Cristina Cerschi Seceleanu

A Methodology for Constructing Correct Reactive Systems

Turku Centre *for* Computer Science

TUCS dissertations
No 68, November 2005

A Methodology for Constructing Correct Reactive Systems

Cristina Cerschi Seceleanu

To be presented - with the permission of the Faculty of Chemical Engineering at Åbo Akademi University - for public criticism, in Auditorium 3102 at the Department of Computer Science at Åbo Akademi University, on December 9th, 2005, at 13.

Åbo Akademi University
Department of Computer Science
Lemminkäisenkatu 14 A
20520 Turku, Finland

2005

Supervisor

Academy Professor Ralph-Johan Back
Department of Computer Science
Åbo Akademi University
Lemminkäisenkatu 14 A
20520 Turku
Finland

Reviewers

Professor Thomas A. Henzinger
École Polytechnique Fédérale de Lausanne
Station 14, CH -1015 Lausanne
Switzerland

Dr. K. Rustan M. Leino
Microsoft Research
One Microsoft Way
Redmond, WA 98052
USA

Opponent

Dr. K. Rustan M. Leino
Microsoft Research
One Microsoft Way
Redmond, WA 98052
USA

ISBN 951-29-4015-9
ISSN 1239-1883

To my family

Acknowledgements

This thesis exists thanks to many people. Words are too weak to express how much everyone whom I will mention has influenced my work and life.

In the beginning of my Ph.D. studies, I was tempted to approach research in the “let’s rush and see what happens” mode. Luckily, I have had Professor Ralph Back as supervisor. I am grateful to him for enforcing quality and structure into my work. For the illuminating discussions, advice, professional help and contribution to our joint papers, he has all my gratitude.

I have been extremely fortunate that Professor Thomas Henzinger, at École Polytechnique Fédérale de Lausanne, Switzerland, and Dr. Rustan Leino, at Microsoft Research, USA, have kindly agreed to review this work. Their thorough review, visionary ideas and excellent comments have helped me improve this dissertation. I thank them for their time, generosity and effort.

Turku Centre for Computer Science (TUCS) and the Department of Computer Science at Åbo Akademi are gratefully acknowledged for the financial support provided for my studies and travels to conferences and summer schools. I am also indebted to the Academy of Finland for its financial support.

Chapter 5 of this thesis is partly built on work carried out together with Ralph and Professor Jan Westerholm. Foremost, I am much obliged to Jan for accepting to be the architect of a nice tool on top of which I could add new constructs. His ever-open door has encouraged me to visit his office and seek scientific advice.

Professor Joakim von Wright deserves the credit for making the Mechanized Reasoning group meetings so enjoyable and attractive. I thank all the members of the group for the stimulating discussions and for letting me learn from them.

To Professor Kaisa Sere I say thanks for being the feminine scientific model in our department. My gratitude goes also to Professor Johan Lilius, the head of the Department of Computer Science at Åbo Akademi, for creating an excellent working environment.

The staff of the department and of TUCS are acknowledged for their work on keeping administrative things running smoothly. Special gratitude to Christel Engblom and Nina Kivinen for promptly answering my various questions.

My research career would have been less interesting without the support given by Professor Matti Jakobsson, the rector of the University of Vaasa. I am deeply indebted to him for accepting me as a researcher within the IT Department at Vaasa University and for encouraging me to enroll in a Ph.D. program.

Luigia Petre and Ivan Porres deserve many thanks for their valuable comments during the Hybrid Systems group meetings, for bearing with my questions, and for

pointing me at the theoretical challenges of hybrid systems modeling and analysis. My work has benefitted a great deal from their generous knowledge sharing.

Professor Hannu Toivonen, the head of the Process Control Laboratory at Åbo Akademi, and Professor Henrik Saxén from the Heat Engineering Laboratory are thanked for showing interest in my work. Hannu provided me with one of the examples used in this thesis. Henrik explained how laws that govern dynamical processes are derived.

TUCS is a better place thanks to my fellow colleagues. To Orieta Celiku, my very good friend, I am grateful for being close, generous, as well as for the interesting discussions, both scientific and not, and quality time together. Her humor, wit and affection have never failed to double my joy and divide my grief.

It has been a pleasure to share the office with Viorel Preoteasa, to whom I am not able to thank enough for the professional help that he, so generously, gave me. Both Diana and Viorel are thanked for many wonderful moments.

Special thanks go to my friend Victor Bos for his good feedback on one of my papers, and also for enjoyable moments. Best wishes for their studies and gratitude for their friendship to Chang Li, Dorina Marghescu, and Luka Milovanov. My heartfelt thanks go to Alina and Dragoş Truşcan for being there for me during all these years, and for the nice time that we shared.

Colleagues like Marina Waldén, Elena Troubitsyna, Dubravka Ilic, Linda Grandell, Patrick Sibelius, Juha Plosila, Marcus Alanen, Herman Norrgrann, Sébastien Lafond, Linas Laibinis, Kim Solin, Adrian Costea, Marius Codrea, and all the others, make the best scientific environment. From them, I also get my daily dose of kindness and wisdom. Anna Mikhajlova, Leonid Mikhajlov, Rimvydas Rukšėnas, and Ion Petre have been examples of excellency in research.

The special people met in conferences and summer schools made those times unforgettable. My thanks go to Ana, Manuela, Anne-Marie, Maria, David, Jonathan, Philipp, for great moments in different places around the world.

To all my friends in Romania, Finland and elsewhere, I am deeply grateful for their closeness (often despite the distance), faith in the happy-end of my endeavors, and also for the special time with them.

I can not imagine completing this thesis without the incredible professional help, support and enthusiasm of my dear husband, Tiberiu. For the great ideas and determination that he invested in our joint paper, I am also grateful.

My mother-in-law, Geta, and my godparents, Rodica and Ionut, are thanked for their constant encouragement, affection and remarkable patience.

The unconditional love, all-times support and devotion of my parents, Marilena and Constantin, made this thesis possible. I am grateful to them for everything that they have taught me. I dedicate this thesis to my family. I also offer them “that kernel of myself that I have saved, somehow”.

Cristina Seceleanu, November 2005
Turku, Finland

Abstract

The challenges of designing *reactive systems*, which are supposed to maintain an ongoing interaction with a possibly unpredictable environment, stem from the need of ensuring the *correctness* of the design at the earliest stage possible. An increasingly appealing mathematical approach towards accomplishing this goal is *formal system construction*. While enjoying the advantages of a rigorous development process, it also frees the designer from the burden of taking into account implementation details, from the beginning.

This thesis proposes a formal *methodology* that aims at constructing correct reactive systems. Our work lies in the area of computerized systems that combine aspects of discrete control, continuous data values and real-time constraints. We address the issues from the perspective of a logical framework called *refinement calculus*. The results are described in terms of various forms of the so-called *action systems*.

Designing for reactivity assumes dealing with composability and concurrency. Targeting the correct execution of concurrent actions, we introduce a *synchronized* semantics for the parallel composition of action systems. The construct mimics the *barrier synchronization* mechanism found as a primitive in concurrent programming languages. We prove that the synchronized composition improves the modular design capabilities of our framework. This translates into being able to carry out refinements of modules, modeled by action systems, in isolation, without knowledge about the details of functionality of the other modules of the parallel environment.

Hybrid control systems are reactive systems characterized by continuous behavior interleaved with discrete control decisions. As a precursor to full formal analysis, simulation of hybrid system models can be used effectively, especially if the state space is represented *symbolically*. We present a simulation tool for *continuous action systems* (CAS), the timed extension of action systems. The simulator is implemented in *Mathematica*, a commercial computer algebra package. Our tool takes a description of any CAS as input, and provides automatically a symbolic simulation of the system, up to a given maximum time.

To cope with the concurrent behavior of hybrid systems, we extend the synchronization execution environment developed for discrete action systems, to their hybrid counterparts. The modularity results at the discrete level hold for the syn-

chronized composition of CAS, too.

Many hybrid systems are defined using *parameters*. The systems are intended to work correctly under specific parametric conditions. These relationships may be hard to find by following an intuitive approach alone. We apply the well-known *invariance rule* to the parametric reachability problem of hybrid systems modeled as CAS. We synthesize constraints on parameters that are sufficient to guarantee the safety property of a relevant hybrid system example.

When timing requirements are set on top of the functional ones, for any type of reactive system, be it discrete or hybrid, we need to find a means to cope with them, in design. This should be done regardless of the respective functional behavior. Being faithful to this viewpoint, we advance a top-down method for the incremental construction of *scheduled real-time systems*, within the refinement calculus framework. We apply the method on two well-known scheduling algorithms, namely *Deadline-Monotonic* and *Earliest-Deadline-First* policies.

A viable controller construction method is known in literature as *controller synthesis*. Synthesis is equivalent to computing the most general model of a controller that satisfies the requirements. Here, we propose a *game-based* method for the *synthesis* of *invariance* and certain *reachability* controllers.

Contents

1 Introduction **1**
1.1 Formal Development of Reactive Systems 2
1.2 Contributions of this Thesis 7
1.2.1 Behavior Control and Modularity of Reactive Systems . . 7
1.2.2 Hybrid Systems Modeling and Analysis 9
1.2.3 Correct-by-Construction Real-Time Schedulers 12
1.2.4 Controller Synthesis for Discrete Systems 13
1.3 Organization 15

2 Background **17**
2.1 Statements 17
2.2 Action Systems 22
2.3 Correctness and Refinement of Action Systems 24
2.4 Continuous Action Systems 28
2.5 A Brief Description of Mathematica 31

3 Synchronized Discrete Reactive Systems **33**
3.1 Interleaved vs. True Concurrency 34
3.2 Traditional Model of Action Systems Execution 34
3.2.1 Example: A Digital Filter 36
3.3 Synchronized Parallel Environments 40
3.3.1 Partitioned Action Systems 41
3.3.2 The Synchronization Operator ($\sharp$) 42
3.4 Module Refinement in Synchronized Environments 45
3.4.1 Refinement Example 45
3.4.2 Trace Refinement of Partitioned Action Systems 47
3.4.3 Modularity 50
3.4.4 Refinement Example Revisited 51
3.5 Summary and Related Work 51

4 Modeling Hybrid Systems **55**
4.1 Timelocking and Zenoness 55
4.2 Adorned Continuous Action Systems 56

4.3 Implementing Continuous Action Systems 58
4.3.1 Example: A Two-Tank System 59
4.4 Synchronized Hybrid Models 62
4.4.1 Parallel Composition of Hybrid Models with Discontinuities 62
4.4.2 Synchronized Continuous Action Systems 65
4.4.3 Example Revisited - Synchronized Design Approach . . . 67
4.5 Summary and Related Work 69

5 Hybrid Systems Analysis 73
5.1 Symbolic Simulation of Hybrid System Behavior 74
5.1.1 Mathematica-based Simulation of Continuous Action Systems 77
5.1.2 Linear Hybrid Models: A Nuclear Reactor Temperature Control System 78
5.1.3 Simulating the Behavior of the Temperature Control System in Mathematica 81
5.1.4 Simulation Results 82
5.1.5 Simulation of the Two-Tank Action System 86
5.2 Parameter Synthesis 90
5.2.1 Applying Deductive Synthesis on the Temperature Control System 92
5.3 Summary and Related Work 102

6 Building Uniprocessor Priority-driven Real-Time Schedulers 105
6.1 Uniprocessor Scheduling of Real-Time Tasks 106
6.2 Generic Approach 108
6.2.1 Enforcing the Required Conditions 108
6.2.2 Deriving the Final Scheduler Model by Refinement 110
6.3 Preemptible Task Model 111
6.4 Fixed-Priority Scheduling: The Deadline-Monotonic Algorithm . 113
6.4.1 Enforcing Conditions for Correct Scheduling 113
6.4.2 Trace Refinement of Continuous Action Systems 120
6.4.3 Implementing the Real-Time System 121
6.4.4 Deriving the Scheduler Component 125
6.5 Non-Preemptible Task Model 129
6.6 Dynamic-Priority Scheduling: The EDF Algorithm 130
6.6.1 Enforcing Conditions for Correct Scheduling 131
6.6.2 Validating the EDF Scheduled System 132
6.7 Summary and Related Work 137

7 Controller Synthesis for Discrete Systems 141
7.1 Game Tree Semantics of Action Systems 142
7.2 Generic Approach to Synthesis 145

7.3 Synthesis of Controllers for Invariance 146
7.3.1 Enforcing the Safety Property 146
7.3.2 Extracting the Control Strategy for Invariance 147
7.3.3 Example: A Producer-Consumer Application 149
7.3.4 The Producer-Consumer Model as an Action System . . . 151
7.3.5 Applying the Synthesis Method 152
7.4 Synthesis of Controllers for Reachability 158
7.4.1 Characterizing Enforcement of Response Properties in Action Systems 158
7.4.2 Proving Enforcement of Weak Response 162
7.4.3 Example: A Data Processing System 163
7.5 Summary and Related Work 171

8 Conclusions and Discussion **175**

9 Appendix **181**
A-1 Proof of Theorem 2 (chapter 3) 182
A-2 Proof of Theorem 3 (chapter 3) 185
A-3 Proof of Corollary 1 (chapter 3) 189
A-4 Refinement of *sys* $\mathcal{F}$ (chapter 3) 190
A-5 Computation of weakest precondition $\mathcal{T}(i).I_t$ (chapter 6) 194
A-6 Proof of $\overline{\mathcal{RTS}}'^s \sqsubseteq \overline{\mathcal{RTS}}^m$ (chapter 6) 196

Chapter 1

Introduction

Unlike *transformational systems* that have to produce specific outputs for given sets of inputs, *reactive systems* are designed to maintain an ongoing interaction with their environment. Therefore, they might be subjected to unexpected changes of input stimuli. Typical examples of reactive systems are: air traffic control systems, programs controlling mechanical devices such as trains, planes, or ongoing processes in nuclear reactors (see Figure 1.1).

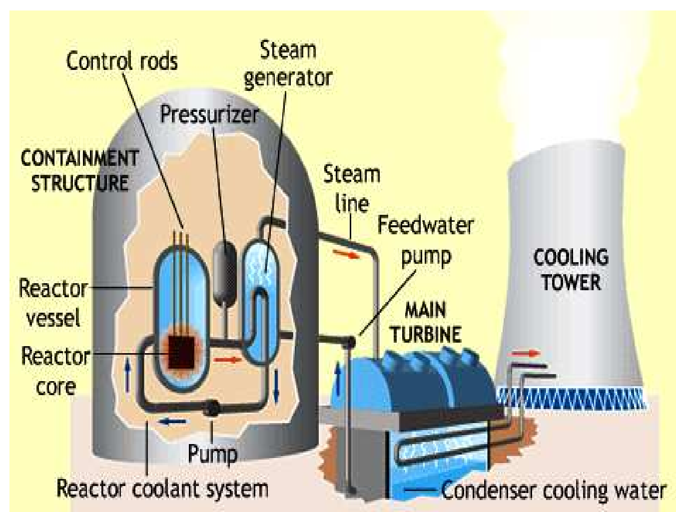


Figure 1.1: A nuclear reactor.

Many systems contain a reactive component, and *control systems* belong to this class. A control system is equipped with a *controller* that observes the state of a *plant* via *sensors*. Based on the acquired information, the controller communicates actions to the plant via *actuators*. Furthermore, reactive systems may be purely *discrete* in nature, meaning that the state space is defined by variables that are

assigned discrete values. Alternatively, they can contain a *continuous* component, usually described by real-valued variables that evolve according to some predefined laws. A mix of the two mentioned behaviors, that is, discrete and continuous, is called a *hybrid* behavior. If timing requirements are added on top of the functional ones, the respective reactive system is a *real-time* system. This means that its correct behavior depends on both the accuracy of the output result, as well as on the time at which the result is delivered.

Since malfunctioning of reactive systems can have dramatic consequences, it is most important to guarantee their *correctness*. To accomplish this goal, *formal methods* come into picture, with their synergetic paradigms, namely *program verification* and *program construction*. In the first case, correctness is established by formally proving that the program behaves according to an initial document, called the *system specification*. If one aims at *constructing* a reactive (control) program in a provably correct manner, applying *refinement* techniques is a feasible option. Refinement-based approaches implement the given system specification, as a program, after a series of transformations called refinements.

Targeting hybrid control systems construction, the two pillars of modern design, *control theory* and *computer science*, have brought tremendous progress in the area. The pair stands complementary. The first solves the *robustness* issues, by specifying adequate control laws that ensure optimal performance. On the other hand, the second answers the question of software correctness, with precision. Through mathematical modeling, numerical experiments, analytical studies and other techniques, control theory meets computer science in an attempt to produce reliable hybrid control programs.

1.1 Formal Development of Reactive Systems

A formal-method-based reactive-system construction assumes the existence of the following:

- a *mathematical formalism* in which the initial system specification is described;
- an underlying *logic*, where the required properties that the system is expected to satisfy are formulated;
- a *construction* technique, which, if applied, leads to a correct and reliable program.

Action Systems Formalism. Suitability for rigorous design methods is a crucial criterion in choosing a specification language. Such a language should serve as the basis for both modeling and rigorous reasoning. It is therefore important that the language has a close relationship with an underlying logic, but can also be grasped by software designers [110].

In this thesis, we describe discrete systems by *action systems*. The latter is a useful formalism for modeling systems, due to its expressive power and developmental clarity. It was introduced by Back and Kurki-Suonio [31, 32], who provided an action-based model of system execution. In this model, atomic actions (that is, actions that are indivisible, such that no intermediate state can be observed) can be executed whenever they are enabled, the selection among them being nondeterministic. The standard form of an action system is the following:

$$\begin{array}{rcl} \mathcal{S}ys(z : T_y) & \triangleq & \mathbf{begin\ var}\ x : T_x \bullet \\ & & \quad x := x_0\ ;\ z := z_0; \\ & & \quad \mathbf{do}\ A_1\ [\!]\ \ldots\ [\!]\ A_i\ [\!]\ \ldots\ [\!]\ A_n\ \mathbf{od} \\ & & \mathbf{end} : p \end{array}$$

Here, $A_1, \ldots, A_i, \ldots, A_n$ are the actions of $\mathcal{S}ys$, z the *global variables*, and x the *local variables*. The *parameters* p represent the constants of the model, thus they do not change their values during execution. We explain action systems in more detail, in chapter 2.

There are various execution models [62, 97, 115, 127] that are close in spirit to action systems. Out of them, Lamport's *Temporal Logic of Actions* (TLA) [115], and Chandy's and Misra's *UNITY* [62] are quite similar to action systems. UNITY and TLA use temporal logics for specification purposes.

Originally, action systems also used temporal logic for reasoning. However, later work by Back, Sere, and von Wright led to their formalization within the *refinement calculus* [22, 30, 33].

It is important to underline the fact that action systems employ the same notation for high-level specifications and their implementations.

Continuous Action Systems. As presented in the previous paragraph, discrete concurrent systems can be modeled by action systems, where a state (described by a collection of state variables) is manipulated by a collection of actions.

Continuous Action Systems (CAS) are an extension of action systems to hybrid systems, being based on a new approach to describe the state of a system. Essentially, the state variables range over functions of time, rather than just over values. The variables are expressed using *lambda abstraction*, for example $(\lambda t \cdot t - 2)$. Given the function f, we write $f.t$ for the function f applied to variable t. In our example, $f.t = t - 2$.

The CAS formalism has been introduced by Back, Petre and Porres [27]. This model allows one to describe both control actions and time advancing behavior, with the same simple mechanism. Consequently, the hybrid, and real-time systems that we are looking at, in chapters 4, 5 and 6, respectively, are modeled by CAS.

The semantics of CAS is given in terms of ordinary action systems, with explicit time, which is measured by a nonnegative real-valued variable now. In the action system translation of a CAS, the variable now is declared, initialized and

advanced, accordingly. We give the formal definition and describe the execution of CAS in chapter 2.

Clock variables (*timers*), which measure the time elapsed since they were set to zero, can be used in a CAS-based model, especially when we describe real-time systems (chapter 6).

Besides CAS, there are many other hybrid formalisms developed to support the description and analysis of real-time, or hybrid systems. Among them, *timed automata* [9], *hybrid automata* [86] and the more general framework of *hybrid input/output automata* [120] have gained high popularity.

Refinement-based Construction Technique. Coined by Dijkstra [69] and Wirth [155], *stepwise refinement* is a method for constructing programs in a provably correct manner.

The construction method based on stepwise refinement involves developing programs through one or more transformations called *refinements*. The first step is to describe the system behavior in a precise, yet abstract manner. This initial form, which is usually nondeterministic, is called the *specification*. Each refinement is then a transformation that adjusts the initial specification, by reducing its nondeterminism. There are cases in which the system behavior is described by a deterministic model, from the start; then, refinements can decrease the level of abstraction, by replacing some parts of the initial program model by more concrete ones, while preserving correctness properties.

Back proposed the *refinement calculus* [21] as a further development of stepwise refinement, based on Dijkstra's *weakest precondition* semantics [70, 71] for the language of *guarded commands*. In Dijkstra's view, the meaning of a guarded command is defined by its weakest precondition. The latter is a predicate computed by a function denoted wp. This function takes as its first argument a program statement, and as its second a predicate, called the *postcondition*. Then, $wp(S, q)$, or $wp.S.q$ denote the weakest precondition of S to establish q. It represents the largest set of initial states σ, such that S executed in σ is guaranteed to terminate in a state that satisfies q.

Since the meaning of a program statement is interpreted as a predicate over the program state, it follows that the properties that the program is required to satisfy can be formulated as predicates. When needed, the requirements can also be *temporal properties*, initially defined by Back and von Wright, within the dually nondeterministic weakest precondition framework [37, 38].

Later work by Back and von Wright [35] extended and improved Back's original work on the refinement calculus. The underlying logic of refinement calculus is *higher-order logic*, which allows for quantification over functions, a very useful feature when reasoning about complex behaviors. Research on refinement calculus has been independently carried out by Morris [131], and Morgan [129], too. The approach to program refinement promoted by Morgan [130] is concise and *calculational*: an initial abstract specification is transformed towards an im-

plementation through *refinement laws*. Early work of Hoare, who introduced the notions of *program correctness* [95] and *data refinement* [96], has also influenced the development of the refinement calculus framework.

Formalisms like the *Vienna Development Method* (VDM) [102] and the *B method* [3] propose a different approach to proving refinements of system models. One needs to first guess a new program (model), and then verify whether the created model is indeed a refinement of the previous one. This is called a *verification*-based formal approach.

a. Algorithmic Refinement. In this thesis, each statement S is identified with a specific *predicate transformer* from postconditions to weakest preconditions. Hence, one can write $S.q$ instead of $wp(S, q)$, or $wp.S.q$.

The refinement calculus introduces a *refinement relation* between statements. The relation is defined in terms of ordering on predicates. Thus S is refined by S', denoted $S \sqsubseteq S'$, if and only if:

$$\forall q \bullet S.q \subseteq S'.q$$

This refinement relation models *algorithmic refinement*, which preserves *total correctness* between program statements on the same state space. Most often, this type of refinement decreases the statement's S degree of nondeterminism.

The refinement relation is a *preorder* (that is, a reflexive and transitive relation). Due to the transitivity property, programs can be developed by a series of refinement steps:

$$S_0 \sqsubseteq S_1 \sqsubseteq \ldots \sqsubseteq S_n$$

since $S_0 \sqsubseteq S_n$ follows from the above.

b. Trace Refinement of Action Systems. The weakest precondition semantics of action systems does not suffice for capturing reactive behavior. A system's interaction with its environment subsumes the necessity of reasoning about sequences of global states. Thus, the semantics of a reactive action system is given in terms of *behaviors* [33], which can also model nonterminating execution. A behavior of an action system,

$$b = (x_0, z_0), (x_1, z_1) \ldots,$$

is a (possibly infinite) sequence of states, where each state has two components. The first component is the *local state* and the second is the *global state*. A *trace* of a behavior is obtained by removing the local state component in each state of a given system, and all finite stuttering (no change of the visible state).

In a general, less formal manner, we say that an action system $\mathcal{C}$ refines $\mathcal{A}$, written as $\mathcal{A} \sqsubseteq \mathcal{C}$, if each trace in $\mathcal{C}$ is a refinement of a trace in $\mathcal{A}$. In practice, we use a special lemma to prove *trace refinement* of action systems [29, 33]. The lemma is presented in chapter 2.

Clearly, trace refinement is more powerful than algorithmic refinement, since it can change the state space of the system. For example, one might benefit from

using an abstract, nonimplementable data structure, in the initial system model. However, this data structure should be further transformed into a more concrete, efficient representation, which can be implemented in a standard programming language. The necessary transformations that lead to an implementable data structure are performed with respect to an *abstraction relation*, which relates the initial data representation to the final one. This type of refinement, called *data refinement*, can be viewed as a particular case of trace refinement.

If, on the other hand, one needs to add functionality to the initial system specification, new actions will have to describe the respective behavior. These actions are expected to implement the stuttering actions, in the initial description, with respect to global variables. Besides introducing new actions, some of the old actions may be refined, such that the behavioral nondeterminism is reduced. This refinement technique is also a special case of trace refinement, being known as *superposition refinement*.

In short, superposition refinement of action systems reduces to the following steps:

- Replacing the initial variables x by an extended list of variables, e.g. $y = x, x'$.
- Changing the initialization of the action system, such that all variables in y are assigned initial values.
- Replacing actions A_i by actions C_i, which are (algorithmic) refinements of the old actions with respect to x, that is,

  $$A_i \sqsubseteq \mathbf{begin}\ x' \bullet C_i\ \mathbf{end},\ \text{for some i},$$

  or adding new actions that do not modify variables in x, that is,

  $$\mathsf{skip} \sqsubseteq \mathbf{begin}\ x' \bullet C_j\ \mathbf{end},\ \text{for some j},$$

  where skip is the statement that models stuttering.
- Introducing a continuation condition, which establishes that any enabled action in the original action system has a corresponding enabled action C_i, or C_j in the resulting system.
- Introducing a termination condition of C_j, which ensures that the execution of any new action, taken separately, terminates eventually.

In case we do not change any data representation, and we also do not superimpose functionality, but rather need to just decrease nondeterminism of actions, then we run into classical algorithmic refinement.

As the informal definitions outlined in this paragraph suggest, trace refinement is a comprehensive form of refinement, which can be specialized, on demand, into data, superposition or plain algorithmic refinement.

Since continuous action systems are special cases of ordinary action systems, we can use the proof and refinement techniques developed for ordinary action systems, for CAS, as well. This allows us to prove correctness of transformations of timed and hybrid models.

1.2 Contributions of this Thesis

The current study presents a methodology for building correct programs for reactive systems, be they discrete systems, hybrid control systems, or real-time schedulers. The methodology comprises several techniques for solving the construction problem, tailored to the nature of the target systems. Both *closed* and *open* reactive systems are considered. By closed systems we mean systems that can be viewed in isolation; their behavior can not be influenced from outside. We treat closed reactive systems where there is a clear distinction between the reactive program and the rest of the system. Nonetheless, we also look at compound closed systems, in which the reactive behavior is not modeled separately. In contrast, the behavior of open reactive systems is influenced by the external environment. Systems that comprise several components that interact with each other are open. Each component taken apart is an open reactive system. We analyze aspects of behavior control and modularity of such reactive systems, which, in addition, are required to respond concurrently to a set of inputs.

Our contribution uses action systems and its hybrid/real-time extension, continuous action systems, as the modeling languages for the discrete, hybrid, and real-time systems that we consider, respectively. The reasoning environment is the refinement calculus, with its correctness-preserving refinement techniques, which we have briefly described in section 1.1.

Even though the present work spans over untimed, and also timed and hybrid systems, we have concentrated on solving the intrinsic, specific construction problems of each class of systems. By exploiting the richness of our favorite formalism at the modeling stage, we carry out the program construction, where needed, under the common umbrella of the established refinement methods. The following sections emphasize the problems that we are analyzing and the main lines of our contribution.

1.2.1 Behavior Control and Modularity of Reactive Systems

Problem Description. To cope with the complexity of reactive system design, *modular reasoning* is a necessity. The entire system model is then composed from smaller parts, the *modules*, which is desirable to be developed independently. This desideratum should be supported by adequate rules and techniques, which guarantee that separate module transformations do not alter the entire system correctness. Moreover, equipped with such reasoning techniques, one may reuse previously designed modules, in a different, yet similar setup.

As detailed in chapter 2, *parallel action systems* are executed by interleaving enabled actions. This model of execution is strongly connected with behavioral nondeterminism. In chapter 3, we exemplify that the interleaved model of concurrency may not suffice, as such, for modeling parallel reactive systems. To ensure the expected behavior, one has to model some sort of communication protocol

among modules. This, in turn, makes the overall system model cumbersome and expensive to implement.

Solution. We propose a solution to the above problem, in the form of a *synchronization mechanism* (that mimics *barrier synchronization* [84]), implying a new virtual execution model for action systems, applicable to both discrete and hybrid designs. The extension to hybrid systems represented as CAS is carried out in chapter 4.

We define the necessary formal concepts, and prove modular refinement results, for the *synchronized parallel action systems*, which is the actual environment that we introduce in chapter 3. By employing it, we eliminate intermediate results that can affect the global state, as the system gives complete answers to the input stimuli. One of the main advantages is the fact that we preserve the internal nondeterminism, thus we allow the modules to execute in any order, yet increasing the external determinacy. Concretely, this means that the observable state is the same, after an execution cycle, regardless of the internal execution order during rounds. The main results of chapter 3 are the following:

- A barrier synchronization mechanism applied to both discrete and hybrid models (action systems and continuous action systems). The mechanism is suitable for designing reactive systems that have to present a simultaneous global response to sets of input stimuli. To achieve this, we introduce a new parallel composition operator (sharp, $\sharp$) that ensures correct outputs to all sets of inputs, without employing communication channels between modules. Consequently, using our mechanism bears the advantage of less coding effort, in practice. The new execution model requires a certain type of action systems that we call *partitioned action systems*, which separate local actions from global actions.
- Proofs of the usefulness of our synchronized parallel environment, with respect to modular design. We show that the capabilities of the action systems framework, for modularity, are improved. This translates into being able to carry out (trace) refinements of modules, in isolation, without knowledge about the invariants of the other modules of the parallel environment. However, the price to pay is that one has to use *proper* invariants for trace refinement. Theorem 3, and Corollary 1 of chapter 3 demonstrate these informal claims.

The most important contribution of chapter 3 is the proof that barrier synchronization can also improve reactive systems modularity, besides their control. We believe that the proposed parallel execution model and its properties could be extended to other state-based formalisms, if one needs to obtain similar benefits.

Related Work. The approximation of concurrency by interleaving is used in most process algebras like CSP [97], CCS [127], as well as in input-output automata [121] and UNITY [62]. The nondeterministic behavior induced by the in-

terleaved model requires solutions for controlling the data flow. On the other hand, resolving control issues reduces the design independence across the different levels of the design process. Several recent studies have analyzed aspects of control and / or composability within different formal frameworks, all of which deal with a certain interleaved environment.

Cavalcanti and Woodcock [55], and Charpentier [63] needed to build new reasoning environments in order to address issues related to correctness and composability of (reactive) systems. Both approaches have strong roots in the weakest precondition semantics of Dijkstra [71].

Bellegarde et al. introduce a similar idea of synchronized parallel composition for event-B systems [42].

In the temporal logic of actions of Lamport [115, 116], synchronization is specified as a way of applying *noninterleaving* to system design. This is reached by employing *joint actions*. The author's conclusion supports our point of view: interleaving "blurs" the distinction between the components used in design.

1.2.2 Hybrid Systems Modeling and Analysis

Problem Description. Hybrid systems combine discrete control with continuous evolutions. The latter are most frequently described by differential equations, which make the state space of the system infinite. This and other reasons like, for example, the interaction of the logic controller with the continuous behavior, make hybrid systems inherently complex and difficult to analyze automatically.

Such systems may exhibit particular undesired behaviors that are not met in purely discrete environments. One known behavioral anomaly is *timelocking*: time is prevented from advancing by infinite executions of discrete transitions. Consequently, the corresponding modeling formalism should be capable of prohibiting such behaviors.

Tool support is needed in order to carry out hybrid system analysis. There is a range of tools that can be used for this purpose: simulators, model-checkers, theorem provers, or combinations of the last two techniques [133].

Simulating a formal model of the hybrid system is very useful, allowing one to find potential trouble spots before proceeding to full formal verification.

Symbolic simulation, introduced by King [105], refers to performing simulation on sets of states represented symbolically. Thus, symbolic simulation differs from regular simulation in that it simultaneously traverses a number of trajectories, rather than a single trajectory through the state space [151]. This allows the simulation of a potentially infinite number of trajectories in one symbolic simulation. Initial logical mistakes can be uncovered by visualizing the behavior of the model, or by inspecting lists of symbolic values of discrete-valued and continuous-valued time variables.

The proof automation mentioned above can either be partial or total. In the partial case, we have *semi-decision* procedures. These algorithms may not always succeed in proving a claim, and may, therefore, require guidance from the user. On

the other hand, totally automated procedures are guaranteed to terminate with a result. However, due to the undecidability of most properties of hybrid systems [6], these algorithms apply only to a restricted class of hybrid systems. Besides, there are many hybrid systems that contain *parameters* in their representation. These systems are supposed to operate correctly only for certain values of the parameters, or if specific relationships between parameters hold. The relationships may sometimes be easy to guess. If this is the case, we can then formally verify the correctness of the guess. Verification of models with parameters is a semi-decisional process that depends on the number of clocks, parameters, and other variables.

Nevertheless, there are also cases where it is not trivial to intuitively formulate the correct parametric relationships, or values of parameters that would ensure a correct functioning of the analyzed system. Then, one needs to deduce them by means of a verification procedure. This method is called *parameter synthesis*. Since an algorithmic approach to parameter synthesis may sometimes fail to deliver the expected constraints, it may be useful to tackle such a design task by means of *deductive reachability analysis*.

Solution. Below, we enumerate the contributions of chapters 4 and 5, which try to address the above mentioned problems.

- In chapter 4 we extend the syntax of continuous action systems [27], such that the absence of *timelocks* is guaranteed. We model the *execute only once* (at the same time-point) concept, for actions, by adorning transitions in the original CAS. Rather than complicating the otherwise simple model of CAS, we push the problem of avoiding timelocks to the implementation level. Our solution aims at enhancing the class of hybrid/real-time systems that can be handled within our framework.
- Still in chapter 4, we adapt the synchronization mechanism introduced in chapter 3, to continuous action systems. *Synchronized parallel CAS* let one compose reactive hybrid models, by employing the new parallel composition operator, '$\sharp$', such that the composition presents a global, concurrent response to the inputs. The CAS modules synchronize on the update of the global variables, after a sequence of execution rounds. All the results established for the discrete case, and outlined in section 1.2.1, hold for the continuous case, too. We show that our model is useful for the design of hybrid systems characterized by behaviors with discontinuities.
- A tool for the *symbolic simulation* of CAS, implemented in Mathematica [156], is proposed in chapter 5. Mathematica is a powerful computer algebra package, also equipped with plotting facilities. We give the flavor of this platform, in chapter 2. Various linear models have been simulated. Symbolic simulation should be used as far as it is possible, or numerical approximation could be applied instead (for nonlinear cases). Besides graphical representation of variables, at the end of the simulation, we also get information about the exact time moments when discrete transitions have been fired. Lists with

symbolic values of all model variables at those particular moments complete the set of simulation results delivered by the tool.

- In chapter 5, we also apply a deductive procedure for parameter synthesis, on a relevant linear hybrid system model, that is, a temperature control system within a nuclear reactor tank. The method is based on superposition of nonconflicting invariants. It reduces to proving that a certain bad condition can not be reached, if certain relationships between parameters hold. The disadvantage is that our method is not automated. The advantage is that one gets more insight about the system behavior, in comparison to parameter synthesis carried out by using model-checkers like HYTECH [87, 88], or TREX [48]. Moreover, the deductive method is not limited by the number of parameters or clocks. Thus, it might be applied to complex parametric hybrid systems.

Related Work. Alur and Henzinger have developed an assume-guarantee principle for reasoning about timed and hybrid modules [12]. The approach uses the concepts of *update rounds* and *time rounds*. The former rounds update the global variables, whereas the latter update all clock variables, by selecting a duration (possibly 0) that the module is prepared to let elapse. In our synchronized environments, we do not make the distinction between rounds updating global variables, and rounds updating time. All global variables (be they discrete valued or continuous valued time variables) and time are updated by a sequence of statements, at the end of the same cycle.

Many simulation packages have been proposed and applied for the systematic analysis of hybrid systems [74, 79, 124, 125]. Out of these, the *Matlab Simulink/Stateflow* tool [124] provides extensive simulation facilities. However, the conducted simulation is based on Matlab's numerical routines. The suite consists of two modeling languages: Simulink, which is used to model the continuous dynamics, and Stateflow, used to specify the discrete control logic. The latter language does not have a precise formal semantics, thus verification of hybrid systems modeled in Stateflow is hampered.

Analysis and verification of parameterized hybrid models is a difficult problem, since their verification is, in general, undecidable. Therefore, automated parameter synthesis can be applied with limitation. Tools that allow synthesis of parameters for hybrid systems, such as HYTECH [87, 88], may fail to terminate due to parameter types, or big number of clocks. Initial bounds on parameters may be set in order to limit the size of the generated state space. On the other hand, tools such as TREX [48] use *on-the-fly* state exploration techniques, thus overcoming nontermination. However, TREX uses *timed automata* [9] as the modeling language, which might be too weak for modeling hybrid systems, since timed automata is a subclass of *linear hybrid automata* [86], which is used in HYTECH.

Aiming for generality, a deductive method, based on a mathematical invariance proof, can be applied as an alternative to algorithmic approaches. It extracts correct

values or relationships between parameters involved in hybrid designs, regardless of the number of clocks or parameters. To gain confidence in manual proofs, we may mechanize the process, by means of a theorem-prover [65].

1.2.3 Correct-by-Construction Real-Time Schedulers

Problem Description. *Scheduling* real-time tasks is by far the research topic that has received most attention in the real-time systems community. The requirements of real-time systems include non-functional properties that need to be preserved during execution. These properties refer to *task deadlines*, which are supposed to be met, at run-time.

In most cases, when building a real-time schedule, it is essential to find a systematic way that guarantees that the tasks are executed such that they will always complete by their deadlines. This way is called a *scheduling policy* [51, 119]. Hence, task priorities need to be assigned with respect to predefined algorithms.

Constructing correct policy-based real-time schedulers is not a trivial job, since analyzing the schedulability conditions of sets of real-time tasks reduces, in most cases, to computing fixed points. This technique is notoriously resource consuming, so that trying to avoid such algorithms might become an attractive idea.

Solution. In consequence, in chapter 6, we propose the following:

- A refinement-based method for building correct-by-construction real-time priority-driven schedulers, for uniprocessor systems. We enforce schedulability properties, as the conjunction of timing requirements (meeting deadlines), mutual exclusive execution of tasks, and policy-related priority assignments. The scheduled system is derived through refinement, starting from an abstract level. In the end, we reach an implementable level, also described as an action system. A decomposition in two separate modules (the scheduler and the set of tasks) is performed as a last step.
- The application of the technique described above to both fixed-priority (e.g., *Deadline-Monotonic*) algorithms, as well as to dynamic-priority algorithms (e.g., *Earliest-Deadline-First*). Preemptible, sporadic, as well as non-preemptible, periodic task models are considered. Moreover, in the Earliest-Deadline-First case, the constructed schedule is validated by simulating the constructed real-time model in Mathematica, up to the least-common-multiple of the task periods.

Related Work. Formal approaches have been recently applied by Kwak et al. [113], who develop symbolic bisimulation algorithms, and Altisen et al. [4], who propose synthesis algorithms, for constructing real-time schedules. A major disadvantage of these approaches is the practical high complexity of the algorithms. Hence, the methods can not accommodate a large number of tasks.

We were motivated by the mentioned inconvenience to find an alternative solution to the model-checking targeted algorithms. In consequence, we have developed a general scheduler construction method, which can be applied to any particular collection of real-time tasks.

A similar work to ours is due to Altisen, Gößler, and Sifakis [5], where fixed point computation algorithms are combined with the incremental application of priority rules that restrict the initial behavior of the real-time model. Nonetheless, the authors model the system in a monolithic fashion, the real-time tasks being represented by uncontrollable transitions in timed automata models. In comparison, our method leads eventually to a two-module implementation of the abstract real-time model.

1.2.4 Controller Synthesis for Discrete Systems

Problem Description. Since many reactive systems are actually control systems, it is essential to tackle the problem of constructing the controller of such a system. The idea behind *controller synthesis* involves changing the level of abstraction of an initial system model, by computing the most general (maximally nondeterministic) controller that satisfies the requirements. Therefore, it is sufficient to start with a nondeterministic, high-level model of the controller. In principle, synthesis steps decrease the controller's nondeterminism. In short, one needs to adjust the initial system representation, by restricting the behavior of the controller, such that all possible transitions that could lead to unsafe states are eliminated.

Most of the synthesis approaches known in literature are algorithmic. They are based on computation of the maximal set of controllable states, using *backward fixed point iteration* of *symbolic predecessor* operators [18, 99, 122]. This strategy implies exploration of the entire state space, in order to find the result. A main drawback could be excessive memory consumption.

Solution. To address this deficit, in chapter 7, we propose a deductive game-based method for solving the problem of sequential control. As distinct from the research carried out in the previous chapters, chapter 7 opens a new gate by pointing to a different direction.

We describe the system as a game between two rival players, the *angel* and the *demon*. The angel represents the controller, and the demon models the plant, or the disturbance. Thus, we work with action systems that contain two kinds of nondeterministic statements, in the form of *angelic choices*, or *angelic nondeterministic assignments*, and *demonic choices*, or *demonic nondeterministic assignments*.

Requirements are modeled as temporal properties, initially defined within the dually nondeterministic weakest precondition framework, by Back and von Wright [34, 37, 38]. The angelic controller has the obligation to enforce the respective temporal property, if it can, regardless of the actions of the demonic plant.

Our method starts by checking whether the angel has a way to win the game with respect to the specified requirement. The latter is an *always* property, when we

synthesize controllers for *invariance* (controllers that must keep the system within a safe set of states). The requirement is a special form of *eventually* property, called *weak response*, when we address synthesis of reliable controllers for *reachability* (controllers that must guide the system into an intended set of states, in finite time). We show that the response property that we are looking at is enforced by proving adequate invariance and termination properties of a fixed-point statement that we define. From our perspective, this is the main difference between the approach presented in chapter 7 and the algorithmic solutions to the controller synthesis problem (for example the one employing *alternating-time temporal logic* [15]): we reduce synthesis to correctness reasoning.

If we succeed in proving that there exists such a strategy for the angel, we extract it, next, by rewriting the respective angelic statement in a certain context resulted from the correctness proofs carried out in the first step. This transformation restricts the choices of the angel to those that establish the requirement. The result is a correct, implementable (maximally nondeterministic) controller, which is guaranteed to preserve the required temporal property for any of its available choices.

Below, we summarize the contribution of chapter 7:

- A deductive method for synthesizing controllers for classes of discrete systems that can be modeled as games with more than one round.
- The application of our technique to both invariance and reachability controller synthesis.
- A new inference rule for verifying the existence of angelic winning strategies under a particular case of reachability control. Our study uses propagation of context information as the main transformation for extracting the angelic winning strategy. In both invariance and reachability cases, by playing the angelic nondeterminism against the demonic one, we get, in the end, a correct-by-construction implementable controller.

Related Work. Viewing a reactive system as a two-player game is not a new idea. It can be traced back to Ramadge and Wonham [139], and Pnueli and Rosner [137]. The authors developed synthesis algorithms for finite-state discrete systems, and showed that finding a winning strategy for the game was equivalent to synthesizing a controller that satisfied the requirements.

For *discrete-event systems*, which is one of the most popular (discrete) frameworks, there are tools for the construction of *supervisory controllers* [68, 157].

In order to overcome the state explosion problem encountered in most algorithmic approaches to controller synthesis [18, 99, 122], Tripakis and Altisen have proposed *on-the-fly* algorithms [154]. However, their algorithms are applicable to finite-state systems only.

If compared to dedicated model-checking algorithms, our approach is a general and less costly technique in terms of computer memory resources. On the other hand, it involves the non-trivial task of finding adequate invariants.

Perhaps the closest work to ours is proposed by Slanina [147], who develops proof rules for safety and response linear temporal logic [136] properties of concurrent reactive games. However, the equivalent of our second synthesis step, that is, extracting the angelic winning strategy, is not apparent. We are just promised that, if the control conditions can be proved valid, by invoking constructive theorem proving methods, the extracted program can be used further to synthesize a control winning strategy. Moreover, the author does not use a two-fold nondeterminism, he rather relies on *existential Hoare triples* [46, 145] to emulate angelic behavior.

1.3 Organization

The thesis builds on published papers that have appeared in conference proceedings and journals. Nevertheless, in the current study we extend and improve on the results of the papers, providing information that is not available in the published material. The list of papers that form the core of this thesis is given below, in chronological order.

- R. J. Back and C. Cerschi. *Modeling and Verifying a Temperature Control System using Continuous Action Systems*. In Proceedings of the 5th International ERCIM Workshop on Formal Methods for Industrial Critical Systems (FMICS'2000), GMD Report 91, pp. 265-286, *ERCIM and GMD*, 2000.
- R. J. Back, C. Cerschi Seceleanu, and J. Westerholm. *Symbolic Simulation of Hybrid Systems*. In Proceedings of the 9th Asia-Pacific Software Engineering Conference (APSEC 2002), pp. 147 - 158, IEEE Computer Society Press, 2002.
- R. J. Back and C. Cerschi Seceleanu. *Contracts and Games in Controller Synthesis for Discrete Systems*. In Proceedings of the 11th IEEE International Conference on the Engineering of Computer-Based Systems (ECBS 2004), pp. 307 - 315, IEEE Computer Society Press, 2004.
- C. Cerschi Seceleanu and T. Seceleanu. *Synchronization Can Improve Reactive Systems Control and Modularity*. Journal of Universal Computer Science (JUCS), 10(10): 1429 - 1468, Springer, 2004.
- C. Cerschi Seceleanu. *Formal Development of Real-Time Priority-Based Schedulers*. In Proceedings of the 12th IEEE International Conference on the Engineering of Computer-Based Systems (ECBS 2005), pp. 263 - 270, IEEE Computer Society Press, 2005.
- C. Cerschi Seceleanu. *Designing Controllers for Reachability*. In Proceedings of the 29th Annual International Computer Software and Applications Conference (COMPSAC 2005), IEEE Computer Society Press, 2005.

The outline of the rest of the dissertation is as follows. Chapter 2 presents the preliminary notions needed and used throughout this study. The reader is briefed

on statements, games, actions systems, and continuous action systems, as well as the refinement techniques available for such models. The background chapter ends with a short overview of the computer algebra tool, *Mathematica* [156].

A new parallel composition operator for action systems, “sharp” (‘$\sharp$’), and an associated execution environment, called the *synchronized parallel environment* are introduced in chapter 3. We compose action system modules by employing the “sharp” operator that we define; moreover, we prove the essential properties that emerge out of this formal perspective. The modular design and refinement of a digital filter serves as the accompanying case-study.

The extended syntax for CAS and its corresponding implementation are proposed in chapter 4. We close the chapter with the *synchronized parallel CAS*, intended for modular design of timed and hybrid systems.

