i, \sigma_{i+1}^{out}) \in \rho_1 \odot \rho_2) \\
\qquad\quad \vee (l_{i+1}^1 = l_{i+1}^2 \in \mathbf{R}_{>0} \wedge q_{i+1}^1 = q_i^1 \wedge q_{i+1}^2 = q_i^2 \wedge \\
\qquad\qquad \sigma_{i+1}^{1,out} = \sigma_i^{1,out} \oplus_{R_1}^{q_i^1} l_{i+1}^1 \wedge \sigma_{i+1}^{2,out} = \sigma_i^{2,out} \oplus_{R_2}^{q_i^2} l_{i+1}^2 \wedge \\
\qquad\qquad \forall 0 \leq t < l_{i+1} : \sigma_i^{1,cout} \oplus_{R_1}^{q_i^1} t \in I_1(q_i^1) \wedge \sigma_i^{2,cout} \oplus_{R_2}^{q_i^2} t \in I_2(q_i^2)))
\end{array}$$

$\Leftrightarrow$ (introduce/remove Q_0, Σ_0, R, I, E, using Definition 6.4)

$$
\begin{array}{l}
\exists (q_0^1, \sigma_0^1) \xrightarrow{l_1^1} (q_1^1, \sigma_1^1) \xrightarrow{l_2^1} \ldots, (q_0^2, \sigma_0^2) \xrightarrow{l_1^2} (q_1^2, \sigma_1^2) \xrightarrow{l_2^2} \ldots : \\
(q_0^1, q_0^2) \in Q_0 \wedge \sigma_0^1 \cup \sigma_0^2 \in \Sigma_0 \wedge \\
\forall i : q_i = (q_i^1, q_i^2) \wedge \sigma_i = \sigma_i^1 = \sigma_i^2 \wedge (l_i^1 = l_i^2 = l_i \in \mathbf{R}_{>0} \vee l_i = l_i^1 \cup l_i^2) \wedge \\
\quad ((l_{i+1}^1 \subseteq L_1 \wedge l_{i+1}^2 \subseteq L_2 \wedge \exists \rho_1 \subseteq \Sigma \times \Sigma_1^{out}, \rho_2 \subseteq \Sigma \times \Sigma_2^{out} : \\
\qquad ((q_i^1, q_i^2), l_{i+1}^1 \cup l_{i+1}^2, \rho_1 \odot \rho_2, (q_{i+1}^1, q_{i+1}^2)) \in E \wedge \\
\qquad (\sigma_i, \sigma_{i+1}^{out}) \in \rho_1 \odot \rho_2) \\
\qquad \vee (l_{i+1}^1 = l_{i+1}^2 \in \mathbf{R}_{>0} \wedge (q_{i+1}^1, q_{i+1}^2) = (q_i^1, q_i^2) \wedge \\
\qquad\quad \sigma_{i+1}^{out} = \sigma_i^{out} \oplus_R^{(q_i^1, q_i^2)} l_{i+1} \wedge \\
\qquad\quad \forall 0 \leq t < l_{i+1} : \sigma_i^{cout} \oplus_R^{(q_i^1, q_i^2)} t \in I((q_i^1, q_i^2))))
\end{array}
$$

$\Leftrightarrow$ (remove/introduce q_i^1, q_i^2, σ_i^1, σ_i^2, l_i^1, l_i^2)

$$
\begin{array}{l}
q_0 \in Q_0 \wedge \sigma_0 \in \Sigma_0 \wedge \forall i : \\
(l_{i+1} \subseteq L \wedge \exists \rho \subseteq \Sigma \times \Sigma^{out} : (q_i, l_{i+1}, \rho, q_{i+1}) \in E \wedge (\sigma_i, \sigma_{i+1}^{out}) \in \rho) \\
\vee (l_{i+1} \in \mathbf{R}_{>0} \wedge q_{i+1} = q_i \wedge \sigma_{i+1}^{out} = \sigma_i^{out} \oplus_R^{q_i} l_{i+1} \\
\quad \wedge \forall 0 \leq t < l_{i+1} : \sigma_i^{cout} \oplus_R^{q_i} t \in I(q_i))
\end{array}
$$

$\Leftrightarrow$ (Definition 6.3)

$$(q_0, \sigma_0) \xrightarrow{l_1} (q_1, \sigma_1) \xrightarrow{l_2} \ldots \in \mathit{Comp}(\mathcal{A})$$

$\Leftrightarrow$

$$(q_0, \sigma_0) \xrightarrow{l_1} (q_1, \sigma_1) \xrightarrow{l_2} \ldots \in \mathit{Comp}(\mathcal{A}_1 \| \mathcal{A}_2) \ .$$

6.2.5 Trace Semantics

While the notion of computations is well-suited to describe the way a single CLHA operates, it is not wise to use computations for specifying the interaction of several automata. Computations contain information about the changes of locations, which should be considered as internal and not observable from outside.

Therefore, we remove the locations from the computations and use the resulting sequences, called *traces,* for our compositional CLHA semantics.

Definition 6.7 (Trace of a CLHA) Given a CLHA $\mathcal{A}$ as in Definition 6.2, a sequence

$$\sigma_0 \xrightarrow{l_1} \sigma_1 \xrightarrow{l_2} \sigma_2 \xrightarrow{l_3} \ldots$$

is called *trace of* $\mathcal{A}$ if and only if there exist $q_0, q_1, q_2, \ldots \in Q$ such that

$$(q_0, \sigma_0) \xrightarrow{l_1} (q_1, \sigma_1) \xrightarrow{l_2} (q_2, \sigma_2) \xrightarrow{l_3} \ldots$$

is a computation of $\mathcal{A}$.

Now we define the semantics of a CLHA by its traces:

Definition 6.8 (Semantics of a CLHA) The *semantics of a CLHA* $\mathcal{A}$, denoted by $[\![\mathcal{A}]\!]$, is the set of all traces of $\mathcal{A}$.

Definition 6.5 introduces the parallel composition of computations. Analogously, we can define the parallel composition of traces (we overload the "$\|$" operator one more).

Definition 6.9 (Parallel composition of traces) Given the two CLHA $\mathcal{A}_1$ and $\mathcal{A}_2$ as in Definition 6.4 and given their sets of traces $t_1 = [\![\mathcal{A}_1]\!]$ and $t_2 = [\![\mathcal{A}_2]\!]$, the *parallel composition of* t_1 *and* t_2, denoted by $t_1 \| t_2$, is defined as follows: The trace $\sigma_0 \xrightarrow{l_1} \sigma_1 \xrightarrow{l_2} \ldots$ is in $t_1 \| t_2$ if and only if there exist traces $\sigma_0^1 \xrightarrow{l_1^1} \sigma_1^1 \xrightarrow{l_2^1} \ldots$ in t_1 and $\sigma_0^2 \xrightarrow{l_1^2} \sigma_1^2 \xrightarrow{l_2^2} \ldots$ in t_2, where for all indices i,

- $\sigma_i = \sigma_i^1 = \sigma_i^2$ and
- $l_i^1 = l_i^2 = l_i \in \mathbf{R}_{>0}$ or $l_i = l_i^1 \cup l_i^2 \wedge l_i^1 \cap L_1^{in} \cap L_2^{out} \subseteq l_i^2 \wedge l_i^2 \cap L_2^{in} \cap L_1^{out} \subseteq l_i^1$

hold.

Similarly to Lemma 6.6 for computations we can show that the parallel composition of traces is compositional:

Lemma 6.10 (Parallel composition) Given two CLHA $\mathcal{A}_1$ and $\mathcal{A}_2$ as in Definition 6.4, we have

$$[\![\mathcal{A}_1]\!] \| [\![\mathcal{A}_2]\!] = [\![\mathcal{A}_1 \| \mathcal{A}_2]\!] \ .$$

Proof. Let $\mathcal{A}_1$, $\mathcal{A}_2$, and $\mathcal{A}$ be given as in Definition 6.4. Then,

$$\sigma_0 \xrightarrow{l_1} \sigma_1 \xrightarrow{l_2} \ldots \in [\![\mathcal{A}_1]\!] \| [\![\mathcal{A}_2]\!]$$

if and only if (by Definition 6.9)

$$\begin{aligned}
&\exists \sigma_0^1 \xrightarrow{l_1^1} \sigma_1^1 \xrightarrow{l_2^1} \ldots \in [\![\mathcal{A}_1]\!], \sigma_0^2 \xrightarrow{l_1^2} \sigma_1^2 \xrightarrow{l_2^2} \ldots \in [\![\mathcal{A}_2]\!] : \\
&\forall i : \sigma_i = \sigma_i^1 = \sigma_i^2 \wedge (l_i^1 = l_i^2 = l_i \in \mathbf{R}_{>0} \vee \\
&\quad l_i = l_i^1 \cup l_i^2 \wedge l_i^1 \cap L_1^{in} \cap L_2^{out} \subseteq l_i^2 \wedge l_i^2 \cap L_2^{in} \cap L_1^{out} \subseteq l_i^1)
\end{aligned}$$

if and only if (by Definition 6.8 and Definition 6.7)

$$\begin{aligned}
&\exists q_0, q_1, \ldots \in Q : \\
&\exists (q_0^1, \sigma_0^1) \xrightarrow{l_1^1} (q_0^1, \sigma_1^1) \xrightarrow{l_2^1} \ldots \in Comp(\mathcal{A}_1), \\
&\ (q_0^2, \sigma_0^2) \xrightarrow{l_1^2} (q_0^2, \sigma_1^2) \xrightarrow{l_2^2} \ldots \in Comp(\mathcal{A}_2) : \\
&\forall i : q_i = (q_i^1, q_i^2) \wedge \sigma_i = \sigma_i^1 = \sigma_i^2 \wedge (l_i^1 = l_i^2 = l_i \in \mathbf{R}_{>0} \vee \\
&\quad l_i = l_i^1 \cup l_i^2 \wedge l_i^1 \cap L_1^{in} \cap L_2^{out} \subseteq l_i^2 \wedge l_i^2 \cap L_2^{in} \cap L_1^{out} \subseteq l_i^1)
\end{aligned}$$

if and only if (by Definition 6.5)

$$\begin{aligned}&\exists q_0, q_1, \ldots \in Q : \\ &(q_0, \sigma_0) \xrightarrow{l_1} (q_1, \sigma_1) \xrightarrow{l_2} \ldots \in \mathit{Comp}(\mathcal{A}_1) \| \mathit{Comp}(\mathcal{A}_2)\end{aligned}$$

if and only if (by Lemma 6.6)

$$\begin{aligned}&\exists q_0, q_1, \ldots \in Q : \\ &(q_0, \sigma_0) \xrightarrow{l_1} (q_1, \sigma_1) \xrightarrow{l_2} \ldots \in \mathit{Comp}(\mathcal{A}_1 \| \mathcal{A}_2)\end{aligned}$$

if and only if (by Definition 6.7)

$$\sigma_0 \xrightarrow{l_1} \sigma_1 \xrightarrow{l_2} \ldots \in [\![\mathcal{A}_1 \| \mathcal{A}_2]\!] \ .$$

6.3 Propositional Linear Temporal Logic

Using the trace semantics of Definition 6.8, we are able to describe the exact behavior of a CLHA. However, in practice a much simpler description language is sufficient, since one is usually interested in an abstract and finite way to describe the system behavior. One possibility is the use of *temporal logic*. This section introduces *propositional linear temporal logic,* which describes sequences of sets of propositions. Linear (time) temporal logic was first introduced by Amir Pnueli in [Pnu77].

6.3.1 Syntax

Definition 6.11 (PLTL syntax) Given a countable set of propositions, the set of *Propositional Linear Temporal Logic* (PLTL) formulae is given by the following BNF notation:

$$\begin{aligned}\langle \mathit{formula} \rangle ::= \ & \langle \mathit{proposition} \rangle \ \mid \ \langle \mathit{formula} \rangle \wedge \langle \mathit{formula} \rangle \ \mid \ \neg \langle \mathit{formula} \rangle \\ & \langle \mathit{formula} \rangle \, \mathcal{U} \, \langle \mathit{formula} \rangle \ \mid \ \bigcirc \langle \mathit{formula} \rangle\end{aligned}$$

Other Boolean operators are defined as abbreviations in the usual way: $\varphi \vee \psi \equiv \neg((\neg\varphi) \wedge (\neg\psi))$, $\varphi \Rightarrow \psi \equiv (\neg\varphi) \vee \psi$, $\varphi \Leftrightarrow \psi \equiv (\varphi \Rightarrow \psi) \wedge (\psi \Rightarrow \varphi)$. The Boolean constant *true* can be abbreviated as $p \vee (\neg p)$ for some proposition p, and $\mathit{false} \equiv \neg \mathit{true}$.

In addition to the temporal operators $\varphi \mathcal{U} \psi$ ("φ until ψ") and $\bigcirc \varphi$ ("next φ"), we introduce two temporal operators as abbreviations: $\Diamond \varphi \equiv \mathit{true} \, \mathcal{U} \, \varphi$ ("eventually φ"), $\Box \varphi \equiv \neg(\Diamond(\neg\varphi))$ ("always φ").

The temporal operators $\mathcal{U}$, $\bigcirc$, $\Diamond$, and $\Box$ have the highest binding power, followed by (in decreasing order) $\neg$, $\wedge$, $\vee$, $\Rightarrow$, and $\Leftrightarrow$.

6.3.2 Semantics

The semantics of a PLTL formula is given by its interpretation over an infinite sequence of sets of propositions.

Notation: Given a sequence $P = P_0 P_1 \ldots$ and $k \in \mathbf{N}$, we denote by P^k the sequence P without its first k elements, i.e., $P_k P_{k+1} \ldots$.

Definition 6.12 (PLTL semantics) Given an infinite sequence of sets of propositions $P = P_0 P_1 \ldots$, the *validity of a PLTL formula φ over P,* denoted as $P \models \varphi$, is defined inductively over the structure of PLTL formulae as follows:

$$
\begin{array}{lll}
P \models p & \text{if and only if} & p \in P_0 \\
P \models \varphi \wedge \psi & \text{if and only if} & P \models \varphi \text{ and } P \models \psi \\
P \models \neg\varphi & \text{if and only if} & \text{not } P \models \varphi \\
P \models \varphi \,\mathcal{U}\, \psi & \text{if and only if} & \exists k \in \mathbf{N} : P^k \models \psi \text{ and} \\
 & & \quad \forall 0 \leq i < k : P^i \models \varphi \\
P \models \bigcirc \varphi & \text{if and only if} & P^1 \models \varphi
\end{array}
$$

Lemma 6.13 (PLTL semantics for $\Diamond$ and $\Box$) By Definition 6.12, we have the following semantics for the $\Diamond$ and $\Box$ operators:

$$
\begin{array}{lll}
P \models \Diamond \varphi & \text{if and only if} & \exists k \in \mathbf{N} : P^k \models \varphi \\
P \models \Box \varphi & \text{if and only if} & \forall k \in \mathbf{N} : P^k \models \varphi
\end{array}
$$

6.3.3 PLTL for CLHA

We want to use PLTL formulae to describe the behavior of CLHA. Since the PLTL semantics operates on sequences of sets of propositions, we need to transform the semantical concept for CLHA (traces as introduced in Definition 6.7) into such sequences. This transformation depends on the kind of information we want to describe in the PLTL formulae. E.g., if we want to specify the changes of synchronization symbols along the run

$$\sigma_0 \xrightarrow{l_1} \sigma_1 \xrightarrow{l_2} \sigma_2 \xrightarrow{l_3} \ldots$$

by a PLTL formula φ, we can use the set of synchronization symbols L as the set of propositions and use the sequence of synchronization symbol sets

$$l_{i_1} l_{i_2} l_{i_3} \ldots$$

(with $i_1 i_2 i_3 \ldots$ being the ordered sequence of indices i with $l_i \subseteq L$, thus leaving out all continuous computation steps) to check the validity of φ.

If we also need to reason about variable evaluations in PLTL formulae, it is necessary to choose abstractions of sets of evaluations like "$x > 0$" as propositions. If each single variable evaluation was encoded as a proposition,

many infinite sets of evaluations (like "$x > 0$" with $type(x) = \mathbf{R}$) could not be expressed in a PLTL formula, since one formula can only contain finitely many propositions.

In the following we assume that we have a function *pseq* which transforms a CLHA run into a sequence of sets of propositions.

Definition 6.14 (PLTL for CLHA) Given a CLHA $\mathcal{A}$ and a PLTL formula φ, the *validity of* φ *over* $\mathcal{A}$, denoted as $\mathcal{A} \models \varphi$, is defined as

$$\mathcal{A} \models \varphi \quad \text{if and only if} \quad \forall t \in [\![\mathcal{A}]\!] : pseq(t) \models \varphi \ .$$

Using the CLHA formalism presented above, components of a linear hybrid system can be specified in a unified framework. To formally verify a component, it can be translated into the input language of a model-checking tool that fits best to the features used in the component, e.g., SMV if the component only uses discrete variables, and KRONOS or UPPAAL if clocks are involved.

6.4 Example

We illustrate the CLHA formalism by modeling one of the product storage tanks of the multi-product batch plant introduced in Chapter 3. This tank stores a liquid product and can be filled and drained via inlet and outlet pipes. The maximal capacity of the tank is 3.0 volume units. Filling the tank increases the volume of its content linearly by 0.05 volume units per second, while draining the tank changes the volume linearly by -0.10 volume units per second. These numbers are approximations of measurements on the actual plant given in the plant description [BKSL00].

6.4.1 CLHA Model

Our CLHA model uses five discrete locations, *idle* (meaning that no in- or outflow occurs), *fill* (filling the tank), *drain* (draining the tank), *both* (filling and draining happens simultaneously), and *error* (after an error has occurred). Four self-explanatory input symbols are used to change the location: *start_fill*, *stop_fill*, *start_drain*, and *stop_drain*. One continuous variable *vol* contains the current liquid volume of the tank.