In chapter 5, we describe and exemplify a *symbolic CAS simulator*, which we have implemented in Mathematica. The tool is exercised on simulating the behavior of a nuclear reactor *temperature control system*. The simulation results are presented as graphs, and also as symbolic lists of state variables values. Next, we describe and exemplify the iterative invariance-based *parameter synthesis* approach, intended for the reliable design of hybrid systems with parameters.

The refinement-based methodology for the incremental construction of uniprocessor real-time schedulers is described in chapter 6. Preemptible, sporadic, and also non-preemptible, periodic task collections are considered for scheduling. We open the chapter with the main real-time scheduling results already known in the literature of uniprocessor scheduling theory. Afterwards, we present the actual scheduler construction method. Next, we apply it for constructing a real-time model scheduled by the fixed-priority *Deadline-Monotonic* algorithm. Trace refinement techniques are then used to reach an efficient real-time system implementation; also, a provably correct decomposition splits the final model in two distinct modules, the scheduler and the real-time tasks. Last but not least, we apply our correct-by-construction method under the assumption of a dynamic-priority scheme, that is, the *Earliest-Deadline-First* policy. We show how to validate the constructed schedule by simulating it up to the least-common-multiple of the task periods. For this, we use the tool described in chapter 5.

Last but not least, we return to the issue of discrete reactive system control. Chapter 7 points to a different direction and introduces our game-based approach to synthesis of controllers for discrete control systems. The first part deals with finding winning strategies for safety games. The second part aims at finding ways to win a particular type of reachability games. Two relevant examples illustrate the application of the theoretical results.

We end our dissertation with conclusions, enumerate the limitations of our methods and also future research directions, in chapter 8.

Some detailed proofs of particular theorems, corollaries, and refinements stated throughout the thesis can be found in the Appendix.

Chapter 2

Background

As already mentioned in the introduction, the *refinement calculus* [21, 35, 129, 131] is a logical framework for reasoning about programs.

In principle, we want to know whether a program is *correct* with respect to a given specification, and how can we improve or *refine* the higher-level model, without loss of correctness. Both specifications and programs can be seen as special cases of a more general notion, that of a *contract* between different agents (programs, modules, systems, users) involved in the computation.

A contract regulates the behavior of an agent, by modeling what the agent is permitted and supposed to do. In the following, we only consider at most two rival agents as participants. Then, the contract reduces to the special case of statements where two distinct kinds of choices are permitted.

In this chapter, we give an overview of statements, and introduce *action systems* as a special kind of statement. Next, we present *continuous action systems*, the extension of discrete action systems, which targets hybrid and real-time systems modeling. This is followed by brief formal descriptions of *correctness* and *refinement* at both the action and system levels. At the end, we present shortly the computer algebra tool, *Mathematica*.

All the notions and the tool description below provide the basis for the remainder of the thesis.

2.1 Statements

Our reasoning framework uses *higher-order logic* as the underlying logic. A *statement* S is built according to the syntax below:

$$
\begin{array}{rcll}
S & ::= & \mathsf{skip} & \text{(stuttering)} \\
 & & \mid \mathsf{abort} & \text{(abort)} \\
 & & \mid \mathsf{magic} & \text{(miracle)} \\
 & & \mid x := e & \text{(assignment)} \\
 & & \mid \{p\} & \text{(assertion)}
\end{array}
$$

$\mid [p]$	(assumption)
$\mid S_1 \,;\, S_2$	(sequential composition)
$\mid$ **if** g **then** S_1 **else** S_2 **fi**	(deterministic conditional)
$\mid S_1 \sqcup S_2$	(angelic choice)
$\mid S_1 \sqcap S_2$	(demonic choice)
$\mid \{x := x' \mid b\}$	(angelic nondeterministic assignment)
$\mid [x := x' \mid b]$	(demonic nondeterministic assignment)
$\mid (\mu X \bullet S)$	(angelic recursion)
$\mid (\nu X \bullet S)$	(demonic recursion)

Here, p ranges over *state predicates* ($\Sigma \to \mathsf{Bool}$), $x := e$ is a *state transformer* ($\Sigma \to \Sigma$), and $x := x' \mid b$ is a *state relation* ($\Sigma \to \Sigma \to \mathsf{Bool}$). In these definitions, Σ is the polymorphic type of the program state. We write $f.x$ for function f applied to argument x.

The statement abort is seen as a failure: the computation has to be *aborted* because something has gone wrong. This statement does not achieve any condition. On the contrary, statement magic achieves whatever condition the program is supposed to, as if a *miracle* has occurred [35].

The *assignment* changes the state according to the state transformer $x := e$. In the grammar, skip is described as *stuttering* (no change of the observable state).

The *assertion* $\{p\}$ leaves the state unchanged if p holds and aborts otherwise (it behaves like: **if** p **then** skip **else** abort **fi**). It then follows that abort is interpreted as the assertion that is impossible to satisfy, that is, $\{\mathsf{false}\}$. The *assumption* $[p]$ also leaves the state unchanged if p holds, but terminates *miraculously* otherwise (that is, it establishes any postcondition, even false). The assumption behaves as: **if** p **then** skip **else** magic **fi** . Hence, magic can be interpreted as the assumption that is impossible to satisfy, that is, $[\mathsf{false}]$.

In the *sequential composition* $S_1 ; S_2$, statement S_1 is first carried out, followed by S_2.

An *angelic choice* $S_1 \sqcup S_2$ allows a controllable entity called *the angel* to choose which statement is to be executed. A *demonic choice* $S_1 \sqcap S_2$ lets an uncontrollable entity called *the demon* to choose between carrying out S_1 or S_2. Note that, if there is only one type of nondeterminism involved, that is, demonic, we denote the respective nondeterministic choice of statements by $S_1 \parallel S_2$.

The *angelic nondeterministic assignment* or *angelic update*, $\{x := x' \mid b\}$, models angelic choices of the final state, among those that satisfy the boolean condition b. In the *demonic nondeterministic assignment* (*demonic update*), $[x := x' \mid b]$, the choice of the final state is demonic. If no such state exists, then the angelic update is aborting, while the demonic update is miraculous. In both assignments, x' characterizes the state after the execution of the statement, and stands for a bound variable whose value should satisfy b.

Our language also permits *recursive statements*, as $(\mu X \bullet S)$ or $(\nu X \bullet S)$. In the first case, the recursive statement is executed so that S is repeated a demonically

chosen (finite) number of times. If S can be executed indefinitely, this infinite execution is a failure (abort) for the angel. In the second case, the execution of $(\nu X \bullet S)$ is similar, with the difference that infinite execution is not a bad thing for our agent.

Semantics of Statements. A *predicate transformer* is a function that maps predicates to predicates $((\Sigma \to \mathsf{Bool}) \to (\Sigma \to \mathsf{Bool}))$. We want the predicate transformer S to map postcondition q to the set of all initial states σ from which S is guaranteed to end in a state of q. Thus, $wp.S.q$ is the *weakest precondition* of S to establish postcondition q.

The pointwise extension of the implication ordering ($\Rightarrow$) on Bool gives us the ordering on predicates ($\subseteq$). Also, conjunction ($\cap$), disjunction ($\cup$), and negation ($\neg$) on predicates are defined by pointwise extension of the corresponding operations on booleans. Similarly, the order ($\sqsubseteq$), and operations ($\sqcap, \sqcup$) on predicate transformers are lifted from their respective counterparts on predicates.

In all the following chapters, except for the last, we choose to use the notation for booleans, for predicates too, rather than the actual set notation (e.g. $q \wedge r$ rather than $q \cap r$, q, r predicates). In the last chapter, we employ the predicate-specific notation, as a distinctive feature of the fact that we work with dual nondeterminism.

Predicates form a *complete boolean lattice*, and so do predicate transformers [35]. Abort is the bottom ($\bot$) and magic is the top ($\top$) of the predicate transformer lattice.

Given the fact that Back's refinement calculus [35] does not make a distinction between program statements and their semantics, each statement S is simply identified with a specific predicate transformer from postconditions to weakest preconditions. Hence, one can write $S.q$ instead of $wp.S.q$.

The intuitive description of statements can be used to justify the following definition of the weakest precondition semantics:

$$\mathsf{skip}.q \triangleq q \tag{2.1}$$

$$\mathsf{abort}.q \triangleq \mathsf{false} \tag{2.2}$$

$$\mathsf{magic}.q \triangleq \mathsf{true} \tag{2.3}$$

$$(x := e).q \triangleq q[x := e] \tag{2.4}$$

$$\{p\}.q \triangleq p \cap q \tag{2.5}$$

$$[p].q \triangleq \neg p \cup q \tag{2.6}$$

$$(S_1 \, ; S_2).q \triangleq S_1.(S_2.q) \tag{2.7}$$

$$(\mathbf{if}\ g\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{fi}\ ).q \triangleq (\neg g \cup S_1.q) \cap (g \cup S_2.q) \tag{2.8}$$

$$(S_1 \sqcup S_2).q \triangleq S_1.q \cup S_2.q \tag{2.9}$$

$$(S_1 \sqcap S_2).q \triangleq S_1.q \cap S_2.q \tag{2.10}$$

$$\{x := x' \mid b\}.q \quad \triangleq \quad (\exists x' \bullet b \cap q[x := x']) \tag{2.11}$$

$$[x := x' \mid b].q \quad \triangleq \quad (\forall x' \bullet b \subseteq q[x := x']) \tag{2.12}$$

These definitions are consistent with Dijkstra's original semantics for the language of guarded commands [71], and with later extensions to it.

Observe that the controllability of the angel is mirrored in the semantics of the angelic choice: for $S_1 \sqcup S_2$ to establish q, it is enough that one of the statements does it. In contrast, the uncontrollable demonic behavior shows in the requirement that both alternatives of the choice $S_1 \sqcap S_2$ should establish q, in order for the demonic choice to do so.

We say that a predicate transformer S is *monotonic* if and only if the following implication holds:

$$\forall p, q \bullet p \subseteq q \Rightarrow S.p \subseteq S.q$$

Furthermore, a predicate transformer S is called *strict* (or *nonmiraculous*) if it preserves false, and *terminating* (or *nonaborting*) if it preserves true. Also, S is said to be *conjunctive* if it preserves nonempty meets of predicates, and *disjunctive* if it preserves nonempty joins of predicates:

$$\begin{array}{rcll} S.\mathsf{false} & = & \mathsf{false} & \text{(S strict)} \\ S.\mathsf{true} & = & \mathsf{true} & \text{(S terminating)} \\ S.(\cap i \in I \bullet q_i) & = & (\cap i \in I \bullet S.q_i),\ I \neq \emptyset & \text{(S conjunctive)} \\ S.(\cup i \in I \bullet q_i) & = & (\cup i \in I \bullet S.q_i),\ I \neq \emptyset & \text{(S disjunctive)} \end{array}$$

A *conjunctive specification* may be a compound statement where all the constituent statements are conjunctive predicate transformers. The statement below is such an example:

$$\{p\} \,;\, (S_1 \, [\!] \, S_2)$$

Fixed points are used to give meaning to recursive definitions. The *least fixed point* of f is denoted by $\mu.f$, while the *greatest fixed point* is denoted by $\nu.f$. The least fixed point of f is characterized by the following two properties:

$$\begin{array}{rcll} f.(\mu.f) & = & \mu.f & \text{(folding least fixed point)} \\ f.x \sqsubseteq x & \Rightarrow & \mu.f \sqsubseteq x & \text{(least fixed point induction)} \end{array} \tag{2.13}$$

The greatest fixed point is characterized by the following dual properties:

$$\begin{array}{rcll} f.(\nu.f) & = & \nu.f & \text{(folding greatest fixed point)} \\ x \sqsubseteq f.x & \Rightarrow & x \sqsubseteq \nu.f & \text{(greatest fixed point induction)} \end{array} \tag{2.14}$$

Let us consider the function $(\lambda X \cdot S)$. The least fixed point of this function is written as

$$\mu.(\lambda X \cdot S) = (\mu X \cdot S)$$

An important particular case of recursion is the while loop, which is defined as the least fixed point of the unfolding function:

$$\mathsf{while}\ g\ \mathbf{do}\ S\ \mathbf{od} \ \triangleq\ (\mu\, X \bullet \mathbf{if}\ g\ \mathbf{then}\ S \,;\, X\ \mathbf{else}\ \mathsf{skip}\ \mathbf{fi}\,)$$

By unfolding, we see that S is repeatedly executed as long as the predicate g holds.

The *guarded iteration statement* generalizes the while loop. It is defined as

$$\begin{aligned}&\mathbf{do}\ g_1 \rightarrow S_1 \ [\!]\ \ldots\ [\!]\ g_n \rightarrow S_n\ \mathbf{od}\ \ \triangleq \\ &\qquad \mathsf{while}\ g_1 \cup \ldots \cup g_n\ \mathbf{do}\ [g_1]\ ;\ S_1 \sqcap \ldots \sqcap [g_n]\ ;\ S_n\ \mathbf{od}\end{aligned}$$

This gives the while loop as a special case, when $n = 1$:

$$\mathbf{do}\ g \rightarrow S\ \mathbf{od}\ = \mathsf{while}\ g\ \mathbf{do}\ S\ \mathbf{od}$$

Here, the predicate g is called *the guard* of the loop, assuming that S is strict.

In general, the semantics of the *guard* gS of a statement S is: $gS = \neg S.\mathsf{false}$ [134], meaning that it characterizes those states from which S behaves non-miraculously (gS guards against miracles).

Prioritizing composition. This construct, introduced by Nelson [134], and adapted for action systems by Sekerinski and Sere [141], is basically a choice operation, where statements are given certain priorities. If a lower level priority statement is enabled, it can be executed only if no other higher level priority statement is enabled.

Then, the prioritizing composition of two statements is defined as:

$$S_1\ //\ S_2\ \triangleq\ S_1\ [\!]\ \neg gS_1 \rightarrow S_2$$

In order of decreasing binding power, the operators are: ; $[\!]$ //.

Quantified composition. Any composition operator can be *quantified*. This applies to the different compositions of statements. For instance, the quantified nondeterministic choice is defined as follows:

$$[\ [\!]\ \ i \in [1..n] : S_i\,]\ \ \triangleq\ \ S_1\ [\!]\ \ldots\ [\!]\ S_n$$

Statements as Games. In a previous paragraph, we have shown how statements can be interpreted as (monotonic) predicate transformers. In this paragraph, we briefly discuss the *game-based* interpretation of statements.

Statements can be defined in terms of a game that is played by two participants, the angel and the demon. The *game semantics* describes how a statement S encodes the *rules* of a *game*. A *play* of the game S is characterized by an *initial state* (σ) of the game, and a *goal* (a postcondition) q that describes the set of final states that are winning positions.

A sequence of an angelic and a demonic update is interpreted as a *two-player* game with the angel and the demon as players:

$$\{x := x' | x < x' \leq x + 2\}\ ;\ [x := x' | x \leq x' \leq x + 2]$$

In the above game, the angel plays first, after which the demon takes its turn. The players are rivals. For a given postcondition q, the angel tries to reach a final state

that satisfies q, starting from the initial state σ. The demon tries to prevent this. In the example, the angelic nondeterministic assignment lets the angel increase the value of the natural variable x, with one or two units, whereas the demonic assignment requires the demon to keep x unchanged, or increase its value also by 1 or 2.

We say that the angel has a *strategy to win* a two-player game, if and only if it has a way of making its choices inside S such that the postcondition q holds in the final state, regardless of how the demon resolves its choices. Note that to win, the angel must be able to reach q, thus nontermination is bad for the angel.

In the above game example, assuming that the initial state satisfies $x = 0$, the angel does have a strategy to win the game with the goal $x \geq 2$, no matter how the demon moves. On the other hand, from that same initial state, the angel can not win the game for postcondition $x = 2$.

2.2 Action Systems

Back and Kurki-Suonio proposed the *action systems* formalism, as a framework for specifying and refining concurrent programs [31, 32]. An action system is in general a collection of *actions* or *guarded commands*, which are executed one at a time.

The generic action system is built according to the following syntax:

$$\mathcal{A}(z : T_z) \triangleq \textbf{begin var } x : T_x \bullet \ Init \, ; \ \textbf{do } A_1 \ [\!]\ \ldots\ [\!]\ A_n \ \textbf{od} \ \textbf{end} : p \quad (2.15)$$

Here, $\mathcal{A}$ contains the declaration of *local* variables x (of type T_x), followed by an *initialization* statement $Init$ and the *actions* $A_1, \ldots, A_n$, grouped within a $\textbf{do}$ - $\textbf{od}$ loop. Variables z (of type T_z) are *global* to the action system. The constants $p = p_1, \ldots, p_m$ are called the *parameters* of system $\mathcal{A}$.

An action (*guarded command*), $A_i \triangleq g_i \to S_i$, S_i strict, is *enabled* and its *body* S_i is executed, if the *guard* g_i evaluates to true. Otherwise, action A_i is called *disabled*. The chosen actions change the values of the variables in a way that is determined by the action body.

The initialization statement assigns starting values to the global and local variables. After that, *enabled* actions are repeatedly chosen and executed. We assume that actions are *atomic*, that is, they are indivisible. In addition, the guard of an action system $\mathcal{A}$ given by (2.15) is denoted by $gg_{\mathcal{A}}$, and it is defined as: $gg_{\mathcal{A}} \triangleq \bigvee_{i=1}^{n} g_i$. The execution of system $\mathcal{A}$ terminates when its guard does not hold anymore, that is, when $\neg gg_{\mathcal{A}}$ holds.

In other words, an action system is an initialized $\textbf{do} - \textbf{od}$ loop:

$$\begin{array}{l} Init \, ; \textbf{do } g_1 \to S_1 \ [\!]\ \ldots\ [\!]\ g_n \to S_n \ \textbf{od} \ \triangleq \\ \quad Init \, ; (\mu X \bullet [g_1] \, ; S_1 \, ; X \sqcap \ldots \sqcap [g_n] \, ; S_n \, ; X \sqcap [\neg(g_1 \vee \ldots \vee g_n)]) \end{array}$$

An action system is not usually regarded in isolation, but rather as a part of a more complex structure, the rest of which, that is, the *environment*, communicates with the action system via *shared* (read and written) variables. In the following, we assume and extend the notations defined in [28].

The set of state variables accessed by some action A is denoted vA, and is composed of the *read* variable set of action A, denoted rA, and the *write* variable set of action A, denoted wA. We have that $vA = rA \cup wA$. We can also build the same sets at the system level, considering the local / global partition of the variables. Thus, for a given action system $\mathcal{A}$, we have the access set, $v\mathcal{A}$, split into the global read / write variables, denoted by $gr\mathcal{A}/gw\mathcal{A}$ and the local read / write variables, denoted by $lr\mathcal{A}/lw\mathcal{A}$. We say that an action A of $\mathcal{A}$ is either *global*, if $gw\mathcal{A} \cap wA \neq \emptyset$, or *local*, if $wA \subseteq lw\mathcal{A}$.

Action systems occupy a central position in this thesis. We will thus take advantage of the expressiveness of this formalism, and use it further for modeling purposes.

Parallel Composition of Action Systems. Several action systems can be combined to form a new action system.

Let our protagonists be $\mathcal{A}$ and $\mathcal{B}$, two action systems of the form

$$\begin{array}{lcl} \mathcal{A}(z_A) & \triangleq & \textbf{begin var } x_A \bullet Init_A \,;\, \textbf{do } g_A \rightarrow S_A \textbf{ od end} \\ \mathcal{B}(z_B) & \triangleq & \textbf{begin var } x_B \bullet Init_B \,;\, \textbf{do } g_B \rightarrow S_B \textbf{ od end} \end{array}$$

Assuming that x_A and x_B are disjoint, the parallel composition [22] of $\mathcal{A}$ and $\mathcal{B}$ is the system $\mathcal{P} = \mathcal{A} \,||\, \mathcal{B}$:

$$\mathcal{P}(z_P) \triangleq \textbf{begin var } x_P \bullet Init_A \,;\, Init_B \,;\, \textbf{do } g_A \rightarrow S_A \,[\!]\, g_B \rightarrow S_B \textbf{ od end}$$

The composed action system essentially combines the variables, the initialization statements and the actions of the two components. The initialization of the common variables z must be consistent[1], that is, they are assigned the same initial values by both initialization statements, $Init_A$ and $Init_B$. In principle, initializations are merely made of assignments; nevertheless, other constructs like conditionals are allowed only if they do not mention in their guards uninitialized (global) variables. Some of the previously global variables of $\mathcal{A}$ and $\mathcal{B}$ may become local variables of $\mathcal{P}$. We add these to the reunion of the individual local variables of $\mathcal{A}$ and $\mathcal{B}$, thus obtaining the set of local variables of $\mathcal{P}$, x_P. The global variables z_P are defined as $z_P \triangleq (z_A \cup z_B) - x_P$.

Given the above formalization, $\mathcal{A} \,||\, \mathcal{B}$ is executed by first initializing the local and global variables, and then interleaving the execution of the enabled actions of $\mathcal{A}$ and $\mathcal{B}$. Termination occurs when both action systems terminate, which means that there is no enabled action, in neither of the systems, that is, $gg_\mathcal{A} \vee gg_\mathcal{B} \equiv \mathsf{false}$.

[1] Alternatively, global variables can have a well-defined value that is expressed as a precondition.

In short, one observes the composition as the single system $\mathcal{P}$.

Prioritizing Composition of Action Systems. Considering the same action systems $\mathcal{A}$ and $\mathcal{B}$ as above, their prioritizing composition is a new action system $\mathcal{U} = \mathcal{A} \mathbin{//} \mathcal{B}$, defined by:

$$\mathcal{U}(z_U) \triangleq \textbf{begin var } x_U \bullet Init_A \,;\, Init_B \,;\, \textbf{do } g_A \rightarrow S_A \mathbin{//} g_B \rightarrow S_B \textbf{ od end}$$

The variables x_U are the reunion of the local variables x_A, x_B, to which the variables hidden from the interface of $\mathcal{U}$ are added. The global variables $z_U = (z_A \cup z_B) - x_U$. The choice between the action of $\mathcal{A}$ and the action of $\mathcal{B}$ is deterministic in the sense that when both are enabled, S_A is executed.

2.3 Correctness and Refinement of Action Systems

Invariance. We say that a predicate $I(vA)$ – I for short – is *preserved* by an action A, if the relation

$$I \Rightarrow A.I$$

holds. Considering the action $A \triangleq g \rightarrow S$, S strict, the above relation translates into

$$g \wedge I \Rightarrow S.I$$

This means that whenever the action A is enabled, and provided that I holds, the execution of the action body S terminates in a state of I.

At the system level, a predicate $I(v\mathcal{A})$ (I for short) is an *invariant of the action system* $\mathcal{A}$, given by (2.15), if:

- it is *established* by $Init$, that is,

$$\mathsf{true} \Rightarrow Init.I, \text{ and if}$$

- it is *preserved* by each action $A_i \triangleq g_i \rightarrow S_i$, that is,

$$g_i \wedge I \Rightarrow S_i.I,\ \forall i \in [1..n]$$

Correctness. Proving correctness of programs can be done by applying the inference rules of *Hoare logic* [95], which hold of any constructs of a programming language. The logical rules let us reduce the correctness of a program statement to the correctness of its components. Dijkstra's predicate transformer method provides an alternative approach aiming for a similar goal.

The approach via Hoare logic is based on primitive assertions of the form $\{p\}\, S\, \{q\}$, where p, q are predicates and S a statement. Operationally, this means that if S is activated in a state where p is true, then q is true of any state in which S might halt [134]. The approach to the *partial correctness* model via predicate

transformers defines $wlp.S.q$ (*weakest liberal precondition*) to be the weakest p such that $\{p\}\, S\, \{q\}$.

Refinement calculus is built around the concept of *total correctness*, that is, $(\forall q \cdot S.q = S.true \wedge wlp.S.q)$. We say that a predicate transformer S is totally correct with respect to precondition p and postcondition q, denoted by $p\ \{\!|S|\!\}\ q$, if and only if $p \subseteq S.q$ holds.

For example, for arbitrary statements S_1, S_2, and predicates p, r, and q, the correctness rule (in the Hoare style) for their sequential composition is the following:

$$\frac{p\ \{\!|S_1|\!\}\ r \qquad r\ \{\!|S_2|\!\}\ q}{p\ \{\!|S_1\,;\,S_2|\!\}\ q}$$

Algorithmic refinement. An action A is *(algorithmically) refined* by the action C, written $A \sqsubseteq C$, if, whenever A establishes a certain postcondition, so does C [22]:

$$A \sqsubseteq C \ \triangleq\ \forall q \bullet A.q \Rightarrow C.q$$

Refinement rules. A *refinement rule* allows us to deduce that a certain refinement $S \sqsubseteq S'$ is valid. For example, adding choices to an angelic assignment and removing choices from a demonic assignment are both valid refinements.

$$b \subseteq b' \quad \models \quad \{x := x' \mid b\} \ \sqsubseteq\ \{x := x' \mid b'\} \tag{2.16}$$
$$b' \subseteq b \quad \models \quad [x := x' \mid b] \ \sqsubseteq\ [x := x' \mid b'] \tag{2.17}$$

Equality "$=$" of statements can be used as refinement.

The rule of *backward propagation of an assertion*, or *pulling an assertion* through an angelic nondeterministic assignment is given below:

$$\{x := x' \mid b\}\,;\{q\} = \{x := x' \mid b \wedge q[x := x']\} \tag{2.18}$$

We also use the rule of *adding context assertions* to a nondeterministic choice of guarded statements:

$$\begin{array}{ll} & \{p\}\,;(g_1 \rightarrow S_1 \ [\!]\ \dots\ [\!]\ g_n \rightarrow S_n) \\ = & \{p\}\,;(g_1 \rightarrow \{g_1 \wedge p\}\,;S_1 \ [\!]\ \dots\ [\!]\ g_n \rightarrow \{g_n \wedge p\}\,;S_n) \end{array} \tag{2.19}$$

Dropping an assertion is also a valid refinement:

$$\{p\}\,;S \sqsubseteq S \tag{2.20}$$

A *local variable introduction* is a refinement of the form:

$$\textbf{begin var}\ y := e'\,;S\ \textbf{end} \sqsubseteq \textbf{begin var}\ x, y := e, e'\,;S'\ \textbf{end} \tag{2.21}$$

where S' is similar to S, possibly with total assignments to x added.

Next, we give one refinement rule for guarded iteration statements:

$$
\begin{array}{l}
\quad g_1 \to S_1 \sqsubseteq g'_1 \to S'_1 \wedge \ldots \wedge g_n \to S_n \sqsubseteq g'_n \to S'_n \\
\quad \wedge\, (g'_1 \vee \ldots \vee g'_n \Rightarrow g_1 \vee \ldots \vee g_n) \\
\Rightarrow \\
\quad \textbf{do}\ g_1 \to S_1 \ [\!]\ \ldots\ [\!]\ g_n \to S_n\ \textbf{od} \ \sqsubseteq \textbf{do}\ g'_1 \to S'_1 \ [\!]\ \ldots\ [\!]\ g'_n \to S'_n\ \textbf{od}
\end{array}
$$

Other refinement laws that we apply in proofs that are outlined in the Appendix are enumerated there.

Data refinement of actions. When carrying out data refinement, *abstract* data structures can be replaced by *concrete* ones (that is, ones more efficiently implementable).

Let us consider an action system that contains an atomic action A that depends on program variables a and z. Variables z are global to the analyzed action system. Next, let $R(a, c, z)$ (simply written as R) be a boolean *abstraction* relation, which links the *abstract* local variables a to the *concrete* local variables c. Additionally, let $I(c, z)$ be a predicate that depends on the concrete and global variables. Then, action A is *data refined* by action C using relation R and predicate I, that is, $A \sqsubseteq_{R,I} C$, if

$$\forall q \bullet R \wedge I \wedge A.q \Rightarrow C.(\exists a \bullet R \wedge I \wedge q),$$

where q is a predicate on the variables a, z, and $(\exists a.R \wedge I \wedge q)$ is a predicate on c, z. If R is the identity relation ($R \ \triangleq\ a = c$), we then write $A \sqsubseteq_I C$. Similarly, if $I \equiv$ true, we write $A \sqsubseteq_R C$. If both are trivial, we run into the usual algorithmic refinement of actions, $A \sqsubseteq C$, as defined previously.

Trace refinement of action systems. The semantics of a reactive action system is given in terms of behaviors [33]. A *behavior* of an action system $\mathcal{A}$ is a sequence of states,

$$b = (x_0, z_0), (x_1, z_1) \ldots,$$

where each state has two components. The first component is the *local state* and the second is the *global state*. Behaviors can be finite or infinite. A finite behavior is called *terminating* if it ends in a proper state, or *aborting* if it ends improperly, indicated by the symbol $\perp$.

A *trace*, $tr(b)$, of a behavior b, is obtained by removing the local state component in each state of a given system, and all finite stuttering (no change of the visible state). We denote the set of all traces of an action system $\mathcal{A}$ by $tr(\mathcal{A})$. An *aborting* trace corresponds to an aborting behavior.

The order relation over traces is an *approximation relation* $\preceq$. A trace t approximates a trace t', $t \preceq t'$, if either $t = t'$, or t is aborting and it is a prefix of t'. An action system $\mathcal{A}$ is *trace refined* by an action system $\mathcal{C}$, $\mathcal{A} \sqsubseteq \mathcal{C}$, if every trace

of $\mathcal{C}$ has an approximating trace in $\mathcal{A}$:

$$\mathcal{A} \sqsubseteq \mathcal{C} \ \triangleq\ \forall c \in tr(\mathcal{C}) \cdot \exists\, a \in tr(\mathcal{A}) \cdot a \preceq c$$

In practice, we use the following lemma to prove trace refinement of action systems [29].

Lemma 1 *Given the action systems*

$$\begin{array}{lcl} \mathcal{A}(z) & \triangleq & \textbf{begin var } a \bullet a, z := a_0, z_0 \,;\ \textbf{do } A \textbf{ od end} \\ \mathcal{C}(z) & \triangleq & \textbf{begin var } c \bullet c, z := c_0, z_0 \,;\ \textbf{do } C \,[\!]\, X \textbf{ od end}, \end{array}$$

let $R(a, c, z)$ be an abstraction relation and $I(c, z)$ an invariant of system $\mathcal{C}$. The concrete system $\mathcal{C}$ (trace) refines the abstract system $\mathcal{A}$, denoted $\mathcal{A} \sqsubseteq_{R,I} \mathcal{C}$, if:

1. *Initialization:* $R(a_0, c_0, z_0) \wedge I(c_0, z_0) \equiv \mathsf{true}$
2. *Main action:* $A \sqsubseteq_{R,I} C$
3. *Auxiliary action:* $\mathsf{skip} \sqsubseteq_{R,I} X$
4. *Continuation condition:* $R \wedge I \wedge gA \Rightarrow gC \vee gX$
5. *Internal convergence:* $R \wedge I \Rightarrow (\textbf{do } X \textbf{ od }).\mathsf{true}$ ■

The first condition of Lemma 1 says that the initializations of the systems $\mathcal{A}$ and $\mathcal{C}$ establish the invariant I and the abstraction relation R. The second condition requires the abstract action A to be (data) refined by the concrete action C, by using I and R. In turn, the third condition indicates that the auxiliary action X is obtained by data refining a skip action. This means that X behaves like skip with respect to the global variables z, which are not supposed to be changed in the refinement. The continuation condition means that whenever the action A of the abstract system $\mathcal{A}$ is enabled, provided that R and I hold, there must be an enabled action in the concrete system C as well. Finally, the fifth condition states that, if R and I hold, the execution of the auxiliary action X, taken separately, must eventually terminate.

Observe that, if we do not add any auxiliary actions, meaning that the refinement assumes only rewriting the existing actions, for optimization purposes, we do not have to check requirements 3, 4 and 5.

Decomposition. One way of developing a concurrent program is to first specify it without taking parallelism into consideration, and then add parallelism subsequently, by decomposing the initial specification.

In this spirit, Sekerinski and Sere [141] introduced a theorem for *prioritizing decomposition* of an action system, which we present below in an adapted form.

Theorem 1 *If action system $\mathcal{A}$ is of the form*

$$\mathcal{A} \ \triangleq\ \textbf{begin var } x, y \bullet [B_0 \wedge C_0]\,;\ \textbf{do } B \,[\!]\, g \rightarrow C \textbf{ od end},$$

where variables x do not occur in C_0, variables y do not occur in B_0, and furthermore variables x do not occur in C, and for some predicate I,

1. Initialization: $B_0 \wedge C_0 \Rightarrow I$,

2. Preservation: (B *preserves* I) $\wedge$ (C *preserves* I),

3. Exit condition: $I \wedge \neg gB \wedge gC \Rightarrow g$,

then

$$\mathcal{A} \sqsubseteq \mathcal{B} \,//\, \mathcal{C}$$

where

$$\begin{array}{lcl} \mathcal{B}(y) & \triangleq & \textbf{begin var } x \bullet\ [B_0 \wedge C_0]\,;\ \textbf{do } B \textbf{ od } \textbf{ end} \\ \mathcal{C}(y) & \triangleq & \textbf{begin } [C_0]\,;\ \textbf{do } C \textbf{ od } \textbf{ end} \end{array} \qquad \blacksquare$$

The *exit condition* ensures that, after eliminating g, action C does not become enabled in $\mathcal{B} \,//\, \mathcal{C}$ when it was not in $\mathcal{A}$, because then $\mathcal{B} \,//\, \mathcal{C}$ would not terminate when $\mathcal{A}$ would.

2.4 Continuous Action Systems

A *continuous action system* (CAS), proposed by Back, Petre and Porres [27], consists of a finite set of variables that can range over discrete or continuous valued time functions, together with a finite set of actions that act upon the variables. The variables form the state of the system. A CAS is of the form

$$\begin{array}{lcl} \mathcal{C}(z : \mathsf{Real}_+ \rightarrow T_z) & \triangleq & \textbf{begin var } x : \mathsf{Real}_+ \rightarrow T_x \bullet Init; \\ & & \qquad \textbf{do } g_1 \rightarrow S_1 \parallel \ldots \parallel g_n \rightarrow S_n \textbf{ od} \\ & & \textbf{end} : p \end{array} \tag{2.22}$$

Here, $x = (x_1, \ldots, x_n)$ are the *local variables* of the system, $Init$ is the initialization statement, while $A_i = g_i \rightarrow S_i$, $i = 1, \ldots, n$ are the *actions* of the system. The variables $z = (z_1, \ldots, z_k)$ are defined in the environment of the CAS and they are called *global* variables. As similar to the discrete case, the constants $p = (p_1, \ldots, p_m)$ are the *parameters* of system $\mathcal{C}$. Real_+ stands for the non-negative reals, and it is used as the time domain.

Intuitively, executing a CAS proceeds as follows. There is an implicit variable now, that shows the present time. Initially $now = 0$. The guards of the actions may refer to the value of now, as may any expressions in the action bodies (but they can not change now). The initialization $Init$ assigns initial time functions to the local and global variables. These time functions describe the default future behavior of the variables. The system will then start evolving according to these functions, with time (as measured by now) moving forward continuously. However, as soon as one of the conditions $g_1, \ldots, g_n$ becomes true, the system chooses one of the *enabled* actions, say $g_i \rightarrow S_i$, for execution. The choice is nondeterministic if there is more

than one such action. The body S_i of the action is then executed. It will usually change some variables by changing their future behavior. Variables that are not changed will behave as before. After the changes stipulated by S_i have been done, the system will evolve to the next time instance when one of the actions is enabled, and the process is repeated. The next time instance when an action is enabled may well be the same as the previous, that is, time needs not progress between the execution of two enabled actions. This is usually the case when the system is doing some (discrete, logical) computation to determine how to proceed next. It is possible that after a certain time instance, none of the actions is enabled anymore. This just means that, after this time instance, time diverges (grows unboundedly).

Note that in our approach actions are selected and executed asynchronously, compared to the hybrid automata formalism [86] where transitions are fired synchronously.

We write $x :- e$ for an assignment, rather than $x := e$, to emphasize that only the future behavior of the variable x is changed to the function e. The past behavior (before now) remains unchanged.

One of the main advantages of this model for hybrid computation is that both discrete and continuous behaviors are described in the same way. In particular, if the variables are only assigned constant functions, then we obtain a discrete computation.

Let $\mathcal{C}$ be the CAS described by (2.22). We explain the meaning of $\mathcal{C}$ by translating it into an ordinary action system. Its semantics is given in terms of the following action system $\overline{\mathcal{C}}$:

$$
\begin{array}{lcl}
\overline{\mathcal{C}}(z : \mathsf{Real}_+ \to T_z) & \triangleq & \textbf{begin}\ \ \textbf{var}\ now : \mathsf{Real}_+, x : \mathsf{Real}_+ \to T_x \bullet \\
 & & \quad now := 0 \,;\, Init \,;\, UT \,; \\
 & & \quad \textbf{do}\ \ g_1.now \to S_1 \,;\, UT \\
 & & \qquad [\!]\ \dots \\
 & & \qquad [\!]\ g_n.now \to S_n \,;\, UT \\
 & & \quad \textbf{od} \\
 & & \textbf{end} : p
\end{array}
\tag{2.23}
$$

Here, the variable now is declared, initialized and updated explicitly. It models the moments of time that are of interest for the system, that is, the starting time and the succeeding moments when some action is enabled. The value of now is updated by the statement UT (**U***pdate***T***ime*),

$$UT \quad \triangleq \quad now := \mathsf{next}.gg.now$$

Above, $gg = g_1 \vee ... \vee g_n$ and next is defined by

$$
\mathsf{next}.gg.t \triangleq \begin{cases} min\{t' \geq t \mid gg.t'\}, & \text{if exists } t' \geq t \text{ such that } gg.t' \\ +\infty, & \text{otherwise.} \end{cases}
$$

Thus, the function next gives a moment of time when at least one action is enabled. Only at such a moment can the future behavior of variables be modified. If no action will be ever enabled, then the second branch of the definition is followed, and the variable now goes to infinity.

We assume in this thesis that the minimum in the definition of next always exists when at least one guard is enabled in the present or future. Continuous action systems that do not satisfy this requirement are considered ill-defined (for example, a guard of the form $now > 0$ is ill-defined).

We define the *future update* $x :- e$ by

$$x :- e \triangleq x := x/now/e$$

where

$$x/t_0/e \triangleq (\lambda t \cdot \mathbf{if}\ t < t_0\ \mathbf{then}\ x.t\ \mathbf{else}\ e.t\ \mathbf{fi}\ )$$

Thus, only the future behavior of x is changed by this assignment.

A CAS is essentially a collection of time functions $x_1, \ldots, x_n$ over the non-negative reals, defined in a stepwise manner. The steps form a sequence of intervals $I_0, I_1, I_2, \ldots$, where each interval I_k is either a left closed interval of the form $[t_i \ldots t_{i+1})$, or a closed interval of the form $[t_i, t_i]$, that is, a point. The action system determines a family of functions, $x_1, \ldots, x_n$, which are defined over this sequence of intervals and points. The extremes of these intervals correspond to the control points of the system where a discrete action is performed.

The behavior of a hybrid system is often described using a system of *differential equations*. Continuous action systems allow for this kind of definitions, by introducing the shorthand $\dot{x} :- f(x)$. This will assign to x a time function that satisfies the given differential equation, such that the function x will evolve continuously.

As an example, if $f = (\lambda t \bullet v)$, where v is a constant value, then we have that

$$\dot{x} :- v \quad \triangleq \quad x :- (\lambda t \cdot x.now + v * (t - now))$$

We can use *clock variables* or *timers* to measure the passage of time and to correlate the execution of an action with time. A clock variable measures the time elapsed since it was set to zero. Assume that c is a time variable of type Real_+. We then use the following definition for resetting the clock c:

$$reset(c) \triangleq c :- (\lambda t \cdot t - now)$$

Since a clock variable is just a regular variable, we can define as many clocks as we need and reset them independently. It is also possible to do arithmetic operations with clock variables, e.g., to use time intervals in guards. These features make the formalism well suited for modeling real-time systems, too.

2.5 A Brief Description of Mathematica

Mathematica [156] is a computer algebra tool, developed at Wolfram Research, USA, which integrates a numeric and symbolic computational engine, graphics system, programming language, documentation system, and advanced connectivity to other applications.

We have chosen Mathematica as a development environment for implementing a symbolic simulation tool for hybrid systems. The tool is described in chapter 5. Since the continuous laws of evolution within hybrid systems are defined in terms of differential equations, it is most important to have analysis tools that can handle solving such equations. This and other reasons that support our choice are enumerated below:

- Mathematica handles complex symbolic calculations that often involve a large number of terms.
- It has advanced capabilities for plotting and visualizing data.
- One can solve equations, differential equations, and minimization problems, numerically or symbolically.
- Mathematica has a simple interface, which can be used directly; however, the package can also be used through a web browser, or by other systems as a back-end computational engine.

Mathematica as a Programming Language. Programmed computations are of great help. Mathematica incorporates a range of programming paradigms, which facilitate writing programs in a flexible and intuitive manner.

In Mathematica, there is no need to predeclare variable types, or dimensions of lists and arrays, to direct memory management, or to compile the programs.

Chapter 3

Synchronized Discrete Reactive Systems

Designing for reactivity entails dealing with communication, composability, concurrency and preemption. Out of these, *concurrency* is related to the fundamental aspects of systems with *multiple*, *simultaneously active* computing agents that interact with one another. The complexity of such systems comes as an inherent byproduct, which leads further to problems concerning the correctness of the steps performed in the development flow.

In this chapter, we concentrate on two major problems regarding the design of reactive systems, be they software or hardware-targeted systems: behavior control and modularity. A feasible design methodology requires that the designer composes the system from parallel concurrent components called *modules*. Such modules are modeled here by action systems. We approach aspects of concurrency and modular design from the perspective of the system–level integrator who has access to a library of predefined modules. His only task is to appropriately connect them in order to get the system functionality.

The built-in interleaving semantics for handling concurrency in action systems goes together with behavioral nondeterminism. Observations of an interleaved model are sequential, therefore the updates of two systems executing in parallel may not be consistent over a set of executions [128]. Hence, though versatile and general, this way of modeling large systems can have a negative impact on the data flow control and the *composability* of the modules that interact concurrently. When plugging modules together, one has to specify additional details about the order in which they exchange information. This requirement may compromise the data abstraction at the interface of a module. We will illustrate these phenomena by examples.

In the following, we provide a solution to the above mentioned problems, by introducing an additional concurrency mechanism for action systems, as a way to describe controllable behavior [60, 61]. The mechanism mimics *barrier synchronization*, a common technique of reaching thread synchronization, found as

a primitive within concurrent programming languages [84]. For this purpose, we define a new parallel composition operator. The concepts that we formulate still rely on the established mathematical techniques underlying action systems. Moreover, the synchronized environment provides the designer with additional means for system development.

Our goal is completed by showing that the new virtual execution environment also enhances the capabilities of our framework, for modular design. Modules may be picked up from existing libraries and just plugged into the system representation. The traditional techniques of trace refinement [33], introduced in chapter 2, are used to ensure that the implementation is correct with respect to a system specification that faithfully captures the system's global reaction to all sets of inputs.

We believe that the result of this chapter's contribution, that is, the positive effect of the proposed synchronized environments on the design modularity, could also be applied to other formal frameworks, to obtain similar benefits.

3.1 Interleaved vs. True Concurrency

In *interleaving* semantics without fairness assumptions, concurrency is equated with nondeterministic sequentiality. Informally, this law can be expressed as below:

$$\begin{aligned} & \mathbf{do}\ Action_1 \mid\mid Action_2\ \mathbf{od} \\ = \ & (Action_1 \ [\!] \ Action_2)\ ;\ (Action_1 \ [\!] \ Action_2)\ ;\ (Action_1 \ [\!] \ Action_2)\ ;\ \ldots \end{aligned}$$

In *true concurrency* semantics [135] this equation does not hold. On the other hand, interleaving semantics are held to be mathematically more tractable, whereas true concurrency semantics are better for dealing with certain properties, such as fairness.

3.2 Traditional Model of Action Systems Execution

Traditionally, action systems are executed in a sequential manner. Parallel executions are modeled by interleaving actions, without fairness considerations. There also exists a parallel execution model for action systems, with fairness conditions [32]. However, in the following, we assume the truly demonic interleaved model as the default one.

The initialization places the system in a stable, starting state. From there, any enabled action may be selected for execution. Only one action is chosen, in a (demonically) nondeterministic way. The statements inside the $\mathbf{do} - \mathbf{od}$ loop of a system $\mathcal{A}$, as illustrated by (2.15), are iterated as long as $gg_A \equiv \mathsf{true}$. Termination is normal if the exit condition ($\neg gg_A$) holds.

Thus, the execution of an action system assumes that there exists a virtual external entity - the **execution controller** (**controller** in short) - which, at any moment, knows what actions are enabled. After initialization, the controller selects for execution any of the enabled actions. After the completion of the action execution,

the system moves to a new state. We call this operation an *execution round*. Notice that an execution round is equivalent to the execution of an action. After this, the controller, which behaves demonically, evaluates the new state, observes the enabled actions and starts another execution round.

Next, let us visually exemplify the above mentioned round-based execution scenario. Consider two action systems $\mathcal{A}$ and $\mathcal{B}$:

$$\begin{array}{ll} & \mathcal{A}(z_A : T_z) \\ \triangleq & \textbf{begin var } x_A : T_x \bullet z_A := z_{A_0} \,;\, x_A := x_{A_0}; \\ & \quad \textbf{do } A_1 \,[\!]\, A_2 \textbf{ od} \\ & \textbf{end} \end{array}$$

$$\begin{array}{ll} & \mathcal{B}(z_B : T_z) \\ \triangleq & \textbf{begin var } x_B : T_x \bullet z_B := z_{B_0} \,;\, x_B := x_{B_0}; \\ & \quad \textbf{do } B_1 \,[\!]\, B_2 \textbf{ od} \\ & \textbf{end} \end{array}$$

An intuitive illustration of the execution of $\mathcal{A}$, $\mathcal{B}$ is shown at the upper part of

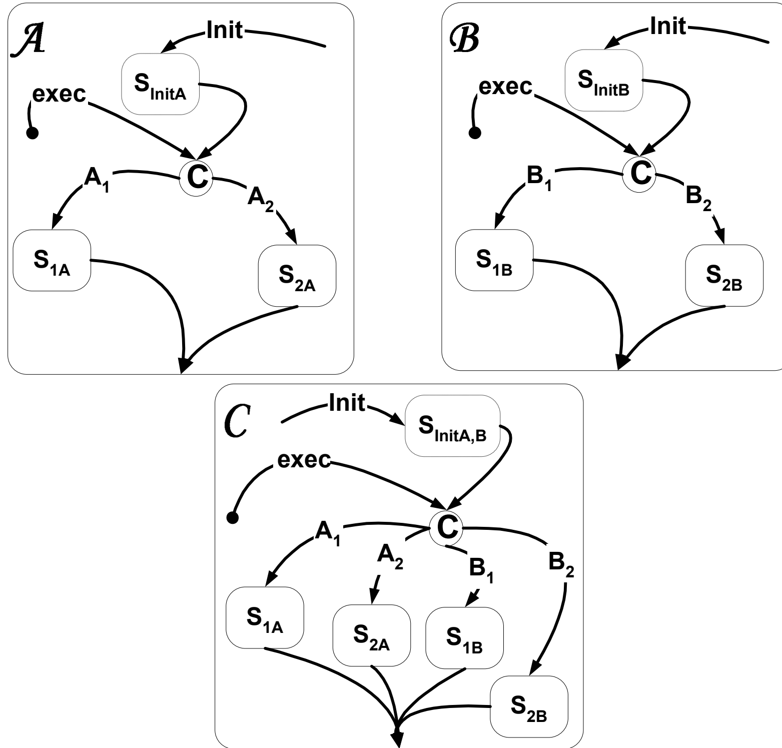


Figure 3.1: Execution Visualization of the Traditional Parallel Composition.

Figure 3.1, as statecharts-like descriptions.

After the initial entry (the transition labeled *Init*), the systems $\mathcal{A}$ and $\mathcal{B}$ evolve either towards the state S_{1A}, or S_{2A}, and S_{1B}, or S_{2B}, respectively, depending on which action is enabled in each of the action systems. This coincides with the end of one execution round. The execution continues (if possible) by reentering the states $Sys_{\mathcal{A}}$ or $Sys_{\mathcal{B}}$, now by taking the (default) transition labeled *exec*, through the choice operator. If both actions (A_1 and A_2, or B_1 and B_2, respectively) are simultaneously enabled, the selection at the choice-point is nondeterministically taken by the controller. The two substates of $Sys_{\mathcal{A}}$ or $Sys_{\mathcal{B}}$ are visualized as OR-states [83].

The parallel composition of $\mathcal{A}$ and $\mathcal{B}$ is specified as the action system $\mathcal{C}$: $\mathcal{C} = \mathcal{A} \mid\mid \mathcal{B}$. Observe that, in the corresponding part of Figure 3.1, the states $S_{1A}, S_{2A}, S_{1B}, S_{2B}$ are collected together as OR sub-states of $Sys_{\mathcal{C}}$. Any execution round takes the system into one of the mentioned states, depending on the enabledness of the composing actions, or on the selection of the controller.

3.2.1 Example: A Digital Filter

Let us illustrate the interleaved execution model by a simple example. We consider the task of modeling a digital *filter* [101]. Briefly, a filter is a module that takes as input a sequence of samples, performs certain operations on it, and delivers as output a corresponding sequence of samples. The incoming sequence is described as $X[n]$, where X is the input signal and n identifies the sample position; a similar notation applies to the output signal Y, for which we have the samples $Y[n]$. The relation between the input and output is given by $Y[n] = \sum_{k=0}^{N-1} h[k] \times X[n-k]$, where the vector $h[0, ..N-1]$ contains the filter *coefficients*. Hence, apart from the incoming current sample of X, $(N-1)$ previous samples are stored in a buffer and can be accessed by the filter. Finally, a filter may have either a software or a hardware implementation. From the above informal description of the filter we can

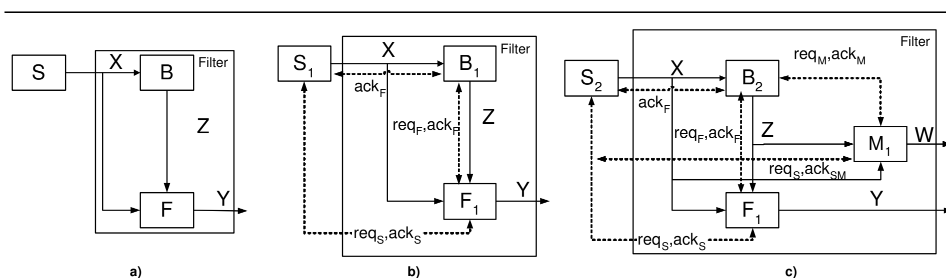


Figure 3.2: Simple filter representation.

identify two modules of such a device: the storage First-In-First-Out (FIFO)-like buffer, and the actual implementation of the filter functionality. In the following, we model the signal source by system $\mathcal{S}$, the buffer by system $\mathcal{B}$, whereas system

$\mathcal{F}$ models the actual filter. This is illustrated in Figure 3.2 a). The complete action system description is the parallel composition of the available modules, that is, $\mathcal{P} = \mathcal{S} \parallel \mathcal{B} \parallel \mathcal{F}$. The composed system as well as the modules are given below. Note that the common global variables are initialized in a consistent way, by all modules. Also, z_0 and y_0 model sequences of values, respectively.

$$\begin{array}{ll} & \mathcal{S}(X : T) \\ \triangleq & \textbf{begin } X := x_0 \, ; \\ & \quad \textbf{do } [X := X' \mid X' \in T] \textbf{ od} \\ & \textbf{end} \end{array} \tag{3.1}$$

$$\begin{array}{ll} & \mathcal{B}(X, Z[0,..N-2] : T) \\ \triangleq & \textbf{begin } X, Z[0,..N-2] := x_0, z_0 \, ; \\ & \quad \textbf{do } Z[0], Z[1], \ldots, Z[N-2] := X, Z[0], \ldots, Z[N-3] \textbf{ od} \\ & \textbf{end} \end{array} \tag{3.2}$$

$$\begin{array}{ll} & \mathcal{F}(X, Z[0,..N-2], Y : T) \\ \triangleq & \textbf{begin } X, Z[0,..N-2], Y := x_0, z_0, y_0 \, ; \\ & \quad \textbf{do } Y := \sum_{k=1}^{N-1} h[k] \times Z[k-1] + h[0] \times X \textbf{ od} \\ & \textbf{end} : h[0,..N-1] \end{array} \tag{3.3}$$

Hence, by computing $\mathcal{P}$, we get:

$$\begin{array}{ll} & \mathcal{P}(Y : T) \\ = & \textbf{begin var } X, Z[0,..N-2] : T \bullet \\ & \quad\quad X, Z[0,..N-2], Y := x_0, z_0, y_0 \, ; \\ & \quad \textbf{do } [X := X' \mid X' \in T] \\ & \quad\quad [\!] \; Z[0], Z[1], \ldots, Z[N-2] := X, Z[1], \ldots, Z[N-3] \\ & \quad\quad [\!] \; Y := \sum_{k=1}^{N-1} h[k] \times Z[k-1] + h[0] \times X \\ & \quad \textbf{od} \\ & \textbf{end} : h[0,..N-1] \end{array} \tag{3.4}$$

Observe first that the interleaved execution of $\mathcal{P}$ does not guarantee that every signal emitted by $\mathcal{S}$ is correspondingly received by $\mathcal{B}$ and $\mathcal{F}$; several executions of $\mathcal{S}$ may be selected, before any of $\mathcal{B}$ or $\mathcal{F}$. Moreover, different values can be assigned to Y for the same sequence of samples provided by $\mathcal{S}$, depending on the

order of selection for execution of the systems $\mathcal{B}$ and $\mathcal{F}$. Only one of these results is correct.

Both problems can be solved by specifying a certain order in which the modules of $\mathcal{P}$ should be executed. This can be achieved by introducing the communication variables req_S, req_F and ack_S, ack_F, and by devising a communication protocol such that the desired order is enforced. Hence, the systems should know about the status of the partners, as indicated by the elements of the communication channel. The systems may be remodeled as $\mathcal{S}_1, \mathcal{B}_1, \mathcal{F}_1$:

$$
\begin{array}{rl}
& \mathcal{S}_1(req_S, ack_S, ack_F : Bool \,;\, X : T) \\
= & \textbf{begin}\ req_S := \mathsf{false} \,;\, ack_S := \mathsf{false} \,;\, ack_F := \mathsf{false} \,;\, X := x_0; \\
& \quad \textbf{do} \\
& \qquad \neg(req_S \vee ack_S \vee ack_F) \qquad\qquad /*\ \text{action } A^1_{S_1}\ */ \\
& \qquad \rightarrow [X := X' \mid X' \in T] \,;\ req_S := \mathsf{true} \\
& \quad\ \ [\!]\ req_S \wedge ack_S \wedge \neg ack_F \qquad\qquad /*\ \text{action } A^2_{S_1}\ */ \\
& \qquad \rightarrow req_S := \mathsf{false} \\
& \quad \textbf{od} \\
& \textbf{end}
\end{array}
$$

$$
\begin{array}{rl}
& \mathcal{B}_1(req_F, ack_F : Bool \,;\, X, Z[0,..N-2] : T) \\
= & \textbf{begin}\ req_F := \mathsf{false} \,;\, ack_F := \mathsf{false} \,;\, X := x_0 \,;\, Z[0,..N-2] := z_0; \\
& \quad \textbf{do} \\
& \qquad req_F \wedge \neg ack_F \qquad\qquad /*\ \text{action } A^1_{B_1}\ */ \\
& \qquad \rightarrow Z[0], Z[1], \ldots, Z[N-2] := X, Z[0], \ldots, Z[N-3]; \\
& \qquad\quad ack_F := \mathsf{true} \\
& \quad\ \ [\!]\ \neg req_F \wedge ack_F \qquad\qquad /*\ \text{action } A^2_{B_1}\ */ \\
& \qquad \rightarrow ack_F := \mathsf{false} \\
& \quad \textbf{od} \\
& \textbf{end}
\end{array}
$$

$$
\begin{array}{rl}
& \mathcal{F}_1(req_S, req_F, ack_S, ack_F : Bool \,;\, X, Z[0,..N-2], Y : T) \\
= & \textbf{begin}\ req_S := \mathsf{false} \,;\, req_F := \mathsf{false} \,;\, ack_S := \mathsf{false} \,;\, ack_F := \mathsf{false}; \\
& \qquad X := x_0 \,;\, Z[0,..N-2] := z_0 \,;\, Y := y_0; \\
& \quad \textbf{do} \\
& \qquad req_S \wedge \neg(req_F \vee ack_F) \wedge \neg ack_S \qquad\qquad /*\ \text{action } A^1_{F_1}\ */ \\
& \qquad \rightarrow Y := \sum_{k=1}^{N-1} h[k] \times Z[k-1] + h[0] \times X \,;\ req_F := \mathsf{true} \\
& \quad\ \ [\!]\ req_F \wedge ack_F \qquad\qquad /*\ \text{action } A^2_{F_1}\ */ \\
& \qquad \rightarrow req_F := \mathsf{false} \,;\, ack_S := \mathsf{true} \\
& \quad\ \ [\!]\ \neg req_S \wedge ack_S \qquad\qquad /*\ \text{action } A^3_{F_1}\ */ \\
& \qquad \rightarrow ack_S := \mathsf{false} \\
& \quad \textbf{od} \\
& \textbf{end} : h[0,..N-1]
\end{array}
$$

The result of their parallel composition, $\mathcal{P}_1 = \mathcal{S}_1 \parallel \mathcal{B}_1 \parallel \mathcal{F}_1$, is illustrated by the

block diagram of Figure 3.2 b), where the communication variables are shown as dotted lines.

By implementing the communication channels, the execution of $\mathcal{P}_1$ becomes a deterministic sequence of activities, given as:

$$A^1_{S_1} \rightarrow A^1_{F_1} \rightarrow A^1_{B_1} \rightarrow A^2_{F_1} \rightarrow A^2_{B_1} \rightarrow A^2_{S_1} \rightarrow A^3_{F_1} \rightarrow A^1_{S_1} \rightarrow \ldots$$

The meaning of the sequence is as follows. First, the system $\mathcal{S}_1$ is deactivated, after which $\mathcal{F}_1$ performs the filtering, and then informs $\mathcal{B}_1$, by setting $req_F := \mathsf{true}$, that it can perform its operation. When this is accomplished, $\mathcal{B}_1$ signals to $\mathcal{F}_1$ ($ack_F := \mathsf{true}$). Next, $\mathcal{F}_1$ signals the end of the operation, to $\mathcal{S}_1$ ($ack_S := \mathsf{true}$), followed by a couple of rounds for resetting the acknowledge signals ack_F (by action $A^2_{B_1}$), and ack_S (by $A^3_{F_1}$). Another sample can now be presented by $\mathcal{S}_1$, and so on.

Consider further that, in the above example, X is an audio signal and $\mathcal{F}_1$ models a low-pass filter. The output of $\mathcal{F}_1$ would go to the woofer speaker of one's audio system. We would also like to have a high-pass filter, the output of which would go to the corresponding speakers of the same audio system. We want to reuse the previously designed modules, and then add one that can detect the high frequencies of the incoming signal. The high-frequency filter is modeled by the new system $\mathcal{M}_1$:

$$\begin{array}{ll}
& \mathcal{M}_1(req_S, req_M, ack_{SM}, ack_M : Bool\,;\, X, Z[0,..N-2], W : T) \\
\triangleq & \textbf{begin}\ req_S := \mathsf{false}\,;\, req_M := \mathsf{false}\,;\, ack_{SM} := \mathsf{false}\,;\, ack_M := \mathsf{false}; \\
& \qquad X := x_0\,;\, Z[0,..N-2] := z_0\,;\, W := w_0; \\
& \quad \textbf{do} \\
& \qquad req_S \wedge \neg(req_M \vee ack_M) \wedge \neg ack_{SM} \\
& \qquad \rightarrow W := \sum_{k=1}^{N-1} h_M[k] \times Z[k-1] + h_M[0] \times X\,;\; req_M := \mathsf{true} \\
& \quad [\!]\ req_M \wedge ack_M \\
& \qquad \rightarrow req_M := \mathsf{false}\,;\, ack_{SM} := \mathsf{true} \\
& \quad [\!]\ \neg req_S \wedge ack_{SM} \\
& \qquad \rightarrow ack_{SM} := \mathsf{false} \\
& \quad \textbf{od} \\
& \textbf{end} : h_M[0,..N-1]
\end{array}$$

Structurally, $\mathcal{F}_1$ and $\mathcal{M}_1$ are the same, the difference residing in the values of the coefficients, $h[0,..N-1]$ and $h_M[0,..N-1]$, respectively. $\mathcal{M}_1$ uses the same input signals, X, Z, and req_S (to which it answers with ack_{SM}).

In order to accommodate the introduction of $\mathcal{M}_1$, the system $\mathcal{B}_1$ has to wait for the two filters to read its data, once a new sample has been issued by $\mathcal{S}_1$. Consequently, we have to change the representation of $\mathcal{B}_1$. The same is required for $\mathcal{S}_1$, since it has to communicate with $\mathcal{M}_1$, too. Thus, S_1 becomes S_2 (here, we omit the description of the latter). The new system $\mathcal{B}_2$ is described as follows, whereas

the whole filter is illustrated in Figure 3.2 c).

$$
\begin{array}{rl}
& \mathcal{B}_2(req_F, ack_F, req_M, ack_M : Bool\ ;\ X, Z[0,..N-2] : T) \\
= & \mathbf{begin}\ req_F := \mathsf{false}\ ;\ ack_F := \mathsf{false}\ ;\ req_M := \mathsf{false}\ ;\ ack_M := \mathsf{false}; \\
& \qquad X := x_0\ ;\ Z[0,..N-2] := z_0; \\
& \quad \mathbf{do} \\
& \quad\quad req_F \wedge req_M \wedge \neg ack_F \wedge \neg ack_M \\
& \quad\quad \to Z[0], \ldots, Z[N-2] := X, \ldots, Z[N-3]\ ;\ \ ack_F, ack_M := \mathsf{true} \\
& \quad [\!]\ \neg req_F \wedge \neg req_M \wedge ack_F \wedge ack_M \\
& \quad\quad \to ack_F, ack_M := \mathsf{false} \\
& \quad \mathbf{od} \\
& \mathbf{end}
\end{array}
$$

Discussion. The interleaving model of execution brings the benefit of a very simple concept. In order to reduce the implicit nondeterministic behavior of the model, inappropriate in certain situations, as shown in our example, one may introduce control channels. These ensure that the data emitted by one source is not missed by any of the intended targets, or that data is processed in a correct manner.

However, there is another aspect of the problem, not yet solved by the exemplified introduction of the communication channels. An observer of the composed system $\mathcal{P}_2 = \mathcal{S}_2 \parallel \mathcal{B}_2 \parallel \mathcal{F}_1 \parallel \mathcal{M}_1$ (the listener, in the example) has access to both output sequences, $Y(n)$ and $W(n)$ (Figure 3.2 c)). Depending on the execution order of $\mathcal{F}_1$ and $\mathcal{M}_1$, until the listener observes the new output $(Y(n+1), W(n+1))$, it also observes the intermediate state, that is, either $(Y(n), W(n+1))$, or $(Y(n+1), W(n))$, which is also an incorrect aspect of the design. A solution is provided, again, by introducing new communication channels, between $\mathcal{F}_1$ and $\mathcal{M}_1$, on one side, and the observer, on the other. What happens if multiple, different observers become necessary in the design?

Any extension / reduction of the design elements requires an internal change of the involved modules. This clearly destroys any hope for a modular design flow and the reuse of components in future projects. We may assign meanings like “data valid”, “operation finished”, etc., to the signals of the communication channels, thus the interleaved approaches are suitable for asynchronous designs [152]. Unfortunately, these signals are global variables of the model. In hardware, generally, this translates into “more wires”; in software, this violates the principle of information hiding [41]. In the following section, we propose a solution to this kind of design issues.

3.3 Synchronized Parallel Environments

Synchronized environment. We want to build an environment in which the response of the system is a collection of the individual module reactions to the input stimuli. The solution that we propose requires that the modules synchronize when

the global variables of the compound system are updated. This is achieved by extending the execution round concept, as described in section 3.2, to an *execution cycle*. A cycle is defined by the activities carried out by the system between two global states: it is a sequence of rounds in which each participating action system updates the local variables, as necessary, followed by a last round, in which, simultaneously, *all* the global variables are updated, accordingly. Note that between rounds, the global state of the system does not change.

From the controller's point of view, we can imagine the following scenario. It selects for execution an enabled action from one action system module. If the action updates global variables, the system is marked as "executed", and no other action can be selected from that system. However, the other participants, or possible external observers, do not see the changes yet. Another action is then selected, from an "unexecuted" action system. The process continues until all the modules are marked "executed". This also signals the end of a cycle, when all the global variables are updated.

3.3.1 Partitioned Action Systems

The translation of the above scenario into our framework requires certain characteristics for the action systems employed in design. These requirements are introduced by Definition 1. Recall from chapter 2 that wS is the set of variables written by an action S, $gw\mathcal{A}$ is the set of global variables written by an action system $\mathcal{A}$, and $lw\mathcal{A}$ models the local variables written by the same action system.

Definition 1 *Consider the action system* $\mathcal{A}$

$$\mathcal{A}(z : T_z) \triangleq \textbf{begin var } x : T_x \bullet Init\,; \textbf{do}\, g_L \rightarrow L \;[\!]\; g_S \rightarrow S\, \textbf{od end} \quad (3.5)$$

We say that $\mathcal{A}$ *is a* **partitioned** *action system if:*

1. $gw\mathcal{A} \subseteq wS$ *- meaning that* S *is the global action of* $\mathcal{A}$*. Notice that* wS *may also contain local variables of* $\mathcal{A}$*.*

2. $wL \subseteq lw\mathcal{A}$ *- meaning that* L *is the local action of* $\mathcal{A}$*.*

3. $(\textbf{do}\, g_L \rightarrow L\, \textbf{od}).(\neg g_L \wedge g_S) \equiv \mathsf{true}$ *- meaning that the execution of* L*, taken separately, terminates, and establishes the precondition for executing* S*.* ■

Notice that the specification $\mathcal{A}$, as given by Definition 1, encodes more visibly than (2.15), the mechanism that triggers the global state changes. Also, in the above definition we forbid nested loops. If $gw\mathcal{A}$ is empty, the action system $\mathcal{A}$ can not function as an actuator, since it does not modify any global variable.

3.3.2 The Synchronization Operator ($\sharp$)

Let us consider n partitioned action systems ($k \in [1..n]$):

$$\mathcal{A}_k(z_k) \triangleq \textbf{begin var } x_k \bullet \mathit{Init}_k \,;\, \textbf{do } g_L^k \rightarrow L_k \,[\!]\, g_S^k \rightarrow S_k \textbf{ od end}, \quad (3.6)$$

for which we also have that $\forall j, k \in [1..n], j \neq k \bullet ((gw\mathcal{A}_j \cap gw\mathcal{A}_k = \emptyset) \wedge (x_j \cap x_k = \emptyset))$ (the action systems do not write on the same global variables and they also have pairwise disjoint local variables).

The *synchronized parallel composition* of the above systems is a new action system $\mathcal{P} = \mathcal{A}_1 \sharp \ldots \sharp \mathcal{A}_n$, given by:

$\mathcal{P}(z)$
$\triangleq$ **begin var** $x : T_x, sel[1,..n] : \mathsf{Bool}, run : \mathsf{Nat} \bullet \mathit{Init};$
do
gg_P
$\rightarrow$

	selection action
$run = 0 \wedge \neg sel[1]$	
$\rightarrow sel[1] := \mathsf{true} \,;\, run := 1$	
$[\!]$ $\ldots$	
$[\!]$ $run = 0 \wedge \neg sel[n]$	
$\rightarrow sel[n] := \mathsf{true} \,;\, run := n$	
$[\!]$ $run = 1 \wedge g_L^1$	module $\mathcal{A}_1$
$\rightarrow L_1$	
$[\!]$ $run = 1 \wedge \neg g_L^1 \wedge g_S^1$	
$\rightarrow wS_1c := wS_1 \,;\, S_1' \,;\, run := 0$	
$[\!]$ $run = 1 \wedge \neg gg_{A_1}$	
$\rightarrow run := 0$	
$[\!]$ $\ldots$	
$[\!]$ $run = n \wedge g_L^n$	module $\mathcal{A}_n$
$\rightarrow L_n$	
$[\!]$ $run = n \wedge \neg g_L^n \wedge g_S^n$	
$\rightarrow wS_nc := wS_n \,;\, S_n' \,;\, run := 0$	
$[\!]$ $run = n \wedge \neg gg_{A_n}$	
$\rightarrow run := 0$	
$[\!]$ $sel \wedge run = 0$	update action
$\rightarrow \mathit{Update} \,;\, sel := \mathsf{false}$	

od
end

The operator '$\sharp$' ("sharp") is called the *synchronization parallel* operator.

The set z of global variables of $\mathcal{P}$ is, initially, the union of the global variables sets of each module: $z = \bigcup_k z_k$. It may be possible that communication between some modules of $\mathcal{P}$ (the composing systems $\mathcal{A}_k$) should not be disclosed at the interface of $\mathcal{P}$. Therefore, the variables that model such channels will be *hidden* within the system $\mathcal{P}$. They will not be mentioned in z.

Further, the local variables x of the new action system $\mathcal{P}$ are the union of the local variables x_k, to which we add the hidden variables. We also add copies (wS_kc) of the original write variables of each action body S_k. They replace the original variables wS_k, therefore we have $S'_k = S_k[wS_k := wS_kc]$. Finally, the list x is completed by adding the array sel and the execution indicator, run. The predicate gg_P is a short notation for the disjunction of the guards of all the actions in the systems $\mathcal{A}_k$: $gg_P \triangleq \bigvee_1^n gg_{A_k}$, where $gg_{A_k} = g_L^k \vee g_S^k$.

The $Init$ statement is the sequential composition of the individual $Init_k$ statements, to which we add the initialization of variables wS_kc, sel and run:

$$\begin{aligned} Init \;\triangleq\;& Init_1 ; \ldots ; Init_n ; wS_1c, \ldots, wS_nc := wS_1, \ldots, wS_n; \\ & run := 0 ; sel := \mathsf{false} \end{aligned}$$

The action $Update$ is given by:

$$Update \triangleq Update_1 ; \ldots ; Update_n, \text{ where } Update_k \triangleq wS_k := wS_kc.$$

The above definition of the '$\sharp$' - based composition says that, whenever there is a change in the input, such a composition of action systems reacts based on the state of all its modules. The result is composed of the individual reactions of each of the modules. The system composition reacts only if at least one module is enabled ($\exists k \in [1..n] \bullet gg_{A_k} \equiv \mathsf{true}$). Moving certain variables to the local level, within the system $\mathcal{P}$, is motivated by the containment of local communication. The variable run identifies the system that is selected for execution. The variable sel stores the information on the executing, or already executed systems. Whenever all of the elements of the array sel become true ($sel = sel[1] \wedge \ldots \wedge sel[n]$), and $run = 0$, we have reached the end of an execution cycle. At this moment, the assignment $sel := \mathsf{false}$ is understood as a shorthand notation for $sel[1] := \mathsf{false}; \ldots; sel[n] := \mathsf{false}$.

The assignment $wS_kc := wS_k$ that precedes the action S'_k takes into account that the (local) variables of wS_k could have been also updated by L_k. As they may also belong to rS_k, their current values must be taken into consideration. In case actions L_k do not modify rS_k, the presence of this assignment is not necessary. The same applies when there are no local actions in a given partitioned action system.

Further, we are able to find useful properties of system $\mathcal{P}$, expressed by the following theorem (the proof is shown in Appendix A-1).

Theorem 2 *Assume that the partitioned action systems $\mathcal{A}_1$ and $\mathcal{A}_2$ are of the form given by (3.5). Then, the synchronized parallel composition $\mathcal{A}_1 \sharp \mathcal{A}_2$ satisfies the following properties:*

(a) *$\mathcal{A}_1 \sharp \mathcal{A}_2$ is a partitioned action system ($\sharp$ preserves partitioning)*

(b) *$\mathcal{A}_1 \sharp \mathcal{A}_2 = \mathcal{A}_2 \sharp \mathcal{A}_1$ (commutativity of $\sharp$)* ■

The Update Action. In a more general view, we would avoid imposing the restriction that $\forall j, k \in [1..n], j \neq k. gw\mathcal{A}_j \cap gw\mathcal{A}_k = \emptyset$. Designs where the systems do not have disjoint sets of global write variables are not necessarily examples of bad designs. A well known example of such situations is the bus-based design of digital systems, where multiple participants in the processing effort share a common resource, the bus lines. Of course, the situation requires a thorough analysis, and there exist multiple solutions that resolve the inevitable conflicts. Therefore, the action $Update$ might not be merely the action that updates the respective variables; instead, it can rather be the action that "solves" such conflicts. Intuitively, this means that the system $\mathcal{P}$ must also allow the designer to separately specify the action $Update$. Relaxing the above mentioned assumption is subject to further studies. Preliminarily, such an approach has been already studied by Seceleanu and Garlan [143], for modeling self-adaptive systems, in multimedia environments.

Execution Visualization. A graphical, statecharts-like representation of the execution model as introduced by the synchronization operator is illustrated in Figure 3.3. Considering two action systems $\mathcal{A}$ and $\mathcal{B}$, each with a single action A and B, respectively, we build the synchronized environment $\mathcal{C} = \mathcal{A} \sharp \mathcal{B}$. After splitting the original actions of $\mathcal{A}$ and $\mathcal{B}$ into local and global actions, the first two illustrations of Figure 3.3 represent the execution of $\mathcal{A}$ and $\mathcal{B}$, as also discussed in section 3.2. There is one difference, seen in the third representation, residing in the mutual exclusion of the transition conditions (g'_L and g'_S); therefore, the controller does not make a selection, the choice being clear. In addition, now, the execution model corresponding to $\mathcal{C}$, differently from the one in Figure 3.1, shows an AND-based statecharts description.

Several execution rounds may be necessary to bring to termination the local activities of each of the synchronized components. This is reflected by the transitions to the border of the corresponding state from the internal L_A or L_B OR-states. Further, before executing the action *Update*, in order to reach the state G_{AB} – that corresponds to the end of an execution cycle - both substates $\mathcal{A}$ and $\mathcal{B}$ have to be exited.

Design Implications. We revisit briefly the example proposed in Section 3.2.1. Consider that instead of the parallel composition $\mathcal{P} = \mathcal{S} \parallel \mathcal{B} \parallel \mathcal{F}$, we write the description of our system as $\mathcal{P} = \mathcal{S} \sharp \mathcal{B} \sharp \mathcal{F}$. Clearly, the modules $\mathcal{S}, \mathcal{B}, \mathcal{F}$ are partitioned action systems, since they have only global actions. When using such a composition, we do not have to add communication channels to any of the modules, which all remain simple, as described by (3.1), (3.2), (3.3) and (3.4). Also, in case of a synchronized environment, the multiplicity of targets stops being an issue for the system $\mathcal{P}$. We can introduce as many $\mathcal{F}$-like systems as required, without modifying $\mathcal{B}$ or $\mathcal{S}$ in order to accommodate the presence of the new modules. Additionally, an external observer will always observe only the state $(Y(n+1), W(n+1))$, regardless of the order in which the systems $\mathcal{F}$ and the corresponding $\mathcal{M}$ ($\mathcal{M}_1$ without the communication variables) are selected for execution.

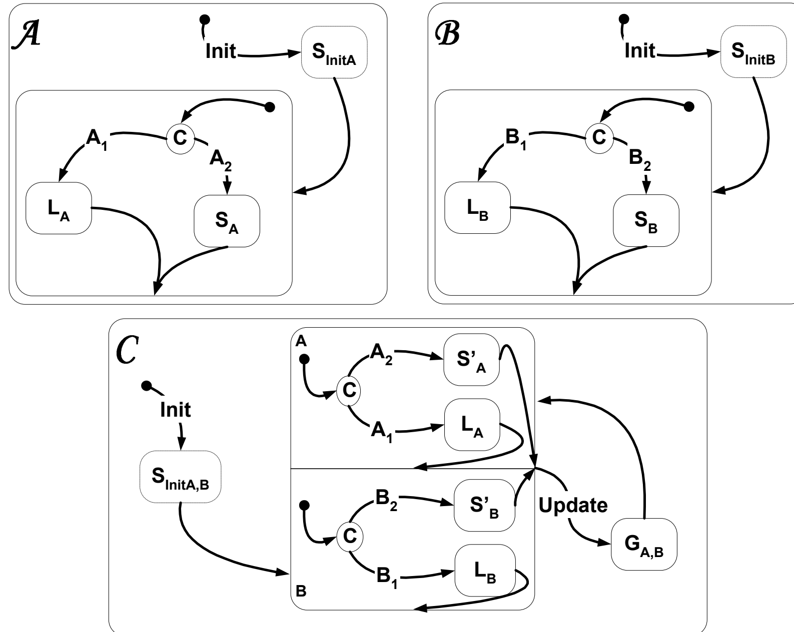


Figure 3.3: Execution Visualization of the Synchronized Composition.

3.4 Module Refinement in Synchronized Environments

Design Process. Faced with the complexity of modern day devices, the designer of such systems has to start the design process at higher levels of abstraction, which may provide him with a simpler model of the whole system. A correct partitioning and identification of the necessary modules is the next step. Crucial to a module-based design context is the possibility to separately analyze and, if necessary, improve the functionality of the modules, optimize them for a given technology, or map them to existing library elements. All these actions involve, most usually, certain transformations of the initial representations. One has to certify that the modifications imposed on the modules represent a correct transformation of the initial specification, with respect to behavior. Within the refinement calculus, the correctness of such transformations is ensured by action-level and system-level refinement rules. In the following, we exemplify how the mentioned rules apply to system design. We analyze the process, both from an interleaved, and also from a synchronized perspective.

3.4.1 Refinement Example

Let us see next how the refinement procedure is applied to the design example outlined in section 3.2.1. Considering a hardware implementation of our example, a direct mapping of the filter functionality on hardware elements (registers,

multipliers, adders, etc) is represented in Figure 3.4 a). Characteristic to this implementation of system $\mathcal{F}$ is the parallel processing and the large area occupied by the hardware elements. A functionally equivalent implementation (Figure 3.4 b)) would result out of a serial representation of the filtering device, which requires a reduced silicon area. We transform the original system $\mathcal{F}$ into $\mathcal{F}_S$, with the action system model given as:

$$
\begin{array}{ll}
 & \mathcal{F}_S(X, Z[0,..N-2], Y : T) \\
\triangleq & \textbf{begin var } temp : T \,;\, step : [0,..N] \bullet \\
 & \quad X, Z, Y := x_0, z_0, y_0 \,;\, temp := 0 \,;\, step := 0; \\
 & \textbf{do} \\
 & \quad step = 0 \\
 & \quad \rightarrow temp := 0 \,;\, step := step + 1 \\
 & \quad [\!]\ step \in [1,..N-1] \\
 & \quad \rightarrow temp := temp + h[step] \times Z[step-1] \,;\, step := step + 1 \\
 & \quad [\!]\ step = N \\
 & \quad \rightarrow Y := temp + X \times h[0] \,;\, step := 0 \\
 & \textbf{od} \\
 & \textbf{end} : h[0,..N-1]
\end{array}
$$

Is $\mathcal{F}_S$ a correct transformation of $\mathcal{F}$? Is the whole system still working according to the functional specification?

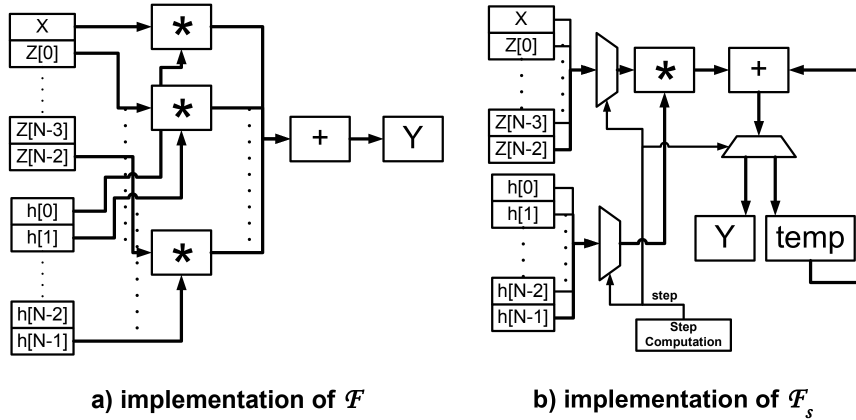


Figure 3.4: The hardware implementation of the filters $\mathcal{F}$ and $\mathcal{F}_S$.

In *isolation*, one may prove (see Appendix A-4), using Lemma 1, that the system $\mathcal{F}_S$ is a refinement of $\mathcal{F}$, $\mathcal{F} \sqsubseteq_I \mathcal{F}_S$, under the invariant I:

$$
I \triangleq (step = 1 \Rightarrow temp = 0) \wedge \bigwedge_{p=2}^{N} (step = p \Rightarrow temp = \sum_{k=1}^{p-1} h[k] \times Z[k-1])
$$

However, the separate refinement of the module $\mathcal{F}$ is not sufficient; I must be an invariant of the whole composition $\mathcal{S} \parallel \mathcal{B} \parallel \mathcal{F}_S$. From a system level point of view, we should check that $\mathcal{S} \parallel \mathcal{B} \parallel \mathcal{F} \sqsubseteq_I \mathcal{S} \parallel \mathcal{B} \parallel \mathcal{F}_S$. Unfortunately, as $\mathcal{B}$ does not *respect* I, the refinement is not possible (see [39] for details about the conditions for refinement). This fact has a simple explanation. Since the controller may choose an enabled action from either $\mathcal{B}$ or $\mathcal{F}_S$, let us suppose that it chooses only actions from $\mathcal{F}_S$, until $step = N$, after which it selects $\mathcal{B}$ for execution. Hence, following the update of Z, the invariant I is no longer valid. The solution towards a correct result comes, again, from employing communication channels as described in section 3.2.1. The invariant I has to be rewritten so as to take into account these channels, and the systems will gain some independence in this respect. Still, the same problems arise when one introduces another filtering unit ($\mathcal{M}$), in which case both the invariant and the system models must be reshaped.

3.4.2 Trace Refinement of Partitioned Action Systems

Definition 2 *Any invariant I is a* **proper**[1] **invariant** *of an action system $\mathcal{A}$, if*

$$\begin{aligned} &\forall\, v \in r\mathcal{A}, v \notin w\mathcal{A} \bullet \forall\, z, z' \bullet \\ &\quad I[w\mathcal{A} := w'\mathcal{A}, v := z] \equiv I[w\mathcal{A} := w'\mathcal{A}, v := z']) \end{aligned} \tag{3.7}$$

■

In the above definition, $w\mathcal{A}$ is the set of variables updated by the module $\mathcal{A}$; variables $r\mathcal{A}$ are read by $\mathcal{A}$. The definition says that the computed value of a proper invariant I does not depend on the variables updated by other modules. If the action system $\mathcal{A}$ is a partitioned one, then the variables $w\mathcal{A}$ should be replaced by the corresponding set of write variables (wL) of the local action, and also by the set of write variables (wS) of the global action. In addition, proving that an invariant is proper within a synchronized environment reduces to showing that the relation (3.7) holds for the global action only. This is due to the fact that variables v do not change after the execution of any local action in the mentioned environment.

Next, we propose a lemma that can be used to prove trace refinement of partitioned action systems.

Lemma 2 *Given the partitioned action systems*

$$\begin{aligned} \mathcal{A}(z) \triangleq\ & \mathbf{begin\ var}\ a \bullet a, z := a_0, z_0\ ; \\ & \qquad \mathbf{do}\ g_L^A \rightarrow L_A \parallel g_S^A \rightarrow S_A\ \mathbf{od}\ \ \mathbf{end}, \\ \mathcal{C}(z) \triangleq\ & \mathbf{begin\ var}\ c \bullet c, z := c_0, z_0\ ; \\ & \qquad \mathbf{do}\ g_L^C \rightarrow L'_A \parallel g_S^C \rightarrow S'_A \parallel g_X \rightarrow X\ \mathbf{od}\ \ \mathbf{end}, \end{aligned}$$

[1] We use the short name *proper* for any invariant that is *suitable* for a synchronized composition of action systems.

let $R(a, c, z)$ be an abstraction relation and $I(c, z)$ a proper invariant of the system $\mathcal{C}$. The system $\mathcal{A}$ is (trace) refined by the system $\mathcal{C}$, $\mathcal{A} \sqsubseteq_{R,I} \mathcal{C}$, if:

1. *Initialization:* $R(a_0, c_0, z_0) \wedge I(c_0, z_0) \equiv \mathsf{true}$
2. *Main actions:* $(g_L^A \to L_A \sqsubseteq_{R,I} g_L^C \to L'_A) \wedge (g_S^A \to S_A \sqsubseteq_{R,I} g_S^C \to S'_A)$
3. *Auxiliary action:* $\mathsf{skip} \sqsubseteq_{R,I} g_X \to X$
4. *Continuation condition:* $R \wedge I \wedge (g_L^A \vee g_S^A) \Rightarrow g_L^C \vee g_S^C \vee g_X$
5. *Partitioning property:* $R \wedge I \Rightarrow (\mathbf{do}\ g_X \to X \ [\!]\ g_L^C \to L'_A\ \mathbf{od}\ ).(\neg(g_X \vee g_L^C) \wedge g_S^C)$

Proof. Since the first four requirements of Lemma 2 are only adaptations of the original four requirements of Lemma 1, we concentrate here on showing that the fifth requirement of Lemma 2 implements the corresponding requirement of the original trace refinement lemma:

$$\begin{array}{ll} & R \wedge I \Rightarrow (\mathbf{do}\ g_X \to X \ [\!]\ g_L^C \to L'_A\ \mathbf{od}\ ).(\neg(g_X \vee g_L^C) \wedge g_S^C) \\ \Rightarrow & R \wedge I \Rightarrow (\mathbf{do}\ g_X \to X\ \mathbf{od}\ ).\mathsf{true} \end{array}$$

We consider the definition of the weakest precondition of a loop, to establish some postcondition Q, as given by Dijkstra [71]:

$$\begin{array}{rcl} \mathrm{wp}(\mathbf{do}\ g \to A\ \mathbf{od}\ , Q) & = & \exists k \geq 0 \bullet H_k, \\ H_0 & = & Q \wedge \neg g, \\ H_k & = & H_0 \vee wp(g \to A, H_{k-1}) \end{array} \quad (3.8)$$

In our context, established by Lemma 2, the new local action of the refined system ($\mathcal{C}$) is:

$$L_{new} = g_X \to X \ [\!]\ g_L^C \to L'_A$$

We need to prove that

$$(\mathbf{do}\ L_{new}\ \mathbf{od}\ ).(\neg(g_X \vee g_L^C) \wedge g_S^C) \Rightarrow (\mathbf{do}\ g_X \to X\ \mathbf{od}\ ).\mathsf{true}$$

holds.

In order to compute $(\mathbf{do}\ L_{new}\ \mathbf{od}\ ).(\neg(g_X \vee g_L^C) \wedge g_S^C)$, we apply (3.8):

$$\begin{array}{rcl} H'_0 & \equiv & \neg(g_X \vee g_L^C) \wedge g_S^C \\ H'_k & \equiv & H'_0 \vee (g_X \to X \ [\!]\ g_L^C \to L'_A).H'_{k-1} \\ & \equiv & \{ \text{ wp rule for choice } \} \\ & & H'_0 \vee ((g_X \to X).H'_{k-1} \wedge (g_L^C \to L'_A).H'_{k-1}) \\ & \Rightarrow & \{ \text{ logic } \} \\ & & H'_0 \vee (g_X \to X).H'_{k-1} \end{array} \quad (3.9)$$

We prove that $H'_k \Rightarrow H'_{k+1}$ (by induction on k). If $k = 0$, then $H'_0 \Rightarrow H'_1$ follows directly from the definition of H'_1 and logic. For $k > 0$, we have:

$$\begin{array}{ll} & H'_k \\ \equiv & \{ \text{ definition } \} \\ & (H'_0 \vee (g_X \rightarrow X).H'_{k-1}) \wedge (H'_0 \vee (g_L^C \rightarrow L'_A).H'_{k-1}) \\ \Rightarrow & \{ \text{ induction hypothesis } H'_{k-1} \Rightarrow H'_k\text{, monotonicity of wp } \} \\ & (H'_0 \vee (g_X \rightarrow X).H'_k) \wedge (H'_0 \vee (g_L^C \rightarrow L'_A).H'_k) \\ \equiv & \{ \text{ definition } \} \\ & H'_{k+1} \end{array} \tag{3.10}$$

Next, by applying (3.8), we compute $(\mathbf{do}\ g_X \rightarrow X\ \mathbf{od}\ ).\mathsf{true}$:

$$\begin{array}{lll} H_0^X & \equiv & \neg g_X \\ H_k^X & \equiv & H_0^X \vee (g_X \rightarrow X).H_{k-1}^X \\ & \equiv & \neg g_X \vee (g_X \rightarrow X).H_{k-1}^X \end{array}$$

In a similar manner as above (by induction and monotonicity of wp), we obtain that

$$H_k^X \Rightarrow H_{k+1}^X \tag{3.11}$$

In the following, we prove that $H'_k \Rightarrow H_k^X$ by induction on k. The case $k = 0$ is easy. For $k > 0$ we have

$$\begin{array}{ll} & H'_k \\ \Rightarrow & \{ \text{ (3.9) } \} \\ & H'_0 \vee (g_X \rightarrow X).H'_{k-1} \\ \Rightarrow & \{ \text{ induction hypothesis on } k = 0 \ \} \\ & H_0^X \vee (g_X \rightarrow X).H'_{k-1} \\ \Rightarrow & \{ \text{ induction hypothesis on } k = k-1\text{, monotonicity of wp} \} \\ & H_0^X \vee (g_X \rightarrow X).H_{k-1}^X \end{array} \tag{3.12}$$

Summing up the results of (3.9), ..., (3.12), we conclude that

$$(\mathbf{do}\ L_{new}\ \mathbf{od}\ ).(\neg(g_X \vee g_L^C) \wedge g_S^C) \Rightarrow (\mathbf{do}\ g_X \rightarrow X\ \mathbf{od}\ ).\mathsf{true}$$

Thus, considering that the requirements 1 to 5 of Lemma 2 are satisfied, also the requirements of Lemma 1 are satisfied, hence, $\mathcal{A} \sqsubseteq_{R,I} \mathcal{C}$. ■

Observe that the fifth requirement of Lemma 2 strengthens the original request (given in Lemma 1), by specifying that not only the auxiliary action $g_X \rightarrow X$, taken in isolation, must terminate, but that the new group of local actions, $g_X \rightarrow X \parallel g_L^C \rightarrow L'_A$ must terminate. Moreover, they must also establish the precondition for the (possibly) new global action $g_S^C \rightarrow S'_A$ to execute.

3.4.3 Modularity

Along the line established by Lemma 2, we prove the following theorem.

Theorem 3 *Consider the synchronized environment* $\mathcal{P} \stackrel{\wedge}{=} \mathcal{A}_1 \sharp \ldots \sharp \mathcal{A}_n$, *where each of the modules preserves the proper invariants* $I_1, \ldots, I_n$, *respectively. We then have that*

$$I = I_1 \wedge \ldots \wedge I_n \wedge \bigwedge_{k \in [1..n]} (sel[k] \wedge run \neq k \Rightarrow I'_k)$$

is a proper invariant of $\mathcal{P}$. *Above,* $I'_k \stackrel{\wedge}{=} I_k[wS_k := wS_kc]$. ■

The theorem states that in a synchronized environment, the global properties of the system are obtained from the individual properties of the modules, as $I \Rightarrow I_1 \wedge \ldots \wedge I_n$. The additional terms of I help us make the connection between the copies of the write variables and the respective original variables, at the moment when the action $Update$ is executed. The theorem is proved in Appendix A-2.

Corollary 1 *Consider the partitioned action systems* $\mathcal{A}_k$, *given by (3.6), and the abstraction relation* R_j. *Assume that, the system* $\mathcal{A}_j$ *preserves its respective proper invariant* I_j. *Then*

$$\frac{\mathcal{A}_j \sqsubseteq_{R_j, I_j} \mathcal{A}'_j}{\mathcal{A}_1 \sharp \ldots \sharp \mathcal{A}_j \sharp \ldots \sharp \mathcal{A}_n \sqsubseteq_{R_j, I_j} \mathcal{A}_1 \sharp \ldots \sharp \mathcal{A}'_j \sharp \ldots \sharp \mathcal{A}_n}, \forall j \in [1 \ldots n]$$

■

The statement of the corollary follows from Theorem 3 and Lemma 2 (see Appendix A-3).

The interpretation of Corollary 1 is that each component of a synchronized parallel composition may be refined in isolation, without knowledge about the invariants of the other components. The system designer may then employ the modules without knowing their respective internal details of functionality. The module designer is responsible for improving the performance of the modules, in total transparency for the integrator designer. This is a consequence of the fact that the systems exchange information at the end of an execution cycle, rather than after each execution round. Observe that, if I_j is a *new* invariant for $\mathcal{A}'_j$, it will just be a new entry in the definition of I, as specified by Theorem 3.

A similar conclusion as ours is reached by Back and von Wright for the parallel composition of action systems [39]. However, this is achieved while requiring that the invariants of all modules are known, and a noninterference relation between them proves to hold. The corresponding noninterference condition corresponds to our requirement that the invariant I_j is proper. Still, checking the properness of an invariant concerns the respective module designer only, who does not have to get information about the other invariants. Therefore, we have increased the independence of the module designer. Nevertheless, the mentioned benefit comes at the expense of needing more constrained action system invariants.

3.4.4 Refinement Example Revisited

Consider the analysis presented in section 3.4.1. If we check $\mathcal{F} \sqsubseteq_I \mathcal{F}_S$ in the context of Lemma 2, meaning that we adopt a synchronized perspective on the composition, we will immediately obtain that $\mathcal{S} \sharp \mathcal{B} \sharp \mathcal{F} \sqsubseteq_I \mathcal{S} \sharp \mathcal{B} \sharp \mathcal{F}_S$ (notice that $\mathcal{F}_S$ is a partitioned action system). Besides this, a previous addition of module $\mathcal{M}$ would not change the refinement, and we could have $\mathcal{S} \sharp \mathcal{B} \sharp \mathcal{F} \sharp \mathcal{M} \sqsubseteq_I \mathcal{S} \sharp \mathcal{B} \sharp \mathcal{F}_S \sharp \mathcal{M}$.