There are two possibilities for the occurrence of an error: The tank overflows if $vol \geq 3.0$, and if $vol \leq 0.0$, the tank has run empty, which we consider as an "underflow" error. If an error occurs, the output symbol *over*, resp., *under* is sent, and the location *error* is entered. If an input symbol arrives "just in time" before an error occurs (*start_drain* or *stop_fill* when $vol = 3.0$ in state *fill*, *start_fill* or *stop_drain* when $vol = 0.0$ in state *drain*, *stop_drain* or *stop_fill* when $vol = 0.0$ in state *both*), the CLHA model chooses nondeterministically if the *error* location is entered or not.

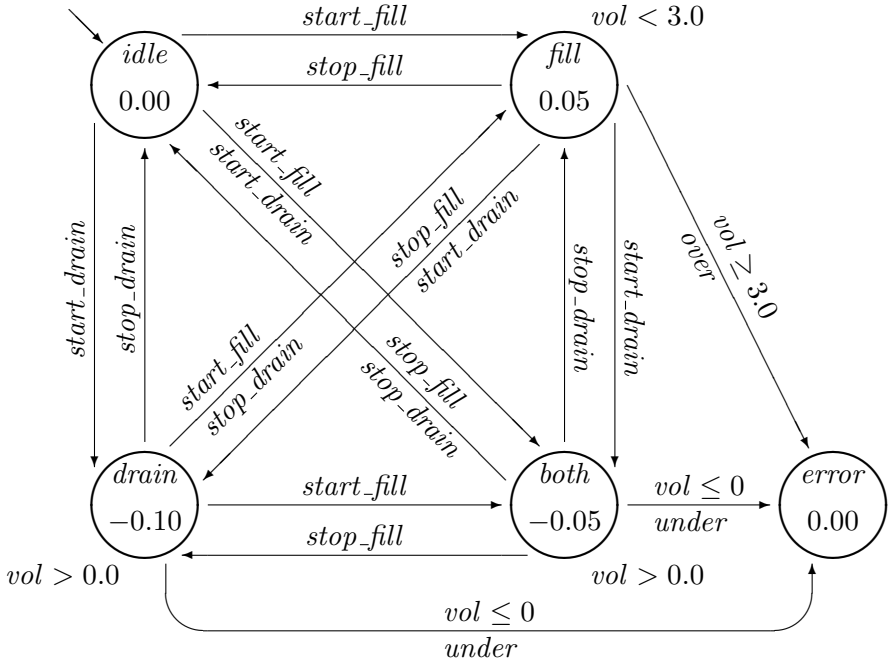


Figure 6.1: CLHA tank model

Figure 6.1 illustrates the CLHA model of the tank. The locations are marked with the flow rates of *vol*, and locations with nonzero rates are labeled with invariants ($vol > 0.0$ and $vol < 3.0$) to force the discrete transition to the *error* location if *vol* leaves its allowed range. Note that in Figure 6.1 and all following CLHA figures in this chapter, we do not display self-loop transitions from the set of edges E which do not change any variables, for the sake of readability.

The system uses two real-valued variables (the variable t will be used in a second CLHA introduced later): $V = \{vol, t\}$, $type(vol) = type(t) = \mathbf{R}$. The model of the tank is given as the CLHA $\mathcal{T} = (Q, \{idle\}, \{vol\}, \Sigma_0, R, I, L, E)$, where

- $Q = \{idle, fill, drain, both, error\}$, $\Sigma_0 = \{\sigma \in \Sigma \,|\, \sigma(vol) = 3.0\}$,
- $R((q, vol)) = \begin{cases} 0.00, \text{ if } q \in \{idle, error\} \\ 0.05, \text{ if } q = fill \\ -0.10, \text{ if } q = drain \\ -0.05, \text{ if } q = both \end{cases}$,
- $I(q) = \begin{cases} \Sigma^{cout}, \text{ if } q \in \{idle, error\} \\ \{\sigma \in \Sigma^{cout} \,|\, \sigma(vol) < 3.0\}, \text{ if } q = fill \\ \{\sigma \in \Sigma^{cout} \,|\, \sigma(vol) > 0.0\}, \text{ if } q \in \{drain, both\} \end{cases}$, and
- $L = L^{in} \cup L^{out}$, with $L^{in} = \{start\_fill, stop\_fill, start\_drain, stop\_drain\}$

and $L^{out} = \{over, under\}$.

The set of edges E is defined as follows: for all $l \subseteq L$, $\rho = (\sigma, \sigma^{out})$ with $\sigma \in \Sigma$, and $q, q' \in Q$,

- $(idle, l, \rho, q') \in E$ if and only if $l \subseteq L^{in}$ and one of the following cases holds:
  - $start\_fill \notin l \wedge start\_drain \notin l \wedge q' = idle$
  - $start\_fill \in l \wedge start\_drain \notin l \wedge q' = fill$
  - $start\_fill \notin l \wedge start\_drain \in l \wedge q' = drain$
  - $start\_fill \in l \wedge start\_drain \in l \wedge q' = both$
- $(fill, l, \rho, q') \in E$ if and only if $l \subseteq L^{in}$ and one of the following cases holds:
  - $stop\_fill \in l \wedge start\_drain \notin l \wedge q' = idle$
  - $stop\_fill \notin l \wedge start\_drain \notin l \wedge q' = fill$
  - $stop\_fill \in l \wedge start\_drain \in l \wedge q' = drain$
  - $stop\_fill \notin l \wedge start\_drain \in l \wedge q' = both$
- $(drain, l, \rho, q') \in E$ if and only if $l \subseteq L^{in}$ and one of the following cases holds:
  - $stop\_drain \in l \wedge start\_fill \notin l \wedge q' = idle$
  - $stop\_drain \in l \wedge start\_fill \in l \wedge q' = fill$
  - $stop\_drain \notin l \wedge start\_fill \notin l \wedge q' = drain$
  - $stop\_drain \notin l \wedge start\_fill \in l \wedge q' = both$
- $(both, l, \rho, q') \in E$ if and only if $l \subseteq L^{in}$ and one of the following cases holds:
  - $stop\_drain \in l \wedge stop\_fill \in l \wedge q' = idle$
  - $stop\_drain \in l \wedge stop\_fill \notin l \wedge q' = fill$
  - $stop\_drain \notin l \wedge stop\_fill \in l \wedge q' = drain$
  - $stop\_drain \notin l \wedge stop\_fill \notin l \wedge q' = both$
- $(error, l, \rho, error) \in E$
- $(q, l, \rho, error) \in E$ if and only if $q \in \{drain, both\}$, $under \in l$, $over \notin l$, and $\sigma(vol) \leq 0.0$
- $(fill, l, \rho, error) \in E$ if and only if $over \in l$, $under \notin l$, and $\sigma(vol) \geq 3.0$

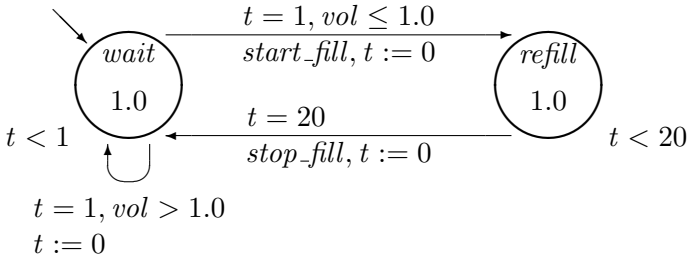


Figure 6.2: CLHA controller model

Now that we have a complete CLHA model of the tank, we add a simple controller to the tank's environment which controls the refilling of the tank in case it impends to run empty. The refilling process always adds a fixed amount of 1.0 units to the tank (which corresponds to draining the contents of one reactor containing the product into the tank). This controller is modeled as follows: every second, the controller checks the volume of the tank's content; if it is below 1.0 units, *start_fill* is sent, and a clock t is started. After 20 seconds ($t = 20$), *stop_fill* is sent. Note that this controller is not a part of the original plant, which has a more complex refilling mechanism based on schedules for product demands and raw material deliveries. Figure 6.2 shows the CLHA model of the controller. The model of the controller is given as the CLHA $\mathcal{C} = (Q^{\mathcal{C}}, \{wait\}, \{t\}, \Sigma_0^{\mathcal{C}}, R^{\mathcal{C}}, I^{\mathcal{C}}, L^{\mathcal{C}}, E^{\mathcal{C}})$, where

- $Q^{\mathcal{C}} = \{wait, refill\}$, $\Sigma_0^{\mathcal{C}} = \{\sigma \in \Sigma \mid \sigma(t) = 0.0\}$,
- $R^{\mathcal{C}}((wait, t)) = R^{\mathcal{C}}((refill, t)) = 1.0$,
- $I^{\mathcal{C}}(wait) = \{\sigma \in \Sigma^{cout} \mid \sigma(t) < 1\}$, $I^{\mathcal{C}}(refill) = \{\sigma \in \Sigma^{cout} \mid \sigma(t) < 20\}$, and
- $L^{\mathcal{C}} = L_{\mathcal{C}}^{in} \cup L_{\mathcal{C}}^{out}$, with $L_{\mathcal{C}}^{in} = \emptyset$ and $L_{\mathcal{C}}^{out} = \{start\_fill, stop\_fill\}$.

The set of edges is defined as follows: for all $l \subseteq L^{\mathcal{C}}$, $\rho = (\sigma, \sigma')$ with $\sigma \in \Sigma$, $\sigma' \in \Sigma^{out}$, and $q, q' \in Q^{\mathcal{C}}$, we have $(q, l, \rho, q') \in E$ if and only if one of the following cases holds:

- $q = q' = wait \wedge l = \emptyset \wedge \sigma(vol) > 1.0 \wedge \sigma'(t) = \sigma(t)$
- $q = q' = wait \wedge l = \emptyset \wedge \sigma(vol) > 1.0 \wedge \sigma(t) = 1 \wedge \sigma'(t) = 0$
- $q = wait \wedge q' = refill \wedge l = \{start\_fill\} \wedge \sigma(vol) \leq 1.0 \wedge \sigma(t) = 1 \wedge \sigma'(t) = 0$
- $q = q' = refill \wedge l = \emptyset \wedge \sigma'(t) = \sigma(t)$
- $q = refill \wedge q' = wait \wedge l = \{stop\_fill\} \wedge \sigma(t) = 20 \wedge \sigma'(t) = 0$

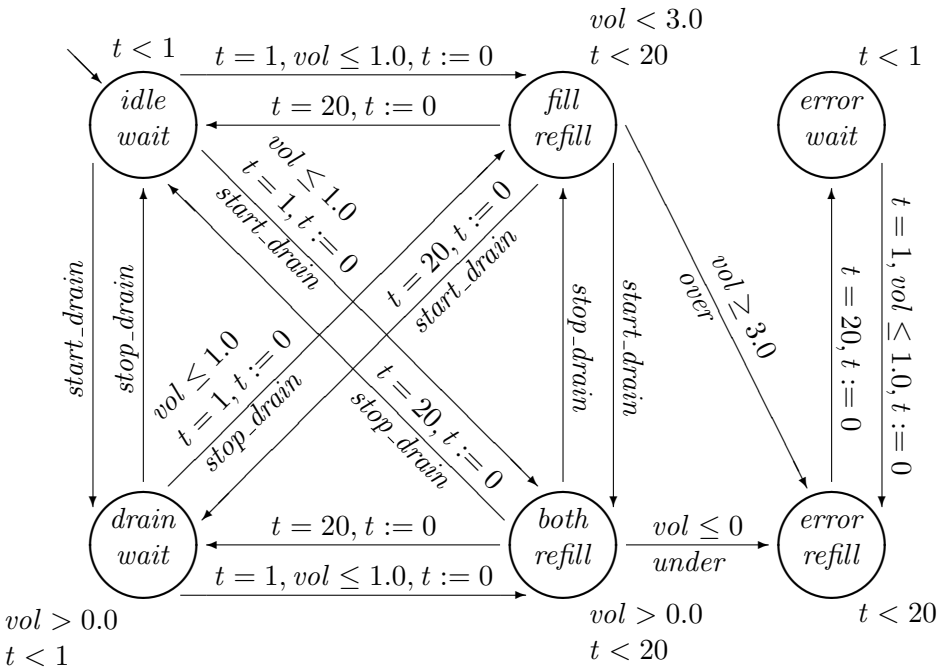


Figure 6.3: CLHA model of $\mathcal{T} \| \mathcal{C}$

The product automaton $\mathcal{T} \| \mathcal{C}$ is shown in Figure 6.3. Product locations which are obviously not reachable from the initial location, e.g., (*idle*, *refill*) are not shown, as well as transitions which are never enabled. For readability we also left out the rates for *vol* and t, the self-loops labeled "$t = 1$, $vol > 1.0$, $t := 0$" at all *wait*-states, and all occurrences of *start_fill* and *stop_fill*.

Figure 6.4 shows how *vol* evolves over time in a scenario where the tank is drained at time 2 (*start_drain* is received) for a duration of 23 seconds. After 20 seconds of draining (i.e., at time 22), *vol* falls below 1.0 units, and the controller sends *start_fill*, which increments the change rate of *vol* from -0.10 to -0.05 units per second. 3 seconds later, the draining stops (*stop_drain*), and the level in the tank rises. After 20 seconds (at time 42) the filling is stopped (*stop_fill*), and *vol* has reached 1.7 units.

6.4.2 Verification With HyTech

CLHA can easily be translated into the input language of the model-checking tool HyTech [HHWT97]. Figure 6.5 shows the HyTech code for the controller $\mathcal{C}$. Note that we multiply the volumes and rates by 100 to get integer values. If we add another automaton to the model which sends *start_drain* at time 2 and *stop_drain* at time 25, a reachability analysis with HyTech computes the following regions, which correspond to the five line segments drawn in Figure 6.4 (g denotes the global time):

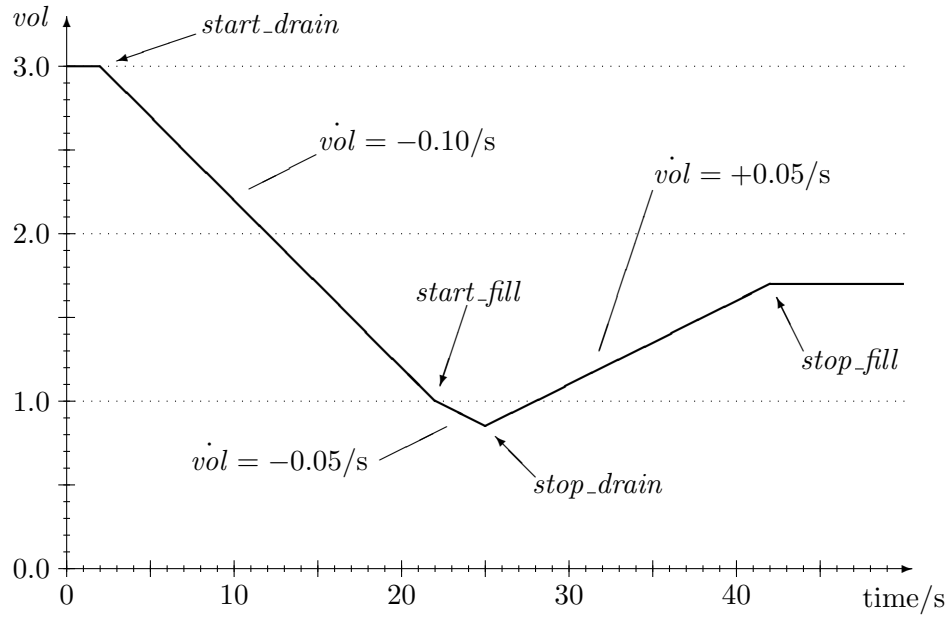


Figure 6.4: A run of the system $\mathcal{T} \| \mathcal{C}$

```
automaton controller
synclabs: start_fill, stop_fill;
initially waiting & t=0;

loc waiting: while t<=1 wait {dt=1}
   when t=1 & vol<=100 sync start_fill do {t'=0} goto refill;
   when t=1 & vol> 100                do {t'=0} goto waiting;

loc refill: while t<=20 wait {dt=1}
   when t=20 sync stop_fill do {t'=0} goto waiting;

end
```

Figure 6.5: HyTech code for the controller $\mathcal{C}$

1. $vol = 300 \wedge g \leq 2$
2. $100 \leq vol \leq 300 \wedge vol + 10g = 320$
3. $85 \leq vol \leq 100 \wedge vol + 5g = 210$
4. $85 \leq vol \leq 170 \wedge 5g = vol + 40$
5. $vol = 170 \wedge g \geq 42$

We can also use HyTech to determine under which conditions the tank can run empty. E.g., if fixed batches of 1.0 units are repeatedly drained from the tank, a delay of at least 10 seconds between two consecutive drainings is needed to prevent an underflow. If the batch size is increased to 2.0 units, an underflow is inevitable, no matter how long the delay is chosen.

Chapter 7

Conclusions

This last chapter summarizes the main topics and results of this thesis and points out future work.

7.1 Summary

This thesis presented methods for the formal verification of control software in the field of chemical process engineering. We focussed on software running on programmable logic controllers (PLCs), in particular, programs written in the standardized programming language sequential function charts (SFC). The verification methods have been illustrated by checking properties of the control software of two chemical batch plants.

The formal verification task can be described as follows: Given some system (in our case, a chemical plant) and a list of requirements, prove formally that the system satisfies these requirements. To perform this task, several steps need to be carried out, all addressed in this thesis:

- The system needs to be modeled in a framework which allows formal reasoning. This framework should have a level of abstraction that makes formal verification with respect to complexity feasible.
- The requirements also need to be formulated in a framework appropriate for the application of formal methods, e.g., temporal logic.
- The model of the system is checked against the formal requirements, e.g., by model checking and/or deductive reasoning.

The feasibility and efficiency of the last step depends heavily on the frameworks and abstraction levels chosen in the first two steps.

A prerequisite for formal verification methods is a clear semantics of the programming languages involved. Therefore, we explained in Chapter 2 the particularities of PLC program execution, and we defined a formal operational semantics for the SFC language.

Chapter 3 described the two case studies considered in this thesis, one implementing a mixing and separation process (experimental batch plant) and the other implementing the concurrent production of two different liquid products (multi-product batch plant).

Two different verification methods have been presented. Both methods first decompose the system into its constituent parts (called modules), i.e., the physical components (e.g., tanks and valves) and the software components (PLC programs written in the SFC language).