3.5 Summary and Related Work

The research presented in this chapter was motivated by an analysis of control aspects and of modular design techniques, as supported by the current action systems formal framework. We exemplified that the interleaved model of concurrency may not suffice, as such, for modeling parallel reactive systems. Additional variables are needed in order to implement the system functionality, correctly. Our solution comes as a synchronization mechanism, implying a new virtual execution model of action systems, applicable to both discrete and hybrid designs. This last claim is supported in chapter 4, where we show how to apply the same synchronization mechanism to models of hybrid systems represented as continuous action systems. We eliminate intermediate results that could affect the global state, as the system gives complete answers to the stimuli provided by the environment. Most importantly, we prove that the modularity capabilities of synchronized action systems are better than those of the parallel composition of such systems.

Related Work. The approximation of concurrency by interleaving is used in most process algebras like CSP [97], CCS [127], as well as in input-output automata [121] and UNITY [62]. The nondeterministic behavior induced by the interleaved model requires solutions for controlling the data flow. However, resolving control issues reduces the design independence across the different levels of the design process. Several recent studies have analyzed aspects of control and / or composability within different formal frameworks, all of which deal with a certain interleaved environment.

Cavalcanti and Woodcock [55], and Charpentier [63] build new reasoning environments in order to address issues related to correctness and composability of (reactive) systems. Both approaches have strong roots in the weakest precondition semantics of Dijkstra [71]. These aspects are already included in our framework, and we have shown how to use them in order to achieve our goals. The main idea is to isolate the local updates performed by an action system from the global ones, which appear to be updates carried out at the same time with the corresponding global variables of the other systems. This is consistent with the views presented by Gupta et al. [81], in the framework of *concurrent constraint programming*.

The product operator of Milner's *Synchronous Calculus of Communicating Systems* (SCCS) [126] offers a somewhat similar approach to synchronization.

However, while we synchronize on the updates of a group of variables, the SCCS approach is based on simultaneous execution of actions, which we only reach in the last execution round of a synchronized composition. Moreover, synchronization *restrictions* must be analyzed for each particular synchronized composition, thus decreasing the possibility of reuse.

Hoare and He use the $sync$ operator for modeling *bulk synchronization*, as a means of controlling concurrent processes [98]. The operator is semantically close to our $sharp$ operator. However, the composition is not exploited towards achieving higher degrees of modularity, it just serves the purpose of improving concurrent behavioral control.

Bellegarde et al. introduce a similar idea of synchronized parallel composition for event-B systems [42]. In contrast to our model, which increases the *external* determinacy, while preserving the *internal* nondeterminism, the event-B solution preserves also the external nondeterminism. Moreover, a *gluing invariant* is necessary when synchronized modules are refined. This requirement comes from the fact that the synchronization is performed only with regard to selected events, collected in a synchronization specification. Therefore, the supplier of modules should also deliver to the system integrator, besides the modules themselves, the synchronization specification. From this point of view, the approach is similar to the one adopted by Back and von Wright [39], where information about the invariants of all modules must be known in order to perform refinements. A more relaxed approach to this problem is given by Butler [53]. The author combines features of the state-based action systems and of the algebraic CSP, in search for modularity.

In the temporal logic of actions of Lamport [115, 116], synchronization is specified as a way of applying *noninterleaving* to system design. This is reached by employing *joint actions*. The author's conclusion supports our point of view: interleaving "blurs" the distinction between the components used in design.

Treharne and Schneider [153] employ CSP processes to control B-machines. The basic problems are raised by the interleaved execution semantics of both formalisms. Playing the state-based formalism (B), against the event-based approach (CSP), one may get a controllable environment for modeling certain applications. Our study shows, on the other hand, that it is possible, within the same state-based framework, to obtain the desired control of behaviors.

An execution mechanism quite close to our synchronized environment is described by the semantics of STATECHARTS [83]. We can identify the execution of local actions that come from a single component, in an execution cycle, as a *compound transition* – CT. There are as many such *nonconflicting* CTs, as modules in a synchronized composition. By adding the initial and final state corresponding to a given execution cycle, we obtain a full CT.

In a similar study targeting modular development of hierarchic reactive systems, Alur and Grosu build their approach based on refinement checking by *assume-guarantee* rules [10]. Compared to our work, the authors benefit from unrestricted compositionality of interleaving composition with respect to refinement.

This property characterizes the language of *reactive modules* [13], which is used to describe the respective modules employed in design. However, the modules have disjoint sets of variables.

In VHDL [19], the update mechanisms for variables and signals are relatively similar to our solution concerning local and global variables. The difference resides in the fact that already executed processes (assimilated to action systems) may be rescheduled for execution, within the same VHDL execution cycle. This is possible due to new values of watched signals, assigned by other processes. The validity of such an approach is supported by the fact that, targeting a hardware implementation, the VHDL designer may assume that eventually, such reaction-triggering events will cease to appear (the combinational logic outputs will eventually settle to some value).

One important remark is that our approach does not necessarily address *synchronous* designs. The existence of a common clock signal is not suggested by any of our constructs. It is true that synchronous designs can be easily obtained from our models. This is furthermore supported by the underlying "synchrony hypothesis", as the time to perform individual actions is assumed to be null. From this perspective, we are close to the synchronous group of languages (Esterel [45], Lustre, Signal, etc.).

By providing the new virtual execution environment, we have tackled two important problems of system design: behavior control and modularity. The essential result of the study is mentioned by Corollary 1. Based on this, we can say that the system level integrator and the module designers gain an increased independency with respect to each other, during the design process. Tools like *Refinement Calculator* [54] could be used for proving the necessary trace refinements in a synchronized context.

We believe that our achievement of using barrier synchronization to increase the modular design capabilities of the action systems framework is a contribution that could be easily adapted to other similar formal environments. The trade off: the action systems involved in design have to respect more constrained invariants.

Chapter 4

Modeling Hybrid Systems

In the previous chapter, we have focused on techniques suited for the rigorous construction of discrete concurrent reactive systems. It is now time to embark on the study of systems that also contain a continuous component. This and the next two chapters will ultimately give a unified view of the discrete and continuous system design.

Describing hybrid systems by continuous action systems, as introduced by Back, Petre and Porres [27], bears the advantage of using the same proof theory as for discrete action systems. This is justified by the fact that the implementation of a CAS is an action system having time as an explicit variable.

Freezing time unless the system execution has finished is an anomaly of a hybrid system model, being known as *timelocking*. Due to the way in which time is advanced in the action system implementation of CAS, such an undesired phenomenon might appear. The original definition of CAS does not contain a clear mechanism that would prevent timelocks from occurring. Therefore, in this chapter we extend the syntax of CAS, by incorporating the *execute only once* concept, for its actions. In this way, we move something that would usually have to be explicitly expressed in a CAS, to an implementation issue in the corresponding action system representation. As a result, the timelock-free behavior is enforced while the CAS model remains simple.

Next, we also adapt the formalization of the *barrier synchronization mechanism* that we have developed in chapter 3 for action systems, to CAS models.

We illustrate our modeling techniques on two simple control systems: a heating-cooling system and a two-tank system.

4.1 Timelocking and Zenoness

When executing a hybrid system, an evolution is followed by a discrete transition or a sequence of discrete transitions. If the hybrid system is modeled as a CAS, the transitions might trigger a change of the future evolution of some continuous-

valued or discrete-valued time variables. Moreover, in traditional CAS, a discrete transition does not take time. Therefore, its execution is represented on the time coordinate as a point.

Hybrid systems can suffer from two undesired behavioral anomalies: *timelocking* [49], and *zenoness* [1]. In principle, we would like to build models that do not expose the mentioned anomalies, since they might determine unexpected behavior of the model, or might compromise simulation.

Traditionally, a timelock occurs when the infinitely repeated execution of a discrete transition prevents the execution of a time-advancing statement [49]. Therefore, in such cases, the time in the model stops, and some discrete transition remains enabled forever. In a CAS, the timelock interpretation is similar to the traditional one, and this will become apparent in section 4.2.

A Zeno behavior appears in situations when, although time keeps progressing during the execution of the model, it is prevented from growing unboundedly.

Since hybrid models that suffer from zenoness can not be realized physically, we will next give a definition of a nonzeno action system, which lets us rule out zeno candidates.

Definition 3 *Assume a CAS denoted by* Sys *and its semantic translation* $\overline{Sys}$*. We say that* $\overline{Sys}$ *is* **nonzeno** *if the following condition holds:*

$$\exists \varepsilon \in \mathsf{Real}_+ \cdot \forall now, now' \in \mathsf{Real}_+, now' = min\{t' \geq now \mid gg_{Sys}.t'\} \cdot \\ now' - now > \varepsilon$$

■

As also argued by Alur and Henzinger [12], one needs to make sure that an execution can be extended to an execution of arbitrary accumulated duration.

4.2 Adorned Continuous Action Systems

Problem description. As presented in chapter 2, any CAS model is explained in terms of a corresponding action system with explicit time. In the latter, time is advanced by evaluating the disjunction of constituent action guards. The current value of time is measured by variable now. The minimum time point that is at least equal to the actual now represents the next transition time. This means that some action (in particular, an action described by a differential equation) could be executed more than once at the same moment of time, which in turn could prevent the execution of any other action, if it never becomes disabled. This scenario implies that time is being locked at now. Thus, the system does not evolve, yet it does not terminate. Note that discrete updates do not suffer from this anomaly.

To illustrate this sort of behavior, we consider the following heating-cooling system modeled as a CAS. Below, we give its action system translation.

$$
\begin{array}{ll}
& \overline{\mathcal{H}C}(\theta : \mathsf{Real}_+ \rightarrow \mathsf{Real}) \\
\triangleq & \mathbf{begin}\ \ \mathbf{var}\ now : \mathsf{Real}_+ \bullet \\
& \quad now := 0\ ;\ \theta :-\ (\lambda t \cdot 4 * t)\ ;\ UT; \\
& \ \mathbf{do} \\
& \qquad \theta.now = 10 \\
& \qquad \rightarrow \theta :-\ (\lambda t \cdot \theta.now - 2 * (t - now))\ ;\ UT \\
& \ \ [\!]\quad \theta.now = 5 \\
& \qquad \rightarrow \theta :-\ (\lambda t \cdot \theta.now + 3 * (t - now))\ ;\ UT \\
& \ \mathbf{od} \\
& \mathbf{end}
\end{array}
\tag{4.1}
$$

In the above model, the first action's guard holds even after the execution of the corresponding action body, which is supposed to decrease the temperature θ whenever it reaches the maximum value of 10. Since the temperature starts decreasing from $\theta.now = 10$, the next minimum now is the same as its immediate predecessor (at now, $\theta.t = \theta.now = 10$). Therefore, looping in the same state at the same moment of time goes on forever. Consequently, the temperature never gets the chance to actually decrease, and for that matter to increase later. The phenomenon triggers nontermination of the execution of the first action. Obviously, simulating models such as (4.1) is not possible.

Solution. From the above example, one can learn that CAS models should have a way of preventing the execution of any action more than once if now has not changed. In this spirit, we provide a means for modeling the *execute only once* (or *single invocation at a new time point*) concept. The mechanism reduces to extending the syntax of CAS, by decorating each transition with 1. This single response problem translates into an implementation issue, within the corresponding action system with explicit time. As a result, the CAS model stays simple.

Even if this solution targets mainly time consuming actions, we extend it to the entire system model, for simplicity and consistency. If an action performs only a discrete computation, the single invocation at a new now mechanism does not compromise functional correctness. It even makes sense, as discrete actions take no time.

Syntactically, a CAS with adorned transitions is as follows:

$$
\begin{array}{lll}
\mathcal{C}(y : \mathsf{Real}_+ \rightarrow T_y) & \triangleq & \mathbf{begin}\ \mathbf{var}\ x : \mathsf{Real}_+ \rightarrow T_x \bullet Init; \\
& & \quad \mathbf{do}\ g_1 \xrightarrow{1} S_1\ [\!]\ \ldots\ [\!]\ g_n \xrightarrow{1} S_n\ \mathbf{od} \\
& & \mathbf{end}
\end{array}
\tag{4.2}
$$

4.3 Implementing Continuous Action Systems

Adorning CAS transitions with single response indicators requires that there is an actual mechanism that implements this concept. We present here two distinct CAS implementations and underline the appropriate situations where each could be employed, respectively.

1. Using a single variable $state$**.** As a first solution, which in fact has been already used [27], one can add a variable $state$ that stores the current state of the system. By keeping track of the system states before and after a discrete transition, through variable $state$ that is set accordingly, we get self-disabling actions in effect.

Let us see how this applies to the heating-cooling action system (4.1).

$$
\begin{array}{ll}
& \overline{\mathcal{HC}}(\theta : \mathsf{Real}_+ \to \mathsf{Real}) \\
\triangleq & \mathbf{begin}\ \ \mathbf{var}\ now : \mathsf{Real}_+, state : \mathsf{Real}_+ \to \{0,1\} \bullet \\
& \quad now := 0; \\
& \quad state :- (\lambda t \cdot 0); \\
& \quad \theta :- (\lambda t \cdot 4 * t)\ ;UT; \\
& \ \mathbf{do} \\
& \qquad state.now = 0 \wedge \theta.now = 10 \\
& \qquad \to state :- (\lambda t \cdot 1); \\
& \qquad\quad \theta :- (\lambda t \cdot \theta.now - 2 * (t - now))\ ; UT \\
& \ [\!] \quad state.now = 1 \wedge \theta.now = 5 \\
& \qquad \to state :- (\lambda t \cdot 0); \\
& \qquad\quad \theta :- (\lambda t \cdot \theta.now + 3 * (t - now))\ ; UT \\
& \ \mathbf{od} \\
& \mathbf{end}
\end{array}
\tag{4.3}
$$

This solution for preventing timelocks works well for systems with a small number of states and with an obvious discrete transition route.

The same disabling mechanism might be difficult to apply to systems with a large number of states, or to nondeterministic systems. In such cases it is not easy, if possible at all, to determine the next state of the system, based on current information.

Alternatively, one could use actions like ($state.now \neq 1 \wedge \ldots \to state :- (\lambda t \cdot 1)\ ; \ldots$), in order to deal with nondeterminism and more complex state machines. Even so, if two (or more) actions of such an action system do not change now, this second solution does not prevent waffling back and forth between those actions, forever, yet at the same now. Hence, the timelocking problem moves from the action level to the group-of-actions level.

Next, we propose a general modeling solution that overcomes the mentioned inconveniences.

2. Using variables $u_1, \ldots, u_n$**.** We translate a decorated CAS described by (4.2)

into an action system with time, as follows.

$$
\begin{array}{ll}
 & \overline{\mathcal{C}}(y : \mathsf{Real}_{+} \rightarrow T_y) \\
\triangleq & \mathbf{begin\ var}\ now, now_c : \mathsf{Real}_{+}, x : \mathsf{Real}_{+} \rightarrow T_x, \\
 & \qquad\qquad u_1, \ldots, u_n : \mathsf{Real}_{+} \rightarrow \mathsf{Bool}\ \bullet \\
 & \quad now := 0 \,;\, Init \,;\, u_1, \ldots, u_n : -(\lambda t \cdot \mathsf{false}) \,;\, UT \,;\, now_c := now; \\
 & \quad \mathbf{do} \\
 & \qquad\quad \neg u_1.now \wedge g_1.now \\
 & \qquad\quad \rightarrow u_1 :- (\lambda t \cdot \mathsf{true}) \,;\, S_1 \,;\, UT \,;\, Check \\
 & \qquad [\!] \quad \ldots \\
 & \qquad [\!] \quad \neg u_n.now \wedge g_n.now \\
 & \qquad\quad \rightarrow u_n :- (\lambda t \cdot \mathsf{true}) \,;\, S_n \,;\, UT \,;\, Check \\
 & \quad \mathbf{od} \\
 & \mathbf{end}
\end{array}
\tag{4.4}
$$

where

$$
\begin{array}{ll}
 & Check \\
\triangleq & \mathbf{if}\ now \neq now_c\ \mathbf{then}\ u_1, \ldots, u_n :- (\lambda t \cdot \mathsf{false}) \,;\, now_c := now\ \mathbf{else}\ \mathsf{skip}\ \mathbf{fi}
\end{array}
$$

Describing the state of the system by local variables $u_1, \ldots, u_n$ forces each action to execute just once at the same now. After one execution, any action becomes disabled until time is advanced. Observe that, by employing this solution, we have eliminated the need of computing next states based on current ones.

In order to avoid the premature termination of system execution, statement $Check$ resets the variables $u_1, \ldots, u_n$, provided that $now \neq now_c$. Here, now_c is the copy of the current time. Consequently, a new execution cycle is enabled. Otherwise, if $now = now_c$, the status-quo is maintained by executing skip. Hence, the system continues its execution only if time progresses, otherwise it terminates.

The u - variables CAS implementation (4.4) could be applied to a hybrid control system with nondeterministic state transitions, like the one that we are going to outline in the next section.

4.3.1 Example: A Two-Tank System

The example system of Figure 4.1 has also been studied by Slupphaug et al. [148], however our version is simplified. It consists of two tanks, that is, the buffer and the supply, a pump with three modes (off, low speed, high speed) used to pump water from the buffer tank into the supply, and two on/off valves, positioned at the inlet of the buffer, and at the outlet of the supply, respectively. The plant receives liquid from an input stream and has to deliver the liquid to an output target.

The safety requirement is equated to keeping the level of the liquid in both tanks between 1 m and 9 m, hence preventing emptying, as well as overflowing of the tanks. In chapter 5 we introduce a simulation tool for CAS. With this tool

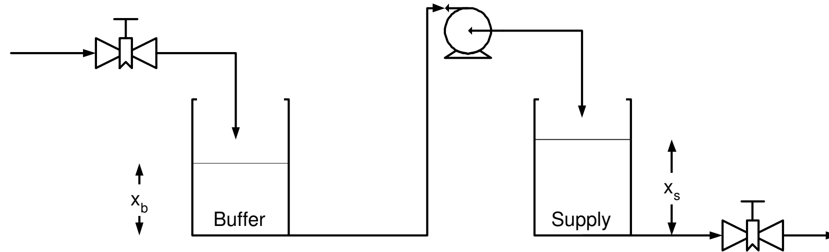


Figure 4.1: Two-tank system.

at hand, we would like to simulate various execution scenarios of the two-tank system. This could help in figuring out control strategies that would, for example, prevent the system from swinging periodically between the minimum and the maximum level. Such a degree of freedom during simulation means that it is desirable to model the system as nondeterministically as possible, in the beginning. In this way, the designer will be able to explore different possible trajectories. The user will have to resolve the nondeterministic choices whenever it is the case, automatically or interactively. The gain out of simulating such a nondeterministic model is an increased intuition for further possible improvements, tailored to one's practical needs.

System modeling. Depending on the liquid levels in the buffer and the supply, the controller should set the valves and use the pump appropriately. Its duty is to regulate the levels in both tanks without violating the safety requirement mentioned above ($1 \leq level_{buf} \leq 9$), ($1 \leq level_{sup} \leq 9$).

Below, we enumerate the model variables:

- x_b, x_s : $\mathsf{Real}_+ \rightarrow \mathsf{Real}$ - denote the liquid levels in the buffer and the supply, respectively;
- u_{iv}, u_{ov} : $\mathsf{Real}_+ \rightarrow \{0, 1\}$ - denote the positions of the inlet and the outlet valves, respectively (0 - closed, 1 - opened);
- u_p : $\mathsf{Real}_+ \rightarrow \{0, 1, 2\}$ - models the position of the pump (0 - off, 1 - low speed, 2 - high speed).

The inflow rate (v_i), outflow rate (v_o), and pump capacity factor (α) are the parameters of the system. For the current analysis we assume that $v_i = v_o = \alpha = 1\ m^3/min$. Nevertheless, we do not substitute the parameter values in the continuous evolutions of variables x_b, x_s of the model. We rather use the parameters as such, for one to be able to run the simulation under different parameter instantiations.

The differential equations that describe the continuous behavior of x_b and x_s are the following:

$$\begin{array}{rcl} \dot{x}_b & = & 1/3.5 * (v_i * u_{iv} - \alpha * u_p) \\ \dot{x}_s & = & 1/2 * (\alpha * u_p - v_o * u_{ov}) \end{array}$$

In the equations above, $3.5\ (m^2)$ is the buffer (bottom) area, and $2\ (m^2)$ is the supply area. Observe that levels x vary inversely proportional to these areas.

Rather than presenting the whole CAS model of the two-tank system, we will enumerate and describe formally only some of its possible actions. Since we are simulating the model in the next chapter, we will give the translated version of the entire CAS there. Note that we do not make a distinction between the controller and the plant in our model; we are in fact modeling a closed hybrid system.

• For example, the system may be in a state where the liquid in the buffer tank has reached the maximum of 9 m. At the same time, suppose that the level of the liquid in the supply is $1 \leq x_s < 9$. Provided that the inlet valve is closed ($u_{iv} = 0$) – to avoid the overflow of the buffer tank – one could use the pump on either low ($u_p = 1$) or high speed ($u_p = 2$). The outlet valve should be opened ($u_{ov} = 1$) for the liquid to be delivered continuously. The timed representations of the two actions that correspond to this situation are given below.

$$\begin{array}{l} x_b.now = 9 \wedge 1 \leq x_s.now < 9 \\ \xrightarrow{1} c :- (\lambda t \cdot t - now)\ ;\ u_{iv} :- (\lambda t \cdot 0); \\ \quad u_p :- (\lambda t \cdot 1)\ ;\ u_{ov} :- (\lambda t \cdot 1); \\ \quad \dot{x}_b :- 1/3.5 * (v_i * u_{iv} - \alpha * u_p) \\ \quad \dot{x}_s :- 1/2 * (\alpha * u_p - v_o * u_{ov}) \end{array}$$

$$\begin{array}{l} x_b.now = 9 \wedge 1 \leq x_s.now < 9 \\ \xrightarrow{1} c :- (\lambda t \cdot t - now)\ ;\ u_{iv} :- (\lambda t \cdot 0); \\ \quad u_p :- (\lambda t \cdot 2)\ ;\ u_{ov} :- (\lambda t \cdot 1); \\ \quad \dot{x}_b :- 1/3.5 * (v_i * u_{iv} - \alpha * u_p) \\ \quad \dot{x}_s :- 1/2 * (\alpha * u_p - v_o * u_{ov}) \end{array}$$

Observe that the actions contain the linear differential equations that characterize the behavior of x_b, x_s, rather than their respective analytic solutions.

• Even in situations when $x_b = 1$, if $1 < x_s.now \leq 9$, the pump could be working at low speed, provided that the inlet valve is opened. This is possible only under the assumption of $v_i = v_o = \alpha$. The action below models such a behavior:

$$\begin{array}{l} x_b.now = 1 \wedge 1 < x_s.now \leq 9 \\ \xrightarrow{1} c :- (\lambda t \cdot t - now)\ ;\ u_{iv} :- (\lambda t \cdot 1); \\ \quad u_p :- (\lambda t \cdot 1)\ ;\ u_{ov} :- (\lambda t \cdot 1); \\ \quad \dot{x}_b :- 1/3.5 * (v_i * u_{iv} - \alpha * u_p) \\ \quad \dot{x}_s :- 1/2 * (\alpha * u_p - v_o * u_{ov}) \end{array}$$

• If $u_{iv} = u_p = u_{ov} = 1$, the controller can decide to close the pump or use it

at high speed, depending on whether the liquid level in each tank is above or below half of maximum, respectively:

$$
\begin{aligned}
&1 < x_b.now \le 4.5 \wedge 4.5 \le x_s.now < 9 \wedge \\
&u_{iv}.now = 1 \wedge u_p.now = 1 \wedge u_{ov}.now = 1 \\
&\xrightarrow{1} c :- (\lambda t \cdot t - now) \,;\, u_p :- (\lambda t \cdot 0); \\
&\quad \dot{x}_b :- 1/3.5 * (v_i * u_{iv} - \alpha * u_p) \\
&\quad \dot{x}_s :- 1/2 * (\alpha * u_p - v_o * u_{ov})
\end{aligned}
$$

$$
\begin{aligned}
&4.5 < x_b.now < 9 \wedge 1 < x_s.now < 4.5 \wedge \\
&u_{iv}.now = 1 \wedge u_p.now = 1 \wedge u_{ov}.now = 1 \\
&\xrightarrow{1} c :- (\lambda t \cdot t - now) \,;\, u_p :- (\lambda t \cdot 2); \\
&\quad \dot{x}_b :- 1/3.5 * (v_i * u_{iv} - \alpha * u_p) \\
&\quad \dot{x}_s :- 1/2 * (\alpha * u_p - v_o * u_{ov})
\end{aligned}
$$

4.4 Synchronized Hybrid Models

As in discrete reactive systems, concurrency, in all its flavors, plays an important role in hybrid designs. The parallel composition of CAS, as described in chapter 2 uses interleaving as the underlying execution mechanism. Thus, composing certain hybrid modules might imply a similar effort for controlling their behavior, as in the discrete case. From a modular design perspective, there are classes of hybrid systems for which the u implementation of CAS (4.4) helps in reducing this effort.

4.4.1 Parallel Composition of Hybrid Models with Discontinuities

In the following, we analyze situations that may appear in the behavior of CAS, when employing a traditional parallel operator between two modules. As we will see, certain malfunctions may be exposed due to the interleaved model of execution, combined with particular timed evolutions.

In practice, there are situations where hybrid systems manifest the so called "discontinuities" with infinite bandwidth, or "sudden changes" [107]. They may be found for instance, when analyzing relays switching and mechanical components engaging/disengaging [149], or in processes that observe hysteresis-like behaviors [158].

The example that we illustrate in the following paragraphs is deliberately small and simple. This makes it easier to discuss implementation issues, without being affected by the actual complexity of a practical example. Therefore, we only give a hypothetical behavioral description of a hybrid system, willing to analyze our action system models in the presence of discontinuities.

Let us consider next the abstract model of a hybrid control system, which evolves according to the function plotted in Figure 4.2, above the time axis. The output Y is either increased or decreased, at different speeds (v_1, v_2), respectively.

The system under analysis behaves according to the following CAS:

$$
\begin{array}{ll}
 & \mathcal{S}_1(Y : \mathsf{Real}_+ \rightarrow \mathsf{Real}_+) \\
\triangleq & \textbf{begin var } dir : \mathsf{Real}_+ \rightarrow \mathsf{Bool} \bullet Y :- (\lambda t \cdot 0) \,;\, dir :- (\lambda t \cdot \mathsf{true}); \\
 & \quad \textbf{do} \\
 & \qquad\quad Y.now = 0 \qquad\qquad\qquad\qquad\qquad\qquad /* \text{ action } A_1^1 */ \\
 & \qquad\quad \xrightarrow{1} Y :- (\lambda t \cdot v_1 * (t - now)); \\
 & \qquad\qquad dir :- (\lambda t \cdot \mathsf{true}) \\
 & \qquad [\!] \;\; (Y.now = Y_0 \wedge dir.now) \vee Y.now = Y_H \quad /* \text{ action } A_1^2 */ \\
 & \qquad\quad \xrightarrow{1} Y :- (\lambda t \cdot Y_H - v_2 * (t - now)); \\
 & \qquad\qquad dir :- (\lambda t \cdot \mathsf{false}) \\
 & \qquad [\!] \;\; Y.now = Y_1 \qquad\qquad\qquad\qquad\qquad\qquad /* \text{ action } A_1^3 */ \\
 & \qquad\quad \xrightarrow{1} \; (Y :- (\lambda t \cdot Y_1 + v_1 * (t - now)) \\
 & \qquad\qquad [\!] \; Y :- (\lambda t \cdot Y_1 - v_2 * (t - now))) \\
 & \quad \textbf{od} \\
 & \textbf{end}
\end{array}
$$

Figure 4.2: The timed evolutions of systems $\mathcal{S}_1$ and $\mathcal{S}_2$.

Notice, in the graphical representation of $\mathcal{S}_1$, that the output Y evolves between 0 and a maximal value Y_H. A second characteristic is the “jump” and the change of direction at point $Y.now = Y_0$; this happens in every instance of the execution. A

descending trend is taken every time the system reaches $Y.now = Y_H$. However, another direction change may happen at point $Y.now = Y_1$, where the system may choose to switch again to an ascending trend towards Y_H, or to continue descending towards 0. The number of direction changes at this point is arbitrary. Observe that the value of local variable dir helps discriminating between time moments represented by t_1 or t_2.

As observers, we are interested in counting how many times the output variable Y reaches certain values. Concretely, in our analysis, we identify one such value with Y_0. Consequently, we attach to $\mathcal{S}_1$ a system that counts the events when $Y = Y_0$. This system may be described as

$$
\begin{array}{ll}
 & \mathcal{S}_2(Y : \mathsf{Real}_+ \to \mathsf{Real}_+, counter : \mathsf{Real}_+ \to \mathsf{Nat}) \\
\triangleq & \textbf{begin}\ counter :- (\lambda t \cdot 0)\ ; Y :- (\lambda t \cdot 0); \\
 & \quad \textbf{do} \\
 & \qquad\quad Y.now = Y_0 \qquad\qquad /*\ \text{action}\ A_2^1 */ \\
 & \qquad\quad \xrightarrow{1} counter :- (\lambda t \cdot counter.now + 1) \\
 & \quad \textbf{od} \\
 & \textbf{end}
\end{array}
$$

Even if the composed system, hybrid system plus counter, is simple enough to be designed as a compound, we choose to design it modularly. This is justified by the fact that we want to create the premises for further extensions; secondly, the counter system $\mathcal{S}_2$ is not regarded as a required functional unit, under normal circumstances.

Interleaved model. At first, we follow the traditional parallel execution model approximated by interleaving, of $S = \mathcal{S}_1 \,||\, \mathcal{S}_2$. At some moment in time (t_1, t_3, t_5 in Figure 4.2), when $Y.now = Y_0$, actions A_1^2 and A_2^1 are simultaneously enabled. If the controller chooses to execute the action A_1^2, first, the variable Y will be updated to Y_H. This disables A_2^1. Thus, the counter misses to record this trajectory change. At time stamps t_2, t_4, when $Y = Y_0$ also holds, the interleaved model allows a correct update of the variable $counter$, as the action of the counter is the only enabled one.

This situation can be solved by adding extra information to the modules, regarding their communication, or by employing other operators on CAS, which could determine the modules to react in a way that produces a correct output. However, either of the solutions implies extra modeling effort. Such a solution also deteriorates the system's modularity, since it requires a thorough analysis of both of the composing action systems, as also illustrated in chapter 3.

Let us see how the synchronized perspective on the composition of CAS modules solves the issues exposed above.

4.4.2 Synchronized Continuous Action Systems

In this section, we give the definition of the synchronized composition of CAS, which is similar in spirit to the one defined for the discrete case, in chapter 3. Nevertheless, the timed composition bears some particular features.

Firstly, we introduce *partitioned continuous action systems*, by the following definition.

Definition 4 *Consider a CAS of the form:*

$$\begin{array}{rcl} \mathcal{A}(z : Real_+ \to T_z) & \triangleq & \mathbf{begin\ \ var}\ x : Real_+ \to T_x \bullet Init\,; \\ & & \quad \mathbf{do}\ \ g_L \xrightarrow{1} L \parallel g_S \xrightarrow{1} S\ \ \mathbf{od} \\ & & \mathbf{end} \end{array} \tag{4.5}$$

We say that $\mathcal{A}$ is a **partitioned** *CAS if:*

1. $gw\mathcal{A} \subseteq wS$ - meaning that S is the global action of $\mathcal{A}$. Notice that wS may also contain local variables of $\mathcal{A}$.

2. $wL \subseteq lw\mathcal{A}$ - meaning that L is the local action of $\mathcal{A}$.

3. $(\mathbf{do}\ g_L \xrightarrow{1} L\ \mathbf{od}\ ).g_S \equiv \mathsf{true}$ - meaning that the execution of L establishes the precondition for executing S. ■

Observe that in (4.5), we have separated the local actions from the global actions in the same way as for partitioned action systems. Moreover, we impose requirements similar to those stated by Definition 1.

In Definition 4, the notation $\mathbf{do}\ g_L \xrightarrow{1} L\ \mathbf{od}$ stands for

$$\mathbf{do}\ \neg u_L.now \wedge g_L.now \to u_L : -(\lambda t \cdot \mathsf{true})\,; L\,; UT\ \mathbf{od}$$

where $UT \triangleq now := min\{t' \geq now \mid gg.t'\}$. The fact that the local action is executed only once at some moment now eliminates the need to require that the corresponding loop terminates. Hence, it is sufficient that $\mathbf{do}\ g_L \xrightarrow{1} L\ \mathbf{od}$ enables the global action S after it executes.

We emphasize the fact that one needs to employ as many u_L-like variables as there are choices in the local action L. This ensures that the execute-only-once concept is implemented for all (possible) sub-actions of L. The procedure does not apply to the global action S, in which case a single u-variable suffices.

The meaning of a partitioned CAS is given by its translation into the following

action system:

$$
\begin{array}{ll}
& \overline{\mathcal{A}}(z : Real_+ \to T_z) \\
\triangleq & \textbf{begin var } now, now_c : Real_+, x : Real_+ \to T_x, \\
& \quad u_L, u_S : Real_+ \to \mathsf{Bool} \bullet \\
& \quad now := 0 \,;\, Init \,;\, u_L, u_S : -(\lambda t \cdot \mathsf{false}) \,;\, UT \,;\, now_c := now; \\
& \textbf{do} \\
& \qquad \neg u_L.now \wedge g_L.now \\
& \qquad \to u_L : -(\lambda t \cdot \mathsf{true}) \,;\, L \,;\, UT \\
& \quad [\!] \ \neg u_S.now \wedge g_S.now \\
& \qquad \to u_S : -(\lambda t \cdot \mathsf{true}) \,;\, S \,;\, UT \\
& \quad [\!] \ now \neq now_c \\
& \qquad \to u_L, u_S : -(\lambda t \cdot \mathsf{false}) \,;\, now_c := now \\
& \textbf{od} \\
& \textbf{end}
\end{array}
\tag{4.6}
$$

Synchronized CAS. Let us consider n partitioned CAS of the form given by (4.5). Their **synchronized parallel composition** is a new system, $\mathcal{P} = \mathcal{A}_1 \sharp \ldots \sharp \mathcal{A}_n$. Its definition is given in terms of the action system $\overline{\mathcal{P}}$:

$$
\begin{array}{ll}
& \overline{\mathcal{P}}(z : \mathsf{Real}_+ \to T_z) \\
\triangleq & \textbf{begin var } x : \mathsf{Real}_+ \to T_x, sel[1..n], \\
& \quad u_L^1, \ldots, u_S^n : \mathsf{Real}_+ \to \mathsf{Bool}, \\
& \quad run : \mathsf{Real}_+ \to \mathsf{Nat}, now, now_c : \mathsf{Real}_+ \bullet \\
& \quad now := 0 \,;\, Init \,;\, UT \,;\, now_c := now; \\
& \textbf{do} \\
& \qquad run.now = 0 \wedge \neg sel[1].now \qquad \text{selection action} \\
& \qquad \to sel[1] :- (\lambda t \cdot \mathsf{true}) \,;\, run :- (\lambda t \cdot 1) \\
& \quad [\!] \ \ldots \\
& \quad [\!] \ run.now = 0 \wedge \neg sel[n].now \\
& \qquad \to sel[n] :- (\lambda t \cdot \mathsf{true}) \,;\, run :- (\lambda t \cdot n) \\
& \quad [\!] \ run.now = 1 \wedge \neg u_L^1.now \wedge g_L^1.now \qquad \text{module } \mathcal{A}_1 \\
& \qquad \to L_1 \,;\, u_L^1 :- (\lambda t \cdot \mathsf{true}) \\
& \quad [\!] \ run.now = 1 \wedge \neg u_S^1.now \wedge g_S^1.now \\
& \qquad\qquad \wedge (u_L^1.now \vee \neg g_L^1.now) \\
& \qquad \to wS_1c :- wS_1 ; S_1' \,;\, run :- (\lambda t \cdot 0) \,;\, u_S^1 :- (\lambda t \cdot \mathsf{true}) \\
& \quad [\!] \ run.now = 1 \wedge \neg u_S^1.now \wedge \neg gg_{A_1}.now \\
& \qquad \to run :- (\lambda t \cdot 0) \,;\, u_S^1 :- (\lambda t \cdot \mathsf{true}) \\
& \quad [\!] \ \ldots \\
& \quad [\!] \ run.now = n \wedge \neg u_L^n.now \wedge g_L^n.now \qquad \text{module } \mathcal{A}_n \\
& \qquad \to L_n \,;\, u_L^n :- (\lambda t \cdot \mathsf{true}) \\
& \quad [\!] \ run.now = n \wedge \neg u_S^n.now \wedge g_S^n.now \\
& \qquad\qquad \wedge (u_L^n.now \vee \neg g_L^n.now) \\
& \qquad \to wS_nc :- wS_n ; S_n' \,;\, run :- (\lambda t \cdot 0) \,;\, u_S^n :- (\lambda t \cdot \mathsf{true})
\end{array}
\tag{4.7}
$$

$[\!]\ \ run.now = n \wedge \neg u_S^n.now \wedge \neg gg_{A_n}.now$
$\rightarrow run :- (\lambda t \cdot 0)\ ;\ u_S^n :- (\lambda t \cdot \mathsf{true})$

$[\!]\ \ sel.now \wedge run.now = 0$ — update action
$\rightarrow Update\ ;\ UT;$
if $now \neq now_c$
then $u_L^1, \ldots, u_S^n, sel :- (\lambda t \cdot \mathsf{false})\ ;\ now_c := now$
else skip
fi

od
end,

$$
\begin{array}{lcl}
Init & \triangleq & Init_1\ ;\ \ldots\ ;\ Init_n\ ;\ wS_1c, \ldots, wS_nc :- wS_1, \ldots, wS_n; \\
 & & \quad run :- (\lambda t \cdot 0)\ ;\ sel, u_L^1, \ldots, u_S^n :- (\lambda t \cdot \mathsf{false}) \\
Update & \triangleq & Update_1\ ;\ \ldots\ ;\ Update_n,\ \text{where } Update_k \triangleq wS_k :- wS_kc \\
S'_k & \triangleq & S_k[wS_k :- wS_kc] \\
gg_{A_k} & \triangleq & g_L^k \vee g_S^k
\end{array}
$$

Recall that the set z of global variables of $\overline{\mathcal{P}}$ is, initially, the union of the global variables sets of each module: $z = \bigcup_k z_k$. If the communication among some modules of $\mathcal{P}$ should not be disclosed at the interface of $\overline{\mathcal{P}}$, the variables that model such channels will be *hidden* within the system $\mathcal{P}$.

Further, the local variables $x = \bigcup_k x_k$, to which we add the hidden variables. We also add copies (wS_kc) of the original write variables of each action body S_k. They replace the original variables wS_k, therefore we have $S'_k = S_k[wS_k :- wS_kc]$. Finally, the list x is completed by adding the array sel and the execution indicator, run.

The above definition of the '$\sharp$' - based composition of CAS says that, whenever there is a change in the input, the composed system reacts based on the state of all its modules.

Observe that time is not advanced unless all the modules have given their respective responses to the input. As distinct from the discrete case, the synchronized parallel composition of partitioned CAS does not need to test gg_P upon the entrance of the loop. This is motivated by the fact that $\overline{\mathcal{P}}$ cannot execute skip actions forever. The conditional statement following $Update$ enforces termination if time has not been advanced by any of the module actions.

Since CAS semantics is given in terms of ordinary action systems, all the modularity results proved in chapter 3 hold for the composition defined by (4.7), too. Consequently, due to Corollary 1, any module can be trace refined independently, provided that its invariant is *proper*.

4.4.3 Example Revisited - Synchronized Design Approach

Let us now compose the systems $\mathcal{S}_1$ and $\mathcal{S}_2$, defined in section 4.4.1, by using the '$\sharp$' operator. It is easy to check that $\mathcal{S}_1$ and $\mathcal{S}_2$ are partitioned CAS. As a result, we

get the new partitioned CAS, $\mathcal{S}_{new} = \mathcal{S}_1 \sharp \mathcal{S}_2$. Next, we translate $\mathcal{S}_{new}$ into $\overline{\mathcal{S}}_{new}$, by applying the definition (4.7).

$$
\begin{array}{ll}
& \overline{\mathcal{S}}_{new}(counter : \mathsf{Real}_+ \to \mathsf{Nat}) \\
\triangleq & \textbf{begin var } Y, Y_c : \mathsf{Real}_+ \to \mathsf{Real}_+, counter_c, run : \mathsf{Real}_+ \to \mathsf{Nat}, \\
& \quad sel[1..n], u_1, u_2 : \mathsf{Real}_+ \to \mathsf{Bool}, now, now_c : \mathsf{Real}_+ \bullet \\
& \quad counter, counter_c, run, Y, Y_c :- (\lambda t \cdot 0); \\
& \quad sel, u_1, u_2 :- (\lambda t \cdot \mathsf{false}) \,;\, UT \,;\, now_c := now; \\
& \textbf{do} \\
& \quad \neg sel[1].now \wedge run.now = 0 \\
& \quad \to sel[1] :- (\lambda t \cdot \mathsf{true}) \,;\, run :- (\lambda t \cdot 1) \\
& \quad [\!] \ \neg sel[2].now \wedge run.now = 0 \\
& \quad \to sel[2] :- (\lambda t \cdot \mathsf{true}) \,;\, run :- (\lambda t \cdot 2) \\
& \quad [\!] \ (run.now = 1 \wedge \neg u_1.now \wedge Y.now = 0 \\
& \quad \to Y_c :- (\lambda t \cdot v_1 * (t - now)); \\
& \qquad dir :- (\lambda t \cdot \mathsf{true}) \\
& \quad [\!] \ run.now = 1 \wedge \neg u_1.now \wedge \\
& \qquad ((Y.now = Y_0 \wedge dir.now) \vee Y.now = Y_H) \\
& \quad \to Y_c :- (\lambda t \cdot Y_H - v_2 * (t - now)); \\
& \qquad dir :- (\lambda t \cdot \mathsf{false}) \\
& \quad [\!] \ run.now = 1 \wedge \neg u_1.now \wedge Y.now = Y_1 \\
& \quad \to \ \ Y_c :- (\lambda t \cdot Y_1 + v_1 * (t - now)) \\
& \qquad [\!] \ Y_c :- (\lambda t \cdot Y_1 - v_2 * (t - now))); \\
& \qquad run :- (\lambda t \cdot 0) \,;\, u_1 :- (\lambda t \cdot \mathsf{true}) \\
& \quad [\!] \ run.now = 1 \wedge \neg u_1.now \wedge \\
& \qquad \neg(Y.now = 0 \vee Y.now = Y_1 \vee \\
& \qquad ((Y.now = Y_0 \wedge dir.now) \vee Y.now = Y_H)) \\
& \quad \to run :- (\lambda t \cdot 0) \,;\, u_1 :- (\lambda t \cdot \mathsf{true}) \\
& \quad [\!] \ run.now = 2 \wedge \neg u_2.now \wedge Y.now = Y_0 \\
& \quad \to counter_c :- (\lambda t \cdot counter.now + 1); \\
& \qquad run :- (\lambda t \cdot 0) \,;\, u_2 :- (\lambda t \cdot \mathsf{true}) \\
& \quad [\!] \ run.now = 2 \wedge \neg u_2.now \wedge Y.now \neq Y_0 \\
& \quad \to run :- (\lambda t \cdot 0) \,;\, u_2 :- (\lambda t \cdot \mathsf{true}) \\
& \quad [\!] \ sel.now \wedge run.now = 0 \\
& \quad \to Y :- Y_c \,;\, counter :- counter_c \,;\, UT; \\
& \qquad \mathsf{if}\ now \neq now_c \\
& \qquad \mathsf{then} \\
& \qquad \quad sel, u_1, u_2 :- (\lambda t \cdot \mathsf{false}) \,;\, now_c := now \\
& \qquad \mathsf{else\ skip} \\
& \qquad \mathsf{fi} \\
& \textbf{od} \\
& \textbf{end}
\end{array}
$$

If we repeat the previously described scenario, at moments $t_1, t_3, t_5, \ldots$, when

$Y.now = Y_0$, the definition of $\overline{\mathcal{S}}_{new}$ lets us preserve the old values of Y. Hence, the synchronization mechanism enables the system $\overline{\mathcal{S}}_2$ to update the variable $counter$ correctly, even when $\overline{\mathcal{S}}_2$ is selected for execution after $\overline{\mathcal{S}}_1$. The order of system execution ceases to be a correctness issue.

It is possible that one can find a solution by considering the traditional parallel composition of $\mathcal{S}_1$ and $\mathcal{S}_2$. One could try, for instance, to bring the variable dir to the interface of $\mathcal{S}_1$, so that it may be read by $\mathcal{S}_2$, which in turn could take a correct decision. However, as concluded also in chapter 3, such a solution requires exposure of internal functionality and presumes knowledge of the internal behavior of modules (in our case, of $\mathcal{S}_1$). Moreover, for such a trick to work, one has to first detect which are the time points when the system's functioning requires a detailed analysis. This would be necessary in order to avoid potential discontinuity-caused errors [107]; in our example, one has to detect the possible malfunction of the counter system in points similar to the (t_1, Y_0) tuple.

In comparison, due to the synchronized semantics, one can design the overall system modularly, by simply plugging $\mathcal{S}_1$ and $\mathcal{S}_2$ together, and without encoding any kind of communication between these modules. Also, in case one needs to add similar modules to the composed system, the synchronized composition lets one reuse the existing modules. At the same time, the global behavior is under control: barrier synchronization ensures correct outputs to all inputs.

4.5 Summary and Related Work

In this chapter we have extended the syntax of continuous action systems [27] by modeling the concept of single point invocation of an action. As a result, our model does not allow the infinite execution of an enabled continuous action, at the same time point. The issue of preventing timelocks is moved to the implementation level. We have proposed a general solution that uses an execution marker for each action, namely, a boolean variable that is set true after the respective action has been executed. Assuming a particular moment now, new execution rounds are allowed as long as there is at least one flag on false. If the executed actions do not consume time, that is, now remains unchanged, and all flags are true, the execution terminates. On the other hand, if now changes, all flags are reset. Consequently, any formerly selected action can be reselected in the future.

Last but not least, we have proposed a *barrier synchronization mechanism* for CAS, similar to the one developed in chapter 3, for discrete systems. Employing the synchronization operator allows for compositionality and modular reasoning of an important class of hybrid systems, that is, systems with *discontinuous changes in continuous variable values*. These arise, in general, in complex systems, such as aircrafts, which often operate in different modes of continuous behavior. When mode changes occur, the continuous dynamics may change abruptly, yet the controller should identify the event and react in consequence. We have shown how

synchronized CAS can faithfully model such systems, at the same time allowing for a modular design perspective.

Related Work. The CAS language is suited for modeling and verification of mission-critical systems, since it allows for the explicit failure of the system (modeled by the “abort” statement). It also allows references to historical values of the variables in guards and expressions (e.g., x.(now - 1)). Such features do not appear in modeling languages like, for example, *hybrid automata* [6, 86] or *hybrid I/O automata* [120].

Rönkkö and Li have introduced *linear hybrid action systems* as a way of modeling hybrid systems with linear continuous behavior [140]. The approach is more restrictive than CAS since only smooth functions (without discontinuities) can be handled. We have shown through the example of section 4.4.1 that CAS allows any type of function to describe the continuous evolutions. Moreover, Rönkkö’s and Li’s approach uses an implicit notion of time, hence the formalism is not intended for modeling real-time systems. Our model facilitates the description of real-time systems, as chapter 6 demonstrates.

The composition of timed systems expressed as communicating processes is also analyzed by Bornot and Sifakis [47], who strive for *maximal progress*: whenever interleaving and synchronization are both possible, synchronization is preferred. In our approach, the property holds by default when one employs the synchronization operator for composing CAS modules.

CHARON is an environment that supports structured hierarchial modeling of hybrid systems [11]. In CHARON a system is described by a collection of *agents* that communicate with their environment via shared variables; the behavior of an agent is called a *mode* and it is basically a hierarchical state machine. CHARON has a formal compositional semantics with a notion of refinement: traces of a mode can be computed from traces of its submodes. The approach assumes an interleaving semantics of discrete updates, whereas updates of analog variables must be synchronized. Synchronized (continuous) action systems allow both non-hierarchical and also hierarchical perspectives. In the latter case, the same shared-variable mechanism can be used for communication among modules.

Compositionality of concurrent hybrid behaviors is also central to models such as hybrid I/O automata [120] and *hybrid modules* [12]. In the latter, Alur and Henzinger have developed an assume-guarantee principle for reasoning about timed and hybrid modules. The authors separate the so called *update rounds*, which take no time, where discrete or clock variables are updated by the modules or environment, from *time rounds*, which have specified durations. Each update round consists of several subrounds. From this point of view, this approach is similar to our synchronized hybrid environment, yet ours differs in that we do not make the distinction between rounds updating global variables, and rounds updating time. All global variables (be they discrete valued or continuous valued time variables)

and time are updated by a sequence of statements, at the end of the same cycle. Moreover, the timed modules employed by Alur and Henzinger are partially ordered, and the respective update rounds follow that ordering. This is especially valid when *await dependencies* occur. We do not use the clause *awaits* in our models, thus the order of module execution is permitted to be nondeterministic but this internal nondeterminism does not affect the visible state.

The *hybrid constraint language* approach (Hybrid cc) [82] to modeling and verification of hybrid systems assumes that various aspects of the given hybrid automaton are expressed as constraints. The technique supports logical concurrency for program construction, allowing a similar preemption construction to Esterel [45].

Chapter 5

Hybrid Systems Analysis

Hybrid control systems can be quite difficult to build, due to the interaction of the continuous system behavior with the discrete controller. Hence, simulating a formal model of the system is most useful, allowing one to find potential trouble spots before proceeding to full formal verification.

In this chapter, we first introduce a symbolic simulation tool for continuous action systems, as a means of analyzing high-level models [25]. We have built the tool in Mathematica [156], a powerful computer algebra package, also equipped with good plotting facilities.

The main problem in carrying out formal analysis of hybrid system models is their infinite state space, which, in turn, is a result of the continuous evolutions involved. There are uncountably many successor states from a given state of a hybrid system. Furthermore, checking whether a hybrid system ever reaches a bad state is undecidable.

Such issues can be overcome by finding suitable abstractions of the continuous dynamics, or of both the discrete transitions and continuous flows, which induce transition systems that can be model-checked for certain properties [151]. In principle, the result of such a procedure is just an assertion that the model behaves as expected, with respect to the verified property.

In contrast, a deductive, iterative approach to formal analysis of hybrid models provides the designer with important insights on the overall system behavior. The drawback is the lack of a "push-button" technique; on the other hand, the gain in understanding is necessary, if one wants to improve the system representation at later design stages.

For parametric models, the outcome of the analysis consists of constraints on parameters, or relationships between parameters, which define the set of all possible values guaranteeing that some system property holds for any possible behavior.

In the following, we also present a deductive way of synthesizing correct parameter values, by using superposition of nonconflicting invariants [24]. Illustrative examples show the proposed approaches at work.

5.1 Symbolic Simulation of Hybrid System Behavior

Prior to, or as an alternative to verification, the simulation of a hybrid system model brings many benefits to the designer, by paving the way towards an error free abstraction. A lot of effort has been devoted to developing simulation tools for hybrid systems, targeting various modeling languages. Such tools include the Hybrid Chi simulator [40], Dymola [74], Shift [79], and Simulink [124].

The contribution of this section is to show how to *simulate* the behavior of hybrid systems that are modeled as CAS. The simulation technique that we use is symbolic. Given the simulation parameters, we represent states using predicates, and we construct the exact analytic functions that describe the behavior of the hybrid system over time (rather than just numeric approximations of the behavior). The simulation method is based on calculating symbolically the next time point when at least one action is enabled, using the minimization capabilities of some programming language. This means that our simulation method is not dependent on choosing a fixed sampling interval, but that the simulation rather proceeds from one interesting time point to the next. These interesting time points can be very dense in times when the behavior changes rapidly, and be sparse at other times.

The Generic CAS Simulator. In this paragraph, we describe the CAS simulator in a generic setting, independent of the programming language used, however with its usability certainly benefitting from having as powerful language as possible.

The symbolic simulation of a CAS consists of three major steps, as follows:

1. Solving each guard separately and finding a list of times in the future when the guard will evaluate to true.
2. Extracting the least of the times in the list for each guard.
3. Collecting the results from step one and two, from all guards, and determining a globally minimal next time. Having found it, one has simultaneously determined whether there is one or several guards satisfied at the respective time moment. If just one guard is satisfied, the corresponding action body is executed, thus changing some of the program variables. If there are several guards simultaneously true, then the user is asked to supply the choice of action to be taken. It is also possible that the machine makes a random choice among the enabled actions.

In the first step, the computation of the solution list for the guards can be arbitrarily complicated depending on the structure of the guards. The guard may involve the solution of higher order algebraic equations, or nonlinear differential equations, or both, in which case analytic solutions to the guards are probably impossible to obtain. In this case, one has to resort to a numerical solution of the guards, e.g., integrate differential equations forward in time using some appropriate numerical scheme. Then, we can still obtain an approximated continuous

solution by interpolating the numerical solution with linear functions between the numerically obtained values.

In case the list of minimum values for the guard from step one is a collection of finite analytic expressions, we will be able to proceed to step two without loss of accuracy. The identification of the minimum value in step two, that is, sorting the list of solutions to a guard, may be numerically cumbersome. The expressions in the list can easily have the tendency of becoming increasingly complicated as time goes on; then, in the end we have to resort to evaluating the minimum values numerically. This immediately makes the comparison of values very close to each other prone to mistakes. The third step is in principle as hard as step two, only now we are comparing the minimum values for each guard with each other.

The usability of the symbolic simulator is thus largely dependent on whether we are able to pass through step one to three, using symbolic expressions.

The main function of the simulator is given below, in pseudocode.

```
S0();                              (* initialize variables *)

now = 0;                             (* start at time t = 0 *)

Max_now = 100;                     (* simulate until Max_now *)

while ( now < Max_now )                 (* loop until Max_now*)
{                                       (* loop through all guards *)

                        (* and find min time when some guard holds *)
  for ( i = 1 ; i ≤ m ; i = i+1 )
    {

                             (* list of solutions to guard i *)
    sol_list = SolveGuard(i);

                  (* find the earliest time in solution list *)
    TentativeNextNow = ExtractMinTime(sol_list);

    NextActionTime = Max_now;
    NoOfNextActions = 0;

                                (* find the globally minimal next time *)
                                   (* and the corresponding actions *)

                                   (* check for any solution *)
    if ( IsNumeric(TentativeNextNow) == False )
    continue;

                                 (* check for later solution *)
    if ( NextActionTime < TentativeNextNow )
    continue;
```

```
          (* we have several guards with earliest time *)

                 (* form list of possible next actions *)
   if ( NextActionTime == TentativeNextNow )
     {
     NoOfNextActions = NoOfNextActions + 1;
     NextActionList[ NoOfNextActions ] = i;
     continue;
     }

                                  (* new earliest time *)
   if ( NextActionTime > TentativeNextNow )
     {
     NextActionTime = TentativeNextNow;
     NoOfNextActions = 1;
     NextActionList[ NoOfNextActions ] = i;
     continue;
     }
   }

                        (* take appropriate next action *)

                             (* no next now available *)
  if ( NoOfNextActions == 0 ) break;

                (* no next now within maximum limit *)
  if ( NextActionTime == Max_now ) break;

               (* next time available, it is unique *)
  if ( NoOfNextActions == 1 )
    {
    now = NextActionTime;
    S_NextActionList[1];
    }

           (* next time available, it is not unique *)
  if ( NoOfNextActions > 1 )
    {
    now = NextActionTime;
    Action_Choice =
      UserSelect(NoOfNextActions,NextActionList);
    S_Action_Choice;
    }
}                                (* end of while loop *)
```

5.1.1 Mathematica-based Simulation of Continuous Action Systems

We have chosen to implement the simulator in *Mathematica* [156], a powerful computer algebra package. Besides allowing us to get symbolic solutions to the time varying behavior of the hybrid system, Mathematica also provides good facilities for visualizing the system evolution as graphs.

The state variable functions are often described by differential equations. The differential equation solver of Mathematica is then very useful, in particular for those cases where it is easy to find an exact solution. If we do not get an analytic solution, we can still get a numeric approximation of the time functions, and use these approximations in our simulation. The approximation will introduce an uncertainty into the simulation, but still allows us to carry out the simulation independently of a fixed sampling interval.

Our tool [25] performs simulation of CAS fully automatically. It is essentially an interpreter with plotting capabilities for CAS, written in the programming language of Mathematica. Our experiences with this tool have been very promising. Besides providing a good visualization of the behavior of hybrid systems, it has also been quite efficient in harnessing the power of Mathematica.

Most of the simulation tools for hybrid systems use either a fixed-step or a variable-step numerical solver to approximate the differential equation by a difference equation [151]. One of the main strengths of our tool is the fact that, at least in the linear case, we do not need to provide a suitable discretization of the continuous system dynamics. As a result, we perform the simulation on the continuous time model, without tolerances.

The tool is parametric, in the sense that the number of guarded actions and the number of functions that are evaluated and plotted are set by the user, depending on the system that one wants to analyze. These parameters are denoted by variables NoOfGuards, NoOfFuncs, respectively. There is also an upper bound for the simulation time, denoted by EndTime, in case the simulation goes on forever.

To run a simulation, the user has to:

- Supply the values of EndTime, NoOfGuards, and NoOfFuncs.
- Specify the initial conditions by giving initial values to all the variables.
- Input the guards and the corresponding action bodies as simple ASCII files, yet using valid Mathematica commands.

The simulation tool then computes the behavior of the model, under these specific choices.

We have applied our simulation technique to a small collection of hybrid systems. In the coming sections, we describe two applications in more detail. The first one models a heat producing nuclear reactor with two cooling rods, and the second one is the two-tank system introduced in chapter 4. The tool has proved to be very useful in these and other cases that we have tried. It has provided a way

of exploiting nondeterminism in specifications, and at most times it has confirmed the a priori intuition about the system behavior.

5.1.2 Linear Hybrid Models: A Nuclear Reactor Temperature Control System

The current case-study has also been analyzed by Alur et al. [7]. The hybrid system is a temperature control system (TCS, for short) for a heat producing reactor. It is described by the temperature as a function of time $\theta(t)$. The reactor starts from the initial temperature θ_0 and heats up at a given rate v_r. Whenever the core reaches the critical temperature θ_M, it is designed to be cooled down by inserting into the core either of two rods, modeled by variables $x_1(t)$ and $x_2(t)$. These are in fact clocks that measure the time elapsed between two consecutive insertions of the same rod, respectively. The cooling proceeds at rate v_1 or v_2 depending on which rod is being used; the cooling stops when the reactor reaches a given minimum temperature θ_m, by releasing the respective inserted rod. The rod used for cooling is then unavailable for a prescribed time T, after which it is again available for cooling.

The object of the simulation is to ascertain that the reactor never reaches the critical temperature θ_M without at least one of the rods available, otherwise a shutdown will be initiated.

The action system $\overline{\mathcal{TCS}}$ (where time is explicitly advanced) consists of a set of initializing statements and a collection of guards and their corresponding action bodies (see Figure 5.3). The last action (action 5) has $abort$ as its body, indicating that the shutdown state is not desired.

Let $\Delta\theta = \theta_M - \theta_m$. Obviously, the time that the coolant needs to increase its temperature from θ_m to θ_M is

$$\tau_r = \Delta\theta / v_r,$$

and the cooling times using $rod1$ and $rod2$ are

$$\tau_1 = \Delta\theta / v_1 \text{ and } \tau_2 = \Delta\theta / v_2,$$

respectively.

The sequence of heating and cooling times is shown in Figure 5.1.

Observe that in our model $\overline{\mathcal{TCS}}$ we have used variable $start$, which denotes the moment when the system starts evolving in a new state. Also, in order to aid intuition about the system's continuous behavior, we have given both the analytic solutions of the differential equations, as well as the actual differential equations that they satisfy. As an alternative representation of the behavior of $\overline{\mathcal{TCS}}$, we give its state transition diagram, in Figure 5.2.

Clearly, if $\tau_r \geq T$ (the temperature rises at a rate slower than the time of recovery of the rods), then the *shutdown* state is not reachable. However, this can

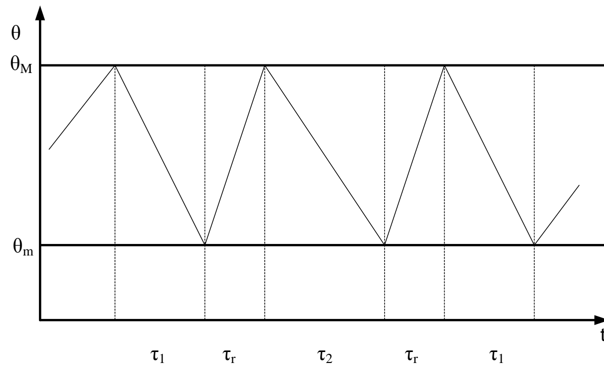


Figure 5.1: The heating and cooling times

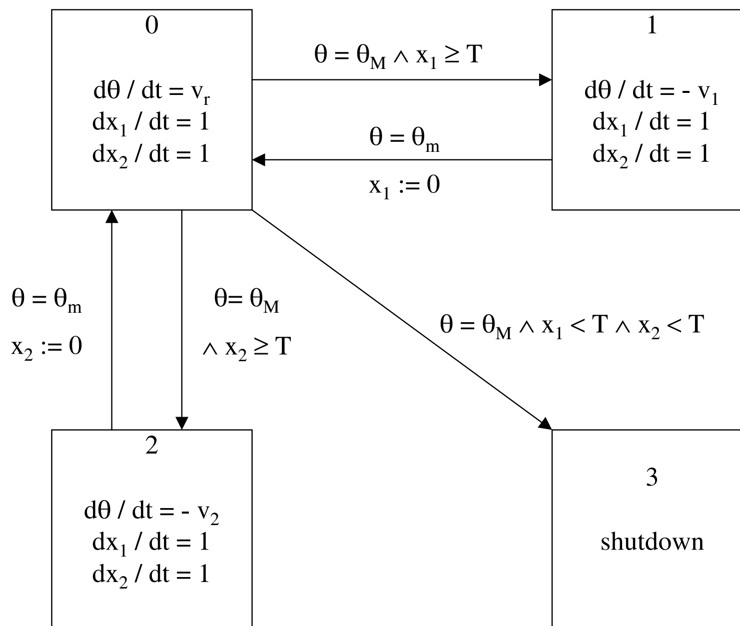


Figure 5.2: The state transition diagram of the temperature control system.

be a too strong condition for not running into the undesired state. Inspecting Figure 5.1 one can find a weaker condition [7]:

$$2\tau_r + \tau_1 \geq T \wedge 2\tau_r + \tau_2 \geq T \qquad (5.1)$$

Relation (5.1) claims that the shutdown state will never be reached if the time between two insertions of the same rod is greater than or equal to the time needed for the rod to recover.

To get a first assurance that condition (5.1) is indeed sufficient, we proceed with

$$
\begin{array}{ll}
& \overline{\mathcal{TCS}} \\
= & \textbf{begin var} \quad x_1, x_2, c : \mathsf{Real}_+ \rightarrow \mathsf{Real}_+; \\
& \quad \theta : \mathsf{Real}_+ \rightarrow \mathsf{Real} \,;\, state : \mathsf{Real}_+ \rightarrow \{0,1,2,3\}; \\
& \quad start, now : Real_+ \bullet now := 0; \\
& \quad state :- (\lambda t \cdot 0) \,;\, c :- (\lambda t \cdot t - now); \\
& \quad x_1 :- (\lambda t \cdot T_1 + c.t) \,;\, x_2 :- (\lambda t \cdot T_2 + c.t); \\
& \quad \theta :- (\lambda t \cdot \theta_0 + v_r * c.t); \\
& \quad start := now \,;\, now := min\{t' \geq now \mid gg.t'\}; \\
& \textbf{do} \quad \{\textbf{action1} : \textbf{cool with rod1}\} \\
& \qquad state.now = 0 \wedge \theta.now = \theta_M \wedge x_1.now \geq T \\
& \qquad \rightarrow c :- (\lambda t \cdot t - now) \,; \qquad /* \dot{c} = 1 \\
& \qquad\quad \theta :- (\lambda t \cdot \theta_M - v_1 * c.t) \,; \quad /* \dot{\theta} = -v_1 \\
& \qquad\quad state :- (\lambda t \cdot 1); \\
& \qquad\quad start := now \,;\, now := min\{t' \geq now \mid gg.t'\} \\
& \quad [\!] \quad \{\textbf{action2} : \textbf{release rod1}\} \\
& \qquad state.now = 1 \wedge \theta.now = \theta_m \\
& \qquad \rightarrow c :- (\lambda t \cdot t - now) \,; \qquad /* \dot{c} = 1 \\
& \qquad\quad x_1 :- (\lambda t \cdot t - now) \,; \qquad /* \dot{x_1} = 1 \\
& \qquad\quad \theta :- (\lambda t \cdot \theta_m + v_r * c.t) \,; \quad /* \dot{\theta} = v_r \\
& \qquad\quad state :- (\lambda t \cdot 0); \\
& \qquad\quad start := now \,;\, now := min\{t' \geq now \mid gg.t'\} \\
& \quad [\!] \quad \{\textbf{action3} : \textbf{cool with rod2}\} \\
& \qquad state.now = 0 \wedge \theta.now = \theta_M \wedge x_2.now \geq T \\
& \qquad \rightarrow c :- (\lambda t \cdot t - now) \,; \qquad /* \dot{c} = 1 \\
& \qquad\quad \theta :- (\lambda t \cdot \theta_M - v_2 * c.t) \,; \quad /* \dot{\theta} = -v_2 \\
& \qquad\quad state :- (\lambda t \cdot 2); \\
& \qquad\quad start := now \,;\, now := min\{t' \geq now \mid gg.t'\} \\
& \quad [\!] \quad \{\textbf{action4} : \textbf{release rod2}\} \\
& \qquad state.now = 2 \wedge \theta.now = \theta_m \\
& \qquad \rightarrow c :- (\lambda t \cdot t - now) \,; \qquad /* \dot{c} = 1 \\
& \qquad\quad x_2 :- (\lambda t \cdot t - now) \,; \qquad /* \dot{x_2} = 1 \\
& \qquad\quad \theta :- (\lambda t \cdot \theta_m + v_r * c.t) \,; \quad /* \dot{\theta} = v_r \\
& \qquad\quad state :- (\lambda t \cdot 0); \\
& \qquad\quad start := now \,;\, now := min\{t' \geq now \mid gg.t'\} \\
& \quad [\!] \quad \{\textbf{action5} : \textbf{shutdown}\} \\
& \qquad state.now = 0 \wedge \theta.now = \theta_M \wedge x_1.now < T \wedge x_2.now < T \\
& \qquad \rightarrow state :- (\lambda t \cdot 3) \,;\, abort \\
& \textbf{od} \\
& \textbf{end} : \theta_0, \theta_m, \theta_M, v_r, v_1, v_2, T_1, T_2, T
\end{array}
$$

Figure 5.3: The TCS action system model

the simulation of the TCS model for two sets of parameters: the first set chosen to satisfy condition (5.1), the second set chosen not to satisfy the same condition. The simulation results should either confirm or deny our assertion. In the second case, at some point in time, the simulation should run into *abort* by executing the action 5 of the $\overline{\mathcal{TCS}}$ action system.

5.1.3 Simulating the Behavior of the Temperature Control System in Mathematica

The starting point for the formulation of the simulation is to take the initializing expressions and the expressions of the guarded actions of $\overline{\mathcal{TCS}}$ as such, with as few numerical or logical manipulations as possible. This confirms our basic strategy of simulating the model as given, thus exposing any possible modeling errors like in the spelling of the model, or in the logic of the guarded actions. The initialization of $\overline{\mathcal{TCS}}$ is implemented in the language of the symbolic manipulation program as

$$\begin{aligned}
\mathsf{now} &= 0; \\
\mathsf{c}[\mathsf{t_}] &= \mathsf{t} - \mathsf{now}; \\
\mathsf{x}_1[\mathsf{t_}] &= \mathsf{T}_1 + \mathsf{c}[\mathsf{t}]; \\
\mathsf{x}_2[\mathsf{t_}] &= \mathsf{T}_2 + \mathsf{c}[\mathsf{t}]; \\
\theta[\mathsf{t_}] &= \theta_0 + \nu_\mathsf{r} * \mathsf{c}[\mathsf{t}]; \\
\mathsf{state}[\mathsf{t_}] &= 0;
\end{aligned}$$

In Mathematica, $t_$ signifies the fact that t is the variable in the function that is being defined. We assume that we start in $state0$, with the rods 1 and 2 both available for cooling, hence the clocks x_1 and x_2 are initialized to the (constant) values T_1 and T_2 (time units), respectively.

As usual, the guards are boolean conditions, which we test for the value of true. In the implemented TCS model, the first guard has the form

```
guard1solution = InequalitySolve[
                     state[t] == 0 &&
                     θ[t] == θM &&
                     x1[t] >= T &&
                     t >= start && t <= EndTime, t
                     ]
```

Here, we are using the Mathematica built-in function InequalitySolve to determine the next moment or moments in time at or after now, when all the conditions of guard 1 become true: the system is in state 0, it has reached the critical temperature and rod1 is available. As a result of solving the simultaneous inequalities we obtain a list called guard1solution, which contains the empty set, or a collection of discrete times and/or finite or infinite ranges of times for which the conditions hold. This list is passed to a subroutine that picks out the earliest time at which guard 1 becomes true.

Similarly, the body of action 1, should we decide to take that action, is given by the following expressions:

$$\begin{aligned}
\mathsf{c}\,[\mathsf{t_}] &= \mathsf{t} - \mathsf{now};\\
\theta\,[\mathsf{t_}] &= \theta_{\mathsf{M}} - \nu_1 * \mathsf{c}\,[\mathsf{t}];\\
\mathsf{state}\,[\mathsf{t_}] &= 1;\\
\mathsf{start} &= \mathsf{now}
\end{aligned}$$

The main task of the simulation is to go through the guards one by one and determine whether they will become true at some point in time in the future. In case there are several solutions to a guard, the minimum of these times is selected, be it a discrete value or the starting value for a closed range. After this, the minimum times for all guards are compared, and the smallest of these with the corresponding action body (or actions bodies) is (are) chosen. In case the next action is one particular action, we will take that action, update the value of now and solve the guards over again. In case several guards become true at the next instance of time, all corresponding action bodies are of course possible, and the user is asked to supply the choice of action to be taken. In addition, a random mode is programmed, in which case a choice among multiple possible actions is made by the simulator.

5.1.4 Simulation Results

The essential information gained by the above procedure is a list of time moments at which some action has been taken in the model, a corresponding list of actions, and lists with symbolic values for the discrete and continuous functions of the TCS hybrid model: the system state, the temperature of the reactor $\theta(t)$ as a continuous piecewise linear function, and similar functions for the clocks $x_1(t)$ and $x_2(t)$. An artificial upper time limit $t_{max} = 100$ was supplied in case the simulation would go on forever. The results presented below are all computed automatically, within seconds.

Parameter set 1. Given the parameter values

$$T_1 = 6, T_2 = 2, T = 6, v_1 = 4, v_2 = 3, v_r = 6, \theta_0 = 0, \theta_m = 3, \theta_M = 15,$$

which satisfy condition (5.1), two of the lists mentioned above are the following:

$$\begin{aligned}
now = \{&0, 5/2, 11/2, 15/2, 23/2, 27/2, 33/2, 37/2,\\
&45/2, 49/2, 55/2, 59/2, 67/2, 71/2, 77/2,\\
&81/2, 89/2, 93/2, 99/2, 103/2, 111/2, 115/2,\\
&121/2, 125/2, 133/2, 137/2, 143/2, 147/2,\\
&155/2, 159/2, 165/2, 169/2, 177/2, 181/2,\\
&187/2, 191/2, 199/2\}
\end{aligned}$$

$$
\begin{aligned}
theta(t) : \{&6t, 25 - 4t, -30 + 6t, 75/2 - 3t, -66 + 6t,\\
&69 - 4t, -96 + 6t, 141/2 - 3t, -132 + 6t,\\
&113 - 4t, -162 + 6t, 207/2 - 3t, -198 + 6t,\\
&157 - 4t, -228 + 6t, 273/2 - 3t, -264 + 6t,\\
&201 - 4t, -294 + 6t, 339/2 - 3t, -330 + 6t,\\
&245 - 4t, -360 + 6t, 405/2 - 3t, -396 + 6t,\\
&289 - 4t, -426 + 6t, 471/2 - 3t, -462 + 6t,\\
&333 - 4t, -492 + 6t, 537/2 - 3t, -528 + 6t,\\
&377 - 4t, -558 + 6t, 603/2 - 3t, -594 + 6t\}
\end{aligned}
$$

Using the first parameter set, the graphical results of the simulation are the plots in Figures 5.4 to 5.6. The vertical lines in the graphs $action(t)$ and $state(t)$ are purposely drawn to guide the reader's eye. In this first case, the simulation did not reveal any unexpected behavior, instead it showed a regular timed behavior of the state variables.

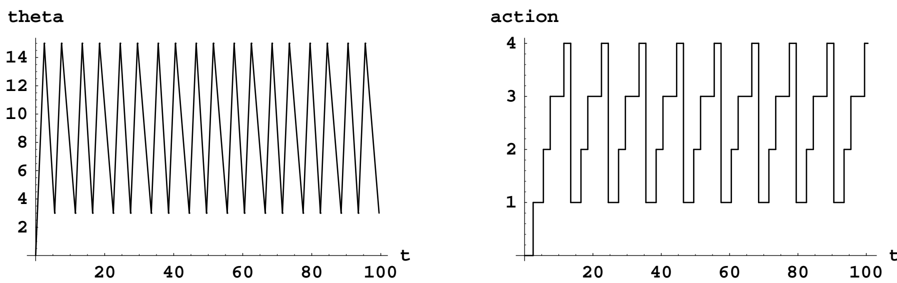


Figure 5.4: The timed behavior of θ and the executed actions (parameter set 1)

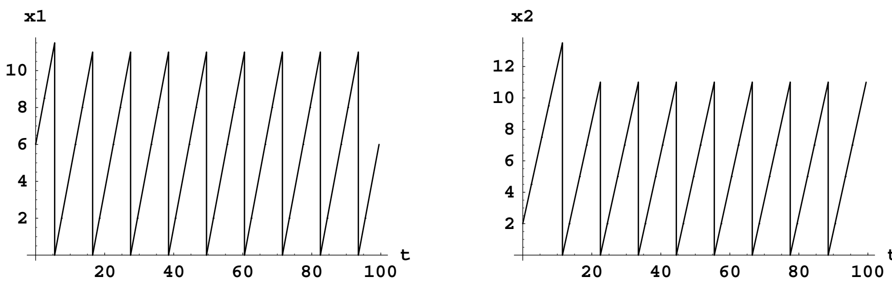


Figure 5.5: The timed behaviors of clocks x_1 and x_2 (parameter set 1)

Parameter set 2. Under a different set of values that violate the condition (5.1), that is, the same set as above except $T = 8$, the simulation shows that the reactor will reach the shutdown state. Action 5 becomes enabled at time $t = 37/2$, since neither of the rods is available (see Figure 5.7). Similar to the first case, here we

also get the graphical representations in time, of all the model variables. Figures 5.7, 5.8 and 5.9 show the respective graphs.

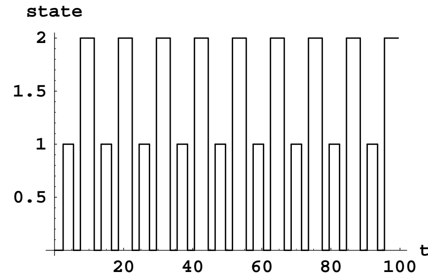


Figure 5.6: The state as a function of time (parameter set 1)

Consequently, the simulation of $\overline{\mathcal{TCS}}$ confirmed our guess: if the parameters do not satisfy condition (5.1), the system will eventually reach the shutdown state.

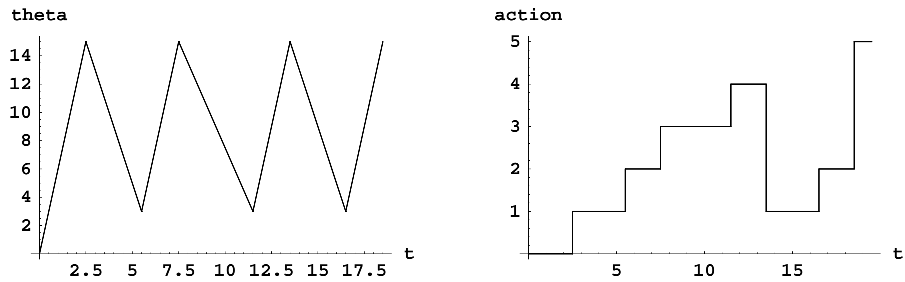


Figure 5.7: The timed behavior of θ and the executed actions (parameter set 2)

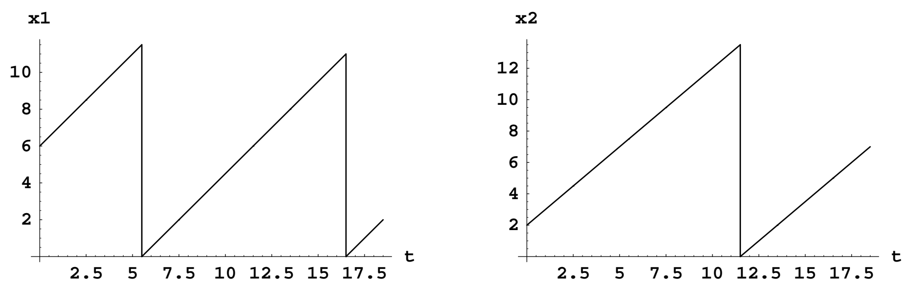


Figure 5.8: The timed behaviors of clocks x_1 and x_2 (parameter set 2)

Parameter set 3. Now, let us replace the heating rate $v_r = 6$, with $v_r = 2$, in the first parameter set, while keeping the other parameters the same. This change gives rise to a simulated situation of having both cooling rods simultaneously available,

at some point. Formally, this translates into actions 2 and 4, of the model in Figure (5.3), becoming enabled at some same moment now. This scenario is exposed by the simulator, which presents the user with the choice dialog box, updated to the current situation. This particular case is shown in Figure 5.10.

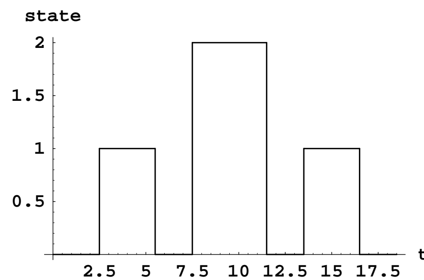


Figure 5.9: The state as a function of time (parameter set 2)

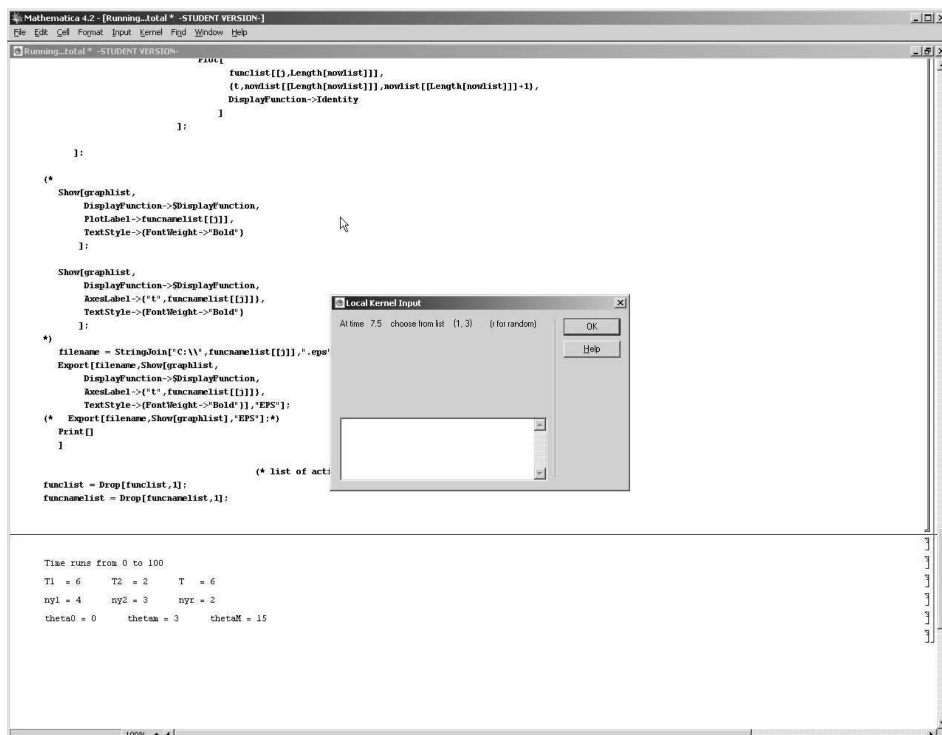


Figure 5.10: The dialog box corresponding to nondeterministic choice.

5.1.5 Simulation of the Two-Tank Action System

We return to the two-tank example of chapter 4 and give its complete action system model. Observe that here, the analytic solutions to the differential equations are given, rather than the equations themselves.

$$
\begin{array}{rl}
& \overline{\mathcal{T}anks} \\
= & \textbf{begin var } \; x_b, x_s : \mathsf{Real}_+ \rightarrow \mathsf{Real}, \\
& \quad u_p : \mathsf{Real}_+ \rightarrow \{0,1,2\}, \\
& \quad u_{iv}, u_{ov} : \mathsf{Real}_+ \rightarrow \{0,1\}, \\
& \quad u_1, \ldots, u_{14} : \mathsf{Real}_+ \rightarrow \mathsf{Bool}, \\
& \quad c : \mathsf{Real}_+ \rightarrow \mathsf{Real}_+, now, now_c : \mathsf{Real}_+ \bullet \\
& \quad now := 0 \,;\, c :- (\lambda t \cdot t) \,;\, x_b :- (\lambda t \cdot 1) \,;\, x_s :- (\lambda t \cdot 1/2 * \alpha * c.t) \,; \\
& \quad u_{iv}, u_p :- (\lambda t \cdot 1) \,;\, u_{ov} :- (\lambda t \cdot 0) \,;\, u_1, \ldots, u_{14} : -(\lambda t \cdot \mathsf{false}) \,; \\
& \quad now := min\{t' \geq now \,|\, gg.t'\} \,;\, now_c := now \,; \\
& \textbf{do} \; [\; [\!]\, 1 \leq i \leq 14 : A_i \,;\, UT \,;\, Check \,] \; \textbf{od} \\
& \textbf{end} : v_i, v_o, \alpha
\end{array}
\qquad (5.2)
$$

where:

$$
\begin{array}{rcl}
A_1 & = & \neg u_1.now \wedge 1 < x_b.now \leq 9 \wedge x_s.now = 1 \\
& & \rightarrow u_1 :- (\lambda t \cdot \mathsf{true}) \,;\, c :- (\lambda t \cdot t - now); \\
& & \quad u_{iv} :- (\lambda t \cdot 1) \,;\, u_p :- (\lambda t \cdot 1) \,;\, u_{ov} :- (\lambda t \cdot 0); \\
& & \quad x_b :- (\lambda t \cdot x_b.now + 2/7 * (v_i - \alpha) * c.t); \\
& & \quad x_s :- (\lambda t \cdot 1 + 1/2 * \alpha * c.t)
\end{array}
$$

$$
\begin{array}{rcl}
A_2 & = & \neg u_2.now \wedge 1 < x_b.now < 9 \wedge 1 < x_s.now < 9 \\
& & \rightarrow u_2 :- (\lambda t \cdot \mathsf{true}) \,;\, c :- (\lambda t \cdot t - now); \\
& & \quad u_{iv}, u_p, u_{ov} :- (\lambda t \cdot 1); \\
& & \quad x_b :- (\lambda t \cdot x_b.now + 2/7 * (v_i - \alpha) * c.t); \\
& & \quad x_s :- (\lambda t \cdot x_s.now + 1/2 * (\alpha - v_o) * c.t)
\end{array}
$$

$$
\begin{array}{rcl}
A_3 & = & \neg u_3.now \wedge (x_b.now > 9 \vee x_s.now > 9) \\
& & \rightarrow \mathsf{abort}
\end{array}
$$

$$
\begin{array}{rcl}
A_4 & = & \neg u_4.now \wedge 1 < x_b.now \leq 4.5 \wedge 4.5 \leq x_s.now < 9 \wedge \\
& & u_{iv}.now = 1 \wedge u_p.now = 1 \wedge u_{ov}.now = 1 \\
& & \rightarrow u_4 :- (\lambda t \cdot \mathsf{true}) \,;\, c :- (\lambda t \cdot t - now); \\
& & \quad u_p :- (\lambda t \cdot 0); \\
& & \quad x_b :- (\lambda t \cdot x_b.now + 2/7 * v_i * c.t); \\
& & \quad x_s :- (\lambda t \cdot x_s.now - 1/2 * v_o * c.t)
\end{array}
$$

$$\begin{aligned}
A_5 \;=\;& \neg u_5.now \wedge 4.5 < x_b.now < 9 \wedge 1 < x_s.now < 4.5 \wedge \\
& u_{iv}.now = 1 \wedge u_p.now = 1 \wedge u_{ov}.now = 1 \\
& \rightarrow u_5 :- (\lambda t \cdot \mathsf{true})\,;\, c :- (\lambda t \cdot t - now); \\
& \quad u_p :- (\lambda t \cdot 2); \\
& \quad x_b :- (\lambda t \cdot x_b.now - 2/7 * (v_i - \alpha) * c.t); \\
& \quad x_s :- (\lambda t \cdot x_s.now + 1/2 * (\alpha - v_o) * c.t)
\end{aligned}$$

$$\begin{aligned}
A_6 \;=\;& \neg u_6.now \wedge 1 < x_b.now < 9 \wedge x_s.now = 9 \\
& \rightarrow u_6 :- (\lambda t \cdot \mathsf{true})\,;\, c :- (\lambda t \cdot t - now); \\
& \quad u_{iv} :- (\lambda t \cdot 1)\,;\, u_p :- (\lambda t \cdot 0)\,;\, u_{ov} :- (\lambda t \cdot 1); \\
& \quad x_b :- (\lambda t \cdot x_b.now + 2/7 * v_i * c.t); \\
& \quad x_s :- (\lambda t \cdot 9 - 1/2 * v_o * c.t)
\end{aligned}$$

$$\begin{aligned}
A_7 \;=\;& \neg u_7.now \wedge x_b.now = 1 \wedge x_s.now = 1 \\
& \rightarrow u_7 :- (\lambda t \cdot \mathsf{true})\,;\, c :- (\lambda t \cdot t - now); \\
& \quad u_{iv} :- (\lambda t \cdot 1)\,;\, u_p :- (\lambda t \cdot 0)\,;\, u_{ov} :- (\lambda t \cdot 0); \\
& \quad x_b :- (\lambda t \cdot 1 + 2/7 * v_i * c.t); \\
& \quad x_s :- (\lambda t \cdot x_s.now)
\end{aligned}$$

$$\begin{aligned}
A_8 \;=\;& \neg u_8.now \wedge x_b.now = 1 \wedge 1 < x_s.now \leq 9 \\
& \rightarrow u_8 :- (\lambda t \cdot \mathsf{true})\,;\, c :- (\lambda t \cdot t - now); \\
& \quad u_{iv} :- (\lambda t \cdot 1)\,;\, u_p :- (\lambda t \cdot 1)\,;\, u_{ov} :- (\lambda t \cdot 1); \\
& \quad x_b :- (\lambda t \cdot 1 + 2/7 * (v_i - \alpha) * c.t); \\
& \quad x_s :- (\lambda t \cdot x_s.now + 1/2 * (\alpha - v_o) * c.t)
\end{aligned}$$

$$\begin{aligned}
A_9 \;=\;& \neg u_9.now \wedge x_b.now = 1 \wedge x_s.now = 9 \\
& \rightarrow u_9 :- (\lambda t \cdot \mathsf{true})\,;\, c :- (\lambda t \cdot t - now); \\
& \quad u_{iv} :- (\lambda t \cdot 1)\,;\, u_p :- (\lambda t \cdot 0)\,;\, u_{ov} :- (\lambda t \cdot 1); \\
& \quad x_b :- (\lambda t \cdot 1 + 2/7 * v_i * c.t); \\
& \quad x_s :- (\lambda t \cdot 9 - 1/2 * v_o * c.t)
\end{aligned}$$

$$\begin{aligned}
A_{10} \;=\;& \neg u_{10}.now \wedge x_b.now = 9 \wedge x_s.now = 1 \\
& \rightarrow u_{10} :- (\lambda t \cdot \mathsf{true})\,;\, c :- (\lambda t \cdot t - now); \\
& \quad u_{iv} :- (\lambda t \cdot 0)\,;\, u_p :- (\lambda t \cdot 2)\,;\, u_{ov} :- (\lambda t \cdot 0); \\
& \quad x_b :- (\lambda t \cdot 9 - 4/7 * \alpha * c.t); \\
& \quad x_s :- (\lambda t \cdot 1 + \alpha * c.t)
\end{aligned}$$

$$\begin{aligned}
A_{11} \;=\;& \neg u_{11}.now \wedge x_b.now = 9 \wedge 1 \leq x_s.now < 9 \\
& \rightarrow u_{11} :- (\lambda t \cdot \mathsf{true})\,;\, c :- (\lambda t \cdot t - now); \\
& \quad u_{iv} :- (\lambda t \cdot 0)\,;\, u_p :- (\lambda t \cdot 1)\,;\, u_{ov} :- (\lambda t \cdot 1); \\
& \quad x_b :- (\lambda t \cdot 9 - 2/7 * \alpha * c.t); \\
& \quad x_s :- (\lambda t \cdot x_s.now + (\alpha - v_o)/2 * c.t)
\end{aligned}$$

$$
\begin{aligned}
A_{12} \;=\; & \neg u_{12}.now \wedge x_b.now = 9 \wedge 1 \leq x_s.now < 9 \\
& \rightarrow u_{12} :- (\lambda t \cdot \mathsf{true})\,;\, c :- (\lambda t \cdot t - now); \\
& \quad u_{iv} :- (\lambda t \cdot 0)\,;\, u_p :- (\lambda t \cdot 2)\,;\, u_{ov} :- (\lambda t \cdot 1); \\
& \quad x_b :- (\lambda t \cdot 9 - 4/7 * \alpha * c.t); \\
& \quad x_s :- (\lambda t \cdot x_s.now + (\alpha - v_o/2) * c.t)
\end{aligned}
$$

$$
\begin{aligned}
A_{13} \;=\; & \neg u_{13}.now \wedge x_b.now = 9 \wedge x_s.now = 9 \\
& \rightarrow u_{13} :- (\lambda t \cdot \mathsf{true})\,;\, c :- (\lambda t \cdot t - now); \\
& \quad u_{iv} :- (\lambda t \cdot 0)\,;\, u_p :- (\lambda t \cdot 0)\,;\, u_{ov} :- (\lambda t \cdot 1); \\
& \quad x_b :- (\lambda t \cdot x_b.now); \\
& \quad x_s :- (\lambda t \cdot 9 - 1/2 * v_o * c.t)
\end{aligned}
$$

$$
\begin{aligned}
A_{14} \;=\; & \neg u_{14}.now \wedge x_b.now < 1 \\
& \rightarrow \mathsf{abort}
\end{aligned}
$$

$$
\begin{aligned}
Check \;=\; & \textbf{if } now \neq now_c \textbf{ then } u_1, \ldots, u_{14} :- (\lambda t \cdot \mathsf{false})\,;\, now_c := now \\
& \textbf{else } \mathsf{skip}
\end{aligned}
$$

In order to visualize the behavior of (5.2), we have implemented the model (5.2) in the language of Mathematica, and simulated it up to $t = 300$ time units, with our symbolic tool introduced in section 5.1.1.

In the scenario that we describe here, we assume that, initially, there is a minimal amount of liquid in the buffer, $x_b = 1\ m$; the inlet valve is opened, the outlet valve is closed, and the pump functions at low speed and transfers liquid to the supply. The liquid level in the supply increases as follows: $x_s :- (\lambda t \cdot 1/2 * \alpha * c.t)$. Also v_i, v_o, α have their nominal values, that is, $v_i = v_o = \alpha = 1\ m^3/min$.

Without changing the semantic model (5.2), we implement the action Check as an If statement, available in Mathematica's programming language. The statement below is executed after each computation of now:

$$
\begin{aligned}
& \mathsf{If}\ [\mathsf{now} \mathrel{!}= \mathsf{nowcopy}, \mathsf{u}_1[\mathsf{t}_], \ldots, \mathsf{u}_{14}[\mathsf{t}_] = 0, \\
& \qquad (\mathsf{u}_1[\mathsf{t}_] = \mathsf{u}_1[\mathsf{t}]\,;\, \ldots\,;\, \mathsf{u}_{14}[\mathsf{t}_] = \mathsf{u}_{14}[\mathsf{t}]\,;\, \mathsf{nowcopy} = \mathsf{now})]
\end{aligned}
$$

The corresponding lists with the simulation results for the described scenario are given in Figure 5.11.

The continuous evolution of variables x_b, x_s, and clock c, as well as the actions of (5.2) that were executed during simulation, are all drawn as graphs in Figures 5.12 and 5.13.

If one refines (5.2), so as to strengthen some of the guards and decrease the behavioral nondeterminism, one may attempt to identify the upper and lower bounds of the system parameters, by simulation. Such information might be useful in case one wants to change parameter values for optimization purposes. Several scenarios have been simulated, out of which it results, as expected, that if $\alpha < 0.5\ m^3/min$, for nominal values $v_i = v_o = 1\ m^3/min$, the levels in both tanks exceed the maximum of $9m$. Hence, action A_3 becomes enabled and the system aborts execution.

```
now list : {   0,   2,  30,  38,  54,  54,  54,  54,  68,  82,
              98, 112, 112, 126, 126, 142, 158, 170, 170, 182,
             198, 214, 222, 222, 230, 246, 246, 260, 274, 290, 300}

clock list : {     t,   -2+t,  -30+t,  -38+t,  -54+t,  -54+t, -54+t,
               -54+t,  -68+t,  -82+t,  -98+t, -112+t, -112+t, -126+t,
              -126+t, -142+t, -158+t, -170+t, -170+t, -182+t, -198+t,
              -214+t, -222+t, -222+t, -230+t, -246+t, -246+t, -260+t,
              -274+t, -290+t,   10}

xb list : { 1,             0.428571 + 2t/7, 26.1429 - 4t/7,  -6.42857 + 2t/7,
            9,            39.8571 - 4t/7,   24.4286 - 2t/7,  39.8571 - 4t/7,
          -18.4286 + 2t/7,  5,              -23 + 2t/7,       41 -2t/7,
          -73 - 4t/7,     -35 + 2t/7,       -35 + 2t/7,        5.57143,
          -39.5714 + 2t/7, 57.5714 - 2t/7,  106.143 - 4t/7,  -49.8571 + 2t/7,
            6.71429,      -54.4286 + 2t/7,   72.4286 - 2t/7, 135.857 - 4t/7,
          -61.2857 + 2t/7, 79.2857 - 2t/7,  149.571 - 4t/7,  -73.2857 + 2t/7,
            5,            -77.8571 + 2t/7,    7.85714}

xs list : {  0.5 t,     1,       -29 + t,     28 - t/2,   -26 + t/2,
            -53 + t,     1,       -26 + t/2,   42 - t/2,  -40 + t/2,
             58 - t/2,   2,       -54 + t/2,   72 - t/2,  72 - t/2,
            -70 + t/2,  88 - t/2,   3,        -82 + t/2, 100 - t/2,
            -98 + t/2, 116 - t/2,   5,        106 + t/2, 124 - t/2,
              1,     -122 + t/2, 138 - t/2, -136 + t/2, 154-t/2,   4}
```

Figure 5.11: Two-tank simulation results as symbolic lists.