The modular approach of Chapter 4 introduced a discrete model for each of the modules which was translated by a compiler into the input language of the model-checking tool SMV. For each property that needed to be proven, only a small subset of all modules sufficient for satisfying the property was translated to SMV and model-checked. This procedure reduced the state space of the model-checking process significantly, and the verification results were computed much faster than in a setting where all modules are included in the verification of each property. In our case, it is even questionable if the full product of all modules can be model-checked at all because of the state-explosion problem. The verification of the experimental batch plant revealed two minor errors in the control programs, and another flaw was found by manual inspection of a control program.

The compositional approach presented in Chapter 5 introduced a discrete model for each of the control programs. Local specifications were given for each module. Then each of the models of a control program was checked against its local specifications by the model checker of the SAL system. The specifications of the modules for physical parts of the system were not verified by model checking; we assumed that these specifications describe the modules' behaviors, and a formal proof of that is not possible anyway, since we have no formal models for devices like tanks or valves. Once we had established the local specifications, these were combined by deduction to gain the global safety properties we are interested in. This approach avoided the state explosion problem by applying model checking to each separate module one by one, thus keeping the state space as small as possible.

These two approaches only used discrete models. Chapter 6 introduced communicating linear hybrid automata (CLHA) as a modeling framework for hybrid systems, which contain discrete as well as continuous components. An example shows a CLHA model of a part of the multi-product batch plant, and we use the hybrid model checker HyTech to compute properties of this model.

7.2 Lessons Learned

For each of the main topics discussed in this thesis, we give a short summary of the observations made.

7.2.1 Programmable Logic Controllers

For the formal treatment of PLC software it is essential to have a clear and unambiguous semantics for PLC programs. We focussed on the programming language sequential function charts (SFC). As examinations of the IEC 61131-3 standard [IEC98] and various PLC programming environments [Bau03, BHLL04] have shown, the semantics of SFCs is far from obvious and implemented differently in different tools, which aggravates the interoperability of PLC software. The research during the construction of the semantics for SFCs revealed details of the program execution which would not have become apparent if we had been content with an intuitive semantics as suggested in the standard.

In our experience most ambiguities play a minor role in the everyday programming of PLCs, but we cannot rule out that an ambiguity will cause problems when the PLC behavior deviates from what the programmer had in mind. Therefore, programmers should be aware of these problems, and some effort should be put into the standard to clarify semantic ambiguities.

7.2.2 Abstraction and Modeling

The abstraction level of models for parts of a plant depends on the level of detail we need for the verification process. In our examples, discrete abstractions of the continuous processes have been chosen, since these have shown to be able to express the properties we were interested in. Since the state explosion problem is always imminent in state-based verification, one should always aim for a safe abstraction which is as coarse as possible, without losing essential information needed to prove the properties.

It is important to notice that the models of hardware parts, irrespective of their level of detail, are often just representations of the expected plant behavior, and therefore, proven properties about these models alone merely state that we have chosen our models in accordance with their expected properties. If we strive for proving reliable properties of the plant, we need verification results obtained in combination with models of the plant that have been created on a formal basis, e.g., PLC program semantics.

Note that a clear distinction just between hardware and software can be difficult, and finer differentiations are possible, e.g., in the hardware of the plant, the PLC control hardware, products processed by the plant, communication infrastructure, human operators, etc.

In our examples, the PLC software is given as SFCs. Since these have a discrete control structure, we keep this structure also in the model. If the SFCs do not have nested structures and only use the N, S, P0, P1 and R qualifiers, there is no need to model the complex action qualifier treatment of SFCs, and a simple transition system is sufficient to represent the SFCs' behavior on an abstract level.

It is not mandatory that all modules of the plant are modeled in the same framework; only if two modules need to be composed for model checking, they should be given in the same language. It is even possible to provide several models for one module, e.g., with different levels of abstractions or the input language of different model checkers.

7.2.3 Modular Verification

The modular verification method presented in Chapter 4 showed that a careful selection of the modules composed for model checking can significantly reduce the time and memory consumption of the verification process. In one case, the verification time has been reduced from 23 hours to a few seconds by splitting the property into several small properties, each of which only requiring the composition of two modules instead of 20 for the original property.

This example shows that there is a lot of potential in optimizing how a model checker is used, even after a certain model has been built.

7.2.4 Compositional Verification

The compositional verification method reduces the algorithmic verification to model checking each model one by one, and thus, the complexity is kept small. Furthermore, different modeling paradigms and different model checkers can be used in the local verification step. More work lies in the construction of local properties and the deductive reasoning undertaken to combine the local properties into global properties.

Consequently, it seems to be appropriate to use the modular or even a global approach for the verification of small systems, avoiding the overhead work of specifying and proving local properties. For systems of high complexity, compositional verification is mandatory, since noncompositional methods are bound to fail because of excessive time and memory requirements. Nevertheless it is important to realize that a compositional approach requires much more efforts in the modeling and specification phase that the modular or global approaches. In our experience, often several iterations of specification and deductive verification are necessary, as local specifications may turn out to be too weak to establish global properties. In that case, the local specification has to be rephrased and proven correct by local model checking. Usually this rephrasing is a refinement step, and in this case, the iteration will eventually terminate, since in the worst case, one ends up with a local specification which describes the full semantics of the module.

Note that the compositional approach does not require a formal model for each module; it is sufficient to provide local specifications. This promotes the use of libraries of modules of which local properties already have been proven. Furthermore, specifications of hardware parts can be given directly,

without having to introduce an algorithmically checkable model on the basis of expected behavior.

7.3 Future Work

The PLC programming languages standardized in IEC 61131-3 are rather new in the field of software verification and have many interesting aspects from the viewpoint of formal methods. Further investigations of their semantics are necessary to advance formal methods for PLC software. Especially the combined use of different languages (e.g., ladder diagrams and function blocks as guards in SFCs) pose challenges for the integration of their semantics.

Further work in the modular approach lies in automated techniques to determine which modules have to be composed to prove a given property, and in other methods for reducing time and memory consumption (like the splitting of a property for several verification steps), and in using other levels of abstraction for the models.

The deductive reasoning in the compositional approach calls for tool support. Theorem proving systems like PVS (Prototype Verification System) [ORS92] can be used to execute and organize the proofs, and their builtin heuristics can even automatize some proof steps. Future work lies in the construction of proof strategies for these tools, which help to complete the deduction part with little manual effort.

Another challenge is the integration of timing and continuous behavior into the verification methods presented. Though the basic concepts will remain the same, the verification tools which are able to handle timed and hybrid models are even more sensitive to complexity issues than tools for purely discrete models.

Appendix A

Condition/Event Systems

This appendix introduces the formal framework used for the models in Chapter 4 and their transformation into SMV code as explained in Appendix B. We introduce *condition/event systems, discrete condition/event systems, named discrete condition/event systems,* and define the *parallel interconnection* of named discrete condition/event systems.

The contents of Appendices A and B have been published as part of a technical report [Luk99b] for the VHS project [VHS].

A.1 Introduction

Condition/event systems (CESs) have been introduced in [SK91]. CESs are a class of continuous-time discrete event systems and can be represented graphically as block diagrams. Communication between CESs takes place by exchanging two different kinds of symbols, called *condition* and *event* symbols. A set of condition or event symbols is called condition or event *alphabet.* Condition symbols are used to describe system states and to enable or disable state changes, whereas event symbols denote certain actions at discrete points in time and can be used to trigger state changes. To describe the semantics of a CES, condition and event *signals* are used. A signal is a function which maps points in time to a condition or event alphabet, i.e., at each moment, one symbol of an alphabet is visible. The behavior of a CES is given by a function which associates with each pair of an input condition signal and an input event signal a nonempty set of possible outputs, which are pairs of output condition signals and output event signals.

This way of describing a system's behavior has some disadvantages. The behavior of a CES is given by relating signals, and one signal describes the current symbol visible at a CES's input or output component for *all* points in time. This is a very abstract view, since normally a dynamic system's output symbols at a certain moment depend only on its internal state and its input symbols at that very moment. Furthermore, we would like to

have an *operational* description of the system using functions mapping input symbols to output symbols (and not relating signals like in CESs), since this makes the implementation and the reasoning about systems much easier. Therefore *discrete condition/event systems* (DCESs) have been introduced, which communicate using the same symbols as CESs, but are described operationally using a finite set of internal states, a transition function, and output functions which compute output symbols using the current internal state and the current input symbols. So a DCES can be understood as a finite automaton having two output functions, one for conditions and one for events. The semantics of a DCES is described as a CES, but not every behavior of a CES is expressible using a DCES.

For composing CESs and DCESs, the notions of *cascade* and *feedback* connections are presented in [SK91]. These connections are very restrictive in the sense that they do not provide many ways of connecting systems, e.g., feedback allows no additional external input signals.

We present a formal framework for composing DCESs. Intuitively, systems are composed by connecting output components to input components, and the signals of the connected components have to be equal. We discuss the following two questions which arise when constructing a system consisting of some interconnected modules (DCESs in our framework):

1. Can we express the semantics of the compound system by using the semantics of its modules?

2. Can we compose the modules into one DCES having the same semantics as the compound system?

For answering these questions, we must of course have a formal definition of the semantics of the compound system. This must not necessarily be expressible in the way we express the semantics of its parts.

The answers to both questions depend on the way the systems are connected. If there are no interconnections at all, i.e., the parts do not exchange any information and are independent from each other, both answers are obviously yes. We simply can use Cartesian products of the parts' signals to form the signals of the compound system (this answers question 1), and we can use a Cartesian product of the parts' sets of states to gain the set of states for the DCES needed in question 2. No synchronization is needed, since the parts have no common signals.

If there are interconnections of the parts, which is usually the case for non-trivial systems, the answers are not that easy to find. Although question 1 might be positively answered, a DCES for question 2 may not exist due to so-called *algebraic loops*. For example, consider the two DCESs $\mathcal{D}_1$ and $\mathcal{D}_2$ in Figure A.1. Both systems are mutually connected by condition signals that can take on the values "a" or "b". $\mathcal{D}_1$ just copies its input (from

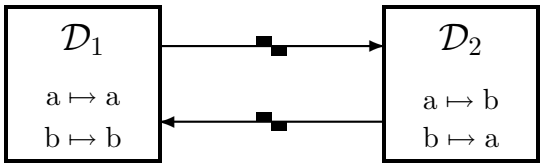


Figure A.1: Example of an algebraic loop

$\mathcal{D}_2$) to its output (to $\mathcal{D}_2$), while $\mathcal{D}_2$ is an inverter, i.e., input "a" yields output "b" and input "b" yields output "a". Since we have no time delays during communication in our model, this connection leads to contradiction; there are no feasible values for the symbols exchanged between $\mathcal{D}_1$ and $\mathcal{D}_2$, and therefore, no DCES for the composed system can be given.

In synchronous languages [Hal93], much research is done to examine such algebraic loops. One way of dealing with this is to demand *unique* solutions for the symbols between $\mathcal{D}_1$ and $\mathcal{D}_2$ (e.g. in LUSTRE [CPHP87] and ESTEREL [BG92]). Systems which do not fulfill this property must not have that kind of mutual connection. Another way is to demand that there exists *at least one* solution (e.g. in Signal [GGB87, BG90]).

If the answer to question 2 is yes, one is interested in a *composition operators* which effectively construct the DCES for the compound system, using the parts and information about the connections between them. In [SK91] two operators are introduces: The *cascade interconnection* operator defines the connection of a DCES's outputs with the inputs of another DCES, and the *feedback connection* operator defines the connection of a DCES's outputs with its own inputs. We define the *parallel interconnection* operator, which allows arbitrary connections among a set of DCESs and can therefore also express feedback and cascade. We define a parallel interconnection operator which handles systems with unique solutions; the case with multiple solutions is discussed in [Luk99b].

A.1.1 Notational Conventions

For a set X, we denote by 2^X the set of all subsets of X. Given a finite set D and a function $r : \mathbf{R}_{\geq 0} \to D$, we call r *finite-variable,* if in any bounded subinterval of $\mathbf{R}_{\geq 0}$, r changes its value only finitely often, i.e, r has only a finite number of discontinuity points. If r is finite-variable, r is furthermore called *right-continuous,* if for all $t \in \mathbf{R}_{\geq 0}$ there exists an $\varepsilon > 0$ such that $r(t + x) = r(t)$ for all $x \in (0, \varepsilon)$. And for any finite-variable r, we define for all $t \in \mathbf{R}_{>0}$ the limit from the left of r at t: $r(t^-) = \lim_{\varepsilon \to 0}^{\varepsilon > 0} r(t - \varepsilon)$.

For any n-tuple x, we denote by x_i the projection of x on its ith component. This notion is extended to Cartesian products of sets and to functions having a Cartesian product as range. The domain of a function f is denoted by $\mathrm{dom}(f)$, and its range by $\mathrm{ran}(f)$.

A.2 The Condition/Event System Framework

Condition/event systems (CESs) are a class of continuous-time discrete event systems and have been introduced in [SK91]. The CES model is widely used to describe the behavior of discrete and hybrid systems. It allows to develop complex systems by means of block diagrams and signal flows, which is standard practice in system theory.

A.2.1 Conditions, Events, and Signals

Let D be a set of symbols. These will be the symbols in the range of condition and event signals.

Definition A.1 (Condition alphabet, event alphabet) A *condition alphabet* is a nonempty, finite set $U \subseteq D$, and its elements are called *condition symbols* or simply *conditions.*[1]

An *event alphabet* is a nonempty, finite set $V \subseteq D$ containing a special *null symbol* $\mathbf{0}_V$. Any $v \in V$ is called *event symbol,* or simply *event,* and if $v \neq \mathbf{0}_V$, it is also called *proper event.*

We write $\mathbf{0}$ for the null symbol $\mathbf{0}_V$, if the event alphabet V is known from the context. A Cartesian product $V = V_1 \times \ldots \times V_n$ of event alphabets is also an event alphabet, and we abbreviate its null symbol $(\mathbf{0}_{V_1}, \ldots, \mathbf{0}_{V_n})$ also with $\mathbf{0}_V$ or $\mathbf{0}$. In the following, we will use the letters U and Y for condition alphabets, and V and Z for event alphabets.

Condition and event alphabets serve as the range for *condition signals* and *event signals,* which are functions mapping points in time to such alphabets. In our framework, time is modeled by the set $\mathbf{R}_{\geq 0}$ of nonnegative real numbers.

Definition A.2 (Condition signal, event signal) A *condition signal* over a condition alphabet U is a right-continuous, finite-variable function $s_U : \mathbf{R}_{\geq 0} \to U$. The set of all condition signals over U is denoted by $C(U)$.

An *event signal* over an event alphabet V is a function $s_V : \mathbf{R}_{\geq 0} \to V$, if $s_V(0) = \mathbf{0}$ and for any bounded subinterval I of $\mathbf{R}_{\geq 0}$, there are only finitely many $t \in I$ with $s_V(t) \neq \mathbf{0}$. The set of all event signals over V is denoted by $E(V)$.

Figure A.2 shows an example of a condition signal s_U over the condition alphabet $U = \{\text{on}, \text{off}, \text{auto}\}$. Figure A.3 shows an example of an event signal s_V over the event alphabet $V = \{\mathbf{0}, \text{start}, \text{stop}\}$.

[1] The reader is warned not to confuse the usage of "condition" as an abbreviation for "condition symbol" with the notion of "condition" for "property", "prerequisite", "Boolean expression", etc.

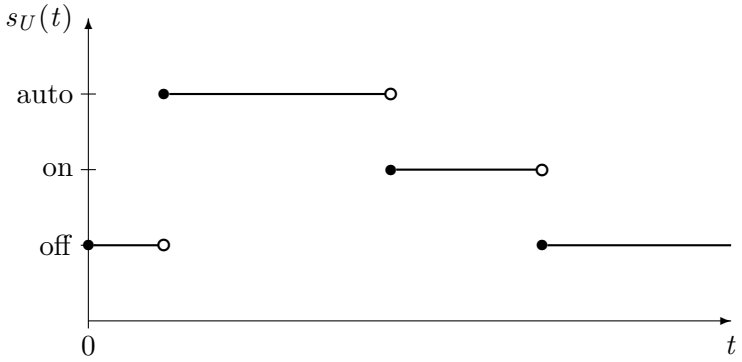


Figure A.2: Example of a condition signal

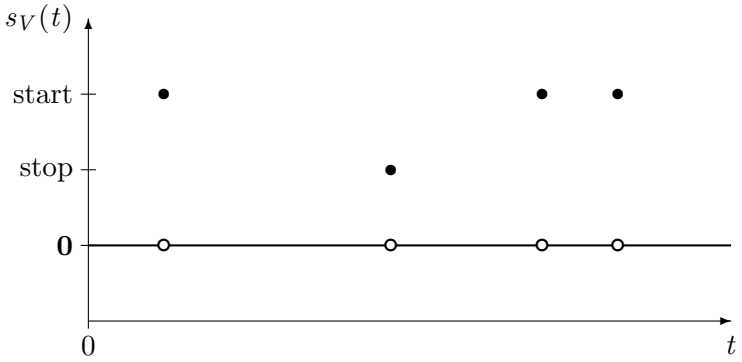


Figure A.3: Example of an event signal

A.2.2 Condition/Event Systems

Now we define condition/event systems. As said before, such a system is characterized by a relation between input and output signals. The relation can also be seen as a function mapping input signals to sets of output signals. So for each input there may be more than one possible output behavior, but there has to be at least one, since CESs are not allowed to refuse generating output (from a practical point of view, you can always measure some values at the CES's output "connectors").

Definition A.3 (Condition/event system) A quintuple $\mathcal{S} = (U, V, Y, Z, S)$ is called *condition/event system* (CES), where U is the *input condition alphabet,* V is the *input event alphabet,* Y is the *output condition alphabet,* Z is the *output event alphabet,* and $S : C(U) \times E(V) \to 2^{C(Y)\times E(Z)} \setminus \{\emptyset\}$ is the *system behavior function.*

We also use the symbol S for the *system behavior relation* $R \subseteq \mathrm{Beh}(U, V, Y, Z) = C(U) \times E(V) \times C(Y) \times E(Z)$ with $(s_U, s_V, s_Y, s_Z) \in R$ if and only if $(s_Y, s_Z) \in S(s_U, s_V)$.