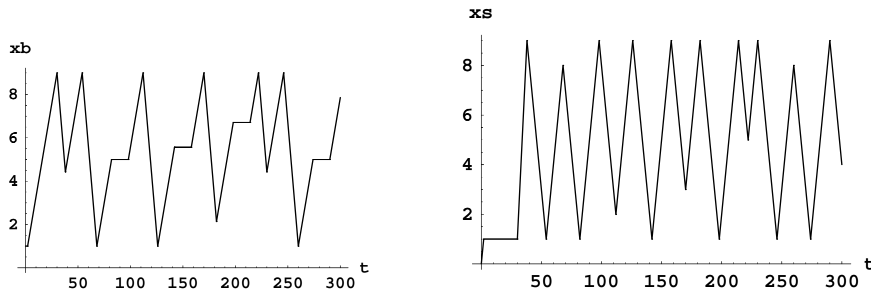


Figure 5.12: Timed evolutions of x_b, x_s.

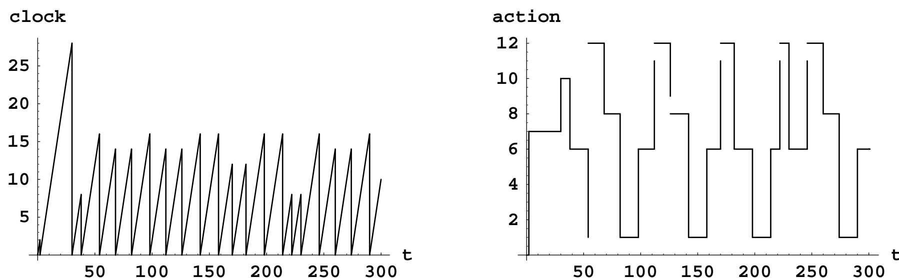


Figure 5.13: Timed evolutions of c and the executed actions.

However, in order to confirm or refute this claim, one should proceed to formal verification, as done in the TCS case-study. The overflow situation is represented graphically in Figure 5.14.

We have also observed that the system tends to swing less between extreme

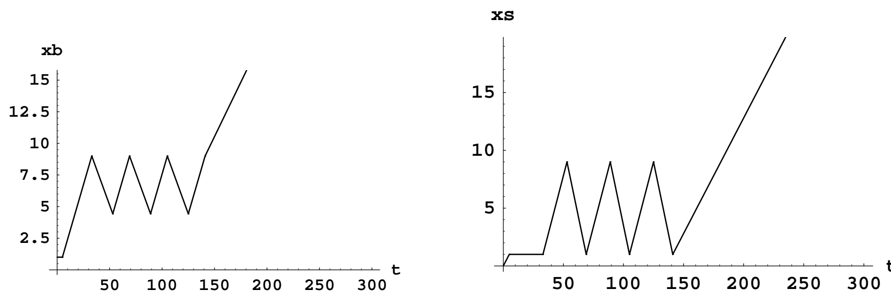


Figure 5.14: Overflow in both tanks ($\alpha = 0.4$ and $v_i = v_o = 1$).

values if initially $x_b > 1$.

Discussion. Complex continuous system dynamics can be described by *nonlinear* differential equations. To be able to simulate and reason about such systems, one needs to find the solutions of the respective equations. Solving the system of equations at run-time is more complicated.

For example, in Mathematica, one can use the function DSolve to find symbolic solutions to ordinary differential equations. Solving a differential equation consists essentially in finding the form of an unknown function. DSolve returns as its result a rule, which gives the independent variable as a pure function. Then, by applying the respective rule to all occurrences of the independent variable in a certain differential equation, by using the Mathematica command expr /. rule, one can extract the solution as a function of time.

For simulating nonlinear CAS models, we have to apply such a procedure at run-time. Our experience has shown that it may be successful only for systems with simple nonlinearities. For more complicated cases, the tool is not able to carry out the simulation by using symbolic solutions. In such cases, one should perhaps resort to numerical solutions, and apply interpolation.

5.2 Parameter Synthesis

As seen in section 5.1.2, the temperature control system is parametric in nature, meaning that it is supposed to work correctly only for specific values of its parameters v_1, v_2, v_r, θ_m, θ_M.

Let us now assume that we are confronted with a situation where it is not trivial to guess the sufficient relation between parameters, which would ensure correct and safe system functioning. In such cases, one needs to find a way of *synthesizing* the right parameter values or relationships. To accomplish this goal, here we apply the well-known deductive method of reachability analysis by proving an

invariance property. We show next how invariance checking can be used to determine the weakest relationships between model parameters. This method is mixed with an incremental way of constructing a sufficiently strong invariant, by adding information to an initial property. The latter encapsulates an approximation of the basic behavior of the hybrid system under analysis.

A state σ' is reachable from the state σ if there is a run of the hybrid system that starts in σ and ends in σ' [7]. Usually, we want to prove that some bad condition g is not reachable. This we can do by proving that some condition I is an invariant of the system, and that $I \Rightarrow \neg g$. Since the target system is parametric, we expect that $I \Rightarrow \neg g$ is not satisfied unless some relation R between parameters holds. As every reachable state satisfies I, this then shows that every reachable state satisfies $\neg g$, that is, a state where g holds cannot be reached.

Formalizing the above scenario, we describe next the proposed parameter synthesis method.

Deductive Reachability Analysis. Assume the action system

$$\begin{aligned}\overline{\mathcal{S}ys}(z : \mathsf{Real}_+ \to T_z) \;\; \triangleq \;\; & \mathbf{begin}\;\; \mathbf{var}\; start, now : \mathsf{Real}_+, x : \mathsf{Real}_+ \to T_x \bullet \\ & \quad now := 0 \,;\, start := now \,;\, S_0 \,;\, UT \,; \\ & \quad \mathbf{do}\; g_1.now \to S_1 \,;\, UT \;[\!]\; \ldots \;[\!]\; g_n.now \to \mathsf{abort}\; \mathbf{od} \\ & \mathbf{end} : p_1, \ldots, p_m,\end{aligned}$$

with $p_1, \ldots, p_m$ parameters, and

$$UT \quad \triangleq \quad start := now \,;\, now := min\{t' \geq now \;\mid\; gg.t'\}$$

Also, assume that I is a fixed invariant of $\overline{\mathcal{S}ys}$. This means that the following conjunction holds:

$$\begin{aligned}&(\mathsf{true}\; \{\!| now := 0 \,;\, start := now \,;\, S_0 \,;\, UT |\!\}\; I) \wedge \\ &(g_1 \wedge I\; \{\!| S_1 \,;\, UT |\!\}\; I) \wedge \ldots \wedge (g_{n-1} \wedge I\; \{\!| S_{n-1} \,;\, UT |\!\}\; I)\end{aligned}$$

Then, the problem of synthesizing the conditions that the parameters should satisfy reduces to finding a sufficient relation $R(p_1, \ldots, p_m)$, other than false, such that

$$(I \Rightarrow \neg g_n) \Leftarrow R(p_1, \ldots, p_m)$$

The construction of the invariant mentioned above follows an incremental pattern, which starts with a basic invariant that characterizes the functional behavior of the hybrid system. Then, this initial invariant is strengthened by other nonconflicting invariants (that is, $I_2 \not\Rightarrow \neg I_1$). Each added property incorporates new information with respect to a state variable.

Hence, our method follows closely the deductive approaches to verification, in which the *rule of invariance* is applied for proving an invariance property [44, 104].

Although the method is suited for such purposes, the creativity required by finding sufficiently strong invariants makes it more difficult to apply.

Abstract interpretation [66] carried out through predicate abstraction has been recently applied by many researchers, for computing approximations of a timed model [89, 142]. The approximation predicates are computed via stepwise refinement. One could for instance use this method as an automated way of finding suitable invariants for parametric CAS.

Alternatively, symbolic reachability analysis techniques, as implemented in the model-checker TREX [48] can be applied for parameter synthesis. Also, the tool HYTECH [87, 88] could be employed for a similar task. Nonetheless, model-checking algorithms may fail to terminate due to several potential causes: number of clocks, parameter types and ways in which parameters are related.

5.2.1 Applying Deductive Synthesis on the Temperature Control System

Returning to our nuclear reactor example, we recall that if the temperature rises to its maximum and can not decrease because no rod is available, a complete shutdown is required.

Let us remind the reader that the following relations hold:

$$\begin{array}{rcll} \Delta\theta & = & \theta_M - \theta_m & \text{(maximum temperature difference)} \\ \tau_r & = & \Delta\theta / v_r & \text{(time to increase } \theta \text{ from } \theta_m \text{ to } \theta_M) \\ \tau_1 & = & \Delta\theta / v_1 & \text{(cooling time using rod1)} \\ \tau_2 & = & \Delta\theta / v_2 & \text{(cooling time using rod2)} \end{array}$$

Here, we assume that we could not figure out the correct relationship, other than $\tau_r \geq T$, of parameters $\tau_r, \tau_1, \tau_2, T$, which guarantees that the shutdown condition never holds. Consequently, we embark on synthesizing the respective relation, by applying the combined method of incremental invariant construction, and invariance proof. The problem is to find the *weakest* parameter relationship. According to the method presented in the previous section, we need to build an invariant that satisfies

$$\begin{array}{ll} & I \Rightarrow \neg(state.now = 0 \wedge \theta.now = \theta_M \wedge x_1.now < T \wedge x_2.now < T) \\ \Leftarrow & R(\tau_r, \tau_1, \tau_2, T) \end{array} \tag{5.3}$$

Proof Technique. In general, as also demonstrated in the previous chapters, one should employ the weakest precondition semantics of statements for proving properties of (continuous) action systems. However, here we decide to resort to proving invariance by *forward analysis*, which assumes computations of *strongest postconditions* of statements, with respect to a given precondition.

Our decision is justified by the fact that timed weakest precondition computations are long and difficult to follow (see Appendix A-5). Even if, in principle,

forward analysis is weaker then backward analysis, we are backed in this choice by the fact that all the statements that we are reasoning about, in this particular example, are assignments, hence they terminate. Also, since there are no undefined expressions, one does not run the risk of not catching abortion of execution of statements. The third reason has to do with lack of automation - currently, we do not have a tool for performing weakest precondition computations on CAS.

Let us assume the Hoare triple, $\{p\}\ S\ \{q\}$, p, q predicates, denoting the *partial* correctness of S with respect to precondition p and postcondition q. Introduced by Dijkstra and Scholten [72], the *strongest postcondition predicate transformer*, denoted by $sp.S.p$, holds in those final states for which there exists a computation controlled by S, which belongs to the class *"initially p"*. Proving the Hoare triple reduces then to showing that $(sp.S.p \;\Rightarrow\; q)$ holds. The *strongest postcondition rules* for the assignment statement, and for sequential composition are as follows:

$$\begin{array}{lll} sp.(x :- (\lambda t \cdot e)).p(x) & \equiv & x = (\lambda t \cdot e) \wedge (\exists x \cdot p(x)) \\ sp.(S_1\,;S_2).p & \equiv & sp.S_2.(sp.S_1.p),\ \forall p \end{array}$$

In the following, we apply this technique as such.

Basic Invariant. We start by generating the statechart of the temperature control system, just to get a first approximation of the invariant. Then, we keep adding information to the system states, in order to figure out an invariant strong enough to ensure safety, provided that some relation $R(\tau_r, \tau_1, \tau_2, T)$ holds.

Figure 5.2 shows the states that the system can be in, and the properties that hold in each state. It is essentially a hybrid automaton view of the temperature control system, and it describes a first system property as follows:

$$\begin{array}{lll} I & \triangleq & (\forall t \in [start, now) \bullet \\ & & (state.start = 0 \Rightarrow (state.t = 0 \wedge \\ & & \qquad d\theta/dt = v_r \wedge \\ & & \qquad dx_1/dt = 1 \wedge \\ & & \qquad dx_2/dt = 1 \wedge \\ & & \qquad \theta.start = \theta_m \wedge (x_1.start = 0 \vee x_2.start = 0))) \\ & \wedge & (state.start = 1 \Rightarrow (state.t = 1 \wedge \\ & & \qquad d\theta/dt = -v_1 \wedge \\ & & \qquad dx_1/dt = 1 \wedge \\ & & \qquad dx_2/dt = 1 \wedge \\ & & \qquad \theta.start = \theta_M \wedge x_1.start \geq T)) \\ & \wedge & (state.start = 2 \Rightarrow (state.t = 2 \wedge \\ & & \qquad d\theta/dt = -v_2 \wedge \\ & & \qquad dx_1/dt = 1 \wedge \end{array} \quad (5.4)$$

$$
\begin{aligned}
&& dx_2/dt = 1 \wedge \\
&& \theta.start = \theta_M \wedge x_2.start \geq T)) \\
\wedge && (state.start = 3 \Rightarrow (\theta.start = \theta_M \wedge x_1.start < T \wedge x_2.start < T)))
\end{aligned}
$$

Lemma 3 *The predicate I defined by (5.4) is an invariant of the action system $\overline{\mathcal{TCS}}$, described in Figure 5.3.* ■

The invariant thus shows the basic continuous behavior in each state, as well as the discrete transitions. It is easy to check that $\overline{\mathcal{TCS}}$ has the properties mentioned in (5.4). By inspecting each guard and action body of $\overline{TCS}$, the fact that I is an invariant follows trivially.

Finding a Stronger Invariant. The invariant I that we have just extracted is not good enough, as condition (5.3) is not satisfied for any $R(\tau_r, \tau_1, \tau_2, T)$ other than false. Thus, we need to strengthen I further.

Adding information on top of the basic behavior, encapsulated in predicate (5.4), leads to a new invariant. We add the property $\theta \leq \theta_M$, which is part of the safety condition, to each state, respectively. Then, we get:

$$
\begin{aligned}
I' \equiv\ & (\forall t \in [start, now) \bullet \\
& \quad (state.start = 0 \Rightarrow (\theta.t \leq \theta_M \wedge \\
& \qquad state.t = 0 \wedge \\
& \qquad d\theta/dt = v_r \wedge \\
& \qquad dx_1/dt = 1 \wedge \\
& \qquad dx_2/dt = 1 \wedge \\
& \qquad \theta.start = \theta_m \wedge (x_1.start = 0 \vee x_2.start = 0))) \\
\wedge & \quad (state.start = 1 \Rightarrow (\theta.t \leq \theta_M \wedge \\
& \qquad state.t = 1 \wedge \\
& \qquad d\theta/dt = -v_1 \wedge \\
& \qquad dx_1/dt = 1 \wedge \\
& \qquad dx_2/dt = 1 \wedge \\
& \qquad \theta.start = \theta_M \wedge x_1.start \geq T)) \\
\wedge & \quad (state.start = 2 \Rightarrow (\theta.t \leq \theta_M \wedge \\
& \qquad state.t = 2 \wedge \\
& \qquad d\theta/dt = -v_2 \wedge \\
& \qquad dx_1/dt = 1 \wedge \\
& \qquad dx_2/dt = 1 \wedge \\
& \qquad \theta.start = \theta_M \wedge x_2.start \geq T)) \\
\wedge & \quad (state.start = 3 \Rightarrow \theta.start = \theta_M \wedge x_1.start < T \wedge x_2.start < T))
\end{aligned}
$$

Lemma 4 *Predicate I' is an invariant of $\overline{TCS}$.*

Proof. Let us show that:

$$\begin{array}{lll} I_\theta & \triangleq & (\forall t \in [start, now) \bullet \\ & & \quad (state.start = 0 \Rightarrow (\theta.t \le \theta_M \wedge state.t = 0)) \\ & \wedge & \quad (state.start = 1 \Rightarrow (\theta.t \le \theta_M \wedge state.t = 1)) \\ & \wedge & \quad (state.start = 2 \Rightarrow (\theta.t \le \theta_M \wedge state.t = 2))) \end{array} \qquad (5.5)$$

is a property of the temperature control system.

We apply standard forward analysis (of computing strongest postconditions as shown above) on the translated model of the TCS. Thus, we have to prove that I_θ, given by (5.5), is established by the initialization statement, and that it is also preserved by each action. We show here the proofs for the initialization statement and for action 1 (cooling with rod1). The calculation of gg needed in the proof is also outlined. We assume that $v_1, v_2, v_r \in \mathsf{Real}_+ - \{0\}, \theta_m \ge 0, \theta_M \ge 0$, and $\theta_0 \le \theta_M$. We also assume that the choice of the rod to use as coolant is demonically nondeterministic, in case both rods are available.

(5.5 a) **Initialization.** We have to prove that $\mathsf{true}\ \{\!| S_0\,;UT |\!\}\ I_\theta$ holds, where S_0 is the initialization statement of $\overline{TCS}$. Because $S_0\,;UT$ terminates, we will actually prove that $\{\mathsf{true}\}\ S_0\,;UT\ \{I_\theta\}$ holds.

The initialization statement establishes the following strongest postcondition, $sp.(S_0\,;UT).true$:

$$\begin{array}{ll} & now = 0 \\ \wedge & state = (\lambda t \cdot 0) \\ \wedge & c = (\lambda t \cdot t) \\ \wedge & x_1 = (\lambda t \cdot T_1 + c.t) \\ \wedge & x_2 = (\lambda t \cdot T_2 + c.t) \\ \wedge & \theta = (\lambda t \cdot \theta_0 + v_r * t) \\ \wedge & start = now \\ \wedge & now' = min\{t' \ge now \mid gg.t'\} \end{array}$$

We assume that $sp.(S_0\,;UT).true$ holds. Next, we need to make sure that the partial invariant I_θ is satisfied after the initial assignments. Thus, we have that

$$\begin{array}{ll} & I_\theta[start := 0, now := now', state :- (\lambda t \cdot 0), \theta :- (\lambda t \cdot \theta_0 + v_r * t)] \\ \equiv & \{\text{ substitute updated variables in the invariant}\} \\ & (\forall t \in [0, now') \bullet \\ & \qquad (\lambda t \cdot 0).0 = 0 \Rightarrow (\lambda t \cdot \theta_0 + v_r * t).t \le \theta_M \wedge (\lambda t \cdot 0).t = 0 \\ & \qquad \wedge (\lambda t \cdot 0).0 = 1 \Rightarrow (\lambda t \cdot \theta_0 + v_r * t).t \le \theta_M \wedge (\lambda t \cdot 0).t = 1 \\ & \qquad \wedge (\lambda t \cdot 0).0 = 2 \Rightarrow (\lambda t \cdot \theta_0 + v_r * t).t \le \theta_M \wedge (\lambda t \cdot 0).t = 2) \end{array}$$

$$\equiv \quad \{\ \lambda \text{ reduction, logic } \}$$
$$(\forall t \in [0, now') \bullet \theta_0 + v_r * t \leq \theta_M)$$
$$\equiv \quad \{now' = min\{t' \geq 0 \mid \theta_0 + v_r * t' = \theta_M\}, \text{assumption } \theta_0 \leq \theta_M\}$$
$$(\forall t, 0 \leq t < (\theta_M - \theta_0)/v_r \bullet v_r * t \leq (\theta_M - \theta_0))$$
$$\equiv \quad \{\ \text{logic}\ \}$$
$$\mathsf{true}$$

In the outlined proof, we had to evaluate the disjunction of the guards, at time t', to be able to depict the exact now'. To make the calculation explicit, let us first see what the expression $gg.t'$ actually translates into:

$$\begin{aligned} gg.t' \quad \equiv \quad & (\lambda t \cdot state.t = 0 \wedge \theta.t = \theta_M \wedge x_1.t \geq T).t' \vee \\ & (\lambda t \cdot state.t = 1 \wedge \theta.t = \theta_m).t' \vee \\ & (\lambda t \cdot state.t = 0 \wedge \theta.t = \theta_M \wedge x_2.t \geq T).t' \vee \\ & (\lambda t \cdot state.t = 2 \wedge \theta.t = \theta_m).t' \vee \\ & (\lambda t \cdot state.t = 0 \wedge \theta.t = \theta_M \wedge x_1.t < T \wedge x_2.t < T).t' \end{aligned}$$

Returning to our particular case, we then have:

$$gg.t'[\theta :- (\lambda t \cdot \theta_0 + v_r * t)]$$
$$\equiv$$
$$gg[\theta :- (\lambda t \cdot \theta_0 + v_r * t)].t'$$
$$\begin{aligned} \equiv \quad & \{(state.t' = 0 \wedge (\lambda t \cdot \theta_0 + v_r * t).t' = \theta_M \wedge x_1.t' \geq T) \vee \\ & (state.t' = 1 \wedge (\lambda t \cdot \theta_0 + v_r * t).t' = \theta_m) \vee \\ & (state.t' = 0 \wedge (\lambda t \cdot \theta_0 + v_r * t).t' = \theta_M \wedge x_2.t' \geq T) \vee \\ & (state.t' = 2 \wedge (\lambda t \cdot \theta_0 + v_r * t).t' = \theta_m) \vee \\ & (state.t' = 0 \wedge (\lambda t \cdot \theta_0 + v_r * t).t' = \theta_M \wedge x_1.t' < T \wedge x_2.t' < T)\} \end{aligned}$$
$$gg[\theta :- (\lambda t \cdot \theta_0 + v_r * t), state :- (\lambda t \cdot 0)].t'$$
$$\equiv \quad \{\lambda\text{-reduction}\}$$
$$\begin{aligned} & (0 = 0 \wedge \theta_0 + v_r * t' = \theta_M \wedge x_1.t' \geq T) \vee \\ & (0 = 1 \wedge \theta_0 + v_r * t' = \theta_m) \vee \\ & (0 = 0 \wedge \theta_0 + v_r * t' = \theta_M \wedge x_2.t' \geq T) \vee \\ & (0 = 2 \wedge \theta_0 + v_r * t' = \theta_m) \vee \\ & (0 = 0 \wedge \theta_0 + v_r * t' = \theta_M \wedge x_1.t' < T \wedge x_2.t' < T) \end{aligned}$$
$$\equiv \quad \{\text{logic}\}$$
$$(0 = 0 \wedge \theta_0 + v_r * t' = \theta_M)$$
$$\equiv \quad \{\text{logic}\}$$
$$\theta_0 + v_r * t' = \theta_M$$

Thus, I_θ holds after the initialization, which means that it holds from moment 0 until the next moment when $\theta = \theta_M$. In the following, we compute the verification condition for the first action (cooling with rod1) and show that it preserves the invariant.

(5.5 b) **Cooling with rod1.** We assume that I_θ holds on $[start, now)$, that g_1 is true and that the local variables have been updated by the assignments of the body of action 1. Thus, we affirm the strongest postcondition $sp.(action1).I_\theta$:

$$
\begin{array}{ll}
& (\exists\, start, now, state, \theta, c \cdot (\forall\, t \in [start, now) \cdot I_\theta)) \\
\wedge & state.now = 0 \wedge \theta.now = \theta_M \wedge x_1.now \geq T \\
\wedge & c' = (\lambda t \cdot t - now) \\
\wedge & \theta' = (\lambda t \cdot \theta_M - v_1 * c.t) \\
\wedge & state' = (\lambda t \cdot 1) \\
\wedge & start' = now \\
\wedge & now' = min\{t' \geq now \mid gg.t'\}
\end{array}
$$

We now check whether the added information I_θ is true after the assignments of action 1. Hence, we have that

$$
\begin{array}{ll}
& I_\theta[start := start', now := now', c :- c', \theta :- \theta', state :- state'] \\
\equiv & \{ \text{ definition of } I'_\theta \} \\
& (\forall t, start' \leq t < now' \bullet \\
& \quad state'.start' = 0 \Rightarrow (\theta'.t \leq \theta_M \wedge state'.t = 0) \wedge \\
& \quad state'.start' = 1 \Rightarrow (\theta'.t \leq \theta_M \wedge state'.t = 1) \wedge \\
& \quad state'.start' = 2 \Rightarrow (\theta'.t \leq \theta_M \wedge state'.t = 2) \\
\equiv & \{ \text{ replace updated variables } state', \theta', \lambda\text{-reduction,} \\
& \quad \text{compute } now' = now + \tau_1 \ \} \\
& (\forall t, now \leq t < (now + \tau_1) \bullet \\
& \quad 1 = 0 \Rightarrow (\theta_M - v_1 * (t - now) \leq \theta_M \wedge 1 = 0) \wedge \\
& \quad 1 = 1 \Rightarrow (\theta_M - v_1 * (t - now) \leq \theta_M \wedge 1 = 1) \wedge \\
& \quad 1 = 2 \Rightarrow (\theta_M - v_1 * (t - now) \leq \theta_M \wedge 1 = 2) \\
\equiv & \{ \text{ logic } \} \\
& (\forall t, now \leq t < (now + \tau_1) \bullet \theta_M - v_1 * (t - now) \leq \theta_M) \\
\equiv & \{\theta = \theta_M - v_1 * (t - now) \text{ is decreasing starting from } \theta_M, v_1 > 0, \\
& \quad (t - now) \geq 0\} \\
& \mathsf{true}
\end{array}
$$

The calculation of $gg.t'$, in this case, is as follows.

$$
\begin{aligned}
& gg[state := (\lambda t \cdot 1)].t' \\
\equiv\ & \{((\lambda t \cdot 1).t' = 0 \wedge \theta.t' = \theta_M \wedge x_1.t' \geq T) \vee \\
& ((\lambda t \cdot 1).t' = 0 \wedge \theta.t' = \theta_m) \vee \\
& ((\lambda t \cdot 1).t' = 0 \wedge \theta.t' = \theta_M \wedge x_2.t' \geq T) \vee \\
& ((\lambda t \cdot 1).t' = 0 \wedge \theta.t' = \theta_m) \vee \\
& ((\lambda t \cdot 1).t' = 0 \wedge \theta.t' = \theta_M \wedge x_1.t' < T \wedge x_2.t' < T)\} \\
\equiv\ & \{\lambda\text{-reduction}\} \\
& (1 = 0 \wedge \theta.t' = \theta_M \wedge x_1.t' \geq T) \vee \\
& (1 = 1 \wedge \theta.t' = \theta_m) \vee \\
& (1 = 0 \wedge \theta.t' = \theta_M \wedge x_2.t' \geq T) \vee \\
& (1 = 2 \wedge \theta.t' = \theta_m) \vee \\
& (1 = 0 \wedge \theta.t' = \theta_M \wedge x_1.t' < T \wedge x_2.t' < T) \\
\equiv\ & \{\text{logic}\} \\
& \theta.t' = \theta_m
\end{aligned}
$$

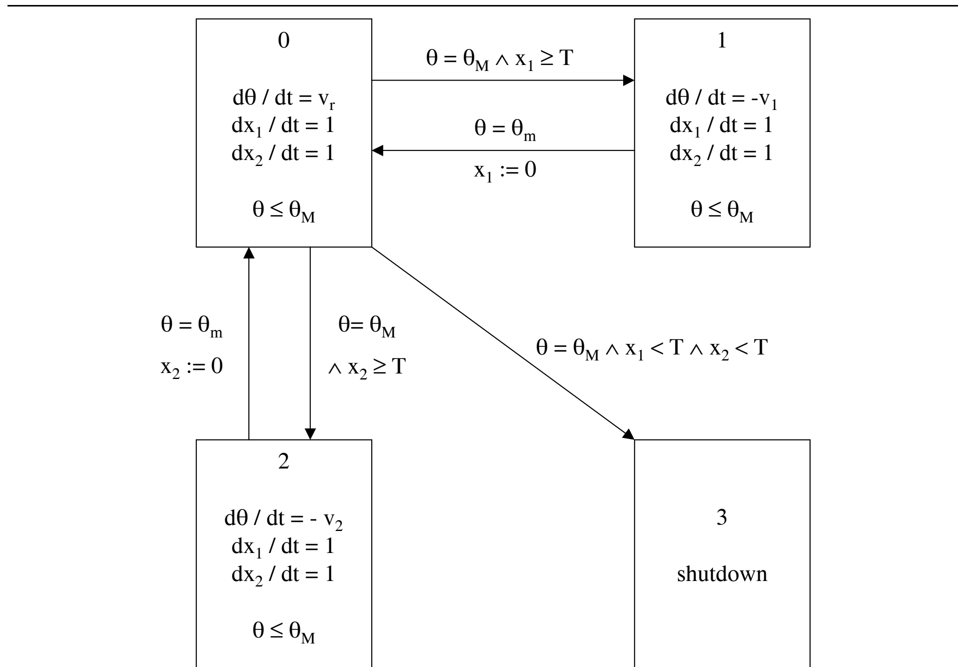


Figure 5.15: TCS state transition diagram with added property, $\theta \leq \theta_M$.

Given the above, we have proved that action 1 preserves I_θ. Consequently,

now, $I' \equiv I \wedge I_\theta$ is preserved after transition $state0 \rightarrow state1$. The proofs for the other possible safe transitions are similar, and are omitted here. The updated state transition diagram is shown in Fig. 5.15. ■

Constructing the Final Invariant and Synthesizing the Constraints on Parameters. Our final goal is to provide sufficient assurance that the system evolves on the safe side. Recall that the safety property reduces to proving first that in any state, $\theta \leq \theta_M$. Moreover, whenever the system is in $state0$ and $\theta = \theta_M$, there should always be (at least) one rod available for cooling:

$$state = 0 \wedge \theta = \theta_M \Rightarrow (x_1 \geq T \vee x_2 \geq T)$$

The latest invariant I' is still too weak to fulfill requirement (5.3), for any other R except $R \equiv$ false. Thus, we keep on adding information and strengthening the invariant. Clearly, the information that is missing addresses clocks x_1 and x_2, which measure the elapsed time since the latest use of rod1 and rod2, respectively.

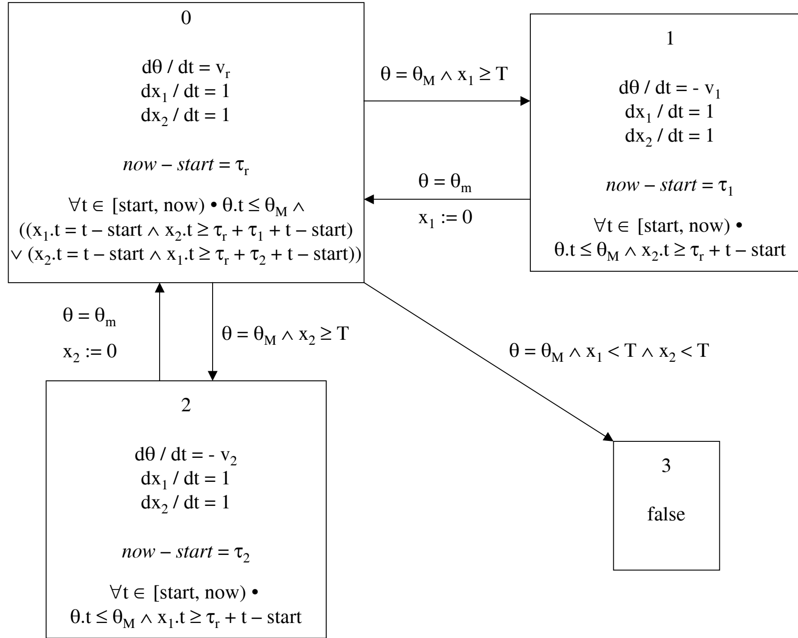


Figure 5.16: TCS statechart with properties regarding x_1, x_2

Besides properties of x_1 and x_2, we also add corresponding information about the time interval between $start$ and now, in each state. For example, considering $state0$, one knows that any transition from this state to any other reachable state is triggered by the equality $\theta = \theta_M$. The necessary time for θ to increase to θ_M is τ_r, therefore it is obvious that $(now - start = \tau_r)$ is a property of $state0$. We carry

out similar judgements for $state1$ and $state2$, and we add this new information to the latest system diagram. As a result, we get the diagram of Fig.5.16.

We denote the new property by I_x:

$$
\begin{aligned}
& I_x \qquad (5.6)\\
\equiv\ & (\forall t \in [start, now) \bullet \\
& \quad (state.start = 0 \Rightarrow \\
& \qquad (((x_1.t = t - start \wedge\ x_2.t \geq \tau_r + \tau_1 + t - start) \\
& \qquad \vee(x_2.t = t - start \wedge\ x_1.t \geq \tau_r + \tau_2 + t - start)) \\
& \qquad \wedge(now - start = \tau_r))) \\
& \wedge (state.start = 1 \Rightarrow \\
& \qquad ((x_2.t \geq \tau_r + t - start) \wedge\ (now - start = \tau_1))) \\
& \wedge (state.start = 2 \Rightarrow \\
& \qquad ((x_1.t \geq \tau_r + t - start) \wedge\ (now - start = \tau_2))) \\
& \wedge (state.start = 3 \Rightarrow \mathsf{false}))
\end{aligned}
$$

Then, we can claim the following lemma.

Lemma 5 *The predicate $I_f \equiv I' \wedge I_x$ is an invariant of $\overline{\mathcal{TCS}}$.*

Proof. For brevity, we are going to prove that condition (5.6) is preserved after releasing rod1, that is, after transition $state1 \rightarrow state0$, when x_1 is reset. The proofs for the initialization statement, and for cooling with rod1 or rod2 are simpler, thus we omit them. However, one needs to choose the values of T_1 and T_2, so that I_x holds right from the beginning. This means that we can have either ($T_1 = \tau_r + \tau_2$ and $T_2 = 0$), or ($T_2 = \tau_r + \tau_1$ and $T_1 = 0$). Even without this choice, the invariant will hold after both rods have been used once.

We assume the following strongest postcondition:

$$
\begin{aligned}
& (\exists\ start, now, state, \theta, c, x_1, x_2 \cdot (\forall\ t \in [start, now) \cdot I' \wedge I_x)) \\
\wedge\quad & state.now = 1 \wedge \theta.now = \theta_m \\
\wedge\quad & c' = (\lambda t \cdot t - now) \\
\wedge\quad & x_1' = (\lambda t \cdot t - now) \\
\wedge\quad & \theta' = (\lambda t \cdot \theta_m + v_r * c.t) \\
\wedge\quad & state' = (\lambda t \cdot 0) \\
\wedge\quad & start' = now \\
\wedge\quad & now' = min\{t' \geq now \mid gg.t'\}
\end{aligned}
$$

After the respective updates, we have that

$$
\begin{aligned}
& I_x[start := start', now := now', state :- state', x_1 :- x_1'] \\
\equiv\ & (\forall t \in [start', now') \bullet \\
& \quad state'.start' = 0 \Rightarrow \\
& \qquad ((x_1'.t = t - start' \wedge x_2.t \geq \tau_r + \tau_1 + t - start') \\
& \qquad \vee (x_2.t = t - start' \wedge x_1'.t \geq \tau_r + \tau_2 + t - start')) \\
& \quad \wedge\ state'.start' = 1 \Rightarrow (x_2.t \geq \tau_r + t - start') \\
& \quad \wedge\ state'.start' = 2 \Rightarrow (x_1.t \geq \tau_r + t - start') \\
& \quad \wedge\ state'.start' = 3 \Rightarrow \mathsf{false}) \\
\equiv\ & \{start' = now,\ state'.now = 0, \text{logic}\} \\
& (\forall t \in [now, now') \bullet \\
& \qquad ((x_1'.t = t - now \wedge x_2.t \geq \tau_r + \tau_1 + t - now) \\
& \qquad \vee (x_2.t = t - now \wedge x_1'.t \geq \tau_r + \tau_2 + t - now)) \\
\equiv\ & \{\text{substituting } x_1'.t = t - now, \text{logic}\} \\
& (\forall t \in [now, now') \bullet x_2.t \geq \tau_r + \tau_1 + t - now) \\
\equiv\ & \{state.start = 1, x_2.t \geq (\tau_r + t - start) \text{ in } [start, now), \\
& \quad \text{thus } x_2.now \geq (\tau_r + now - start) \Rightarrow x_2.now \geq \tau_r + \tau_1, \\
& \quad dx_2/dt = 1 \text{ in } [now, now')\} \\
& \mathsf{true}
\end{aligned}
$$

Above,

$$
\begin{aligned}
& gg[state :- (\lambda t \cdot 0)].t' \\
\equiv\ & \{\text{substitution}\} \\
& (0 = 0 \wedge \theta.t' = \theta_M \wedge x_1.t' \geq T) \vee \\
& (0 = 1 \wedge \theta.t' = \theta_m) \vee \\
& (0 = 0 \wedge \theta.t' = \theta_M \wedge x_2.t' \geq T) \vee \\
& (0 = 2 \wedge \theta.t' = \theta_m) \vee \\
& (0 = 0 \wedge \theta.t' = \theta_M \wedge x_1.t' < T \wedge x_2.t' < T) \\
\equiv\ & \{\text{logic}\} \\
& \theta.t' = \theta_M
\end{aligned}
$$

Therefore, we have proved our claim.

Action 4 (release rod2) is symmetric to action 2, hence, following the same line of proof, the invariant also holds after transition $state2 \rightarrow state0$. ■

We are left with showing that I_f is sufficient to satisfy (5.3), that is

$$(I_f \Rightarrow \neg g_5) \Leftarrow R(\tau_r, \tau_1, \tau_2, T) \tag{5.7}$$

Condition (5.7) reduces to:

$$
\begin{array}{lll}
 & & (\forall t \in [start, now) \bullet \\
 & & \quad state.t = 0 \Rightarrow (((x_1.t = t - start \wedge x_2.t \geq \tau_r + \tau_1 + t - start) \\
 & & \qquad \vee (x_2.t = t - start \wedge x_1.t \geq \tau_r + \tau_2 + t - start)) \\
 & & \qquad \wedge\, now - start = \tau_r)) \\
 & \Rightarrow & \neg(state.now = 0 \wedge \theta.now = \theta_M \wedge x_1.now < T \wedge x_2.now < T) \\
\equiv & & \{now - start = \tau_r \text{ in } state0\} \\
 & & (x_1.now = \tau_r \wedge x_2.now \geq 2\tau_r + \tau_1) \vee \\
 & & (x_2.now = \tau_r \wedge x_1.now \geq 2\tau_r + \tau_2) \\
 & \Rightarrow & (x_1.now \geq T \vee x_2.now \geq T) \\
\Leftarrow & & \{\text{logic}\} \\
 & & (x_2.now \geq 2\tau_r + \tau_1 \vee x_1.now \geq 2\tau_r + \tau_2) \\
 & \Rightarrow & (x_1.now \geq T \vee x_2.now \geq T) \\
\equiv & & \{\text{logic}\} \\
 & & 2\tau_r + \tau_1 \geq T \wedge 2\tau_r + \tau_2 \geq T
\end{array}
$$

This result implies further that, if parameters v_1, v_2, v_r, and T are chosen to satisfy the synthesized relation, the undesired shutdown state is not reachable. ■

5.3 Summary and Related Work

In this chapter we have, first of all, presented a simulation tool for hybrid systems modeled as CAS. We have built the tool using Mathematica, a commercial program [156]. The tool takes a description of any CAS as input, and provides automatically a symbolic simulation of the system, up to a given maximum time. The restrictions on the simulation are essentially those of Mathematica. Nevertheless, more efficient algorithms for evaluating action systems guards need to be implemented.

Symbolic manipulation is an efficient way of simulating a model execution. Plotting the discrete and also continuous model variables as functions of time, with infinite precision, makes the simulation available even without knowing the sampling period to be used for the actual implementation. Thus, in many cases our tool eliminates the need for introducing tolerances in the model. This is true especially when the physical phenomena of the hybrid system are described by linear differential equations. In case the hybrid model is nonlinear, Mathematica could solve the respective nonlinear differential equations either symbolically or numerically. It then follows that, in case we get a numerical solution, we need to introduce tolerances in our action system model and rely on an approximation of the behavior of the variables.

We have applied our simulation technique on two case studies: the temperature control system of a nuclear reactor core, which uses two independent rods for cooling, and a two-tank control system. In the first example, given a certain set of parameters, the objective of the simulation has been to make sure that the reactor never reaches a critical temperature without at least one of the cooling rods being available, to avoid a shutdown of the reactor. The simulation results helped in correlating the model with the actual system behavior.

One of the main advantages of using CAS for modeling hybrid systems is that we now have both a solid proof technique for proving properties of the systems, as well as a powerful simulation technique that we can use to analyze and explore the systems. Simulation can either be used as a precursor to more comprehensive proofs, to iron out bugs in the model, or as an alternative to a complete correctness proof.

Still in this chapter, we have showed how the usual deductive technique of proving inductive invariants is applied on action systems with explicit time. We have done this in order to synthesize sufficient parametric conditions that guarantee a correct operation of the analyzed hybrid system. The method relies on strengthening an initial invariant, by adding information regarding system variables. We stop when the invariant is strong enough to make the undesired set of states not reachable. We show that this happens if a certain parametric relationship holds. The relationship follows from proving that the invariant implies the negation of the abort condition. Nevertheless, there is no guarantee that the successive strengthening steps will ever achieve this goal, nor is there any guidance as to which predicates to use for strengthening. Even so, the fact that the method seems to work on complex hybrid system models might lure one into employing an additional tool that would facilitate finding suitable predicates. Details on this issue are provided in the next paragraph.

We have applied the above method for temperature control, giving a formal proof for a safety property.

Related Work. Many simulation packages have been proposed and applied for the systematic analysis of hybrid systems [74, 79, 124, 125]. Comprehensive overviews and comparisons of some of the most popular simulation tools for hybrid systems, such as DYMOLA, SHIFT, SIMULINK/STATEFLOW, GPROMS, BASIP etc. are given by Kowalewski et al. [109], and Mosterman [132].

Reachability analysis for hybrid systems is among the most important and difficult problems. Alur, Henzinger et al. propose algorithmic analysis of such systems, modeled as hybrid automata [7, 14]. Their techniques are based on constructing the reachable region of linear hybrid models. The authors also provide decidability and undecidability results for classes of linear hybrid systems (see also [108]). For general hybrid systems, the algorithmic analysis can be applied with certain limitations.

Alur et al. [7] use symbolic model-checking techniques, for reachability anal-

ysis of timed automata (see also [92]). The construction techniques are illustrated on the temperature control system that we have also analyzed. The tool KRONOS [67] is used to automate the computation of the characteristic set of state predicates, under particular values of the parameters.

In comparison with the cited approaches, we give a general mathematical proof to parametric reachability problem. The method is based on traditional strongest postcondition computation, and it is applicable even in those cases where relationships between parameters can not be guessed. Parametric verification can be handled by model-checking tools like UPPAAL [117], and parameter synthesis can be automatically carried out by, for example, HYTECH [87, 88] or TREX [48]. UPPAAL can not be used to synthesize constraints on parameters, one should guess the respective parametric relationship instead, and instantiate it for verification. On the other hand, model-checkers HYTECH and TREX are able to perform parameter synthesis, in some cases. For example, as stated by Henzinger et al. [93], systems with complex relationships between multiple parameters and timing constants can quickly lead to arithmetic overflow, when analyzed with HYTECH. In contrast, analysis with a single parameter is often successful.

Whenever reachability construction fails, the reachability verification method can be applied [94]. First, the user has to guess (heuristically) the reachable region, and then verify that the guess is correct. The method is almost fully automated (there are no automated guess heuristics), but in case the guessed region is not directly inductive, new variables and constraints have to be added.

In principle, dedicated model-checking techniques [7] provide only an assertion that the hybrid system model satisfies a safety property. Our approach, like other deductive approaches to reachability verification, relies on proving an invariance property. The method offers useful key insights on the system behavior. In practice, this may be important if one wants to improve the functionality of the system, at later stages. Adding information to system states might ease the process of refinement. A major drawback is the task of finding appropriate invariants. Tools for guided invariant generation, like the one within SAL [43] can help the designer overcome this shortcoming. In SAL, the underlying technique of invariant construction is based on a combination of least and greatest fixed-point computation of reachable states [150].

The verification methodology based on *abstracted* automata developed by Puri and Varaiya [138] faces the inconvenience that the created abstractions depend on the property to be proved. Different properties may require different abstractions of the same hybrid system.

Recently, important progress has been done in the emergent area of reachability analysis of hybrid systems via *predicate abstraction* [8, 43]. The technique is based on abstracting the infinite state-space of a hybrid system into a finite representation, by identifying each state of the abstract state space with a truth assignment to the abstraction predicates [80]. Then, if a temporal logic formula holds for the abstracted system, it also holds for the original one.

Chapter 6

Building Uniprocessor Priority-driven Real-Time Schedulers

As proved in chapter 3, in a concurrent system, it is not necessary to specify the exact order in which processes execute. The general behavior of the program exhibits significant nondeterminism. If the program is correct then its functional outputs will be the same regardless of internal behavior or implementation details.

While the program's outputs will be identical to all the possible interleavings, the timing behavior will vary considerably [52]. If one of the processes has a strict deadline, then perhaps only interleavings in which that process is executed first will meet the program's temporal requirements. A *real-time system* needs to restrict the nondeterminism found within concurrent systems [52]. This process is known as *scheduling*.

A *real-time scheduler* can be seen as a controller of sets of real-time tasks. It establishes the order in which tasks are dispatched, decides the starting time of execution, for each individual task, and it also regulates the task's access to shared resources.

Task priorities are assigned with respect to predefined algorithms. However, their programming discipline is not well supported by existing development tools [78]. Thus, a mathematically proven correct-by-construction method for building generic real-time schedulers could be beneficial. We introduce, in this chapter, such a precise method [58]. Nevertheless, this requires work at a high level of abstraction, that is, without consideration of functionality.

Considering n real-time tasks, $\mathcal{T}(1), \mathcal{T}(2), \ldots, \mathcal{T}(n)$ that we assume schedulable, we want to derive a scheduler that controls the tasks such that all of their executions complete by the respective deadlines. We call this condition the *timeliness condition* and we model it as predicate q_t. Given the fact that the scheduler gives priorities to tasks according to some scheduling algorithm, we denote this

policy condition as predicate q_{pol}. We should also require from the scheduler to grant the CPU to no more than one task at a time. This is the *mutually exclusive execution condition*, q_{me}.

In short, we start with a conjunctive specification of the collection of tasks. This model asserts the specific initial states that the system should start from. Further, we *enforce* the above mentioned predicates that guarantee a correctly scheduled system, by applying the refinement rules of assertion propagation and dropping an assertion (see chapter 2). Consequently, the resulting system representation is free of assertions, which is indeed what we aim for.

The correctness conditions that should be enforced by the scheduler are modeled as *always* properties (the scheduler should respect $\Box\,(q_t \wedge q_{me} \wedge q_{pol}))$. To be able to construct a correct scheduler, we generalize to CAS the expected result proved by Back and von Wright [38]: enforcement of *always* properties reduces to invariance proofs on action systems.

The method leads, eventually, to an implementation of the real-time system model. We exemplify the proposed construction strategy on the *Deadline-Monotonic* (DM) scheduling policy, which is described in the following section. We consider tasks to be independent; we also disregard any other resource allocation than the CPU.

Employing a similar technique, we also tackle the development of *Earliest - Deadline - First* (EDF) scheduling programs, for periodic tasks (the mechanism underlying the EDF algorithm is also discussed in section 6.1). The construction is completed by a simulation-based validation method, where the constructed model is simulated up to the least-common-multiple of the periods of the participating tasks. The simulation is carried out with our CAS symbolic simulator introduced in chapter 5.

6.1 Uniprocessor Scheduling of Real-Time Tasks

In this section, we review the definitions and results of the real-time scheduling theory, which are essential to understanding the rest of the chapter.

Real-time systems are systems whose correctness depends on both the accuracy of the output result, as well as on the time at which the latter is delivered.

A real-time task $\mathcal{T}(i)$ is in general characterized by the following attributes:

- *minimum inter-arrival time*, $P[i]$,
- *worst-case execution time*, $E[i]$,
- *deadline*, $D[i]$, and
- *priority*, $pr[i]$.

We restrict our analysis to tasks with $D[i] \leq P[i]$.

The temporal requirements of a *hard* real-time system imply that *all* the participating tasks complete the execution by their deadlines. *Soft* real-time programs

accept arbitrary omissions in meeting deadlines, at run-time, yet within some predefined tolerances with respect to completion times. Moreover, the system performance degrades directly proportional to tardiness.

Tasks may be *periodic*, meaning that they arrive at fixed intervals equal to the periods $P[i]$, respectively; they can also be *sporadic*, that is, the tasks arrive irregularly, yet no sooner than $P[i]$ (the minimum inter-arrival time). The timing behavior of a sporadic task is exemplified in Figure 6.1 [20].

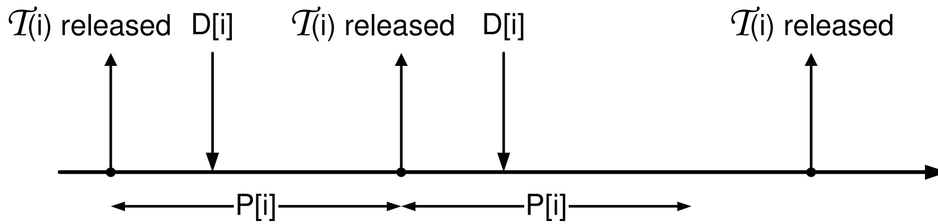


Figure 6.1: The execution of a sporadic task.

With priority-based scheduling, a high-priority process may be released during the execution of a lower priority one. There are cases when tasks run to completion without interruption, no matter if a higher-priority task is waiting while the current task is executing; such tasks are called *non-preemptible*. When a task can be interrupted from running, at any time, by a higher priority task, we say that the respective task is *preemptible*.

A *fixed-priority* scheduling policy relies on assigning priorities to tasks, offline. These priorities do not change during system execution. One of the most popular such algorithm is the *Deadline-Monotonic* (DM) scheme, introduced by Leung and Whitehead [118].

Deadline-Monotonic. Let us consider that n *preemptible*, (hard) real-time tasks, $\mathcal{TS} = \{\mathcal{T}(1), .., \mathcal{T}(n)\}$ execute on a single CPU. Under the mentioned assumption, $D[i] \leq P[i], \forall i \in [1..n]$, an optimal set of priorities can be obtained such that $(D[i] < D[j] \vee (D[i] = D[j] \wedge i < j)) \Rightarrow pr[i] > pr[j]$, for all tasks $\mathcal{T}(i), \mathcal{T}(j)$. This means that the priorities of tasks are in the reverse order from their deadlines. If the CPU is free, the highest priority process among the waiting processes is scheduled.

We say that a task is *feasible* (or *schedulable*) if any of its instances finishes execution before or at most at its deadline. Joseph and Pandaya [103] proposed a method to determine the *feasibility* (or *schedulability*) of a task by computing its worst-case *completion time* (response time), according to the equation:

$$R[i] \quad = \quad E[i] + B[i] + \sum_{j \in \mathbf{hp}(i)} \lceil R[i]/P[j] \rceil * E[j] \qquad (6.1)$$

Above, $\mathbf{hp}(i)$ is the set of higher priority tasks than $\mathcal{T}(i)$, and $B(i)$ is the maximum blocking time caused by a concurrency control protocol protecting shared data. The

ceiling value $\lceil R[i]/P[j] \rceil$ is the smallest integer greater or equal to $R[i]/P[j]$. The last term ($\sum_{j \in \mathbf{hp}(i)} \lceil R[i]/P[j] \rceil * E[j]$) of equation (6.1) measures the interference of higher priority tasks with the execution of task $\mathcal{T}(i)$. The interference consists of the computation time of all higher priority tasks that are released before $D[i]$ [20]. The (smallest) solution of equation (6.1) can be obtained by computing the sequence $r[i]^n, n \in \mathsf{Nat}$ defined by the recurrence relation below:

$$r[i]^{n+1} = E[i] + B[i] + \sum_{j \in \mathbf{hp}(i)} \lceil r[i]^n / P[j] \rceil * E[j] \tag{6.2}$$

Above, $r[i]^0$ is given an initial value of 0. The computation stops when a fixed point is reached, that is, $r[i]^{n+1} = r[i]^n$. This value is the worst-case response time $R[i]$. A task $\mathcal{T}(i)$ is feasible if $R[i] \leq D[i]$. If all the participating tasks are feasible then the entire task set is declared *schedulable* by the DM algorithm.

Task priorities can also be decided online, by employing a *dynamic priority* scheduling policy.

Earliest-Deadline First. For *dynamic* priority schedulers, the priority of a task is recomputed at run-time. If scheduled by the *Earliest-Deadline-First* (EDF) algorithm, the runnable processes are executed in the order determined by the *absolute* deadlines of the processes, the next process to run being the one with the shortest (nearest) deadline. Although it is usual to know the relative deadlines of each process (e.g. 10 ms after release), the absolute deadlines are calculated at run-time, hence the scheme is described as dynamic.

Within the context of uniprocessor scheduling, it has been shown by Liu and Layland [119] that EDF, which at each instant of time chooses for execution the currently-active job with the smallest computed deadline, is an *optimal* algorithm. This means that any feasible task system is guaranteed to be successfully scheduled using EDF.

Schedulability analysis is the process of determining whether a collection of tasks can be scheduled in such a manner that all task instances will complete by their deadlines. When assuming $D[i] \leq P[i]$, the EDF schedulability analysis turns out to be somewhat less straightforward than in the fixed-priority case.

6.2 Generic Approach

6.2.1 Enforcing the Required Conditions

To facilitate the formal analysis, we start by specifying the real-time system as an unscheduled collection of tasks. Next, we construct a correct, implementable scheduled system, under the assumption that the tasks are schedulable by some supported algorithm. In the end, we decompose the obtained real-time system, by refinement, into two modules: the CAS that models the set of tasks and the CAS that models the scheduler.

Assume that we have a way of describing the timeliness, policy and mutual exclusion predicates, as q_t, q_{pol}, q_{me}, respectively. We will show later in this chapter their formal definitions. Since all three correctness conditions have to hold for any behavior of the system, it follows intuitively that they can be expressed as a global "*always*" ($\Box$) temporal property:

$$\Box\, q \;\triangleq\; \Box\, (q_t \wedge q_{me} \wedge q_{pol})$$

Given the definition of the "always" property over discrete behaviors [38], we can extend it to *timed behaviors*.

A timed behavior is a sequence of states, where each state is a tuple of the form (x, y), with $x : \mathsf{Real}_+ \rightarrow T_x$ the local variables, and $y : \mathsf{Real}_+ \rightarrow T_y$ the global ones. We can say that, for all *timed behaviors* b, we have:

$$b \models \Box\, q \qquad \text{iff} \qquad (\forall i \cdot b_i \in q),$$

where q is a predicate.

In order to effectively enforce the scheduling predicates on sets of real-time tasks, we apply the invariant-based inference rule for proving enforcement of "always" properties [38], to the timed case. This is shown in Lemma 6.

Lemma 6 *Assume the following action system*:

$$\begin{array}{rl} \overline{\mathcal{RTS}} \triangleq & \textbf{begin var } start, now : \mathsf{Real}_+, x : \mathsf{Real}_+ \rightarrow T \bullet Init\,; UT\,; \\ & \quad \textbf{do } g_1.now \rightarrow S_1\,; UT \,[\!]\, \ldots \,[\!]\, g_n.now \rightarrow S_n\,; UT \textbf{ od } \textbf{ end} \end{array}$$

Then, scheduler correctness properties can be proved using invariants, as follows:

$$\frac{p \Rightarrow I \quad g_1.now \wedge I \;\{\!|S_1;UT|\!\}\; I \ldots g_n.now \wedge I \;\{\!|S_n;UT|\!\}\; I \quad I \Rightarrow q}{p \;\{\!|\; \textbf{do } g_1.now \rightarrow S_1;UT \,[\!]\, \ldots \,[\!]\, g_n.now \rightarrow S_n;UT \textbf{ od } |\!\}\; \Box\, q}$$

where p, q *are predicates that depend on time, and*

$$UT \quad \triangleq \quad start := now\,; now := min\{t' \geq now \;\mid\; gg.t'\}$$

The variable "start" denotes the beginning of the next time interval triggered by a discrete transition.

Proof.

$$\begin{array}{cl} & p\,\{\!|\textbf{do } g_1.now \rightarrow S_1\,; UT \,[\!]\, \ldots \,[\!]\, g_n.now \rightarrow S_n\,; UT \textbf{ od} |\!\}\, \Box\, q \\ \equiv & \{\text{correctness rule [38], rule (2.10)}\} \\ & p \Rightarrow (\nu x \cdot q \wedge (\neg g_1.now \vee S_1.(UT.x)) \wedge \ldots \wedge (\neg g_n.now \vee S_n.(UT.x))) \\ \Leftarrow & \{\text{assumptions: } p \Rightarrow I, I \Rightarrow q\} \\ & I \Rightarrow (\nu x \cdot I \wedge (\neg g_1.now \vee S_1.(UT.x)) \wedge \ldots \wedge (\neg g_n.now \vee S_n.(UT.x))) \\ \Leftarrow & \{\text{greatest fixed point induction rule (2.14)}\} \\ & I \Rightarrow I \wedge (\neg g_1.now \vee S_1.(UT.I)) \wedge \ldots \wedge (\neg g_n.now \vee S_n.(UT.I)) \\ \equiv & \{\text{logic, assumptions } g_i.now \wedge I \Rightarrow S_i.(UT.I), \forall\, i \in [1..n]\} \\ & \mathsf{true} \end{array}$$

■

Even if the result of the above lemma is not at all surprising, it is nevertheless useful. If the correctness assertions required by the rule of Lemma 6 hold, it follows that the required constraints are enforced. Consequently, a feasible schedule can be constructed.

The first step is to find a predicate $I \triangleq I_t \wedge I_{me} \wedge I_{pol}$. Predicate $I_t \Rightarrow q_t$ has to be an invariant in order to guarantee timely completion of tasks executions. In addition, two other types of constraints have to be enforced, for the real-time system to operate correctly. They are the mutual exclusion, and the scheduling policy conditions, that is, $I_{me} \Rightarrow q_{me}$, and $I_{pol} \Rightarrow q_{pol}$, respectively. The first one ensures that no more than one task at a time is granted the CPU, while the second imposes the order in which tasks are given priorities for execution.

Since we aim for a scheduler that preserves the predicate I described above, we model the loop of the system $\overline{\mathcal{RTS}}$ as $\mathbf{do}\ \{Choice.I\}\,; Choice\ \ \mathbf{od}$, where

$$Choice \triangleq g_1.now \rightarrow S_1\,; UT \parallel \ldots \parallel g_n.now \rightarrow S_n\,; UT$$

Here, $\{Choice.I\}$ represents the initial states that we are interested in. The information supplied by the computed precondition $\{Choice.I\}$ is then used to refine the initial model, as shown in section 6.2.2.

Rather than carrying out the schedulability analysis on a complete real-time system model, we assume that the set of tasks is schedulable and proceed to a stepwise, correctness-preserving scheduler development. Then, our goal is to construct a scheduler that schedules the system to meet all deadlines, according to the supported scheduling algorithm. In case tasks are to be scheduled by a fixed-priority policy, we use the existing results of scheduling theory, presented at the beginning of this chapter.

For a dynamic-priority scheme, the priorities are assigned at run-time, thus the schedulability of the task set can not be checked offline, unless we consider only integer points [159]. So, there is no established schedulability condition that we can assume when starting to build the scheduled system. However, it is argued that one approach for validating the schedulability of a task set is the *simulation* of the model, for a sufficiently long time, until the real-time system is in the periodic state [52]. This interval of time is called the *feasibility interval*. We apply this method of validation, by simulating the real-time CAS model that we construct, for the *Earliest-Deadline-First* protocol. To accomplish this goal, we use the CAS symbolic simulation tool, built in Mathematica.

6.2.2 Deriving the Final Scheduler Model by Refinement

As mentioned previously, our method of constructing the real-time system assumes that, initially, we specify the latter as $\mathbf{do}\ \{Choice.I\}\,; Choice\ \ \mathbf{od}$, where $Choice$ is the nondeterministic model of the task set. Next, we apply stepwise refinement, targeting a final representation that is scheduled by the rules of a particular policy.

The generic refinement steps are as follows:

$$
\begin{array}{cl}
& \{Choice.I\} \,;\, (g_1.now \rightarrow S_1 \,;\, UT \;[\!]\; \ldots \;[\!]\; g_n.now \rightarrow S_n \,;\, UT) \\
= & \{\text{rule}(2.19)\} \\
& \{Choice.I\} \,;\, (g_1.now \rightarrow \{g_1.now \wedge Choice.I\} \,;\, S_1 \,;\, UT \;[\!] \\
& \ldots \;[\!]\; g_n.now \rightarrow \{g_n.now \wedge Choice.I\} \,;\, S_n \,;\, UT) \\
\sqsubseteq & \{\text{rewrite in context, strengthen guards}\} \\
& \{Choice.I\} \,;\, (g'_1.now \rightarrow \{g_1.now \wedge Choice.I\} \,;\, S_1 \,;\, UT \;[\!] \\
& \ldots \;[\!]\; g'_n.now \rightarrow \{g_n.now \wedge Choice.I\} \,;\, S_n \,;\, UT) \\
\sqsubseteq & \{\text{rule } (2.20)\} \\
& g'_1.now \rightarrow S_1 \,;\, UT \;[\!]\; \ldots \;[\!]\; g'_n.now \rightarrow S_n \,;\, UT \\
= & \{\text{ notation }\} \\
& Choice_f
\end{array}
\qquad (6.3)
$$

The idea is to remove all assertions, yet ensuring that I holds after dropping them. The result is an implementable model. In order to decrease the level of abstraction, and eventually reach a more efficient implementation, we go on and trace refine the real-time model that contains the loop $\mathbf{do}\; Choice_f\; \mathbf{od}$. These transformations are supported by trace refinement rules for CAS. The rules are introduced later in this chapter.

As a last step, we decompose the refined real-time system, in two modules. We thus get a two-module implementation of the initial model, which consists of the scheduler and the real-time tasks. In principle, the separate representation of the online scheduler gives one the possibility to improve the performance of, or add functionality to the scheduler, without necessarily modifying the task model. Thus, a two-module description increases the flexibility in design, and the reusability of the real-time system components, by decoupling scheduling issues from task behavior. In order to decompose the real-time system model correctly, we apply the prioritizing decomposition theorem, introduced by Sekerinski and Sere [141], and presented in chapter 2.

In short, the development method that we propose combines the *precondition analysis* technique with *program derivation* [35].

6.3 Preemptible Task Model

By definition, a preemptible task is one that can be interrupted from running at some point in time. We model a generic preemptible, sporadic task $\mathcal{T}(i), i \in [1..n]$, as a choice among five guarded actions defined by (6.4).

The model below does not encode any explicit scheduling algorithm, it rather assumes a virtual scheduler. We abstract from the functional behavior, since it is not relevant within this context. Each task can be in one of the four possible states, sl (*sleeping*), wt (*waiting for the CPU*), ex (*executing*), and pt (*preempted*). If no task has been released, the *arrival* clock c_a measures the time from 0 to each task's release; after that, the same clock records the time elapsed between two consecutive arrivals of each task. The vector variable *ofs* denotes the *arrival offset* of each task,

that is, the arbitrary number of time units (including 0) beyond $P[i]$, which might pass before the next occurrence of $\mathcal{T}(i)$. We record the execution time of each task by clock c_e, and the task preemption time, by clock c_p. All three clocks c_a, c_e, and c_p are vectors of length equal to the number of tasks.

$$
\begin{array}{lcl}
\mathcal{T}(i) & \hat{=} & \quad state[i].now = sl \wedge c_a[i].now = P[i] + \mathit{ofs}[i].now \\
 & & \quad \rightarrow c_a[i] :- (\lambda t \cdot t - now) \,;\, state[i] :- (\lambda t \cdot wt) \,;\, UT \\
 & [\!] & \quad state[i].now = wt \\
 & & \quad \rightarrow c_e[i] :- (\lambda t \cdot t - now) \,;\, state[i] :- (\lambda t \cdot ex) \,;\, UT \\
 & [\!] & \quad state[i].now = ex \wedge c_e[i].now = E[i] \\
 & & \quad \rightarrow c_e[i] :- (\lambda t \cdot 0) \,;\, c_p[i] :- (\lambda t \cdot 0); \\
 & & \quad\quad [\, \mathit{ofs}[i] :- x' \mid \forall t \geq now \cdot x'.t \in \mathsf{Real}_+ \,]; \\
 & & \quad\quad state[i] :- (\lambda t \cdot sl) \,;\, UT \\
 & [\!] & \quad state[i].now = ex \wedge c_e[i].now < E[i] \\
 & & \quad \rightarrow c_e[i] :- (\lambda t \cdot c_e[i].now); \\
 & & \quad\quad c_p[i] :- (\lambda t \cdot c_p[i].now + t - now); \\
 & & \quad\quad state[i] :- (\lambda t \cdot pt) \,;\, UT \\
 & [\!] & \quad state[i].now = pt \\
 & & \quad \rightarrow c_e[i] :- (\lambda t \cdot c_e[i].now + t - now); \\
 & & \quad\quad c_p[i] :- (\lambda t \cdot c_p[i].now) \,;\, state[i] :- (\lambda t \cdot ex) \,;\, UT
\end{array}
\tag{6.4}
$$

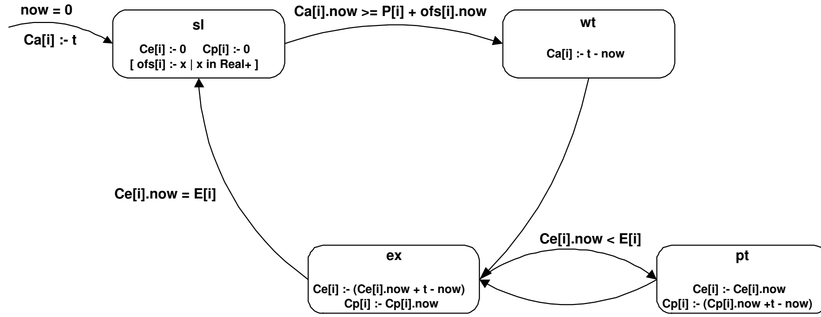


Figure 6.2: Preemptible task behavior as an STD.

Since the tasks are sporadic, their actual arrival times are not periodic; instead, successive arrivals of the same task are separated by no less than $P[i]$ time units, respectively. We model this behavior by requiring the clock $c_a[i]$ to be equal to $P[i] + \mathit{ofs}[i].now$, for the task $\mathcal{T}(i)$ to become available. If we consider the collection of available tasks, and the initial model of a task (6.4), should $\mathcal{T}(i)$ wait for the CPU, it could start executing right away (the guard of the second action holds) or, in case some other tasks are simultaneously waiting, $\mathcal{T}(i)$ could be postponed for an arbitrary time (since the choice of an action out of several enabled ones is nondeterministic). When selected, the task changes its state to ex, and clock $c_e[i]$ is reset. Upon completion of execution, when $c_e[i].t$ is $E[i]$, the respective task returns to state *sl*, the execution and preemption clocks are both set to 0,

and the offset variable is nondeterministically assigned a nonnegative real value. If the task is executing and, implicitly, its permission is removed by the virtual scheduler, the task takes the transition to state *pt*, the execution clock is frozen, $c_e[i] :- (\lambda t \cdot c_e[i].now)$, and $c_p[i]$ starts increasing linearly with time. When the scheduler restores the task's permission to execute, the latter returns to *ex*, and the clock $c_e[i]$ starts evolving again while $c_p[i]$ is frozen. The vector variable $state$ stores the current state of each task. Note that, in (6.4), we assume the worst-case execution time of the task, thus, we check for $c_e[i].now = E[i]$, in order to establish if the task has finished its execution. Observe also that the choice is deterministic, since the guarded actions of task $\mathcal{T}(i)$ are mutually exclusive. The described task behavior is graphically represented as the state-transition diagram (STD) of Figure 6.2.

By employing the quantified nondeterministic choice operator on the participating tasks, we can further describe the real-time task set, as the action system:

$$
\begin{aligned}
\overline{\mathcal{RTS}} \;\; \triangleq \;\; & \textbf{begin var } start, now : \mathsf{Real}_+, \\
& \quad state : \mathsf{array}\ [1..n]\ \textbf{of}\ (\mathsf{Real}_+ \rightarrow \{sl, wt, ex, pt\}), \\
& \quad ofs, c_a, c_e, c_p : \mathsf{array}\ [1..n]\ \textbf{of}\ (\mathsf{Real}_+ \rightarrow \mathsf{Real}_+) \bullet \\
& \quad now := 0 \,;\, state :- (\lambda t \cdot wt) \,;\, c_a :- (\lambda t \cdot t) \,; \\
& \quad [\, ; 1 \le i \le n \cdot [\, ofs[i] :- x' \mid \forall t \cdot x'.t \in \mathsf{Real}_+ \,]\,] \,; \qquad (6.5) \\
& \quad c_e, c_p :- (\lambda t \cdot 0) \,;\, UT \,; \\
& \quad \textbf{do}\ [\, \| \, 1 \le i \le n \cdot \{\mathcal{T}(i).I\} \,;\, \mathcal{T}(i)\,]\ \textbf{od} \\
& \textbf{end}
\end{aligned}
$$

Initially, the tasks are deemed to share a critical instant. That is why, at $now = 0$, they are all waiting for execution. The initial future assignments of clocks c_e, c_p stand for $[\ ; i \cdot c_e[i], c_p[i] :- (\lambda t \cdot 0), (\lambda t \cdot 0)\ ]$. The arrival clock elements $c_a[i]$ start increasing linearly with time, and the offsets $ofs[i]$ are assigned arbitrary nonnegative real values.

6.4 Fixed-Priority Scheduling: The Deadline-Monotonic Algorithm

The generic procedure outlined in the previous sections is applied first to the construction of a real-time system scheduled by the Deadline-Monotonic algorithm.

6.4.1 Enforcing Conditions for Correct Scheduling

Timeliness, Mutual Exclusion, and DM Policy Conditions as Safety Requirements. We work under the following assumptions:

- the tasks are preemptible;
- $R[i] \le D[i]$, where $R[i]$ is the worst-case completion time of each task, and is computed by applying equations (6.1) and (6.2).

When checking schedulability of a set of real-time sporadic tasks, we can approximate each sporadic task with a periodic one, of period equal to $P[i]$ [20].

The second assumption implies that the set of tasks under analysis is schedulable. Nonetheless, there also exists theorem-prover support for verifying schedulability of sets of preemptible sporadic tasks, as the concurrency protocol of *Priority-Ceiling* has been formalized and verified in PVS [73].

According to the method introduced in sections 6.2.1 and 6.2.2, we construct the DM scheduled system by following the steps below.

• Firstly, we define the timeliness property that we want to enforce on the action system (6.5). We express this condition as the following predicate:

$$\begin{aligned} q_t \;\triangleq\;\; & \forall i \cdot \forall t \in [start, now) \cdot \\ & \quad state[i].start = ex \Rightarrow c_a[i].t + R[i] - (c_e[i].t + c_p[i].t) \leq D[i] \end{aligned}$$

In the above relation, we require that, if a task has just started to execute ($state[i].start = ex$), the clock $c_a[i].t$ should ensure that at the end of execution the deadline $D[i]$ is not exceeded, even under the worst-case response-time scenario (that is, time to completion equals $R[i]$). The case when the task has been preempted (at least once) and it returns to execution is also considered, by adding the value of the preemption clock, $c_p[i]$, to the current value of the execution clock $c_e[i]$. Next, we need to find a predicate $I_t \Rightarrow q_t$, and then compute $\mathcal{T}(i).I_t$. We choose:

$$\begin{aligned} I_t \;\equiv\;\; & \forall i \cdot \forall t \in [start, now) \cdot \\ & \quad (state[i].start = ex \Rightarrow \\ & \quad\quad (state[i].t = ex \wedge c_a[i].t - (c_e[i].t + c_p[i].t) \leq D[i] - R[i] \;\wedge \\ & \quad\quad\quad c_a[i].t = c_a[i].start + t - start)) \\ & \wedge (state[i].start = ex \wedge c_e[i].start = 0 \Rightarrow c_p[i].t = 0) \end{aligned}$$

• Secondly, we perform the precondition analysis step. By successive application of rules (2.10), (2.7), (2.4), the computed weakest precondition, $\mathcal{T}(i).I_t$, is as follows.

$$\begin{aligned} & \mathcal{T}(i).I_t \\ \equiv\; & \{\text{succesive application of rules (2.10), (2.7), (2.6), (2.4)}\} \\ & \quad (\forall j \neq i \cdot \mathcal{T}(i).I_t^j) \\ & \wedge (state[i].now = wt \Rightarrow \\ & \quad (\forall now' \cdot now' = min\{t' \geq now \mid c_e[i].t' \leq E[i]\} \Rightarrow \\ & \quad\quad (\forall t \in [now, now') \cdot c_a[i].t - (t - now) \leq D[i] - R[i] \wedge \\ & \quad\quad\quad c_a[i].t = c_a[i].now + t - now))) \\ & \wedge (state[i].now = pt \Rightarrow \\ & \quad (\forall now' \cdot now' = min\{t' \geq now \mid c_e[i].t' \leq E[i]\} \Rightarrow \\ & \quad\quad (\forall t \in [now, now') \cdot \\ & \quad\quad\quad c_a[i].t - (c_e[i].now + (t - now) + c_p[i].now) \leq D[i] - R[i] \wedge \\ & \quad\quad\quad c_a[i].t = c_a[i].now + t - now))) \end{aligned}$$

where

$$
\begin{aligned}
I_t^j \equiv\ & \forall t \in [start, now) \cdot \\
& \quad (state[j].start = ex \Rightarrow \\
& \qquad (state[j].t = ex \wedge c_a[j].t - (c_e[j].t + c_p[j].t) \leq D[j] - R[j] \ \wedge \\
& \qquad\quad c_a[j].t = c_a[j].start + t - start)) \\
& \wedge (state[j].start = ex \wedge c_e[j].start = 0 \Rightarrow c_p[j].t = 0)
\end{aligned}
$$

The detailed computation can be found in Appendix A-5.

• Thirdly, we derive the real-time model that preserves I_t. We do this by propagating the available context information, given as assertions, into the respective guards, as outlined in section 6.2.2. We return to $\overline{\mathcal{RTS}}$ and work on the generic task model, $\mathcal{T}(i)$. Out of refining in context, the nondeterministic behavior of the entire set of tasks is constrained with respect to I_t. This reflects in strengthening the second and fifth action guards of each task described by (6.4), respectively, by imposing upper bounds on clocks $c_a[i]$, for guard two, and $c_a[i] - (c_e[i] + c_p[i])$ for the last guard ($i \in [1..n]$).

Let us denote: $new_now \triangleq now' = min\{t' \geq now \mid c_e[i].t' \leq E[i]\}$. We then have:

$$
\begin{aligned}
& \{\mathcal{T}(i).I_t\} \,;\, \mathcal{T}(i) \\
\sqsubseteq\ & \{\text{substitute task model, rule (2.19), weaken assertion, logic}\} \\
& \{\mathcal{T}(i).I_t\}; \\
& (\ldots \\
& [\!]\ state[i].now = wt \\
& \quad \rightarrow \{\ \forall\, now' \cdot new_now \Rightarrow (\forall t \in [now, now')\cdot \\
& \qquad\qquad c_a[i].t - (t - now) \leq D[i] - R[i] \wedge \\
& \qquad\qquad c_a[i].t = c_a[i].now + (t - now)) \\
& \qquad \}; \\
& \qquad c_e[i] :- (\lambda t \cdot t - now) \,;\, state[i] :- (\lambda t \cdot ex); \\
& \qquad start := now \,;\, now := min\{t' \geq now \mid gg.t'\} \qquad (6.6) \\
& \ldots \\
& [\!]\ state[i].now = pt \\
& \quad \rightarrow \{\ \forall\, now' \cdot new_now \Rightarrow (\forall t \in [now, now')\cdot \\
& \qquad\qquad c_a[i].t = c_a[i].now + (t - now)) \wedge \\
& \qquad\qquad c_a[i].t - (c_e[i].now + (t - now) + c_p[i].now) \leq D[i] - R[i] \\
& \qquad \}; \\
& \qquad c_e[i] :- (\lambda t \cdot c_e[i].now + t - now) \,;\, c_p[i] :- (\lambda t \cdot c_p[i].now); \\
& \qquad state[i] :- (\lambda t \cdot ex) \,;\, start := now; \\
& \qquad\qquad now := min\{t' \geq now \mid gg.t'\} \\
&)
\end{aligned}
$$

$$
\begin{array}{ll}
\sqsubseteq & \{\text{pull assertions through guards, drop assertions}\} \\
& \ldots \\
& \| \; state[i].now = wt \wedge c_a[i].now \leq D[i] - R[i] \\
& \quad \rightarrow c_e[i] :- (\lambda t \cdot t - now) \, ; state[i] :- (\lambda t \cdot ex); \\
& \qquad start := now \, ; now := min\{t' \geq now \mid gg.t'\} \\
& \ldots \\
& \| \; state[i].now = pt \wedge c_a[i].now - (c_e[i].now + c_p[i].now) \leq D[i] - R[i] \\
& \quad \rightarrow c_e[i] :- (\lambda t \cdot c_e[i].now + (t - now)) \, ; c_p[i] : -(\lambda t \cdot c_p[i].now); \\
& \qquad state[i] :- (\lambda t \cdot ex) \, ; start := now \, ; now := min\{t' \geq now \mid gg.t'\} \\
= & \{\text{notation}\} \\
& \mathcal{T}'(i)
\end{array}
$$

Due to Lemma 6, if we replace the new task form, $\mathcal{T}'(i)$, of (6.6), in (6.5), we get a real-time system model where the safety property $\Box \; q_t$ holds. The "three-step" technique has led to a representation that preserves the timeliness predicate I_t (it is easy to check on the refined version of (6.5), where one replaces $\mathcal{T}(i)$ with $\mathcal{T}'(i)$, that I_t is indeed an invariant).

The refinement for timeliness is summarized in table (6.7).

	$\mathcal{T}(i)$		
action	initial guard	guard strengthening predicate	(6.7)
$wt \rightarrow ex$	$state[i].now = wt$	$c_a[i].now \leq D[i] - R[i]$	
$pt \rightarrow ex$	$state[i].now = pt$	$c_a[i].now - (c_e[i].now + c_p[i].now) \leq D[i] - R[i]$	

Nevertheless, not only deadline-related constraints matter, but also being able to ensure, for each real-time task, the mutually exclusive access to CPU resources. Consequently, we are not happy with a scheduled system that lets two tasks execute simultaneously. Therefore, we go on and enforce the mutual exclusion safety condition.

$$
\begin{array}{lll}
q_{me} & \triangleq & \forall i \cdot \forall t \in [start, now) \cdot \\
& & \quad state[i].start = ex \Rightarrow (\forall j \neq i \cdot state[j].t \neq ex)
\end{array}
$$

An important system property is the fact that its state does not change during the time interval $[start, now)$. Hence, condition q_{me} is added with this property to get I_{me}:

$$
\begin{array}{lll}
I_{me} & \equiv & \forall i \cdot \forall t \in [start, now) \cdot \\
& & \quad (state[i].start = ex \Rightarrow (\forall j \neq i \cdot state[j].t \neq ex)) \\
& & \wedge \; (state[i].start = s \Rightarrow state[i].t = s), \; s \in \{sl, wt, pt\}
\end{array}
$$

Following a similar line of refinement as in (6.6), after propagation of the corresponding context information, that is, $\mathcal{T}(i).I_{me}$, we get an improved version of the task model, which guarantees mutually exclusive execution. The weakest pre-

condition for the task $\mathcal{T}(i)$ to establish I_{me} is given below:

$$
\begin{array}{ll}
& \mathcal{T}(i).I_{me} \\
\equiv & \{\text{succesive application of rules (2.10), (2.7), (2.6), (2.4)}\} \\
& \quad (\forall j \neq i \cdot \mathcal{T}(i).I^{j}_{me}) \\
& \wedge\ (state[i].now = wt \Rightarrow \\
& \qquad (\forall now' \cdot now' = min\{t' \geq now \mid c_e[i].t' \leq E[i]\} \Rightarrow \\
& \qquad\quad (\forall t \in [now, now') \cdot (\forall j \neq i \cdot state[j].t \neq ex)))) \\
& \wedge\ (state[i].now = pt \Rightarrow \\
& \qquad (\forall now' \cdot now' = min\{t' \geq now \mid c_e[i].t' \leq E[i]\} \Rightarrow \\
& \qquad\quad (\forall t \in [now, now') \cdot (\forall j \neq i \cdot state[j].t \neq ex))))
\end{array}
$$

In the above, I^{j}_{me} is similar to I_{me}, with the difference that the index i is replaced by j.

Table (6.8) shows the guards that are affected by the second refinement.

	$\mathcal{T}'(i)$	
action	$\mathcal{T}(i)$ guard	guard strengthening predicate
$wt \rightarrow ex$	$state[i].now = wt$ $\wedge\ c_a[i].now \leq D[i] - R[i]$	$\forall j \neq i \cdot state[j].now \neq ex$
$pt \rightarrow ex$	$state[i].now = pt$ $\wedge\ c_a[i].now - (c_e[i].now + c_p[i].now)$ $\leq D[i] - R[i]$	$\forall j \neq i \cdot state[j].now \neq ex$

(6.8)

We agree with the fact that the result of enforcing mutual exclusion is straightforward, thus we could have added this condition from the start, when modeling the preemptible task. However, for the sake of consistency, we have chosen to carry out a similar refinement as for the timeliness predicate.

Enforcing q_t and q_{me} on the task set is still not enough for ensuring a correct scheduling. The first condition does guarantee that each task, taken separately, terminates before or at its deadline, but it does not ensure the same in case multiple tasks are simultaneously waiting for their turn. Not only deciding on the right dispatch time of each task, but also the correct task execution order, based on priorities, falls into the responsibility area of the embedded scheduler. Consequently, we introduce the vector variable pr that models the fixed priority of each task, respectively and we define the policy-related condition q_{pol}. The latter specifies that the task chosen for execution is always the one with the highest priority of all waiting or already preempted tasks.

$$
\begin{array}{lll}
q_{pol} & \triangleq & \forall i \cdot \forall t \in [start, now) \cdot \\
& & \quad (state[i].start = ex \Rightarrow \\
& & \qquad (\forall j \neq i \cdot (state[j].start = sl \wedge c_a[j].start < P[j]) \vee \\
& & \qquad ((state[j].t = wt \vee state[j].t = pt) \wedge pr[j].t < pr[i].t))) \\
& & \wedge\ (state[i].start = pt \Rightarrow \\
& & \qquad (\exists j \neq i \cdot state[j].t = wt \wedge pr[i].t < pr[j].t))
\end{array}
$$

Once a task $\mathcal{T}(i)$ has started to execute, all the other tasks with lower priorities than $\mathcal{T}(i)$ are either waiting or being preempted, or else they have not been released yet, that is, $state[j].start = sl \wedge c_a[j].start < P[j]$. In this way, the list of waiting tasks is updated when the scheduler selects the highest priority task for execution. We choose $I_{pol} \Rightarrow q_{pol}$ as follows:

$$I_{pol} \equiv q_{pol} \wedge (\forall j \neq i \cdot pr[j].start < pr[i].start \Rightarrow pr[j].t < pr[i].t)$$

Next, we conduct a combined technique of precondition calculation and program derivation, as for the other two properties. The refinement is shown in the table of Figure (6.3).

By applying logic, we can simplify the first and the third guards in Figure 6.3. Hence, the predicate $(\forall j \neq i \cdot state[j].now \neq ex)$ can be dropped from the corresponding guards of actions $wt \to ex$ and $pt \to ex$.

	$\mathcal{T}^s(i)$	
action	$\mathcal{T}'(i)$ guard	guard strengthening predicate
$wt \to ex$	$state[i].now = wt$ $\wedge\, c_a[i].now \leq D[i] - R[i]$ $\wedge\, (\forall j \neq i \cdot state[j].now \neq ex)$	$\forall j \neq i \cdot (state[j].now = sl \wedge c_a[j].now < P[j])$ $\vee\, ((state[j].now = wt \vee state[j].now = pt)$ $\wedge\, pr[j].now < pr[i].now)$
$ex \to pt$	$state[i].now = ex$ $\wedge\, c_e[i].now < E[i]$	$\exists j \neq i \cdot state[j].now = wt$ $\wedge\, pr[i].now < pr[j].now$
$pt \to ex$	$state[i].now = pt$ $\wedge c_a[i].now - (c_e[i].now + c_p[i].now) \leq D[i] - R[i]$ $\wedge\, (\forall j \neq i \cdot state[j].now \neq ex)$	$\forall j \neq i \cdot (state[j].now = sl \wedge c_a[j].now < P[j])$ $\vee\, ((state[j].now = wt \vee state[j].now = pt)$ $\wedge\, pr[j].now < pr[i].now)$

Figure 6.3: Task refinement for policy-related property I_{pol}.

Out of the above successive refinements, we get the final form of the scheduled preemptible task

$$\mathcal{T}^s(i) \triangleq A_1^i \parallel A_2^i \parallel A_3^i \parallel A_4^i \parallel A_5^i,$$

$$A_1^i = state[i].now = sl \wedge c_a[i].now = P[i] + ofs[i].now \to c_a[i] :- (\lambda t \cdot t - now) \,;\, state[i] :- (\lambda t \cdot wt) \,;\, UT$$

$$g_{21}^i \equiv state[i].now = wt \wedge c_a[i].now \leq D[i] - R[i]$$

$$g_{22}^i \equiv (\forall j \neq i \cdot (state[j].now = sl \wedge c_a[j].now < P[j]) \vee ((state[j].now = wt \vee state[j].now = pt) \wedge pr[j].now < pr[i].now))$$

$$A_2^i = g_{21}^i \wedge g_{22}^i \to c_e[i] :- (\lambda t \cdot t - now) \,;\, state[i] :- (\lambda t \cdot ex) \,;\, UT$$

$$
\begin{aligned}
A_3^i \quad = \quad & state[i].now = ex \wedge c_e[i].now = E[i] \\
& \to c_e[i] :- (\lambda t \cdot 0) \,;\, c_p[i] :- (\lambda t \cdot 0); \\
& \quad [\, ofs[i] :- x' \mid \forall t \geq now \cdot x'.t \in \mathsf{Real}_+ \,]; \\
& \quad state[i] :- (\lambda t \cdot sl) \,;\, UT
\end{aligned}
$$

$$
\begin{aligned}
g_4^i \quad \equiv \quad & state[i].now = ex \wedge c_e[i].now < E[i] \wedge \\
& (\exists j \neq i \cdot state[j].now = wt \wedge pr[i].now < pr[j].now)
\end{aligned}
$$

$$
\begin{aligned}
A_4^i \quad = \quad & g_4^i \\
& \to c_e[i] :- (\lambda t \cdot c_e[i].now) \,;\, c_p[i] :- (\lambda t \cdot c_p[i].now + t - now); \\
& \quad state[i] :- (\lambda t \cdot pt) \,;\, UT
\end{aligned}
$$

$$
g_{51}^i \quad \equiv \quad state[i].now = pt \wedge c_a[i].now - c_e[i].now - c_p[i].now \leq D[i] - R[i]
$$

$$
\begin{aligned}
g_{52}^i \quad \equiv \quad & (\forall j \neq i \cdot (state[j].now = sl \wedge c_a[j].now < P[j]) \vee \\
& \qquad\qquad ((state[j].now = wt \vee state[j].now = pt) \wedge \\
& \qquad\qquad\quad pr[j].now < pr[i].now)) \\
A_5^i \quad = \quad & g_{51}^i \wedge g_{52}^i \\
& \to c_e[i] :- (\lambda t \cdot c_e[i].now + (t - now)) \,;\, c_p[i] :- (\lambda t \cdot c_p[i].now); \\
& \quad state[i] :- (\lambda t \cdot ex) \,;\, UT
\end{aligned}
$$

Combining n such tasks results in the correct-by-construction scheduled real-time system.

Last, we refine the system model (6.5), by introducing the local variable pr, which is initialized such that tasks are assigned priorities in the reverse order from their deadlines. These fixed values are established by a demonic nondeterministic assignment. The scheduled real-time system is described as follows:

$$
\begin{aligned}
& \overline{\mathcal{RTS}}^s \\
\triangleq \quad & \textbf{begin var } start, now \colon \mathsf{Real}_+, \\
& \qquad\qquad state \colon \mathsf{array}\ [1..n]\ \textbf{of}\ (\mathsf{Real}_+ \to \{sl, wt, ex, pt\}), \\
& \qquad\qquad ofs, c_a, c_e, c_p : \mathsf{array}\ [1..n]\ \textbf{of}\ (\mathsf{Real}_+ \to \mathsf{Real}_+), \\
& \qquad\qquad pr : \mathsf{array}[1..n]\ \textbf{of}\ (\mathsf{Real}_+ \to \mathsf{Nat}) \bullet \\
& \quad now := 0 \,;\, state :- (\lambda t \cdot wt) \,;\, c_e, c_p :- (\lambda t \cdot 0) \,;\, c_a :- (\lambda t \cdot t) \,; \\
& \quad [\, ; 1 \leq i \leq n \cdot [\, ofs[i] :- x' \mid \forall t \geq now \cdot x'.t \in \mathsf{Real}_+ \,]\,] \,; \\
& \quad [\, pr :- p' \mid \forall t \geq now \cdot \forall i \cdot p'[i].t \neq 0 \wedge (\forall j \in [1..n] \cdot \\
& \qquad (D[i] < D[j] \vee (D[i] = D[j] \wedge i < j)) \Rightarrow p'[i].t > p'[j].t) \,] \,;\, UT \,; \\
& \quad \textbf{do}\ [\ \| \, 1 \leq i \leq n \cdot (A_1^i \,\|\, A_2^i \,\|\, A_3^i \,\|\, A_4^i \,\|\, A_5^i)\,]\ \textbf{od} \\
& \textbf{end}
\end{aligned}
\tag{6.9}
$$

By simple inspection of the above actions, one can conclude that $I \triangleq I_t \wedge I_{pol}$ is indeed an invariant of the real-time system described by (6.9): it is established

by the initialization and it is preserved by each action (as a consequence of our development method).

Based on Lemma 6, the invariance property is sufficient for inferring that

$$\Box\, q \;\triangleq\; \Box\,(q_t \wedge q_{pol})$$

has been enforced on the initial real-time system model, generating the correctly scheduled system $\overline{\mathcal{RTS}}^s$.

Although correct, the real-time model (6.9) does not provide an efficient implementation. For example, one needs to compare pairs of tasks, in an exhaustive manner, in order to pick up the one with the highest priority, and select it for execution. This is a time consuming operation, especially when dealing with large sets of tasks. Therefore, we would like to optimize the system representation, yet ensuring that the correctness properties, which we have just enforced, are preserved. To accomplish this goal, we apply trace refinement. As a result, the task-comparing operation is replaced by a function *Max* that establishes, quicker and simpler, the highest priority task, out of a *list* of waiting (or preempted) tasks. For example, in Mathematica, there exists such a function that picks up the maximum element of an array, and delivers it as output.

6.4.2 Trace Refinement of Continuous Action Systems

In this section, we adapt the main proof obligations that one should discharge while carrying out trace refinement of action systems, to CAS.

As explained in chapter 2, a trace of an action system is a sequence of observable states. Within the context of CAS, the observable states are given by evaluating the state functions at consecutive moments recorded by variable now.

Here, we introduce the notions of *behavior* and *trace* for CAS. A behavior of a CAS is a sequence of states observed at consecutive moments now:

$$b = ((x.now_1, z.now_1), \ldots, (x.now_k, z.now_k), \ldots)$$

Above, x denotes the *local* state, and z the *global* state, and $(now_1, \ldots, now_k, \ldots)$ is a sequence of consecutive (not necessarily different) transition times.

A trace of behavior b is obtained by removing the local state component in each state of a given CAS, and all finite stuttering (no change in z).

Assume that two CAS, $\mathcal{A}$ and $\mathcal{C}$ are translated into their respective semantic definitions given by action systems $\overline{\mathcal{A}}$, and $\overline{\mathcal{C}}$. Let us further suppose that the local variables of $\overline{\mathcal{C}}$, $x_{\mathcal{C}} = x_{\mathcal{A}} \cup y$, where $x_{\mathcal{A}}$ are the local variables of $\overline{\mathcal{A}}$ and y are some auxiliary variables. We assume next that X is the *auxiliary* action that modifies variables of y, and that I is a predicate over the observable states of the concrete system $\overline{\mathcal{C}}$. This means that I refers only to the values of the system variables at moment now.

We denote by $S_{\mathcal{A}}\,;UT$ the action of $\overline{\mathcal{A}}$, and by $S_{\mathcal{C}}\,;UT$ the one of $\overline{\mathcal{C}}$. Then, we say that $\overline{\mathcal{A}}$ is (trace) refined by $\overline{\mathcal{C}}$ (using I), $\overline{\mathcal{A}} \sqsubseteq_I \overline{\mathcal{C}}$, if the following conditions hold:

$$
\begin{aligned}
true &\Rightarrow (Init_{\mathcal{C}}\,;UT).I && (6.10)\\
I &\Rightarrow (S_{\mathcal{C}}\,;UT).I && (6.11)\\
(S_{\mathcal{A}}\,;UT) &\sqsubseteq_I (S_{\mathcal{C}}\,;UT) && (6.12)\\
I \wedge g_{\mathcal{A}}.now &\Rightarrow g_{\mathcal{C}}.now \vee g_X.now && (6.13)\\
\mathsf{skip} &\sqsubseteq_I X && (6.14)\\
I &\Rightarrow (\mathbf{do}\ X\ \mathbf{od}\ ).\mathsf{true} && (6.15)
\end{aligned}
$$

Above,

$$(S_{\mathcal{A}}\,;UT) \sqsubseteq_I (S_{\mathcal{C}}\,;UT) \equiv (\forall t \cdot \forall q \cdot I \wedge (S_{\mathcal{A}}\,;UT).q \Rightarrow (S_{\mathcal{C}}\,;UT).(I \wedge q))$$

The first two relations ensure that I is established at $now = 0$, and preserved by the actions of the concrete system $\overline{\mathcal{C}}$, which is necessary to guarantee that the system evolves towards the next now starting from a state of I. Condition (6.12) requires that the main action of the abstract system is refined by the main action of the concrete system, using I. Relation (6.13) guarantees that whenever an action is enabled in the abstract system, one of the actions of the concrete system, be it a refinement of the old action or the fresh auxiliary action, is also enabled. This means that once an initial real-time model is free of deadlocks, the refined system is also free of deadlocks. The last but one condition says that the auxiliary action behaves like skip with respect to the global variables, while preserving I. Finally, the last condition says that the execution of the auxiliary action, taken separately, terminates eventually, whenever I holds.