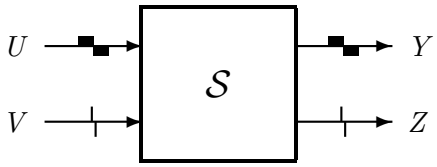


Figure A.4: A block diagram of a condition/event system

For the purpose of brevity, we will use the symbols s_U, s_V, s_Y, and s_Z to denote the four projections of a given behavior $s \in \mathrm{Beh}(U, V, Y, Z)$ on its components.

Figure A.4 shows a block diagram of a condition/event system. We use an arrow like ⟶ to denote the flow of a condition signal and ⟶ for an event signal flow. If we need systems with more than these four signals, we use Cartesian products like $U = U_1 \times U_2$ as alphabets, and we draw signal flows for each U_i in the block diagrams (see, for example, the block diagrams in Section 4.4.1). Formally, we access a certain alphabet U_i, its symbols $u_i \in U_i$, or its signals $s_{U_i} \in C(U_i)$ by using a projection on the ith component of U, $(u_1, u_2) \in U$, or $(s_{U_1}, s_{U_2}) \in C(U)$.

A.2.3 Discrete Condition/Event Systems

With the notion of condition/event systems we have introduced a model describing a relation of input and output signals. However, it is in practice very difficult to relate signals, since they range over the infinite time domain $\mathbf{R}_{\geq 0}$.

Moreover, changes in the *condition* input might force the CES to generate a proper output event at the time the condition change occurs. This behavior is not intended; only input *events* should be able to force output events. And since one specifies behaviors over the whole time domain, you can even build systems that are able to look into the future, e.g., a CES that sends an event as a reaction to an input event exactly one time unit *before* this input occurs.

Strange behaviors of such nature can be avoided by demanding certain properties like *causality, time-change invariance,* and *spontaneity* [SK91].

Another way of avoiding this, and, which is most important, a way of describing a CES in an *operational* way, is the model of *discrete* condition/event systems. These systems communicate with their environment via condition and event signals just like CESs, but an internal finite transition system is used to compute the current output symbols from the current input symbols at each point in time. The input conditions can enable or disable state changes in the transition system, and input events may force transitions.

Definition A.4 (Discrete condition/event system) A *discrete condition/event system* (DCES) $\mathcal{D} = (U, V, X, Y, Z, f, g, h, x_0)$ consists of an input condition alphabet U, an input event alphabet V, an output condition alphabet Y, an output event alphabet Z, and

- a finite set X of *states,*
- a *state transition function* $f : X \times U \times V \to 2^X \setminus \{\emptyset\}$,
- a *condition output function* $g : X \times U \to Y$,
- an *event output function* $h : X \times X \times V \to Z$, and
- an initial state $x_0 \in X$.

We give a short intuitive description of how DCESs work. The formal semantics can be found in Definition A.7. The function f is used to determine the next state of the transition system. As parameters f has the current state and the current condition and event input symbols. A nonempty set of states is computed by f, and one of these is nondeterministically chosen as the next state. The condition output symbol is generated by g from the current state and input condition symbol. The function h produces the event output symbol; parameters are the current state, the next state (which was chosen from the set of possible ones), and the current event input symbol. Note that g and h allow no nondeterministic choice; this is possible in f only.

Now we introduce two properties which lead to an interesting subclass of DCESs, the *well-behaved* DCESs.

Definition A.5 (Well-behaved DCES) Let $\mathcal{D}$ be a DCES as in Definition A.4. $\mathcal{D}$ is called *well-behaved,* if it satisfies

- *stuttering:* $x \in f(x, u, \mathbf{0})$, and
- *output triggering:* $h(x, x, \mathbf{0}) = \mathbf{0}$,

for all $x \in X$ and $u \in U$.

Stuttering describes that the system is always able to stay in the same state as long as no proper input event occurs. This ensures that changes in the input condition *cannot* force any transitions. Output triggering means that the system does not generate a proper output event if it stays in the same state and no proper input event occurs. This prevents the system from generating infinitely many proper events in finite time, since in a finite time interval only finitely many state changes are allowed (this is required in Definition A.6 below).

The operative nature of DCESs, well-behavedness, and its semantics (defined below) ensure the properties mentioned above (causality, time-change

invariance, spontaneity), which avoid the "strange behaviors". In the following, we will therefore use well-behaved DCESs only, without explicitly mentioning it. However, when constructing a DCES by giving explicit definitions for f, g, and h, we do have to show that the DCES is well-behaved.

To define the semantics for DCESs, we define *runs* of DCESs. Intuitively, a run is a function mapping points in time to the state of the DCES's transition system at that time. A run r always starts at the initial state. At all moments $t \in \mathbf{R}_{\geq 0}$, the value of the condition output signal $s_Y(t)$ is computed by g using as parameters the current state $r(t)$ and the current value of the condition input signal $s_U(t)$. At each point in time $t \in \mathbf{R}_{>0}$, a set of states is generated by f using the last state $r(t^-)$, the input condition symbol $s_U(t^-)$ just before time t, and the current input event symbol $s_V(t)$. One of these states is nondeterministically chosen as the new state $r(t)$. The value of the event output signal $s_Z(t)$ is computed by h using the last state $r(t^-)$, the current state $r(t)$ and the current value of the event input signal $s_V(t)$.

Note that nothing is said about the event output symbol at time 0, since the definition of event signal implies $s_Z(0) = \mathbf{0}$.

The usage of $s_U(t^-)$ as parameter for f emphasizes that conditions cannot force transitions; if the condition input signal changes just at time t, still the old value will be used. So this change can only influence transitions *after* time t, but not the transition *at* time t.

Definition A.6 (Run of a DCES) Let $\mathcal{D}$ be a DCES as in Definition A.4. We call a right-continuous and finite-variable function $r : \mathbf{R}_{\geq 0} \to X$ *run* of $\mathcal{D}$ over $s \in \mathrm{Beh}(U, V, Y, Z)$, if and only the following conditions are satisfied by r:

1. $r(t) \in f(r(t^-), s_U(t^-), s_V(t))$ for all $t \in \mathbf{R}_{>0}$,
2. $s_Y(t) = g(r(t), s_U(t))$ for all $t \in \mathbf{R}_{\geq 0}$,
3. $s_Z(t) = h(r(t^-), r(t), s_V(t))$ for all $t \in \mathbf{R}_{>0}$, and
4. $r(0) = x_0$.

Now we can assign a semantics to DCESs. The semantics of a DCES $\mathcal{D}$ is a CES $\mathcal{S}_\mathcal{D}$ with a behavior relation containing all the behaviors matching to possible runs of $\mathcal{D}$.

Definition A.7 (Semantics of a DCES) Let $\mathcal{D}$ be a DCES as in Definition A.4. The *semantics of* $\mathcal{D}$ is given by the CES $\mathcal{S}_\mathcal{D} = (U, V, Y, Z, S_\mathcal{D})$ with $s \in S_\mathcal{D}$ if and only if there exists a run r of $\mathcal{D}$ over s, for all $s \in \mathrm{Beh}(U, V, Y, Z)$.

There is one crucial question which arises immediately after having defined the semantic relation between CESs and DCESs: Can every condition/event system be described by a discrete condition/event system? Or, speaking formally, can we find for every CES $\mathcal{S}$ a DCES $\mathcal{D}$ such that $\mathcal{S}_\mathcal{D} = \mathcal{S}$? The answer is obviously no, as shown in the following example.

Example A.8 We define a simple CES which has only trivial inputs and outputs except for the event output. The only proper output event is a "ping" which can be observed at every time point $t \in \mathbf{N} \setminus \{0\}$, i.e., the time difference between two "pings" is exactly one time unit. Let $\mathcal{S} = (U, V, Y, Z, S)$ be the CES with the trivial condition alphabets[2] $U = Y = \{0\}$, event alphabets $V = \{\mathbf{0}\}$, $Z = \{\mathbf{0}, \text{ping}\}$, and, for all $s \in \text{Beh}(U, V, Y, Z)$,

$$s \in S \quad \text{if and only if} \quad s_Z(t) = \begin{cases} \text{ping}, & \text{if } t \in \mathbf{N} \setminus \{0\} \\ \mathbf{0} & , \text{otherwise} \end{cases}.$$

Note that the trivial alphabets allow only one possibility for s_U, s_V, and s_Y.

This CES cannot be modeled as a DCES, since DCESs have no means to do quantitative timing. In other words, one can build a DCES that outputs a "ping" from time to time, but there is no way to measure time, so the time difference between two "pings" cannot be forced to be exactly one time unit.

A.3 The Parallel Interconnection

The *cascade* and *feedback* connections presented in [SK91] for the composition of CESs and DCESs have the disadvantage that the syntactic requirements for the systems are very strong, as we must have pairs of matching input and output components, even if in some directions we do not have any information to be passed. Furthermore, these two connection operators, even when used iteratively and in combination, are not general enough to express any conceivable connection between several systems.

This shows that we need a different, more flexible way to describe connections between condition/event systems. The solution we present is the *parallel interconnection,* which describes an almost arbitrary connection of CESs, and, by defining the *parallel interconnection operator,* we are able to consider arbitrary connections among DCESs (there are some sensible syntactic restrictions, though).

Connections are described by assigning *names* to the input and output components of all systems. Components which share the same name are

[2] In our examples, we use $\{0\}$ as a trivial condition alphabet. This is quite arbitrary; one can use any singleton $U \subseteq D$.

considered to be connected. An intuitive semantics for a system of CESs connected in such a way is obvious: for every name occurring in the system, one signal has to be provided. If a name is the name of an output component of a CES, this CES provides the signal for that name, and this signal is used for all input components with that name. If a name is used for input components only, the signal for that name must be an external input signal for the system. The output signals of the system are the output signals of all CESs. This parallel interconnection of CESs will be given formally in Definition A.18.

The notion of names makes new definitions for syntax and semantics of CESs and DCESs necessary, but the flexibility we gain for constructing interconnected systems is worth the effort.

A.3.1 Adding Component Names to C/E Systems

In the previous sections, we described a condition/event system's inputs and outputs by Cartesian products of condition or event alphabets. We accessed single components by projecting elements of these products on a certain index. To make dealing with many components easier and to provide a natural way of connecting systems, we change the definition of CESs in the following way: All input and output components of a CES are identified by names. Each name is associated with one condition or event alphabet. Each alphabet may be associated with more than one name. Components with the same names are considered to be connected later on.

The following definitions put these considerations on a formal basis. From now on, let $\mathcal{N}$ be a global set of identifiers, called *names.* We associate with each name $\mathsf{N} \in \mathcal{N}$ either a condition or an event alphabet, denoted by $\alpha(\mathsf{N})$. We call N *condition name* (respectively *event name*), if $\alpha(\mathsf{N})$ is a condition alphabet (respectively event alphabet).

Previously we used Cartesian products of alphabets like $U_1 \times U_2$ if a CES or DCES needed to have two independent alphabets in one of its signals. With the new notion of names, we simply use a set of names for such purposes, like for the case above the set $\bar{\mathsf{U}} = \{\mathsf{U}_1, \mathsf{U}_2\}$ with $\alpha(\mathsf{U}_1) = U_1$ and $\alpha(\mathsf{U}_2) = U_2$. We use symbols like U, V, Y, Z for names, and symbols like $\bar{\mathsf{U}}$, $\bar{\mathsf{V}}$, $\bar{\mathsf{Y}}$, $\bar{\mathsf{Z}}$ for sets of names.

For describing the current value of a system's component named with the set $\bar{\mathsf{A}}$, we introduce the notion of an *evaluation,* which is a function mapping the names of $\bar{\mathsf{A}}$ to the condition or event symbols in the alphabets associated with the names in $\bar{\mathsf{A}}$.

Definition A.9 (Evaluation) For a set $\bar{\mathsf{A}} \subseteq \mathcal{N}$ of names, we define

$$\Sigma_{\bar{\mathsf{A}}} = [\bar{\mathsf{A}} \xrightarrow{\text{type}} \bigcup_{\mathsf{N} \in \bar{\mathsf{A}}} \alpha(\mathsf{N})]$$

as the set of all type-respecting functions φ mapping names to conditions or events, where type-respecting in this case means $\varphi(\mathsf{N}) \in \alpha(\mathsf{N})$ for all $\mathsf{N} \in \bar{\mathsf{A}}$. Such a function φ is called *evaluation* of $\bar{\mathsf{A}}$.

Note that there is only one evaluation for the empty set of names, and this is the function $\emptyset$ having an empty domain, so we have $\Sigma_\emptyset = \{\emptyset\}$. If $\varphi(\mathsf{N}) = \mathbf{0}_{\alpha(\mathsf{N})}$ for all $\mathsf{N} \in \bar{\mathsf{A}}$, we use the symbol $\mathbf{0}_{\bar{\mathsf{A}}}$ (or simply $\mathbf{0}$) for φ. For $\mathsf{N} \in \mathcal{N}$, we define Σ_N as a shortcut for $\Sigma_{\{\mathsf{N}\}}$. Furthermore, we define some operations on evaluations.

Definition A.10 (Operations on evaluations) For evaluations φ and ψ, we define the following operations:

- *union:* If φ and ψ coincide on the intersection of their domains, $\varphi \cup \psi$ denotes their union.
- *restriction:* For $\bar{\mathsf{A}} \subseteq \mathrm{dom}(\varphi)$, the restriction of φ on $\bar{\mathsf{A}}$ is denoted by $\varphi|_{\bar{\mathsf{A}}}$. For the sake of convenient notation, we allow $\bar{\mathsf{A}} = \emptyset$, resulting in $\varphi|_{\bar{\mathsf{A}}} = \emptyset$. For $\mathsf{N} \in \mathcal{N}$, $\varphi|_\mathsf{N}$ is an abbreviation for $\varphi|_{\{\mathsf{N}\}}$.

These operations are extended in the obvious way to named signals and named behaviors, which are defined below.

We use the evaluations introduced above to define *named condition/event systems*, which operate on *named condition/event signals.*

Definition A.11 (Named condition/event signal) Let $\bar{\mathsf{U}} \subseteq \mathcal{N}$ be a finite set of condition names. A function $s_{\bar{\mathsf{U}}} : \mathbf{R}_{\geq 0} \to \Sigma_{\bar{\mathsf{U}}}$ is called *named condition signal* over $\bar{\mathsf{U}}$, if it is right-continuous and finite-variable. The set of all named condition signals over $\bar{\mathsf{U}}$ is denoted by $C(\bar{\mathsf{U}})$.

Let $\bar{\mathsf{V}} \subseteq \mathcal{N}$ be a finite set of event names. A function $s_{\bar{\mathsf{V}}} : \mathbf{R}_{\geq 0} \to \Sigma_{\bar{\mathsf{V}}}$ is called *named event signal* over $\bar{\mathsf{V}}$, if $s_{\bar{\mathsf{V}}}(0) = \mathbf{0}$ and in every bounded subinterval I of $\mathbf{R}_{\geq 0}$ there are only finitely many points $t \in I$ with $s_{\bar{\mathsf{V}}}(t) \neq \mathbf{0}$. The set of all named event signals over $\bar{\mathsf{V}}$ is denoted by $E(\bar{\mathsf{V}})$.

For the signal over an empty set of names we use the symbol $\emptyset$, denoting the function $s_\emptyset : \mathbf{R}_{\geq 0} \to \Sigma_\emptyset$ with constant value $\emptyset$.

Now we are ready to define named condition/event systems. We ensure that each name in $\mathcal{N}$ is used at most once in one system, since we do not want a system to become connected to itself (feedback can still be expressed by introducing a second system copying its inputs to its outputs).

Definition A.12 (Named condition/event system) Let $\bar{\mathsf{U}}$ and $\bar{\mathsf{Y}}$ be finite sets of condition names, and let $\bar{\mathsf{V}}$ and $\bar{\mathsf{Z}}$ be finite sets of event names. Let all four sets be pairwise disjoint. The quintuple $\mathcal{S} = (\bar{\mathsf{U}}, \bar{\mathsf{V}}, \bar{\mathsf{Y}}, \bar{\mathsf{Z}}, S)$ is called *named condition/event system*, where $S : C(\bar{\mathsf{U}}) \times E(\bar{\mathsf{V}}) \to 2^{C(\bar{\mathsf{Y}}) \times E(\bar{\mathsf{Z}})} \setminus \{\emptyset\}$ is the *system behavior function.* We also use the symbol S for the *system*

behavior relation $R \subseteq \mathrm{Beh}(\bar{\mathsf{U}}, \bar{\mathsf{V}}, \bar{\mathsf{Y}}, \bar{\mathsf{Z}}) = C(\bar{\mathsf{U}}) \times E(\bar{\mathsf{V}}) \times C(\bar{\mathsf{Y}}) \times E(\bar{\mathsf{Z}})$ with $(s_{\bar{\mathsf{U}}}, s_{\bar{\mathsf{V}}}, s_{\bar{\mathsf{Y}}}, s_{\bar{\mathsf{Z}}}) \in R$ if and only if $(s_{\bar{\mathsf{Y}}}, s_{\bar{\mathsf{Z}}}) \in S(s_{\bar{\mathsf{U}}}, s_{\bar{\mathsf{V}}})$.

Sometimes, $S_{\mathcal{S}}$ is used to denote the behavior function/relation of the named CES $\mathcal{S}$. For the sake of brevity, we will use the symbols $s_{\bar{\mathsf{U}}}$, $s_{\bar{\mathsf{V}}}$, $s_{\bar{\mathsf{Y}}}$, and $s_{\bar{\mathsf{Z}}}$ to denote the four projections of a given $s \in \mathrm{Beh}(\bar{\mathsf{U}}, \bar{\mathsf{V}}, \bar{\mathsf{Y}}, \bar{\mathsf{Z}})$ on its components.

Now we give the definition of *named discrete condition/event systems*. Note that besides the introduction of names, we now also have a set of initial states instead of only one initial state.

Definition A.13 (Named DCES, well-behaved) A *named discrete condition/event system* $\mathcal{D} = (\bar{\mathsf{U}}, \bar{\mathsf{V}}, X, \bar{\mathsf{Y}}, \bar{\mathsf{Z}}, f, g, h, X_0)$ consists of a finite set $\bar{\mathsf{U}}$ of *condition input names,* a finite set $\bar{\mathsf{V}}$ of *event input names,* a finite set $\bar{\mathsf{Y}}$ of *condition output names,* a finite set $\bar{\mathsf{Z}}$ of *event output names,*

- a finite set X of *states,*
- a *state transition function* $f : X \times \Sigma_{\bar{\mathsf{U}}} \times \Sigma_{\bar{\mathsf{V}}} \to 2^X \setminus \{\emptyset\}$,
- a *condition output function* $g : X \times \Sigma_{\bar{\mathsf{U}}} \to \Sigma_{\bar{\mathsf{Y}}}$,
- an *event output function* $h : X \times X \times \Sigma_{\bar{\mathsf{V}}} \to \Sigma_{\bar{\mathsf{Z}}}$, and
- a nonempty set $X_0 \subseteq X$ of initial states,

where $\bar{\mathsf{U}}$, $\bar{\mathsf{V}}$, $\bar{\mathsf{Y}}$, and $\bar{\mathsf{Z}}$ are pairwise disjoint. The named DCES $\mathcal{D}$ is called *well-behaved,* if it satisfies

- *stuttering:* $x \in f(x, u, \mathbf{0})$ and
- *output triggering:* $h(x, x, \mathbf{0}) = \mathbf{0}$,

for all $x \in X$ and $u \in \Sigma_{\bar{\mathsf{U}}}$.

In the following we will use well-behaved named DCESs only. To define the semantics for a named DCES, we define *runs* of named DCESs. Notice that in contrast to Definition A.6, we have $r(0) \in X_0$ as start of the run.

Definition A.14 (Run of a named DCES) Let $\mathcal{D}$ be given as in Definition A.13. We call a right-continuous and finite-variable function $r : \mathbf{R}_{\geq 0} \to X$ *run* of $\mathcal{D}$ over $s \in \mathrm{Beh}(\bar{\mathsf{U}}, \bar{\mathsf{V}}, \bar{\mathsf{Y}}, \bar{\mathsf{Z}})$, if and only if

1. $r(t) \in f(r(t^-), s_{\bar{\mathsf{U}}}(t^-), s_{\bar{\mathsf{V}}}(t))$ for all $t \in \mathbf{R}_{>0}$,
2. $s_{\bar{\mathsf{Y}}}(t) = g(r(t), s_{\bar{\mathsf{U}}}(t))$ for all $t \in \mathbf{R}_{\geq 0}$,
3. $s_{\bar{\mathsf{Z}}}(t) = h(r(t^-), r(t), s_{\bar{\mathsf{V}}}(t))$ for all $t \in \mathbf{R}_{>0}$, and

4. $r(0) \in X_0$.

Now we can assign a semantics to named DCESs. Like in Definition A.7, it is given via the notion of runs.

Definition A.15 (Semantics of a named DCES) Let $\mathcal{D}$ be given as in Definition A.13. The *semantics of* $\mathcal{D}$ is given by the named CES $\mathcal{S}_\mathcal{D} = (\bar{\mathsf{U}}, \bar{\mathsf{V}}, \bar{\mathsf{Y}}, \bar{\mathsf{Z}}, S_\mathcal{D})$ with $s \in S_\mathcal{D}$ if and only if there exists a run r of $\mathcal{D}$ over s, for all $s \in \mathrm{Beh}(\bar{\mathsf{U}}, \bar{\mathsf{V}}, \bar{\mathsf{Y}}, \bar{\mathsf{Z}})$.

A.3.2 Graphical Descriptions of Discrete C/E Systems

The definition of a DCES through its four name sets, the state sets X and X_0, and the functions f, g, and h, as in Definition A.13 is precise but hard to read. We already know block diagrams as a means of displaying a system's inputs and outputs. In addition, we show next how the remaining components of a DCES can be described by a *transition diagram.*

We illustrate the specification of a DCES by graphical means using the following system which will also be used as an example in Section B.1.5:

$$\begin{aligned} \mathit{Switch} = (&\{\mathsf{Request}\}, \{\mathsf{Update}\}, \{x_0, x_1\}, \{\mathsf{Status}\}, \{\mathsf{Change}\}, \\ &f, g, h, \{x_0\}) \end{aligned}$$

with the alphabets $\alpha(\mathsf{Request}) = \{\mathrm{start}, \mathrm{stop}\}$ and $\alpha(\mathsf{Status}) = \{\mathrm{off}, \mathrm{on}\}$, and $\alpha(\mathsf{Update}) = \alpha(\mathsf{Change}) = \{\mathbf{0}, 1\}$, and for all $x, x' \in \{x_0, x_1\}$, $u \in \Sigma_{\mathsf{Request}}$, and $v \in \Sigma_{\mathsf{Update}}$,

$$f(x,u,v) = \begin{cases} \{x_1\}, \text{ if } x = x_0 \wedge u(\mathsf{Request}) = \mathrm{start} \wedge v(\mathsf{Update}) = 1 \\ \{x_0\}, \text{ if } x = x_1 \wedge u(\mathsf{Request}) = \mathrm{stop} \wedge v(\mathsf{Update}) = 1 \\ \{x\} \text{ , otherwise} \end{cases},$$

$$g(x,u) = \begin{cases} \{(\mathsf{Status}, \mathrm{off})\}, \text{ if } x = x_0 \\ \{(\mathsf{Status}, \mathrm{on})\}, \text{ if } x = x_1 \end{cases}, \text{ and}$$

$$h(x,x',v) = \begin{cases} \{(\mathsf{Change}, 1)\}, \text{ if } x' \neq x \\ \{(\mathsf{Change}, \mathbf{0})\}, \text{ otherwise} \end{cases}.$$

The block diagram of the DCES *Switch* defines the input and output name sets as well as the respective alphabets. The transition diagram shown in Figure A.6 defines the remaining components of *Switch*.

Each circle in the transition diagram represents one state of the DCES, in our case, x_0 and x_1. Each circle marked with a small arrow (x_0) is an initial state. Since g does not depend on the input condition and has only one component, this output condition (off or on) is displayed in the middle of the circle representing the respective state.

Finally, the functions f and g are defined through the transitions in the transition diagram. An edge from a state x to a state x' denotes that there

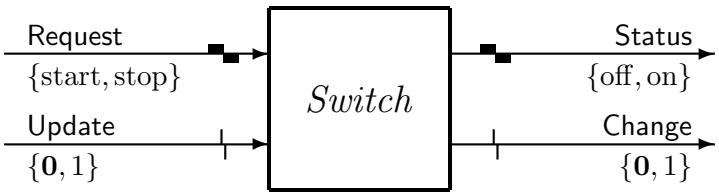


Figure A.5: Block diagram of *Switch*

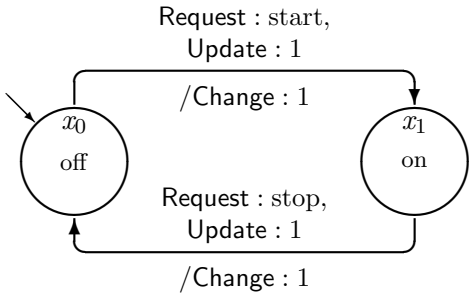


Figure A.6: The transition system of *Switch*

are u, v such that $x' \in f(x, u, v)$. The possible values for u and v are described by the labeling of that edge. In our example, the transition from x_0 to x_1 describes the first line in the definition of f, and the transition from x_1 to x_0 the second line. Everything after the slash (/) is used to indicate the event output evaluation generated by h during that transition.

Note that the parts of u and v not fixed in the labeling may have any value.

If for some x, u, and v there is no edge leaving state x having a labeling describing u and v, we assume $f(x, u, v) = \{x\}$, i.e., there is no state change in state x for the input evaluations u and v. Furthermore, we assume $x \in f(x, u, \mathbf{0})$ for any x and u to obtain stuttering. And if there is no edge from state x to x' describing the event output evaluation of some name N for some event input evaluation v, we assume $h(x, x', v)(\mathsf{N}) = \mathbf{0}$, i.e., there is no proper event generated. With these default values, stuttering and output triggering can be checked easily.

A.3.3 The Parallel Interconnection of C/E Systems

Having introduced component names for single condition/event systems, we are now interested in connecting some systems. As mentioned before, all components sharing the same name will be connected to each other.

Figure A.7 shows an example with three systems (we do not distinguish between conditions and events in this example). The component names of $\mathcal{S}_1$, $\mathcal{S}_2$, and $\mathcal{S}_3$ completely determine the connections between them. All

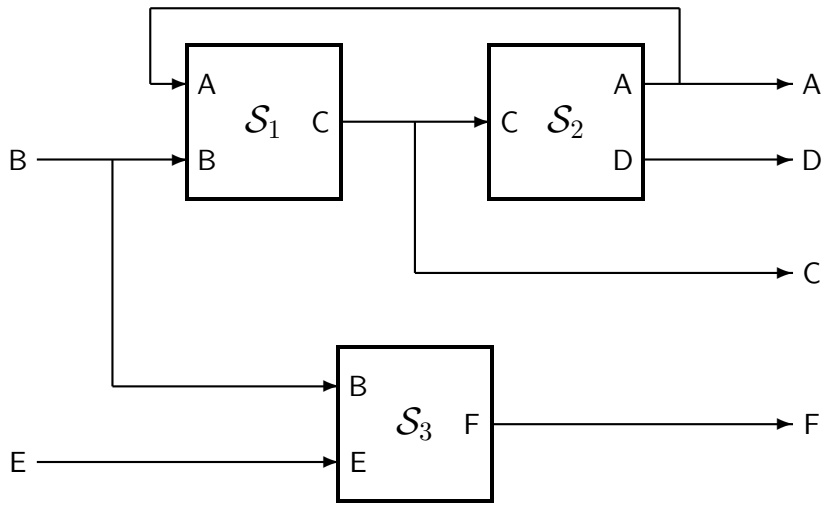


Figure A.7: Example of a parallel interconnection

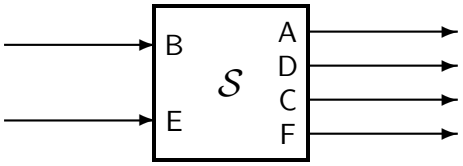


Figure A.8: The composition of the system in Figure A.7

outputs of the single systems will become outputs of the composed system, in this case A, D, C, and F. The inputs of the composed system are all inputs of the single systems for which there exist no output of the same name. This results in the input names B and E. Figure A.8 shows a block diagram of the composed system.

Remark A.16 In control theory, there is at least one more possibility to describe connections between systems: One can use a function that describes for each component of a system how this component is connected to other systems. Such functions have some advantages. For example, one can use one name for two different components even if they are *not* connected. This supports the reuse of modules from a library of systems without having to rename the components.

Though we do not use this method, it is clear that both methods are equally expressive and can be transformed into each other.

In Definition A.12, we assured that a name is used at most once within one system. For a set of systems to be connected to each other, we also have to ensure that in the whole set of systems all output components must have different names, since outputs must not be connected to each other. This

requirement is formalized in the following definition.

Definition A.17 (Consistent set of named CESs) For $i \in \{1, \ldots, n\}$, let $\mathcal{S}_i = (\bar{\mathsf{U}}_i, \bar{\mathsf{V}}_i, \bar{\mathsf{Y}}_i, \bar{\mathsf{Z}}_i, S_i)$ be a named C/E system. The set $\{\mathcal{S}_1, \ldots, \mathcal{S}_n\}$ is called *consistent,* if

$$\left(\bar{\mathsf{Y}}_i \cup \bar{\mathsf{Z}}_i\right) \cap \left(\bar{\mathsf{Y}}_j \cup \bar{\mathsf{Z}}_j\right) = \emptyset$$

holds for all $i, j \in \{1, \ldots, n\}$ with $i \neq j$.

The parallel interconnection of C/E systems can now be defined. This is only a descriptive definition; we do not reason about the existence of a parallel interconnection for any given set of C/E systems.

Definition A.18 (Parallel interconnection) For $i \in \{1, \ldots, n\}$, let $\mathcal{S}_i = (\bar{\mathsf{U}}_i, \bar{\mathsf{V}}_i, \bar{\mathsf{Y}}_i, \bar{\mathsf{Z}}_i, S_i)$ be a named C/E system. Let $\{\mathcal{S}_1, \ldots, \mathcal{S}_n\}$ be *consistent.* A named C/E system $\mathcal{S} = (\bar{\mathsf{U}}, \bar{\mathsf{V}}, \bar{\mathsf{Y}}, \bar{\mathsf{Z}}, S)$ is called *parallel interconnection* of $\mathcal{S}_1, \ldots, \mathcal{S}_n$, if the following conditions hold:

1. The names of the components of $\mathcal{S}$ are those of the components of the single systems, except the inputs for which there exists an output with the same name:

$$\begin{array}{ll} \bar{\mathsf{U}} = \left(\bigcup_{i=1}^{n} \bar{\mathsf{U}}_i\right) \setminus \left(\bigcup_{i=1}^{n} \bar{\mathsf{Y}}_i\right), & \bar{\mathsf{Y}} = \bigcup_{i=1}^{n} \bar{\mathsf{Y}}_i, \\ \bar{\mathsf{V}} = \left(\bigcup_{i=1}^{n} \bar{\mathsf{V}}_i\right) \setminus \left(\bigcup_{i=1}^{n} \bar{\mathsf{Z}}_i\right), & \bar{\mathsf{Z}} = \bigcup_{i=1}^{n} \bar{\mathsf{Z}}_i. \end{array}$$

2. For any $(s_{\bar{\mathsf{U}}}, s_{\bar{\mathsf{V}}}, s_{\bar{\mathsf{Y}}}, s_{\bar{\mathsf{Z}}}) \in \mathrm{Beh}(\bar{\mathsf{U}}, \bar{\mathsf{V}}, \bar{\mathsf{Y}}, \bar{\mathsf{Z}})$, we have $(s_{\bar{\mathsf{U}}}, s_{\bar{\mathsf{V}}}, s_{\bar{\mathsf{Y}}}, s_{\bar{\mathsf{Z}}}) \in S$ if and only if there exist $(s_{\bar{\mathsf{U}}_i}, s_{\bar{\mathsf{V}}_i}, s_{\bar{\mathsf{Y}}_i}, s_{\bar{\mathsf{Z}}_i}) \in S_i$, for all $i \in \{1, \ldots, n\}$, such that the following two conditions are satisfied:

   (a) Connected components of the single systems share the same signals: For all $i, j \in \{1, \ldots, n\}$ we have

$$\begin{array}{l} s_{\bar{\mathsf{U}}_i}|_{\bar{\mathsf{U}}_i \cap \bar{\mathsf{Y}}_j} = s_{\bar{\mathsf{Y}}_j}|_{\bar{\mathsf{U}}_i \cap \bar{\mathsf{Y}}_j}, \text{ and} \\ s_{\bar{\mathsf{V}}_i}|_{\bar{\mathsf{V}}_i \cap \bar{\mathsf{Z}}_j} = s_{\bar{\mathsf{Z}}_j}|_{\bar{\mathsf{V}}_i \cap \bar{\mathsf{Z}}_j}. \end{array}$$

   (b) The signals of the single systems match those of $\mathcal{S}$: For all $i \in \{1, \ldots, n\}$ we have

$$\begin{array}{l} s_{\bar{\mathsf{U}}_i}|_{\bar{\mathsf{U}}_i \cap \bar{\mathsf{U}}} = s_{\bar{\mathsf{U}}}|_{\bar{\mathsf{U}}_i \cap \bar{\mathsf{U}}}, \quad s_{\bar{\mathsf{Y}}_i} = s_{\bar{\mathsf{Y}}}|_{\bar{\mathsf{Y}}_i}, \text{ and} \\ s_{\bar{\mathsf{V}}_i}|_{\bar{\mathsf{V}}_i \cap \bar{\mathsf{V}}} = s_{\bar{\mathsf{V}}}|_{\bar{\mathsf{V}}_i \cap \bar{\mathsf{V}}}, \quad s_{\bar{\mathsf{Z}}_i} = s_{\bar{\mathsf{Z}}}|_{\bar{\mathsf{Z}}_i}. \end{array}$$

A.3.4 The Parallel Interconnection of Discrete C/E Systems

In Definition A.17, we had to ensure that in a set of named CESs which are connected to each other, all output names are different. For discrete systems, this is done analogously in the following definition.

Definition A.19 (Consistent set of named DCESs) Given a set of named DCESs $\mathcal{D}_i$ with component name sets $\bar{\mathsf{U}}_i$, $\bar{\mathsf{V}}_i$, $\bar{\mathsf{Y}}_i$, and $\bar{\mathsf{Z}}_i$, for $i \in \{1, \ldots, n\}$, the set $\{\mathcal{D}_1, \ldots, \mathcal{D}_n\}$ is called *consistent*, if

$$\left(\bar{\mathsf{Y}}_i \cup \bar{\mathsf{Z}}_i\right) \cap \left(\bar{\mathsf{Y}}_j \cup \bar{\mathsf{Z}}_j\right) = \emptyset$$

holds for all $i, j \in \{1, \ldots, n\}$ with $i \neq j$.

For the remainder of this section, let $n \in \mathbf{N} \setminus \{0\}$, and, for all $i \in \{1, \ldots, n\}$, let $\mathcal{D}_i = (\bar{\mathsf{U}}_i, \bar{\mathsf{V}}_i, X_i, \bar{\mathsf{Y}}_i, \bar{\mathsf{Z}}_i, f_i, g_i, h_i, X_{0,i})$ be a named DCES.

The following definition introduces a classification of component names which distinguishes external and internal names.

Definition A.20 (Component name sets) We define some sets of input and output component names of $\mathcal{D}_1, \ldots, \mathcal{D}_n$:

$$\begin{aligned}
\mathsf{C}^{\mathsf{in}} &= \left(\bigcup_{i=1}^n \bar{\mathsf{U}}_i\right) \setminus \left(\bigcup_{i=1}^n \bar{\mathsf{Y}}_i\right), & \mathsf{E}^{\mathsf{in}} &= \left(\bigcup_{i=1}^n \bar{\mathsf{V}}_i\right) \setminus \left(\bigcup_{i=1}^n \bar{\mathsf{Z}}_i\right), \\
\mathsf{C}^{\mathsf{io}} &= \left(\bigcup_{i=1}^n \bar{\mathsf{U}}_i\right) \cap \left(\bigcup_{i=1}^n \bar{\mathsf{Y}}_i\right), & \mathsf{E}^{\mathsf{io}} &= \left(\bigcup_{i=1}^n \bar{\mathsf{V}}_i\right) \cap \left(\bigcup_{i=1}^n \bar{\mathsf{Z}}_i\right), \\
\mathsf{C}^{\mathsf{out}} &= \bigcup_{i=1}^n \bar{\mathsf{Y}}_i\,, & \mathsf{E}^{\mathsf{out}} &= \bigcup_{i=1}^n \bar{\mathsf{Z}}_i\,.
\end{aligned}$$

The set C^{in} contains all condition input component names for which there exists no output component of that name. These are the names of the condition signals which are not produced within the system; thus, these signals are *external* and have to be provided by the environment of the compound system. The set C^{io} contains all condition component names for which there exist input and output components of that name. These are the names for the *internal* condition signals which are exchanged by the systems. The set $\mathsf{C}^{\mathsf{out}}$ contains all output condition component names (it is a superset of C^{io}). The sets E^{in}, E^{io}, and $\mathsf{E}^{\mathsf{out}}$ have analog meanings for events. Using the notation of Definition A.18, we have $\mathsf{C}^{\mathsf{in}} = \bar{\mathsf{U}}$, $\mathsf{E}^{\mathsf{in}} = \bar{\mathsf{V}}$, $\mathsf{C}^{\mathsf{out}} = \bar{\mathsf{Y}}$, and $\mathsf{E}^{\mathsf{out}} = \bar{\mathsf{Z}}$.

Looking at the example in Figures A.7 and A.8, this definition leads to $\mathsf{C}^{\mathsf{in}} = \{\mathsf{B}, \mathsf{E}\}$, $\mathsf{C}^{\mathsf{io}} = \{\mathsf{A}, \mathsf{C}\}$, and $\mathsf{C}^{\mathsf{out}} = \{\mathsf{A}, \mathsf{C}, \mathsf{D}, \mathsf{F}\}$ (provided we only have conditions).

We use a *unique solution property* to ensure that there exists a discrete system with the same semantics as a parallel interconnection of the single systems' semantics.

Theorem A.21 Let $\{\mathcal{D}_1, \ldots, \mathcal{D}_n\}$ be consistent. A named DCES $\mathcal{D}$ can be effectively constructed such that $\mathcal{S}_{\mathcal{D}}$ is a parallel interconnection of $\mathcal{S}_{\mathcal{D}_1}, \ldots, \mathcal{S}_{\mathcal{D}_n}$, if the following *unique solution property* holds for $\mathcal{D}_1, \ldots, \mathcal{D}_n$:

- For all $x_1 \in X_1, \ldots, x_n \in X_n$, $u \in \Sigma^{\mathsf{in}}_{\mathsf{C}}$, there exists a unique $y \in \Sigma_{\mathsf{C}^{\mathsf{io}}}$ such that for all $i \in \{1, \ldots, n\}$, we have

$$g_i\left(x_i, (u \cup y)|_{\bar{\mathsf{U}}_i}\right)|_{\bar{\mathsf{Y}}_i \cap \mathsf{C}^{\mathsf{io}}} = y|_{\bar{\mathsf{Y}}_i \cap \mathsf{C}^{\mathsf{io}}}.$$

- For all $x_1, x_1' \in X_1, \ldots, x_n, x_n' \in X_n$, $v \in \Sigma_{\mathsf{E}}^{\mathsf{in}}$, there exists a unique $z \in \Sigma_{\mathsf{E}^{\mathsf{io}}}$ such that for all $i \in \{1, \ldots, n\}$, we have

$$h_i\left(x_i, x_i', (v \cup z)|_{\bar{\mathsf{V}}_i}\right)|_{\bar{\mathsf{Z}}_i \cap \mathsf{E}^{\mathsf{io}}} = z|_{\bar{\mathsf{Z}}_i \cap \mathsf{E}^{\mathsf{io}}}.$$

Before we prove this theorem, we take a look at the possible number of solutions for the evaluations $y \in \mathsf{C}^{\mathsf{io}}$ and $z \in \mathsf{E}^{\mathsf{io}}$.

Example A.22 Reconsider the system in Figure A.1. We define the DCESs $\mathcal{D}_1$ and $\mathcal{D}_2$ formally, this time in a parameterized version (cf. Figure A.9).

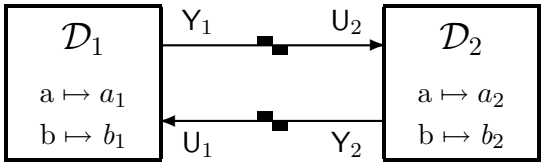


Figure A.9: Mutually connected DCESs

For $i \in \{1, 2\}$, let $\mathcal{D}_i = (\{\mathsf{U}_i\}, \emptyset, \{x\}, \{\mathsf{Y}_i\}, \emptyset, f_i, g_i, h_i, \{x\})$ be a named DCES with $\alpha(\mathsf{U}_i) = \alpha(\mathsf{Y}_i) = \{\mathrm{a}, \mathrm{b}\}$, and

- $f_i(x, u, \emptyset) = \{x\}$ for all $u \in \Sigma_{\mathsf{U}_i}$,
- $g_i(x, u)(\mathsf{Y}_i) = \begin{cases} a_i, \text{ if } u(\mathsf{U}_i) = \mathrm{a} \\ b_i, \text{ if } u(\mathsf{U}_i) = \mathrm{b} \end{cases}$, and
- $h_i(x, x, \emptyset) = \emptyset$.

We have only one state x and empty sets of event names, so the functions f_i and h_i are constant, and well-definedness is trivially fulfilled.[3] We characterize the connections between $\mathcal{D}_1$ and $\mathcal{D}_2$ by demanding $\mathsf{U}_1 = \mathsf{Y}_2 \neq \mathsf{Y}_1 = \mathsf{U}_2$.

Since $\mathcal{D}_1$ and $\mathcal{D}_2$ do not have any event names in common (there are no event names and alphabets at all), we have $\mathsf{E}^{\mathsf{io}} = \emptyset$, and the second condition of unique solution property is trivially fulfilled; the solution is always $z = \emptyset \in \Sigma_{\mathsf{E}^{\mathsf{io}}}$. Since there are no external condition inputs, we have $\mathsf{C}^{\mathsf{in}} = \emptyset$, and the first condition reduces to finding a unique $y \in \Sigma_{\{\mathsf{U}_1, \mathsf{U}_2\}}$ such that

$$g_1(x, y|_{\mathsf{U}_1}) = y|_{\mathsf{U}_2} \quad \text{and} \quad g_2(x, y|_{\mathsf{U}_2}) = y|_{\mathsf{U}_1}.$$

Now consider the following three setups:

1. $a_1 = \mathrm{a}$, $b_1 = \mathrm{b}$, and $a_2 = \mathrm{b}$, $b_2 = \mathrm{a}$.
2. $a_1 = \mathrm{a}$, $b_1 = \mathrm{a}$, and $a_2 = \mathrm{a}$, $b_2 = \mathrm{a}$.

[3] For output triggering, note that $\mathbf{0}_\emptyset = \emptyset$ (the evaluation with the empty domain).

3. $a_1 = \mathrm{a}$, $b_1 = \mathrm{b}$, and $a_2 = \mathrm{a}$, $b_2 = \mathrm{b}$.

For each of the three cases, we take a look at the solutions for the two equations above.

Case 1 has no solutions for y, since the two equations of the unique solution property are contradictory. No DCES $\mathcal{D}$ can be given such that $\mathcal{S}_\mathcal{D}$ is a parallel interconnection of $\mathcal{S}_{\mathcal{D}_1}$ and $\mathcal{S}_{\mathcal{D}_2}$. We have the existence of an algebraic loop; this is the case discussed in the introduction using Figure A.1.

Case 2 has the unique solution $y = \{\mathsf{U}_1 \mapsto \mathrm{a}, \mathsf{U}_2 \mapsto \mathrm{a}\}$, and, if Theorem A.21 is sound, a DCES for the overall system can be effectively constructed, see upcoming Example A.24.

In case 3 we have two solutions: $y = \{\mathsf{U}_1 \mapsto \mathrm{a}, \mathsf{U}_2 \mapsto \mathrm{a}\}$ and $y = \{\mathsf{U}_1 \mapsto \mathrm{b}, \mathsf{U}_2 \mapsto \mathrm{b}\}$. Theorem A.21 does not cover this case, and we do not consider multiple solutions here. See [Luk99b] for discussions on multiple solutions.

The previous example could have been formulated using events instead of conditions. But this is a bit more complex, since well-behavedness (output triggering, to be precise) would force us to use at least two states in each DCES, or, if we have only one state, to use an external trigger event. Otherwise, we would not be able to generate any proper output event.

Now we prove Theorem A.21.

Proof. Let $\mathcal{D}_1, \ldots, \mathcal{D}_n$ be consistent, let the unique solution property hold, and let $G(x_1, \ldots, x_n, u) \in \Sigma_{\mathsf{C}^{\mathsf{io}}}$, respectively, $H(x_1, \ldots, x_n, x'_1, \ldots, x'_n, v) \in \Sigma_{\mathsf{E}^{\mathsf{io}}}$ be the unique solution y, respectively, z, for $x_1, x'_1 \in X_1, \ldots, x_n, x'_n \in X_n$, $u \in \Sigma^{\mathsf{in}}_{\mathsf{C}}$, and $v \in \Sigma^{\mathsf{in}}_{\mathsf{E}}$.

We construct a named DCES $\mathcal{D}$ such that $\mathcal{S}_\mathcal{D}$ is a parallel interconnection of $\mathcal{S}_{\mathcal{D}_1}, \ldots, \mathcal{S}_{\mathcal{D}_n}$. Let $\mathcal{D} = (\bar{\mathsf{U}}, \bar{\mathsf{V}}, X, \bar{\mathsf{Y}}, \bar{\mathsf{Z}}, f, g, h, X_0)$ be the named DCES with component names $\bar{\mathsf{U}} = \mathsf{C}^{\mathsf{in}}$, $\bar{\mathsf{V}} = \mathsf{E}^{\mathsf{in}}$, $\bar{\mathsf{Y}} = \mathsf{C}^{\mathsf{out}}$, $\bar{\mathsf{Z}} = \mathsf{E}^{\mathsf{out}}$, states $X = X_1 \times \ldots \times X_n$, $X_0 = X_{0,1} \times \ldots \times X_{0,n}$, and, for all $x = (x_1, \ldots, x_n) \in X$, $x' = (x'_1, \ldots, x'_n) \in X$, $u \in \Sigma_{\bar{\mathsf{U}}}$, $v \in \Sigma_{\bar{\mathsf{V}}}$,

- $x' \in f(x, u, v)$ if and only if for all $i \in \{1, \ldots, n\}$:
  $x'_i \in f_i(x_i, (u \cup G(x, u))|_{\bar{\mathsf{U}}_i}, (v \cup H(x, x', v))|_{\bar{\mathsf{V}}_i})$,
- $g(x, u) = cup_{i=1}^n g_i(x_i, (u \cup G(x, u))|_{\bar{\mathsf{U}}_i})$, and
- $h(x, x', v) = cup_{i=1}^n h_i(x_i, x'_i, (v \cup H(x, x', v))|_{\bar{\mathsf{V}}_i})$.

Condition 1 of Definition A.18 holds obviously by Definition A.20. For proving condition 2, let $(s_{\bar{\mathsf{U}}}, s_{\bar{\mathsf{V}}}, s_{\bar{\mathsf{Y}}}, s_{\bar{\mathsf{Z}}}) \in \mathrm{Beh}(\bar{\mathsf{U}}, \bar{\mathsf{V}}, \bar{\mathsf{Y}}, \bar{\mathsf{Z}})$. We have

$$(s_{\bar{\mathsf{U}}}, s_{\bar{\mathsf{V}}}, s_{\bar{\mathsf{Y}}}, s_{\bar{\mathsf{Z}}}) \in S_\mathcal{D}$$

if and only if (by Definition A.15)

$\exists r : \mathbf{R}_{\geq 0} \to X$ (right-continuous, finite-variable) :
1. $\forall t \in \mathbf{R}_{>0} : r(t) \in f(r(t^-), s_{\bar{\mathsf{U}}}(t^-), s_{\bar{\mathsf{V}}}(t))$
2. $\forall t \in \mathbf{R}_{\geq 0} : s_{\bar{\mathsf{Y}}}(t) = g(r(t), s_{\bar{\mathsf{U}}}(t))$
3. $\forall t \in \mathbf{R}_{>0} : s_{\bar{\mathsf{Z}}}(t) = h(r(t^-), r(t), s_{\bar{\mathsf{V}}}(t))$
4. $r(0) \in X_0$

if and only if (by construction of f, g, h, and X_0 of the named DCES $\mathcal{D}$)

$\exists r : \mathbf{R}_{\geq 0} \to X$ (right-continuous, finite-variable) :
$\forall i \in \{1, \ldots, n\}$:
1. $\forall t \in \mathbf{R}_{>0} : r(t)_i \in f_i\Big(r(t^-)_i, (s_{\bar{\mathsf{U}}}(t^-) \cup G(r(t^-), s_{\bar{\mathsf{U}}}(t^-)))|_{\bar{\mathsf{U}}_i},$
$(s_{\bar{\mathsf{V}}}(t) \cup H(r(t^-), r(t), s_{\bar{\mathsf{V}}}(t)))|_{\bar{\mathsf{V}}_i}\Big)$
2. $\forall t \in \mathbf{R}_{\geq 0} : s_{\bar{\mathsf{Y}}}(t)|_{\bar{\mathsf{Y}}_i} = g_i\Big(r(t)_i, (s_{\bar{\mathsf{U}}}(t) \cup G(r(t), s_{\bar{\mathsf{U}}}(t)))|_{\bar{\mathsf{U}}_i}\Big)$
3. $\forall t \in \mathbf{R}_{>0} : s_{\bar{\mathsf{Z}}}(t)|_{\bar{\mathsf{Z}}_i} = h_i\Big(r(t^-)_i, r(t)_i,$
$(s_{\bar{\mathsf{V}}}(t) \cup H(r(t^-), r(t), s_{\bar{\mathsf{V}}}(t)))|_{\bar{\mathsf{V}}_i}\Big)$
4. $r(0)_i \in X_{0,i}$

if and only if (we split the run r into the runs $r_1, \ldots, r_n$ (we abbreviate $r_1(\cdot), \ldots, r_n(\cdot)$ by $\bar{r}(\cdot)$ in the parameters of G and H), and move the signal restrictions into the arguments of the unions)

$\exists r_i : \mathbf{R}_{\geq 0} \to X_i$, for $i \in \{1, \ldots, n\}$ (right-continuous, finite-variable) :
$\forall i \in \{1, \ldots, n\}$:
1. $\forall t \in \mathbf{R}_{>0}$:
$r_i(t) \in f_i\Big(r_i(t^-), (s_{\bar{\mathsf{U}}}(t^-)|_{\bar{\mathsf{U}}_i \cap \bar{\mathsf{U}}} \cup G(\bar{r}(t^-), s_{\bar{\mathsf{U}}}(t^-))|_{\bar{\mathsf{U}}_i \cap \bar{\mathsf{C}}^{\mathsf{io}}}),$
$(s_{\bar{\mathsf{V}}}(t)|_{\bar{\mathsf{V}}_i \cap \bar{\mathsf{V}}} \cup H(\bar{r}(t^-), \bar{r}(t), s_{\bar{\mathsf{V}}}(t))|_{\bar{\mathsf{V}}_i \cap \bar{\mathsf{E}}^{\mathsf{io}}})\Big)$
2. $\forall t \in \mathbf{R}_{\geq 0} : s_{\bar{\mathsf{Y}}}(t)|_{\bar{\mathsf{Y}}_i} = g_i\Big(r_i(t), (s_{\bar{\mathsf{U}}}(t)|_{\bar{\mathsf{U}}_i \cap \bar{\mathsf{U}}} \cup G(\bar{r}(t), s_{\bar{\mathsf{U}}}(t))|_{\bar{\mathsf{U}}_i \cap \bar{\mathsf{C}}^{\mathsf{io}}})\Big)$
3. $\forall t \in \mathbf{R}_{>0} : s_{\bar{\mathsf{Z}}}(t)|_{\bar{\mathsf{Z}}_i} = h_i\Big(r_i(t^-), r_i(t),$
$(s_{\bar{\mathsf{V}}}(t)|_{\bar{\mathsf{V}}_i \cap \bar{\mathsf{V}}} \cup H(\bar{r}(t^-), \bar{r}(t), s_{\bar{\mathsf{V}}}(t))|_{\bar{\mathsf{V}}_i \cap \bar{\mathsf{E}}^{\mathsf{io}}})\Big)$
4. $r_i(0) \in X_{0,i}$

if and only if (by Definition A.15, and by the fact that the unique solutions G and H determine for all systems the signal components with names in C^{io} and E^{io})

$\exists (s_{\bar{\mathsf{U}}_i}, s_{\bar{\mathsf{V}}_i}, s_{\bar{\mathsf{Y}}_i}, s_{\bar{\mathsf{Z}}_i}) \in S_{\mathcal{D}_i}$ (for all $i \in \{1, \ldots, n\}$) :
(a) $\forall i, j \in \{1, \ldots, n\}$:
$s_{\bar{\mathsf{U}}_i}|_{\bar{\mathsf{U}}_i \cap \bar{\mathsf{Y}}_j} = s_{\bar{\mathsf{Y}}_j}|_{\bar{\mathsf{U}}_i \cap \bar{\mathsf{Y}}_j}$,
$s_{\bar{\mathsf{V}}_i}|_{\bar{\mathsf{V}}_i \cap \bar{\mathsf{Z}}_j} = s_{\bar{\mathsf{Z}}_j}|_{\bar{\mathsf{V}}_i \cap \bar{\mathsf{Z}}_j}$.
(b) $\forall i \in \{1, \ldots, n\}$:
$s_{\bar{\mathsf{U}}_i}|_{\bar{\mathsf{U}}_i \cap \bar{\mathsf{U}}} = s_{\bar{\mathsf{U}}}|_{\bar{\mathsf{U}}_i \cap \bar{\mathsf{U}}}, \quad s_{\bar{\mathsf{Y}}_i} = s_{\bar{\mathsf{Y}}}|_{\bar{\mathsf{Y}}_i},$
$s_{\bar{\mathsf{V}}_i}|_{\bar{\mathsf{V}}_i \cap \bar{\mathsf{V}}} = s_{\bar{\mathsf{V}}}|_{\bar{\mathsf{V}}_i \cap \bar{\mathsf{V}}}, \quad s_{\bar{\mathsf{Z}}_i} = s_{\bar{\mathsf{Z}}}|_{\bar{\mathsf{Z}}_i}$.

Thus, $\mathcal{S}_{\mathcal{D}}$ is a parallel interconnection of $\mathcal{S}_{\mathcal{D}_1}, \ldots, \mathcal{S}_{\mathcal{D}_n}$.

The construction in this proof is the basis for our parallel interconnection operator for DCESs.

Definition A.23 (Discrete parallel interconnection) Let $\{\mathcal{D}_1, \ldots, \mathcal{D}_n\}$ be consistent, and let the unique solution property of Theorem A.21 hold. The named DCES $\mathcal{D}$ gained from the construction above is called the *discrete parallel interconnection* of $\mathcal{D}_1, \ldots, \mathcal{D}_n$, denoted by $\mathcal{D}_1 \| \ldots \| \mathcal{D}_n$.

In the following example, two small DCESs are composed using the parallel interconnection operator.

Example A.24 Consider again the DCES $\mathcal{D}_1$ and $\mathcal{D}_2$ of Example A.22. We take a look at case 2, where $a_1 = \mathrm{a}$, $b_1 = \mathrm{a}$, and $a_2 = \mathrm{a}$, $b_2 = \mathrm{a}$. As stated already, the unique solution property of Theorem A.21 holds, and the solutions are $G(x, \emptyset) = \{\mathsf{U}_1 \mapsto \mathrm{a}, \mathsf{U}_2 \mapsto \mathrm{a}\}$ and $H(x, x, \emptyset) = \emptyset$. So we can define

$$\mathcal{D} = \mathcal{D}_1 \| \mathcal{D}_2,$$

and by Definition A.23 above and the construction in the proof of Theorem A.21, we get the DCES $\mathcal{D} = (\emptyset, \emptyset, \{(x, x)\}, \bar{\mathsf{Y}}, \emptyset, f, g, h, \{(x, x)\})$ with $\bar{\mathsf{Y}} = \{\mathsf{U}_1, \mathsf{U}_2\}$, $\alpha(\mathsf{U}_1) = \alpha(\mathsf{U}_2) = \{\mathrm{a}, \mathrm{b}\}$, and

- $f((x, x), \emptyset, \emptyset) = \{(x, x)\}$,
- $g((x, x), \emptyset) = \{\mathsf{U}_1 \mapsto \mathrm{a}, \mathsf{U}_2 \mapsto \mathrm{a}\}$, and
- $h((x, x), (x, x), \emptyset) = \emptyset$.

Thus, $\mathcal{D}$ has one state, no inputs, and two condition output components named U_1 and U_2; both deliver constantly the condition symbol a.

There is only one run of $\mathcal{D}$; so by Definition A.15, we have as formal semantics of $\mathcal{D}$ the CES $\mathcal{S}_{\mathcal{D}} = (\emptyset, \emptyset, \bar{\mathsf{Y}}, \emptyset, S_{\mathcal{D}})$ with

$$S_{\mathcal{D}} = \{((\emptyset, \emptyset, s_{\bar{\mathsf{Y}}}, \emptyset) \mid s_{\bar{\mathsf{Y}}}(t) = \{\mathsf{U}_1 \mapsto \mathrm{a}, \mathsf{U}_2 \mapsto \mathrm{a}\} \text{ for all } t \in \mathbf{R}_{\geq 0}\}.$$

Appendix B

Verification of Discrete Condition/Event Systems

This appendix shows how to use the model checker SMV [McM93, McM00] to verify discrete condition/event systems. A set of DCESs is translated into an SMV input file describing the parallel interconnection of these DCESs. Then the SMV model checker is used to verify properties of this system which are described as a CTL formula [CE82].

B.1 Transforming a System of DCESs into the SMV framework

Given a set of discrete condition/event systems, we transform each DCES into an SMV module. While communication between condition/event systems is achieved by using condition and event signals, SMV modules communicate via shared variables. Therefore, we introduce a variable for each event output occurring in one of the systems. This variable is local to and written by the module of the system generating the respective event, and it can be read by the modules of other systems having this event as input.

Conditions are handled differently. Since a condition output symbol only depends on a system's current state and its current condition input symbols, we do not need to introduce a variable, but use SMV's `DEFINE` declaration to describe the mapping of the state and the input symbols to the output symbol in a direct way. As an advantage of this construction, no variable is needed to store the condition output symbol, which can significantly decrease memory and time consumption during the model checking process. The `DEFINE` declaration will be given in the module of the system generating the respective condition, and the modules of systems having this condition as input can access it.

B.1.1 Conversion of Identifiers

The syntax of SMV only allows letters (`a`, ..., `z`, `A`, ..., `Z`), digits (`0`, ..., `9`), the hyphen (`-`), and the underscore symbol (`_`) in its identifiers, so some conversion might be necessary when transforming a DCES into SMV code. In the following we implicitly assume that such a conversion is performed when needed. This will be the case if a non-`typewriter` font face appears in the SMV code shown.

B.1.2 Translating a DCES into an SMV Module

Let $\mathcal{D} = (\bar{\mathsf{U}}, \bar{\mathsf{V}}, X, \bar{\mathsf{Y}}, \bar{\mathsf{Z}}, f, g, h, X_0)$ be a named DCES with

$$\bar{\mathsf{U}} = \{\mathsf{U}_1, \ldots, \mathsf{U}_{n_U}\}, \bar{\mathsf{V}} = \{\mathsf{V}_1, \ldots, \mathsf{V}_{n_V}\}, X = \{x_1, \ldots, x_n\},$$
$$\bar{\mathsf{Y}} = \{\mathsf{Y}_1, \ldots, \mathsf{Y}_{n_Y}\}, \bar{\mathsf{Z}} = \{\mathsf{Z}_1, \ldots, \mathsf{Z}_{n_Z}\}, X_0 = \{x_{0,1}, \ldots, x_{0,m}\},$$

and some identifier *Name* assigned to the system.

First we generate the module header. The name of the module is composed by appending `_module` to *Name*. The parameters of the module are all input names of $\mathcal{D}$.

`MODULE` *Name*`_module(`U_1`,...,`U_{n_U}`,`V_1`,...,`V_{n_V}`)`

If there are no parameters, i.e., $\bar{\mathsf{U}} = \bar{\mathsf{V}} = \emptyset$, we have to omit the parentheses.

We need one local variable `state` to store the control state of the system:

```
VAR
  state: {x_1,...,x_n};
```

For each of the output event names of $\mathcal{D}$ we introduce one variable storing the symbol of that component:

```
VAR
  Z_1: {symbols(Z_1)};
  ⋮
  Z_{n_Z}: {symbols(Z_{n_Z})};
```

Here *symbols*(N) denotes a comma separated list of the symbols in $\alpha(\mathsf{N})$. If N is an event name (here this is always the case), this list includes `0`, denoting the null symbol $\mathbf{0}$. If there are no output events ($\bar{\mathsf{Z}} = \emptyset$), we omit this `VAR` declaration.

The variable `state` is initialized non-deterministically with one of the initial states:

```
INIT
  state in {x_{0,1},...,x_{0,m}}
```

All output event variables are initialized with the null symbol:

```
INIT
  Z_1 = 0 & ... & Z_{n_Z} = 0
```

If there are no event outputs ($\bar{\mathsf{Z}} = \emptyset$), we omit this `INIT` declaration.

For each condition output name Y_i, $i \in \{1, \ldots, n_Y\}$, we examine g to generate a `DEFINE` declaration computing the condition output symbol:

```
DEFINE
  Y_1 :=
    case state=x_0: ... ;
         ⋮
         state=x_n: ... ;
    esac;
  ⋮
  Y_{n_Y} :=
    case state=x_0: ... ;
         ⋮
         state=x_n: ... ;
    esac;
```

The "..." above must be replaced by SMV expressions (consisting of further `case ... esac` constructs reading the values of the variables $\mathsf{U}_1, \ldots, \mathsf{U}_{n_U}$) computing the respective condition output symbol for Y_i.

If for some $i \in \{1, \ldots, n_Y\}$ and $j \in \{1, \ldots, n\}$ the condition output symbol for Y_i at state x_j does not depend on the condition inputs (or there are no condition inputs at all, i.e., $\bar{\mathsf{U}} = \emptyset$), then $g(x_j, \cdot)(\mathsf{Y}_i)$ is constant, and we can simply replace the "..." above by this constant. Figure B.5 on page 143 illustrates this. If there are no condition outputs ($\bar{\mathsf{Y}} = \emptyset$), we omit this `DEFINE` declaration.

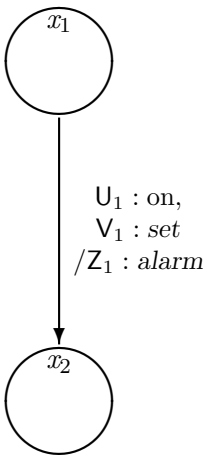
Finally, we have to implement the transition function f and the event output function h. We make use of SMV's `TRANS` declaration which describes a transition relation between the current variable assignments and the variable assignments after one transition step. A transition step in SMV models a transition of the DCES in the following way: The variable assignments before the step model the control state and the condition input symbols at time t^-, and the variable assignments after the step describe the control state and the events at time t.

Thus, we can access the control state at time t^- by using `state`, a condition input symbol at time t^- by using its name U_i, and for time t, we access the state by using `next(state)` and an event with name N by using `next(N)`. Figure B.1 shows the resulting `TRANS` declaration.

The inner block from "`(`U_1 `=` u_1" to "`) |`" has to be written for all $u_1 \in \alpha(\mathsf{U}_1)$, ..., $u_{n_U} \in \alpha(\mathsf{U}_{n_U})$, $v_1 \in \alpha(\mathsf{V}_1)$, ..., $v_{n_V} \in \alpha(\mathsf{V}_{n_V})$, and $x_1' \in f(x_1, \{\mathsf{U}_1 \mapsto u_1, \ldots, \mathsf{U}_{n_U} \mapsto u_{N_U}\}, \{\mathsf{V}_1 \mapsto v_1, \ldots, \mathsf{V}_{n_V} \mapsto v_{N_V}\})$. Analogously this has to be done for the states $x_2, \ldots, x_n$.

```
TRANS
  (state = x_1) &
  ( (U_1 = u_1 &
     ⋮
     U_{n_U} = u_{n_U} &
     next(V_1) = v_1 &
     ⋮
     next(V_{n_V}) = v_{N_V} &
     next(state) = x'_1 &
     next(Z_1) = h(x_1, x'_1, {V_1 ↦ v_1, ..., V_{n_V} ↦ v_{N_V}})(Z_1) &
     ⋮
     next(Z_{n_Z}) = h(x_1, x'_1, {V_1 ↦ v_1, ..., V_{n_V} ↦ v_{N_V}})(Z_{n_Z}) &
    ) |
    ⋮
  ) |
  ⋮
  (state = x_n) &
  (
   ⋮
  )
```

Figure B.1: The TRANS declaration representing the functions f and h




```
TRANS
  ⋮
  (state = x_1) &
  (
    ⋮
   (U_1 = on &
    next(V_1) = set &
    next(state) = x_2 &
    next(Z_1) = alarm
   ) |
   ⋮
```

Figure B.2: A DCES transition and its SMV representation

Example B.1 Figure B.2 shows a transition of a DCES and the resulting piece of SMV code.

B.1.3 Combining the SMV Modules

Now all modules have to be composed into one interconnected system. We connect the modules in same way as the parallel interconnection in Section A.3 does: If a module M generates the output for a name N, all other modules having N as an input read the local variable N of M to obtain their input value.

If there are inputs for which there exists no matching output in the system, these inputs are considered to be external, i.e., their values must be provided by some environment. In SMV we define global variables for these external inputs; by default these can take on any value of their respective domains. This leads to a "chaotic" behavior of the environment, which is a typical test situation for open systems. Alternatively, we can use some additional SMV code to control these variables if we want to model a more restrictive behavior of the environment.

We do not have to introduce global variables for outputs, since these are stored in the modules' local variables. If we need to access them for model checking purposes, they can be read directly in a CTL formula (cf. Example B.2 on page 142).

Let $\mathcal{D}_i = (\bar{\mathsf{U}}_i, \bar{\mathsf{V}}_i, X_i, \bar{\mathsf{Y}}_i, \bar{\mathsf{Z}}_i, f_i, g_i, h_i, X_{0,i})$ be a named DCES, identified as $Name_i$, for $i \in \{1, \ldots, n\}$. Like in Definition A.20 we define the external inputs to our system:

$$\mathsf{C}^{\mathsf{in}} = \left(\bigcup_{i=1}^{n} \bar{\mathsf{U}}_i\right) \setminus \left(\bigcup_{i=1}^{n} \bar{\mathsf{Y}}_i\right) \text{ and } \mathsf{E}^{\mathsf{in}} = \left(\bigcup_{i=1}^{n} \bar{\mathsf{V}}_i\right) \setminus \left(\bigcup_{i=1}^{n} \bar{\mathsf{Z}}_i\right).$$

We number the elements of these sets like this:

$$\mathsf{C}^{\mathsf{in}} = \{\mathsf{C}^{\mathsf{in}}_1, \ldots, \mathsf{C}^{\mathsf{in}}_l\} \text{ and } \mathsf{E}^{\mathsf{in}} = \{\mathsf{E}^{\mathsf{in}}_1, \ldots, \mathsf{E}^{\mathsf{in}}_m\}.$$

The SMV program starts with the `main` module.

```
MODULE main
```

Now we define the variables of this module. These variables are the global variables for the other modules.

We introduce a global variable for each element of C^{in} and E^{in}:

```
VAR
  C_1^in : {symbols(C_1^in)};
  ⋮
  C_l^in : {symbols(C_l^in)};
  E_1^in : {symbols(E_1^in)};
  ⋮
  E_m^in : {symbols(E_m^in)};
```

If there are no external inputs ($\mathsf{C}^{\mathsf{in}} = \mathsf{E}^{\mathsf{in}} = \emptyset$), we omit this `VAR` declaration.

Since conditions are not given as variables but only as `DEFINE` declarations, we have to introduce the condition symbols we want use in our system to SMV's syntax checker. Therefore, we define a dummy variable that lists all these symbols:

```
VAR
  conditionsymbols: {symbols(⋃_{i=1}^{n} Ū_i ∪ Ȳ_i)};
```

If there are no conditions in the system, we omit this `VAR` declaration.

Now we instantiate all the DCESs as modules. This is the place where the outputs of one module can be connected to the inputs of other modules. We define a helpful function

$$Prod(\mathsf{N}) = \begin{cases} \mathsf{N}, \text{ if } \mathsf{N} \in \mathsf{C}^{\mathsf{in}} \cup \mathsf{E}^{\mathsf{in}} \\ Name_i\,.\,\mathsf{N}, \text{ if } \mathsf{N} \in \bar{\mathsf{Y}}_i \cup \bar{\mathsf{Z}}_i \text{ for an } i \in \{1, \ldots, n\} \end{cases},$$

which, given a name N, either delivers this name itself if N is an external input, or composes the identifier of the DCES producing the signal named N, a dot, and N. Thus, *Prod* is an SMV expression pointing either to the global variable we defined for that component, or giving us access to the local variable of the module containing the output symbol we are looking for.

Now we can instantiate all the modules, and let *Prod* do the work of making the right connections:

```
VAR
  Name_1: Name_1_module(Prod(U^1_1), ..., Prod(U^1_{n^1_U}),
                        Prod(V^1_1), ..., Prod(V^1_{n^1_V}));
  ⋮
  Name_n: Name_n_module(Prod(U^n_1), ..., Prod(U^n_{n^1_U}),
                        Prod(V^n_1), ..., Prod(V^n_{n^n_V}));
```

with $\bar{\mathsf{U}}_i = \{\mathsf{U}^i_1, \ldots, \mathsf{U}^i_{n^i_U}\}$ and $\bar{\mathsf{V}}_i = \{\mathsf{V}^i_1, \ldots, \mathsf{V}^i_{n^i_V}\}$, for $i \in \{1, \ldots, n\}$. If $\mathcal{D}_i$ has no inputs ($\bar{\mathsf{U}}_i = \bar{\mathsf{V}}_i = \emptyset$), its module has no parameters, and we have to omit the parentheses as well.

The last thing we have to do is to initialize the external input events with the null event:

```
INIT
  E^in_1 = 0 & ... & E^in_m = 0
```

If there are no external input events ($\mathsf{E}^{\mathsf{in}} = \emptyset$), we omit this `INIT` declaration.

This completes the transformation of a set of DCESs into an SMV input file. One or more `SPEC` declarations can now be used to provide the SMV model checker with some properties of the system we want to check. Section B.2 discusses this is detail.

B.1.4 Well-Definedness

The transformation of a set of DCESs into SMV code can be regarded as an implementation of the parallel interconnection introduced in Section A.3. However, the parallel interconnection is only well-defined if the set of DCESs is consistent (Definition A.19), and if the unique solution property (Theorem A.21) or a multiple solution property holds.

Consistency can be checked easily by inspecting the systems' output names. The unique/multiple solution property, which must be fulfilled to avoid algebraic loops in the system, is much more difficult to check. Some algebraic loops can be detected by SMV. Since we implement conditions and events differently in SMV, we look at them separately when checking the unique/multiple solution property.

Condition outputs are given as `DEFINE` declarations, and SMV allows no circular definitions there. Since algebraic loops are always based on a circular definition, all algebraic loops using condition signals are detected by SMV, and the input file will be rejected with a "recursively defined" error message. Even if there is a circular definition without an algebraic loop, like in the cases 2 and 3 of Example A.22, SMV will reject the input file. This is no severe restriction, since in most systems the condition output does not depend on the condition input, but is only an abstraction of the system's control state. In this case there are no condition-based algebraic loops.

Event outputs are given within the `TRANS` declaration. If there is an algebraic loop based on events, the transition relation generated by SMV is not total, i.e., for some inputs and some control states of the modules, there is no successor state, and a deadlock occurs. SMV detects such a deadlock state only if this state is unavoidable, i.e., it is eventually reached no matter how nondeterministic choices (including external inputs) are made. For example, if an external input event Update : 1 causes an algebraic loop, and Update is not fixed by some SMV construct, SMV does not generate the Update : 1 event, and the deadlock state remains undetected. To be on the safe side, checking the unique or multiple solution property (beforehand or using SMV) is recommended.

B.1.5 A Complete Example

We illustrate the translation of a DCES into SMV code using the DCES *Switch* already used in Section A.3.2.

The functionality of *Switch*, defined by Figure B.4, is as follows: The system is either in the control state x_0 or x_1, and this information is passed to the environment by the condition output symbol Status : off or Status : on. Initially, *Switch* is at state x_0. If the environment wants to change the state of *Switch* to x_1, it sets the condition Request : start and sends the event Update : 1, which immediately triggers the state change in *Switch*.

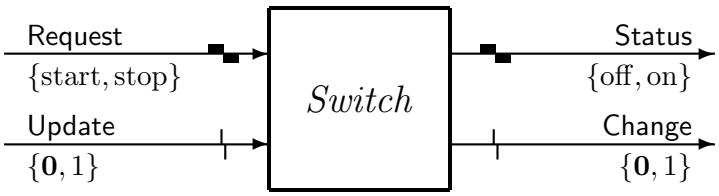


Figure B.3: Block diagram of *Switch*

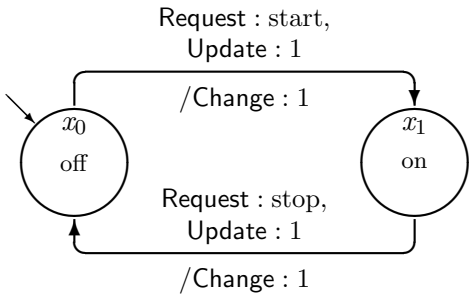


Figure B.4: The transition system of *Switch*

Analogously, Request : stop and Update : 1 will force the state transition back to x_0. Note that changing Request without sending Update : 1 does not change the state.

Whenever the state of *Switch* changes, the event Change : 1 is generated. So if Update : 1 is sent twice without a change of Request in between, there is at most one Update : 1 event.

We apply the transformation to SMV on the DCES *Switch*. This leads to the code shown in Figure B.5.

Obviously the expression representing the transition relation can be written in a more compact way.

B.2 Verification with SMV

SMV accepts CTL formulae [CE82] for checking system properties. The syntax used by SMV is given in [McM00]. Safety and liveness properties can be checked, and the evaluation of formulae can be restricted to fair paths using a second formula to describe all paths which are considered to be fair.

Example B.2 We want to verify that whenever the DCES *Switch* generates the Change : 1 event, an Update : 1 event must occur simultaneously. To verify this, we add the following lines to the module `main` of the SMV input

```
MODULE main

VAR  Request: {start,stop};
     Update: {0,1};

VAR  conditionsymbols: {off,on,start,stop};

VAR  Switch: Switch_module(Request,Update);

INIT  Update=0


MODULE Switch_module(Request,Update)

VAR  state: {x_0,x_1};

VAR  Change: {0,1};

INIT  state in {x_0}

DEFINE  Status := case
                    state=x_0: off;
                    state=x_1: on;
                  esac;

INIT  Change=0

TRANS
  (state=x_0) &
  ( (Request=start & next(Update)=0 & next(state)=x_0 & next(Change)=0) |
    (Request=stop  & next(Update)=0 & next(state)=x_0 & next(Change)=0) |
    (Request=start & next(Update)=1 & next(state)=x_1 & next(Change)=1) |
    (Request=stop  & next(Update)=1 & next(state)=x_0 & next(Change)=0)
  ) |
  (state=x_1) &
  ( (Request=start & next(Update)=0 & next(state)=x_1 & next(Change)=0) |
    (Request=stop  & next(Update)=0 & next(state)=x_1 & next(Change)=0) |
    (Request=start & next(Update)=1 & next(state)=x_1 & next(Change)=0) |
    (Request=stop  & next(Update)=1 & next(state)=x_0 & next(Change)=1)
  )
```

Figure B.5: The SMV code for the DCES *Switch*

file:

```
SPEC
  AG (Switch.Change=1 -> Update=1)
```

The output of SMV is:

```
-- specification AG (Switch.Change = 1 -> Update = 1) is true
```

Alternatively, we can put the lines above into the module `Switch_module`, but since `Change` is a local variable there, we have to write `Change=1` instead of `Switch.Change=1`.

Now we try to prove that whenever an an Update : 1 event occurs, a Change : 1 event is generated:

```
SPEC
  AG (Update=1 -> Switch.Change=1)
```

The output of SMV is:

```
-- specification AG (Update = 1 -> Switch.Change = 1) is false
-- as demonstrated by the following execution sequence
state 1.1:
Request = stop
Update = 0
conditionsymbols = stop
Switch.Status = off
Switch.state = x_0
Switch.Change = 0

state 1.2:
Update = 1
```

The execution sequence shows that if we are at state x_0, the input condition is Request : *stop*, and the event Update : 1 occurs, no Change : 1 event is generated, which is a counterexample to our given specification.

Bibliography

[ACH+95] Rajeev Alur, Costas Courcoubetis, Nicolas Halbwachs, Thomas A. Henzinger, Pei-Hsin Ho, Xavier Nicollin, Alfredo Olivero, Joseph Sifakis, and Sergio Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138(1):3–34, 1995.

[ACHH93] Rajeev Alur, Costas Courcoubetis, Thomas A. Henzinger, and Pei-Hsin Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In *Hybrid Systems*, volume 736 of *Lecture Notes in Computer Science*, pages 209–229. Springer-Verlag, 1993.

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, April 1994.

[Bau00] Nanette Bauer. A demonstration plant for the control and scheduling of multi-product batch operations. Available at http://www-verimag.imag.fr/VHS/CS7/cs7descr.ps, 2000.

[Bau03] Nanette Bauer. *Formale Analyse von Sequential Function Charts*. PhD thesis, Lehrstuhl für Anlagensteuerungstechnik, Universität Dortmund, 2003.

[BG90] Albert Benveniste and Paul Le Guernic. Hybrid dynamical systems theory and the SIGNAL language. *IEEE Transactions on Automatic Control*, 35(5):535–546, May 1990.

[BG92] Gérard Berry and Georges Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, November 1992.

[BGL+00] Saddek Bensalem, Vijay Ganesh, Yassine Lakhnech, César Muñoz, Sam Owre, Harald Rueß, John Rushby, Vlad Rusu, Hassen Saïdi, N. Shankar, Eli Singerman, and Ashish Tiwari. An overview of SAL. In C. Michael Holloway, editor, *LFM*

2000: Fifth NASA Langley Formal Methods Workshop, pages 187–196, Hampton, VA, USA, June 2000. NASA Langley Research Center.

[BHL02] Nanette Bauer, Ralf Huuck, and Ben Lukoschus. A stopwatch semantics for hybrid controllers. In *b '02: The XV. IFAC World Congress, Barcelona, Spain, July 21–26, 2002*, 2002.

[BHLL00a] Sébastien Bornot, Ralf Huuck, Yassine Lakhnech, and Ben Lukoschus. An abstract model for sequential function charts. In René Boel and Geert Stremersch, editors, *Discrete Event Systems: Analysis and Control, Proceedings of WODES 2000: 5th Workshop on Discrete Event Systems, Ghent, Belgium, August 21–23, 2000*, The Kluwer International Series in Engineering and Computer Science, pages 255–264, Boston, Dordrecht, London, 2000. Kluwer Academic Publishers.

[BHLL00b] Sébastien Bornot, Ralf Huuck, Yassine Lakhnech, and Ben Lukoschus. Verification of sequential function charts using SMV. In Hamid R. Arabnia, editor, *PDPTA 2000: International Conference on Parallel and Distributed Processing Techniques and Applications, Monte Carlo Resort, Las Vegas, Nevada, USA, June 26–29, 2000*, volume V, pages 2987–2993. CSREA Press, June 2000.

[BHLL04] Nanette Bauer, Ralf Huuck, Sven Lohmann, and Ben Lukoschus. Sequential Function Charts: Die Notwendigkeit formaler Analyse. *atp – Automatisierungstechnische Praxis*, 2004. Accepted for publication.

[BKSL00] Nanette Bauer, Stefan Kowalewski, Guido Sand, and Thomas Löhl. A case study: Multi product batch plant for the demonstration of control and scheduling problems. In Sebastian Engell, Stefan Kowalewski, and Janan Zaytoon, editors, *ADPM 2000: The 4th International Conference on Automation of Mixed Processes: Hybrid Dynamic Systems, Dortmund, Germany, September 18–19, 2000*, Berichte aus der Automatisierungstechnik, pages 383–388, Aachen, Germany, 2000. Universität Dortmund, Lehrstuhl Anlagensteuerungstechnik, Shaker Verlag.

[CE82] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons for branching time temporal logic. In Dexter Kozen, editor, *Logics of Programs Workshop, IBM Watson Research Center, Yorktown Heights, New York,*

May 1981, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer-Verlag, 1982.

[CGP00] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, January 2000.

[CPHP87] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John Plaice. LUSTRE: a declarative language for programming synchronous systems. In *14th ACM Symposium on Principles of Programming Languages*, München, January 1987.

[DA92] René David and Hassane Alla. *Petri Nets and Grafcet: Tools for Modelling Discrete Event Systems*. Prentice Hall, June 1992.

[Dij00] Edsger W. Dijkstra. The end of computing science? *Communications of the ACM*, 44(3):92, 2000.

[dRdBH+01] Willem-Paul de Roever, Frank de Boer, Ulrich Hannemann, Jozef Hooman, Yassine Lakhnech, Mannes Poel, and Job Zwiers. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*. Number 54 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, UK, November 2001.

[EKKP95] Sebastian Engell, Stefan Kowalewski, Bruce Krogh, and Jörg Preußig. Condition/event systems: A powerful paradigm for timed and untimed discrete models of technical systems. In Felix Breitenecker and Irmgard Husinsky, editors, *EUROSIM '95, Vienna, Austria, September 11–15, 1995*, pages 421–426. Elsevier, 1995.

[FMS01] Scott Fluhrer, Itsik Mantin, and Adi Shamir. Weaknesses in the key scheduling algorithm of RC4. In *8th Workshop on Selected Areas in Cryptography*, number 2259 in Lecture Notes in Computer Science, pages 1–24. Springer-Verlag, August 2001.

[Fre23] Gottlob Frege. Logische Untersuchungen. Dritter Teil: Gedankengefüge. *Beiträge zur Philosophie des Deutschen Idealismus*, 3(1):36–51, 1923. English translation in [Fre77].

[Fre77] Gottlob Frege. Compound thoughts. In P. Geach and N. Black, editors, *Logical Investigations*. Blackwells, Oxford, 1977.

[FSE+01] Goran F. Frehse, Olaf Stursberg, Sebastian Engell, Ralf Huuck, and Ben Lukoschus. Verification of hybrid controlled processing systems based on decomposition and deduction.

In *ISIC 2001: 2001 IEEE International Symposium on Intelligent Control, Mexico City, Mexico, September 5–7, 2001*, pages 150–155. IEEE Control Systems Society, IEEE Press, 2001.

[FSE+02] Goran Frehse, Olaf Stursberg, Sebastian Engell, Ralf Huuck, and Ben Lukoschus. Modular analysis of discrete controllers for distributed hybrid systems. In *b '02: The XV. IFAC World Congress, Barcelona, Spain, July 21–26, 2002*, 2002.

[GGB87] Thierry Gauthier, Paul Le Guernic, and Löic Besnard. Signal, a delarative language for synchronous programming of real-time systems. In *Proc. 3rd Conf. on Functional Programming Languages and Computer Architectures*, volume 274. Springer-Verlag, 1987.

[Hal93] Nicolas Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic Publishers, 1993.

[HHWT97] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. HyTech: a model checker for hybrid systems. *International Journal on Software Tools for Technology Transfer*, 1:110–122, 1997.

[HLFE02] Ralf Huuck, Ben Lukoschus, Goran Frehse, and Sebastian Engell. Compositional verification of continuous-discrete systems. In S. Engell, G. Frehse, and E. Schnieder, editors, *Modelling, Analysis and Design of Hybrid Systems*, volume 279 of *Lecture Notes in Control and Information Sciences*, pages 225–244. Springer-Verlag, 2002.

[HLL01] Ralf Huuck, Ben Lukoschus, and Yassine Lakhnech. Verifying untimed and timed aspects of the experimental batch plant. *European Journal of Control*, 7(4):400–415, September 2001. Special Issue: Verification of Hybrid Systems – Results of a European Union Esprit Project.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985.

[Huu03] Ralf Huuck. *Software Verification for Programmable Logic Controllers*. PhD thesis, Christian-Albrechts-Universität zu Kiel, Germany, April 2003. http://e-diss.uni-kiel.de/diss_726/.

[IEC92] International Electrotechnical Commission, Technical Committee No. 848. *IEC 60848: Preparation of function charts for control systems*, 1992.

[IEC98] International Electrotechnical Commission, Technical Committee No. 65. *Programmable Controllers – Programming Languages, IEC 61131-3*, second edition, November 1998. Committee draft.

[Int94] Pentium® processors: Statistical analysis of floating point flaw. Intel white paper, Intel Corporation, November 1994.

[Jon83] Cliff B. Jones. Tentative steps towards a development method for interfering programs. *ACM Transactions on Programming Languages and Systems*, 5(4):596–619, 1983.

[Kow98] Stefan Kowalewski. Description of VHS case study 1 "experimental batch plant", July 1998. Draft.

[KS98] Stefan Kowalewski and Olaf Stursberg. The batch evaporator: A benchmark example for safety analysis of processing systems under logic control. In *WODES '98: 4th International Workshop on Discrete Event Systems, Cagliari, Italy, August 26–28, 1998*, pages 307–307, London, 1998. IEE, IEE Publishing.

[Lam94] Leslie Lamport. *LaTeX: A Document Preparation System*. Addison-Wesley Publishing Company, second edition, 1994.

[Lev95] Nancy G. Leveson. *Safeware: System Safety and Computers*. Addison-Wesley, 1995.

[LL+96] Jacques-Louis Lions, Lennart Lübeck, et al. Ariane 5: Flight 501 failure – report by the inquiry board. Press Release 33-1996, European Space Agency, Paris, July 1996.

[LPY97] Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997.

[LSVW99] Nancy Lynch, Roberto Segala, Frits W. Vaandrager, and H.B. Weinberg. Hybrid I/O automata. Technical Report CSI-R9907, Computing Science Institute, University of Nijmegen, April 1999.

[LT93] Nancy G. Leveson and Clark S. Turner. An invenstigation of the Therac-25 accidents. *IEEE Computer*, 26(7):18–41, July 1993.

[Luk99a] Ben Lukoschus. An abstract model of VHS case study 1 (experimental batch plant). Technical Report TR-ST-99-2, Chair of Software Technology, Institute of Computer Science and

Applied Mathematics, Christian-Albrechts-University of Kiel, May 1999.

[Luk99b] Ben Lukoschus. Composition and verification of condition/ event systems. Technical Report TR-ST-99-1, Chair of Software Technology, Institute of Computer Science and Applied Mathematics, Christian-Albrechts-University of Kiel, May 1999.

[MC81] Jayadev Misra and K. Mani Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, 7(7):417–426, 1981.

[McM93] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publisher, 1993.

[McM00] Kenneth L. McMillan. *The SMV system*. Carnegie Mellon University, November 2000. Manual for SMV version 2.5.4.

[Moo65] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.

[ORS92] Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.

[OY93] Alfredo Olivero and Sergio Yovine. *KRONOS: A Tool for Verifying Real-Time Systems. User's Guide and Reference Manual*. Verimag, Grenoble, France, 1993.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Symposium on Foundations of Computer Science*, pages 46–57, 1977.

[QS82] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in CESAR. In M. Dezani-Ciancaglini and U. Montanari, editors, *Proceedings of the 5th International Symposium on Programming, Turin, April 6–8, 1982*, pages 337–350. Springer-Verlag, 1982.

[SIR01] Adam Stubblefield, John Ioannidis, and Aviel D. Rubin. Using the Fluhrer, Mantin, and Shamir attack to break WEP. Technical Report TD-4ZCPZZ, AT&T Labs, August 2001.

[SK91] Ramavarapu S. Sreenivas and Bruce H. Krogh. On condition/event systems with discrete state realizations. In *Discrete*

Event Dynamic Systems: Theory and Applications 1, pages 209–236. Kluwer Academic Publishers, Boston, USA, 1991.

[VHS] Verification of Hybrid Systems – European ESPRIT long-term research project 26270. http://www-verimag.imag.fr/VHS/.

[WS04] Guy Webster and Donald Savage. Mars exploration rover mission status. JPL/NASA News Release 2004-052, February 2004.

[ZdBdR84] Job Zwiers, Arie de Bruin, and Willem-Paul de Roever. A proof system for partial correctness of dynamic networks of processes (extended abstract). In Edmund Clarke and Dexter Kozen, editors, *Logic of Programs Workshop, Carnegie Mellon University, Pittsburgh, PA, June 6–8, 1983*, volume 164 of *Lecture Notes in Computer Science*, pages 513–527. Springer-Verlag, 1984.

[Zwi89] Job Zwiers. *Compositionality and Partial Correctness*, volume 321 of *Lecture Notes in Computer Science*. Springer-Verlag, 1989.