6.4.3 Implementing the Real-Time System

Getting back to our target, that is, an efficient implementation of the real-time system model (6.9), we perform the transformations presented below.

• **Step 1.** We introduce the auxiliary local vector variable q, which stores the tasks' priorities, respectively; when a task i is in state wt, its priority is added to q, at $q[i]$; when the same task has finished execution, its priority is removed from q, by assignment $q[i] :- (\lambda t \cdot 0)$. The actions that modify $q[i]$ are as follows.

$$
\begin{aligned}
A_1'^i \;=\;& state[i].now = sl \wedge c_a[i].now = P[i] + ofs[i].now\\
& \rightarrow c_a[i] :- (\lambda t \cdot t - now)\,;state[i] :- (\lambda t \cdot wt)\,;q[i] :- pr[i]\,;UT\\
A_3'^i \;=\;& state[i].now = ex \wedge c_e[i].now = E[i]\\
& \rightarrow c_e[i] :- (\lambda t \cdot 0)\,;c_p[i] :- (\lambda t \cdot 0)\,;\\
& \quad [\, ofs[i] :- x' \mid \forall t \geq now \cdot x'.t \in \mathsf{Real}_+ \,]\\
& \quad state[i] :- (\lambda t \cdot sl)\,;q[i] :- (\lambda t \cdot 0)\,;UT
\end{aligned}
$$

• **Step 2.** We conduct the following refinements:

$$A_2^i \sqsubseteq_I A_2'^i, \quad A_4^i \sqsubseteq_I A_4'^i, \quad A_5^i \sqsubseteq_I A_5'^i,$$

where

$$\begin{aligned}
I \;\triangleq\; & \forall i \cdot \forall now \cdot \\
& \quad (state[i].now = sl \Rightarrow q[i].now = 0) \\
& \wedge ((pr[i].now = Max(q.now) \wedge pr[i].now = q[i].now \\
& \qquad \wedge (\forall j \neq i \cdot (state[j].now = sl \wedge c_a[j].now < P[j]) \\
& \qquad\quad \vee ((state[j].now = wt \vee state[j].now = pt) \\
& \qquad\quad \wedge pr[j].now < pr[i].now \wedge q[j].now = pr[j].now))) \\
& \vee (pr[i].now \neq Max(q.now) \wedge pr[i].now = q[i].now \\
& \qquad \wedge (\exists j \neq i \cdot state[j].now = wt \wedge pr[j].now > pr[i].now \\
& \qquad\quad \wedge q[j].now = pr[j].now)))
\end{aligned}$$

and

$$\begin{aligned}
g_2'^i \;\equiv\; & pr[i].now = Max(q.now) \wedge \\
& (\forall j \neq i \cdot (state[j].now = sl \wedge c_a[j].now < P[j]) \;\vee \\
& \qquad state[j].now = wt \vee state[j].now = pt) \wedge \\
& state[i].now = wt \wedge c_a[i].now \leq D[i] - R[i]
\end{aligned}$$

$$\begin{aligned}
A_2'^i \;=\; & g_2'^i \\
& \rightarrow c_e[i] :- (\lambda t \cdot t - now) \,;\, state[i] :- (\lambda t \cdot ex) \,;\, UT
\end{aligned}$$

$$\begin{aligned}
g_4'^i \;\equiv\; & pr[i].now \neq Max(q.now) \wedge \\
& state[i].now = ex \wedge c_e[i].now < E[i]
\end{aligned}$$

$$\begin{aligned}
A_4'^i \;=\; & g_4'^i \\
& \rightarrow c_e[i] :- (\lambda t \cdot c_e[i].now) \,;\, c_p[i] :- (\lambda t \cdot c_p[i].now + (t - now)) \,; \\
& \quad state[i] :- (\lambda t \cdot pt) \,;\, UT
\end{aligned}$$

$$\begin{aligned}
g_5'^i \;\equiv\; & pr[i].now = Max(q.now) \wedge \\
& (\forall j \neq i \cdot (state[j].now = sl \wedge c_a[j].now < P[j]) \;\vee \\
& \qquad state[j].now = wt \vee state[j].now = pt) \wedge \\
& state[i].now = pt \wedge c_a[i].now - (c_e[i].now + c_p[i].now) \leq D[i] - R[i] \\
A_5'^i \;=\; & g_5'^i \\
& \rightarrow c_e[i] :- (\lambda t \cdot c_e[i].now + t - now) \,;\, c_p[i] :- (\lambda t \cdot c_p[i].now) \,; \\
& \quad state[i] :- (\lambda t \cdot ex) \,;\, UT
\end{aligned}$$

For example, proving the truth of relation $A_2^i \sqsubseteq_I A_2'^i$ boils down to showing that the relations below hold:

$$I \wedge g_2'^i \Rightarrow g_2^i \tag{6.16}$$

$$\begin{aligned} &\forall q \cdot I \wedge g_2'^i \wedge (c_e[i] :- (\lambda t \cdot t - now)\,;\, state[i] :- (\lambda t \cdot ex)\,;\, UT).q \\ &\quad \Rightarrow (c_e[i] :- (\lambda t \cdot t - now)\,;\, state[i] :- (\lambda t \cdot ex)\,;\, UT).(I \wedge q) \end{aligned} \tag{6.17}$$

A similar decomposition into refinement of guards and action bodies applies to the other two refinements, that is, of actions A_4^i and A_5^i, respectively.

Since relation (6.17) can be easily proved, we are left to discharge requirement (6.16), for the refinement $A_2^i \sqsubseteq_I A_2'^i$ to be valid. The proof is sketched below:

$$\begin{aligned}
& I \wedge pr[i].now = Max(q.now) \wedge \\
& (\forall j \neq i \cdot (state[j].now = sl \wedge c_a[j].now < P[j]) \vee \\
& \qquad state[j].now = wt \vee state[j].now = pt) \wedge \\
& state[i].now = wt \wedge c_a[i].now \leq D[i] - R[i]) \\
\equiv \quad & \{\text{substitute } I, \text{logic}\} \\
& pr[i].now = Max(q.now) \wedge pr[i].now = q[i].now \wedge \\
& (\forall j \neq i \cdot (state[j].now = sl \wedge c_a[j].now < P[j]) \vee \\
& \qquad ((state[j].now = wt \vee state[j].now = pt) \wedge \\
& \qquad pr[j].now < pr[i].now \wedge q[j].now = pr[j].now))) \wedge \\
& (state[i].now = sl \Rightarrow q[i].now = 0) \wedge \\
& state[i].now = wt \wedge c_a[i].now \leq D[i] - R[i] \wedge \ldots \\
\Rightarrow \quad & \\
& state[i].now = wt \wedge c_a[i].now \leq D[i] - R[i] \wedge \\
& \quad (\forall j \neq i \cdot (state[j].now = sl \wedge c_a[j].now < P[j]) \vee \\
& \qquad ((state[j].now = wt \vee state[j].now = pt) \wedge pr[j].now < pr[i].now))
\end{aligned}$$

The dots in the above proof stand for the part of I that treats all the other tasks except $\mathcal{T}(i)$. We have also checked that $A_4^i \sqsubseteq_I A_4'^i$ and $A_5^i \sqsubseteq_I A_5'^i$ are met; however, we omit the proofs here. Consequently, requirement (6.12) is fulfilled.

By replacing the refined actions in the real-time system model (6.9), we get the new scheduled action system as:

$$\begin{aligned}
& \overline{\mathcal{RTS}}'^s \qquad\qquad (6.18) \\
\triangleq \quad & \textbf{begin var } start, now : \mathsf{Real}_+, \\
& \qquad state : \mathsf{array}\ [1..n]\ \textbf{of}\ (\mathsf{Real}_+ \rightarrow \{sl, wt, ex, pt\}), \\
& \qquad \mathit{ofs}, c_a, c_e, c_p : \mathsf{array}\ [1..n]\ \textbf{of}\ (\mathsf{Real}_+ \rightarrow \mathsf{Real}_+), \\
& \qquad pr, q : \mathsf{array}[1..n]\ \textbf{of}\ (\mathsf{Real}_+ \rightarrow \mathsf{Nat}) \bullet
\end{aligned}$$

$$
\begin{array}{l}
now := 0 \,;\, state :- (\lambda t \cdot wt) \,;\, c_a :- (\lambda t \cdot t); \qquad \text{Init} \\
c_e, c_p :- (\lambda t \cdot 0); \\
[\,;\, i \cdot \; [\, ofs[i] :- x' \mid \forall t \cdot x'.t \in \mathsf{Real}_+ \,]\,]; \\
[\, pr :- p' \mid \forall t \geq now \cdot \forall i \cdot p'[i].t \neq 0 \wedge (\forall j \in [1..n] \cdot \\
\quad (D[i] < D[j] \vee (D[i] = D[j] \wedge i < j)) \Rightarrow p'[i].t > p'[j].t) \,]; \\
[\,;\, i \cdot q[i] :- pr[i] \,]; \\
[I] \,;\, UT; \\
\mathbf{do} \\
\quad [\; \| \; 1 \leq i \leq n : (A'^i_1 \,\|\, A'^i_2 \,\|\, A'^i_3 \,\|\, A'^i_4 \,\|\, A'^i_5) \,] \\
\mathbf{od} \\
\mathbf{end}
\end{array}
$$

In (6.18), the array q is initialized with the task priorities, respectively, since all tasks are concurrently waiting for the CPU, at time $now = 0$.

The initialization statement assumes the predicate I. We are doing this because, essentially, I requires a correct implementation of the function Max and we assume that this requirement holds of any programming language that has Max as a built-in function. Hence, relation (6.10) is fulfilled. Along the same line, relation (6.11) also holds, meaning that each (new) action of $\overline{\mathcal{RTS}}'^s$ preserves I. We give here only the sketch of the respective proof for A'^i_1. The rest of the proofs are similar.

$$
\begin{array}{l}
\quad g'^i_1 \wedge I \\
= \{\text{substitute } g'^i_1\} \\
\quad (state[i].now = sl \wedge c_a[i].now = P[i] + ofs[i].now) \wedge I \\
\Rightarrow \{\text{compute } A'^i_1.I\} \\
\quad (now = min\{t' \geq now \mid state[i].t' = wt \wedge t' - now \leq D[i] - R[i]\} \Rightarrow \\
\qquad (pr[i].now = Max(q.now) \wedge \\
\qquad\quad (\forall j \neq i \cdot (state[j].now = sl \wedge c_a[j].now < P[j]) \vee \\
\qquad\qquad\qquad ((state[j].now = wt \vee state[j].now = pt) \wedge \\
\qquad\qquad\qquad\quad pr[j].now < pr[i].now \wedge q[j].now = pr[j].now))) \wedge \\
\qquad (state[i].now = sl \Rightarrow q[i].now = 0) \\
\quad \wedge (\forall j \neq i \cdot I^j))
\end{array}
$$

The predicate I^j incorporates the information in I about tasks indexed by $j \neq i$, which is not modified by the assignments in A'^i_1.

Considering the fact that we have proved relations (6.10), (6.11) and (6.12), we conclude that

$$\overline{\mathcal{RTS}}^s \sqsubseteq_I \overline{\mathcal{RTS}}'^s$$

Note that requirements (6.13), (6.14) and (6.15) need not be checked, since we have not introduced any auxiliary actions. Hence, one can view $\overline{\mathcal{RTS}}'^s$ as a more efficient representation of the constructed model $\overline{\mathcal{RTS}}^s$.

6.4.4 Deriving the Scheduler Component

The real-time scheduled system $\overline{\mathcal{RTS}}^f$ that we have constructed is based on the simplistic assumption of independent tasks. Clearly, this should be relaxed at later stages of design, as task interaction is needed in many meaningful applications. Processes can interact safely by either some form of protected shared data (using, for example, semaphores, monitors or protected objects), or directly (using some form of rendezvous) [52]. Therefore, extra requirements have to be added to the real-time scheduled system, regarding resource management protocols, all of which are connected to scheduling.

As one might immediately imagine, new refinements need to be carried out; in principle, they would mostly affect the behavior of the scheduler. Under this scenario, the single-block real-time system model would ultimately become much more complicated. Hence, it can be beneficial to get a modular description of the system, which might ease future design tasks.

Targeting a two-module implementation, we carry out the following refinement steps on model $\overline{\mathcal{RTS}'}^s$:

- Introduce a local variable $ok : \mathsf{array}\ [1..n]\ \mathbf{of}\ (Real_+ \rightarrow \mathsf{Bool})$. This vector variable encodes the permission for execution given by the scheduler to each task; $ok[i]$ is set or reset according to the DM scheduling policy rules.
- Strengthen the guards of actions A'^i_1, A'^i_3.
- Consider the following predicate ($k \in [1, ..n]$):

$$
\begin{aligned}
I_1 \quad \equiv \quad & \forall i \cdot \forall now \cdot \\
& (ok[i].now \wedge state[i].now = wt \Rightarrow \\
& \quad (pr[i].now = Max(q.now) \wedge c_a[i].now \le D[i] - R[i] \wedge \\
& \quad\quad (\forall j \neq i \cdot (state[j].now = sl \wedge c_a[j].now < P[j]) \vee \\
& \quad\quad\quad state[j].now = wt \vee state[j].now = pt))) \\
\wedge \quad & (ok[i].now \wedge state[i].now = pt \Rightarrow \\
& \quad (pr[i].now = Max(q.now) \wedge \\
& \quad\quad c_a[i].now - (c_e[i].now + c_p[i].now) \le D[i] - R[i] \wedge \\
& \quad (\forall j \neq i \cdot (state[j].now = sl \wedge c_a[j].now < P[j]) \vee \\
& \quad\quad state[j].now = wt \vee state[j].now = pt))) \\
\wedge \quad & (\neg ok[i].now \wedge state[i].now = ex \Rightarrow \\
& \quad (pr[i].now \neq Max(q.now) \wedge c_e[i].now < E[i]))
\end{aligned}
$$

By using I_1, which can be easily shown to satisfy relations (6.10),(6.11) on $\overline{\mathcal{RTS}}^m$ given below, we can prove

$$A'^i_2 \sqsubseteq_{I_1} A^i_{22m}, \quad A'^i_4 \sqsubseteq_{I_1} A^i_{42m}, \quad A'^i_5 \sqsubseteq_{I_1} A^i_{52m}$$

The actions A^i_{22m}, A^i_{42m} and A^i_{52m} are also described below. The proofs that the conditions (6.10 - 6.15) of trace refinement are met are sketched in Appendix (A-6).

Eventually, the successive application of the above steps leads to the following refinement

$$\begin{array}{ll} & \overline{\mathcal{RTS}'}^s \\ \sqsubseteq_{I_1} & \{\text{rule (2.21), trace refinement}\} \\ & \overline{\mathcal{RTS}}^m \end{array}$$

where

$$\begin{array}{rl} & \overline{\mathcal{RTS}}^m \qquad\qquad (6.19) \\ = & \mathbf{begin\ var}\ start, now : \mathsf{Real}_+, \\ & \quad state : \mathsf{array}\ [1..n]\ \mathbf{of}\ (\mathsf{Real}_+ \to \{sl, wt, ex, pt\}), \\ & \quad ofs, c_a, c_e, c_p : \mathsf{array}\ [1..n]\ \mathbf{of}\ (\mathsf{Real}_+ \to \mathsf{Real}_+), \\ & \quad pr, q : \mathsf{array}[1..n]\ \mathbf{of}\ (\mathsf{Real}_+ \to \mathsf{Nat}), \\ & \quad ok : \mathsf{array}[1..n]\ \mathbf{of}\ (\mathsf{Real}_+ \to \mathsf{Bool}) \bullet \end{array}$$

Init

$$\begin{array}{l} now := 0\ ;\ state :- (\lambda t \cdot wt)\ ;\ c_a :- (\lambda t \cdot t); \\ c_e, c_p :- (\lambda t \cdot 0)\ ;\ ok :- (\lambda t \cdot \mathsf{false}); \\ [\ ;\ i \cdot [\ ofs[i] :- x' \mid \forall t \geq now \cdot x'.t \in \mathsf{Real}_+\ ]\ ]; \\ [pr :- p' \mid \forall t \geq now \cdot \forall i \cdot p'[i].t \neq 0 \wedge (\forall j \in [1..n] \cdot \\ \quad (D[i] < D[j] \vee (D[i] = D[j] \wedge i < j)) \Rightarrow p'[i].t > p'[j].t)]; \\ [\ ;\ i \cdot q[i] :- pr[i]]; \\ [I]\ ;\ UT; \end{array}$$

$$\begin{array}{l} \mathbf{do} \\ \ [\ \| \ 1 \leq i \leq n : (A^i_{11m} \,\|\, A^i_{12m} \,\|\, A^i_{21m} \,\|\, A^i_{22m} \,\|\, A^i_{3m} \,\| \\ \qquad\qquad A^i_{41m} \,\|\, A^i_{42m} \,\|\, A^i_{51m} \,\|\, A^i_{52m})\ ] \\ \mathbf{od} \\ \mathbf{end}, \end{array}$$

$$\begin{array}{rcl} A^i_{11m} & = & \neg ok[i].now \wedge state[i].now = sl \wedge c_a[i].now = P[i] + ofs[i].now \\ & & \to c_a[i] :- (\lambda t \cdot t - now)\ ;\ state[i] :- (\lambda t \cdot wt)\ ;\ q[i] :- pr[i]\ ;\ UT \end{array}$$

$$\begin{array}{rcl} A^i_{12m} & = & ok[i].now \wedge state[i].now = sl \\ & & \to ok[i] :- (\lambda t \cdot \mathsf{false})\ ;\ UT \end{array}$$

$$\begin{array}{rcl} A^i_{21m} & = & \neg ok[i].now \wedge pr[i].now = Max(q.now) \wedge \\ & & (\forall j \neq i \cdot (state[j].now = sl \wedge c_a[j].now < P[j]) \vee \\ & & \qquad state[j].now = wt \vee state[j].now = pt) \wedge \\ & & state[i].now = wt \wedge c_a[i].now \leq D[i] - R[i] \\ & & \to ok[i] :- (\lambda t \cdot \mathsf{true})\ ;\ UT \end{array}$$

$$A^i_{22m} = ok[i].now \wedge state[i].now = wt$$
$$\rightarrow c_e[i] :- (\lambda t \cdot t - now)\ ;\ state[i] :- (\lambda t \cdot ex)\ ;\ UT$$

$$A^i_{3m} = ok[i].now \wedge state[i].now = ex \wedge c_e[i].now = E[i]$$
$$\rightarrow c_e[i], c_p[i] :- (\lambda t \cdot 0), (\lambda t \cdot 0)\ ;\ [\ ofs[i] :- x' \mid \forall t \cdot x'.t \in \mathsf{Real}_+\ ]$$
$$state[i] :- (\lambda t \cdot sl)\ ;\ q[i] :- (\lambda t \cdot 0)\ ;\ UT$$

$$A^i_{41m} = ok[i].now \wedge pr[i].now \neq Max(q.now) \wedge$$
$$state[i].now = ex \wedge c_e[i].now < E[i]$$
$$\rightarrow ok[i] :- (\lambda t \cdot \mathsf{false})\ ;\ UT$$

$$A^i_{42m} = \neg ok[i].now \wedge state[i].now = ex$$
$$\rightarrow c_e[i] :- (\lambda t \cdot c_e[i].now)\ ;\ c_p[i] :- (\lambda t \cdot c_p[i].now + t - now)\ ;$$
$$state[i] :- (\lambda t \cdot pt)\ ;\ UT$$

$$A^i_{51m} = \neg ok[i].now \wedge pr[i].now = Max(q.now) \wedge$$
$$(\forall j \neq i \cdot (state[j].now = sl \wedge c_a[j].now < P[j]) \vee$$
$$state[j].now = wt \vee state[j].now = pt) \wedge$$
$$state[i].now = pt \wedge c_a[i].now - (c_e[i].now + c_p[i].now) \leq D[i] - R[i]$$
$$\rightarrow ok[i] :- (\lambda t \cdot \mathsf{true})\ ;\ UT$$

$$A^i_{52m} = ok[i].now \wedge state[i].now = pt$$
$$\rightarrow c_e[i] :- (\lambda t \cdot c_e[i].now + t - now)\ ;\ c_p[i] :- (\lambda t \cdot c_p[i].now)\ ;$$
$$state[i] :- (\lambda t \cdot ex)\ ;\ UT$$

It is easy to observe that we can rewrite the loop of $\overline{\mathcal{RTS}}^m$ as follows.

$$\begin{array}{l} \mathbf{do} \\ \quad [\ \| \ 1 \leq i \leq n \cdot (A^i_{11m} \| A^i_{22m} \| A^i_{3m} \| A^i_{42m} \| A^i_{52m})\ ] \\ \| \ g \rightarrow (A^i_{12m} \| A^i_{21m} \| A^i_{41m} \| A^i_{51m}) \\ \mathbf{od} \end{array} \quad (6.20)$$

where

$$g \triangleq \neg\, \mathsf{gg}\ [\ \| \ 1 \leq i \leq n \cdot (A^i_{11m} \| A^i_{22m} \| A^i_{3m} \| A^i_{42m} \| A^i_{52m})\ ]$$

By gg we denote the disjunction of the respective action guards.

Our goal is to decompose $\overline{\mathcal{RTS}}^m$, correctly. For this, we apply the *prioritizing decomposition* theorem (see chapter 2).

Let us consider the trivial invariant $I_2 \equiv \mathsf{true}$. We identify the condition g, mentioned in Theorem 1, with g within the loop (6.20). Next, we identify guards gB and gC of the same theorem, as follows:

$$\begin{array}{lcl} gB & \equiv & \mathsf{gg}\ [\ \llbracket\ 1 \le i \le n \cdot (A^i_{11m} \ \llbracket\ A^i_{22m} \ \llbracket\ A^i_{3m} \ \llbracket\ A^i_{42m} \ \llbracket\ A^i_{52m})\ ] \\ gC & \equiv & \mathsf{gg}\ (A^i_{12m} \ \llbracket\ A^i_{21m} \ \llbracket\ A^i_{41m} \ \llbracket\ A^i_{51m}) \end{array}$$

Under these assumptions, it is easy to verify that the requirements of Theorem 1 are straightforwardly met. Since the invariant is trivial, it is established by the initialization of $\overline{\mathcal{RTS}}^m$; also the actions of the new loop (6.20) preserve I_2 because they terminate. The exit condition $\neg gB \wedge gC \Rightarrow g$ is also immediate. Thus, we can safely decompose the action system (6.19), without loss of correctness.

Eventually, due to Theorem 1, we end up with a two-module implementation of the real time system $\overline{\mathcal{RTS}}^m$, which has been our design goal. The actual scheduler is one of the modules, and the collection of tasks is the other one, each represented by a corresponding CAS. The whole system description is given as

$$\mathcal{RTS}^m = \mathcal{TS} \ // \ \mathcal{S}ched,$$

with

$$\begin{array}{l} \mathcal{TS}\ (ok : \mathsf{array}[1..n]\ \mathbf{of}\ (\mathsf{Real}_+ \to \mathsf{Bool}), \\ \quad state : \mathsf{array}[1..n]\ \mathbf{of}\ (\mathsf{Real}_+ \to \{sl, wt, ex\}), \\ \quad c_a, c_e, c_p : \mathsf{array}\ [1..n]\ \mathbf{of}\ (\mathsf{Real}_+ \to \mathsf{Real}_+), \\ \quad pr, q : \mathsf{array}[1..n]\ \mathbf{of}\ (\mathsf{Real}_+ \to \mathsf{Nat})) \end{array}$$

$$\begin{array}{ll} = & \mathbf{begin\ var}\ ofs : \mathsf{array}\ [1..n]\ \mathbf{of}\ (\mathsf{Real}_+ \to \mathsf{Real}_+) \bullet \\ & \quad ok :- (\lambda t \cdot \mathsf{false})\ ;\ state :- (\lambda t \cdot wt); \qquad \text{Init} \\ & \quad c_a :- (\lambda t \cdot t)\ ;\ c_e, c_p :- (\lambda t \cdot 0), (\lambda t \cdot 0); \\ & \quad [pr :- p' \mid \forall t \cdot \forall i \cdot p'[i].t \neq 0 \wedge (\forall j \cdot (D[i] < D[j] \vee \\ & \qquad\qquad (D[i] = D[j] \wedge i < j)) \Rightarrow p'[i].t > p'[j].t)]; \\ & \quad [\ ;\ i \cdot q[i] :- pr[i]]\ ;\ [\ ;\ i \cdot\ [ofs[i] :- x' \mid \forall t \cdot x'.t \in \mathsf{Real}_+]\ ]; \\ & \quad [I]; \\ & \quad \mathbf{do}\ [\ \llbracket\ 1 \le i \le n \cdot \mathcal{T}^m(i)\ ]\ \mathbf{od} \\ & \mathbf{end}, \end{array}$$

$$\begin{array}{ll} & \mathcal{T}^m(i) \\ = & \\ & state[i].now = sl \wedge c_a[i].now = P[i] + ofs[i].now \wedge \neg ok[i].now \\ & \to c_a[i] :- (\lambda t \cdot t - now)\ ;\ state[i] :- (\lambda t \cdot wt)\ ;\ q[i] :- pr[i] \\ \llbracket & ok[i].now \wedge state[i].now = wt \\ & \to c_e[i] :- (\lambda t \cdot t - now)\ ;\ state[i] :- (\lambda t \cdot ex) \end{array}$$

$$
\begin{array}{ll}
[\!] & ok[i].now \wedge state[i].now = ex \wedge c_e[i].now = E[i] \\
& \to c_e[i] :- (\lambda t \cdot 0) \,;\, c_p[i] :- (\lambda t \cdot 0) \,;\, q[i] :- (\lambda t \cdot 0) \,; \\
& \quad [ofs[i] :- x' \mid \forall t \geq now \cdot x'.t \in \mathsf{Real}_+] \,;\, state[i] :- (\lambda t \cdot sl) \\
[\!] & \neg ok[i].now \wedge state[i].now = ex \wedge c_e[i].now < E[i] \\
& \to c_e[i] :- (\lambda t \cdot c_e[i].now) \,;\, c_p[i] :- (\lambda t \cdot c_p[i].now + t - now) \,; \\
& \quad state[i] :- (\lambda t \cdot pt) \\
[\!] & ok[i].now \wedge state[i].now = pt \\
& \to c_e[i] :- (\lambda t \cdot c_e[i].now + t - now) \,;\, c_p[i] :- (\lambda t \cdot c_p[i].now) \,; \\
& \quad state[i] :- (\lambda t \cdot ex),
\end{array}
$$

$$
\begin{array}{l}
\mathcal{S}ched\,(ok : \mathsf{array}[1..n] \text{ of } (\mathsf{Real}_+ \to \mathsf{Bool}), \\
\quad state : \mathsf{array}[1..n] \text{ of } (\mathsf{Real}_+ \to \{sl, wt, ex\}), \\
\quad c_a, c_e, c_p : \mathsf{array}\ [1..n] \text{ of } (\mathsf{Real}_+ \to \mathsf{Real}_+), \\
\quad pr, q : \mathsf{array}[1..n] \text{ of } (\mathsf{Real}_+ \to \mathsf{Nat}))
\end{array}
$$

$$
\begin{array}{ll}
= & \textbf{begin } \mathsf{Init}'; \\
& \quad \textbf{do } [\ [\!]\ 1 \leq i \leq n \cdot \\
& \qquad \neg ok[i].now \wedge pr[i].now = Max(q.now) \wedge \\
& \qquad (\forall j \neq i \cdot (state[j].now = sl \wedge c_a[j].now < P[j]) \vee \\
& \qquad state[j].now = wt \vee state[j].now = pt) \wedge \\
& \qquad state[i].now = wt \wedge c_a[i].now \leq D[i] - R[i] \\
& \qquad \to ok[i] :- (\lambda t \cdot \mathsf{true}) \\
& \quad\ [\!]\ ok[i].now \wedge state[i].now = sl \\
& \qquad \to ok[i] :- (\lambda t \cdot \mathsf{false}) \\
& \quad\ [\!]\ ok[i].now \wedge pr[i].now \neq Max(q.now) \wedge \\
& \qquad state[i].now = ex \wedge c_e[i].now < E[i] \\
& \qquad \to ok[i] :- (\lambda t \cdot \mathsf{false}) \\
& \quad\ [\!]\ \neg ok[i].now \wedge pr[i].now = Max(q.now) \wedge \\
& \qquad (\forall j \neq i \cdot (state[j].now = sl \wedge c_a[j].now < P[j]) \vee \\
& \qquad state[j].now = wt \vee state[j].now = pt) \wedge \\
& \qquad state[i].now = pt \wedge c_a[i].now - (c_e[i].now + c_p[i].now) \leq D[i] - R[i] \\
& \qquad \to ok[i] :- (\lambda t \cdot \mathsf{true}) \\
& \quad\ ] \\
& \quad \textbf{od} \\
& \textbf{end}
\end{array}
$$

Note that Init' of $\mathcal{S}ched$ is the initialization statement of $\mathcal{TS}$, without the assignment of variable ofs, which is local to $\mathcal{TS}$.

6.5 Non-Preemptible Task Model

Assume that we are given n periodic, non-preemptible real-time tasks. A simple model of a task $\mathcal{T}(i)$ that belongs to such a collection is as follows.

$$
\begin{array}{rcll}
\mathcal{T}(i) & \triangleq & & state[i].now = sl \wedge c_a[i].now = P[i] \\
& & & \rightarrow c_a[i] :- (\lambda t \cdot t - now) \,;\, state[i] :- (\lambda t \cdot wt) \,;\, UT \\
& & [\!] & state[i].now = wt \\
& & & \rightarrow c_e[i] :- (\lambda t \cdot t - now) \,;\, state[i] :- (\lambda t \cdot ex) \,;\, UT \\
& & [\!] & state[i].now = ex \wedge c_e[i].now = E[i] \\
& & & \rightarrow c_e[i] :- (\lambda t \cdot 0) \,;\, state[i] :- (\lambda t \cdot sl) \,;\, UT
\end{array} \tag{6.21}
$$

Distinctly from the preemptible model, each non-preemptible task can be in one of three possible states, sl (*sleeping*), wt (*waiting for the CPU*), and ex (*executing*). The clock c_a measures the time between two consecutive arrivals of each task, respectively. We record the execution time of each task by clock vector c_e.

The tasks are periodic, therefore they arrive strictly at $P[i]$, respectively. In the initial model (6.21), if the task $\mathcal{T}(i)$ is waiting for the CPU, it could start executing right away. Alternatively, given the fact that the real-time system consists of n such tasks and that the choice among the waiting ones is nondeterministic, $\mathcal{T}(i)$ could also wait an arbitrary time before starting execution. Since the guard of the second action holds, the task eventually changes its state to ex, and clock $c_e[i]$ is reset. Upon completion of execution, when $c_e[i].t$ is $E[i]$ (worst-case execution time), the respective task returns to sleep.

In the next section, we proceed to the incremental construction of an EDF scheduled system. However, since the steps are similar to the ones of the previously analyzed fixed-priority case, we will just highlight the key points of the construction method, underlying their specificity.

6.6 Dynamic-Priority Scheduling: The EDF Algorithm

Recall from the beginning of the chapter that for *dynamic* priority schedulers, the priority of a task is recomputed at run-time. If scheduled by the *Earliest-Deadline-First* algorithm, the runnable tasks are executed in the order determined by the absolute deadlines of the processes, the next process to run being the one with the shortest (nearest) deadline.

The real-time system model employing non-preemptible tasks is, initially, close to the one given by (6.5), with the difference that now we quantify the choice of n non-preemptible tasks of the kind modeled by (6.21). We also remove the initial assignment of task priorities, since in this case the priorities are decided online.

$$
\begin{array}{rcl}
\overline{\mathcal{RTS}} & \triangleq & \textbf{begin var } start, now : \mathsf{Real}_+, \\
& & \quad state : \mathsf{array}\ [1..n]\ \textbf{of}\ (\mathsf{Real}_+ \rightarrow \{sl, wt, ex\}), \\
& & \quad c_a, c_e : \mathsf{array}\ [1..n]\ \textbf{of}\ (\mathsf{Real}_+ \rightarrow \mathsf{Real}_+) \bullet \\
& & \quad state :- (\lambda t \cdot wt) \,;\, c_a :- (\lambda t \cdot t) \,;\, c_e :- (\lambda t \cdot 0) \,;\, UT \,; \\
& & \textbf{do}\ [\ [\!]\ 1 \leq i \leq n : \{\mathcal{T}(i).I\} \,;\, \mathcal{T}(i)\ ]\ \textbf{od} \\
& & \textbf{end}
\end{array} \tag{6.22}
$$

Note that we initialize the system similarly to the DM real-time system: at time 0, all tasks are in the ready state, waiting for the permission to execute.

6.6.1 Enforcing Conditions for Correct Scheduling

Enforcement of the correctness conditions proceeds in the same way as for the fixed-priority case.

We model the timeliness condition as follows.

$$\begin{aligned} q_t \;\triangleq\; & \forall i \cdot \forall t \in [start, now) \cdot \\ & \quad state[i].start = ex \Rightarrow c_a[i].t - c_e[i].t \leq D[i] - E[i] \end{aligned}$$

Along the lines described in section 6.2.1, q_t is enforced if we find an action system description, starting from (6.22), such that some predicate $I_t \Rightarrow q_t$ is an invariant of the transformed action system. Condition I_t is the following:

$$\begin{aligned} I_t \;\equiv\; & \forall i \cdot \forall t \in [start, now) \cdot \\ & \quad state[i].start = ex \Rightarrow \\ & \quad\quad (state[i].t = ex \wedge c_a[i].t - c_e[i].t \leq D[i] - E[i] \wedge \\ & \quad\quad\quad c_a[i].t = c_a[i].start + t - start) \end{aligned}$$

The EDF policy predicate is:

$$\begin{aligned} I_{pol} \;\equiv\; & \forall i \cdot \forall t \in [start, now) \cdot \\ & \quad (state[i].start = ex \Rightarrow \\ & \quad\quad (\forall j \neq i \cdot (state[j].start = sl \wedge c_a[j].start < P[j]) \vee \\ & \quad\quad\quad (state[j].t = wt \wedge D[j] - c_a[j].start > D[i] - c_a[i].start))) \\ & \wedge (state[i].start = s \Rightarrow state[i].t = s),\; s \in \{sl, wt\} \end{aligned}$$

The condition I_{pol} requires that any task in state ex remains in that state during the interval $[start, now)$; besides, a task has either been the only task waiting, or if other tasks have also been simultaneously waiting, the task that has started executing has the shortest absolute deadline (that is, the smallest difference between the relative deadline $D[i]$ and the arrival time) of them all.

Next, we need to refine the initial action system (6.22), in the context given by the computed weakest precondition $\{\mathcal{T}(i).I\}$, such that I_t, and I_{pol} become invariants of the refined system. We enforce the above predicates on task model (6.21), by propagating $\{\mathcal{T}(i).I\}$. For brevity, we skip the refinement process, but we present its result. After having carried out enforcement, we get the following

transformed model, with a strengthened guard:

$$
\begin{array}{rll}
& \mathcal{T}(i)^f & \\
= & state[i].now = sl \wedge c_a[i].now = P[i] & \\
& \rightarrow c_a[i] :- (\lambda t \cdot t - now) \; ; state[i] :- (\lambda t \cdot wt) \; ; UT & \\
[\!] & state[i].now = wt \wedge c_a[i].now \leq D[i] - E[i] \wedge & \\
& (\forall j \neq i \cdot (state[j].now = sl \wedge c_a[j].now < P[j]) \vee & (6.23) \\
& (state[j].now = wt \wedge D[j] - c_a[j].now > D[i] - c_a[i].now)) & \\
& \rightarrow c_e[i] :- (\lambda t \cdot t - now) \; ; state[i] :- (\lambda t \cdot ex) \; ; UT & \\
[\!] & state[i].now = ex \wedge c_e[i].now = E[i] & \\
& \rightarrow c_e[i] :- (\lambda t \cdot 0) \; ; state[i] :- (\lambda t \cdot sl) \; ; UT &
\end{array}
$$

Let us exemplify in the next paragraph, how we can confirm or deny the schedulability of a simple three-task real-time system, scheduled by EDF.

6.6.2 Validating the EDF Scheduled System

As tasks are periodically released, the schedule has an infinite length. Due to the periodicity of tasks, the whole schedule is also periodic of period $P = LCM(P[1], \ldots, P[n])$, where LCM is the least-common-multiple of the tasks periods (feasibility interval). Then, the feasibility (or schedulability) analysis problem can be solved by simulation of the scheduling algorithm (in our case EDF) until the task set is in the periodic state [52]. This implies further that, if all tasks complete by their deadlines during the feasibility interval, there will be no unpleasant surprises in the future.

Example. We consider building an EDF schedule for the task set of Figure 6.4.

The correct model of our three task real-time system is an instantiation of the generic model (6.22), where each constituent task is of the form given by (6.23). In order to validate this model, we use the simulation technique. As outlined above, we need to simulate (6.22), up to the least-common-multiple of task periods, which is $t = 60$, in our case. For this purpose, we use our CAS symbolic simulator.

As previously, one has to translate the guards and action bodies of the three-task model, in the language of Mathematica. Then, the main routine goes through the guards, evaluates their boolean expressions, and based on the results, computes the minimum moment of time as the next now.

Task	Period	Deadline	Execution Time
T1	20	16	3
T2	15	15	3
T3	10	10	4

Figure 6.4: Three Periodic Tasks.

Implementation of the Model. In the Mathematica implementation, we have modeled the possible values, sl, wt, ex, of the discrete-valued time variables

$state1, state2, state3$, of each task, as $0, 1, 2$, respectively. The continuous-valued time variables are the arrival clocks, denoted by $ca1, ca2, ca3$.

The simulation time limit $t_{max} = 60$ was supplied to the tool, and so were the task parameters, $D[i], E[i], P[i]$, as given in Figure 6.4.

For example, the $sl \rightarrow wt$ (that is, $0 \rightarrow 1$) action guard and body, of the first task, are implemented by:

```
solution = InequalitySolve[
                state1 [t] == 0 &&
                ca1 [t] == P1 &&
                t >= start && t <= EndTime, t
                ]
```

and

```
ca1 [t_]     =  t − now;
state1 [t_]  =  1;
start        =  now
```

The guard of the action that schedules each task, respectively, is implemented by using task comparison of absolute deadlines.

Surely, for larger number of tasks, we need to first refine the model in order to get a more efficient representation (as we have done for the DM scheduled system). Then, we can use the Mathematica function Min, on the list variable that contains the differences $\mathsf{D[i] - ca[i].now}$, $\forall \mathsf{i} \in [1..3]$, of the current waiting tasks.

The actual implementation of the alternative solution mentioned above (tailored to our example) is as follows. We model the queue of waiting tasks as the variable $\mathsf{q[t_]}$ and we initialize it to: $\mathsf{q[t_] = \{D1, D2, D3\}}$. Next, we introduce a list containing all three arrival clocks: $\mathsf{ca = \{ca1[t], ca2[t], ca3[t]\}}$, where each clock is modeled as the respective function of time $\mathsf{cai[t_]}$, $\mathsf{i} \in \{1..3\}$. The list with the differences $\mathsf{D[i] - ca[i].now}$ is implemented by the following structure: $\mathsf{list = Table[\{\ \}, \{i, Length[q[t]]\}]}$. Since the tasks are all waiting at time 0, we fill the list accordingly: $\mathsf{list = Insert[list, Simplify[q[i] - ca[i]], i]}$.

Whenever a task $\mathcal{T}(i)$ has arrived, it's deadline is inserted in q, at position i. Also, the difference between its deadline and its arrival clock is inserted into the corresponding list. If a task has completed execution, it is dropped from q. After that, one needs to shift the positions of the remaining tasks, accordingly, however we do not show this part in the code excerpt presented below. When the scheduler decides the maximum priority task, it computes the minimum of all current elements of list.

Below, we give the Mathematica code of the above actions, for the first task only.

- **Body of action sl → wt**:

$$\begin{aligned} \mathsf{ca1[t_]} &= \mathsf{t - now;} \\ \mathsf{state1[t_]} &= \mathsf{1;} \\ \mathsf{q[t_]} &= \mathsf{Insert[q[t], D1, 1]} \\ \mathsf{list} &= \mathsf{Insert[list, Simplify[q[1] - ca[1]], 1]} \\ \mathsf{start} &= \mathsf{now} \end{aligned}$$

- **Guard of action wt → ex**:

```
solution = InequalitySolve[
                    state1[t] == 1 &&
                    ca1[t] <= D1 − E1 &&
                    D1 − ca1[t] == Min[list, {i, Length[list]}]
                    t >= start && t <= EndTime, t
                    ]
```

- **Body of action ex → sl**:

$$\begin{aligned} \mathsf{ce1[t_]} &= \mathsf{0;} \\ \mathsf{state1[t_]} &= \mathsf{0;} \\ \mathsf{q[t_]} &= \mathsf{Drop[q[t], \{1\}]} \\ \mathsf{Length[q[t]]} &= \mathsf{Length[q[t]] - 1} \\ \mathsf{start} &= \mathsf{now} \end{aligned}$$

The implemented model has three guarded actions that abort the simulation in case any of the deadlines is missed. For example, the guard of such action that checks for overrun of deadline, for task 1, is as follows:

```
solution = InequalitySolve[
                    state1 [t] == 2 &&
                    ce1 [t] == E1 &&
                    t − start > D1 &&
                    t >= start && t <= EndTime, t
                    ]
```

and its body

```
Print["Stop, task1 missed deadline", now];
Abort[ ]
```

Simulation results. The essential information delivered by the simulation procedure contains: a list of time moments (now) at which an action has been executed in the model, a corresponding list of actions (which we omit here), and lists with symbolic values for the state of each task at each particular transition time, and for the clocks $ca1, ca2, ca3$ as (continuous) linear functions of time. Figure 6.5 shows the mentioned symbolic lists.

One can see, in Figure 6.5, based on the values of variables $state1$, $state2$, $state3$, at each moment now given in the corresponding now list, the order in which tasks have been executed. The tasks share their critical instant, at time 0. Thus, initially, all of them are in $state = 1$, waiting for the CPU. So, as expected, the first to execute is task 3, since at time 0 it has the earliest deadline, after which task 2 and task 1 follow. Notice that, at time $now = 60$, the simulation scenario starts repeating.

At the end of the simulation period, we also get the functions of time, $state1$, $state2$, $state3$, $ca1$, $ca2$, $ca3$, represented as graphs. The plotted functions are presented in Figures 6.6 - 6.8.

```
now list :
{0,0,4,10,10,13,13,15,16,16,20,20,20,20,23,23,27,30,30,30,33,33,37,40,40,40,43,45,45,
49,50,50,53,53,57,57,60,60}

state1 : {1,1,1,1,1,1,2,2,0,0,1,1,1,1,1,1,1,1,1,2,0,0,0,1,1,1,1,1,1,1,1,2,0,0,0,0,0}

state2 : {1,1,1,1,2,0,0,1,1,1,1,1,1,2,0,0,0,1,1,1,1,1,1,1,1,2,0,1,1,1,1,1,1,1,1,2,0}

state3 : {1,2,0,1,1,1,1,1,1,2,2,0,1,1,1,2,0,0,1,1,1,2,0,0,1,1,1,1,2,0,1,1,1,2,0,0,0}

ca1 :

{t,t,t,t,t,t,t,t,t,t,-20+t,-20+t,-20+t,-20+t,-20+t,-20+t,-20+t,-20+t,-20+t,-20+t,
-20+t,-20+t,-20+t,-40+t,-40+t,-40+t,-40+t,-40+t,-40+t,-40+t,-40+t,-40+t,-40+t,-40+t,
-40+t,-40+t,20}

ca2 :

{t,t,t,t,t,t,t,-15+t,-15+t,-15+t,-15+t,-15+t,-15+t,-15+t,-15+t,-15+t,-15+t,-30+t,
-30+t,-30+t,-30+t,-30+t,-30+t,-30+t,-30+t,-30+t,-30+t,-45+t,-45+t,-45+t,-45+t,-45+t,
-45+t,-45+t,-45+t,-45+t,15}

ca3 :

{t,t,t,-10+t,-10+t,-10+t,-10+t,-10+t,-10+t,-10+t,-10+t,-10+t,-20+t,-20+t,-20+t,
-20+t,-20+t,-20+t,-30+t,-30+t,-30+t,-30+t,-30+t,-30+t,-40+t,-40+t,-40+t,-40+t,-40+t,
-40+t,-50+t,-50+t,-50+t,-50+t,-50+t,-50+t,10}
```

Figure 6.5: The symbolic lists for now, $state1$, $state2$, $state3$, $ca1$, $ca2$, $ca3$.

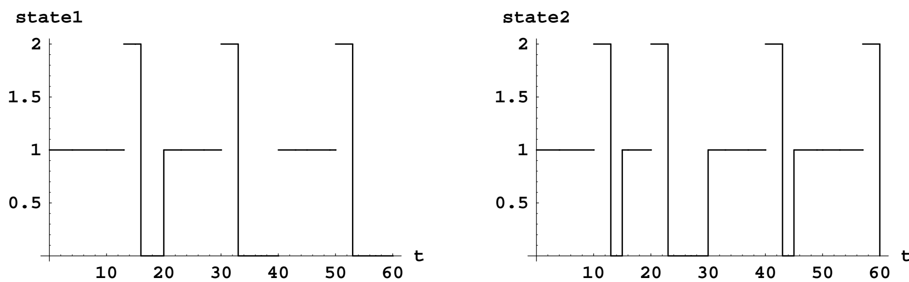


Figure 6.6: The graphs of $state1$, $state2$.

In Figure 6.9, we present the timing diagram of our constructed schedule. The schedule is based on the results of the symbolic simulation; we have just represented the task execution order and the specific transition times that the tool has

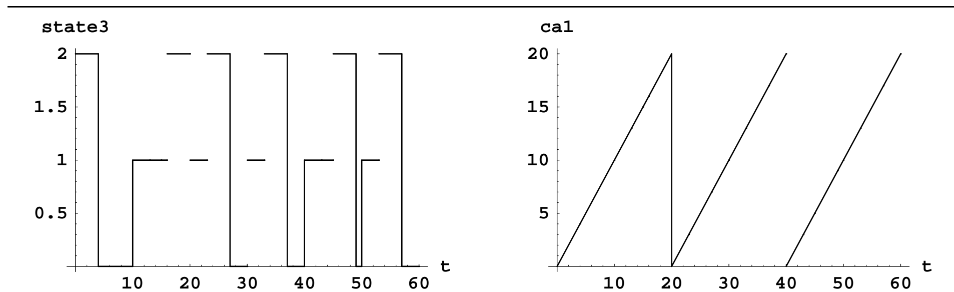


Figure 6.7: The graphs of $state3$, $ca1$.

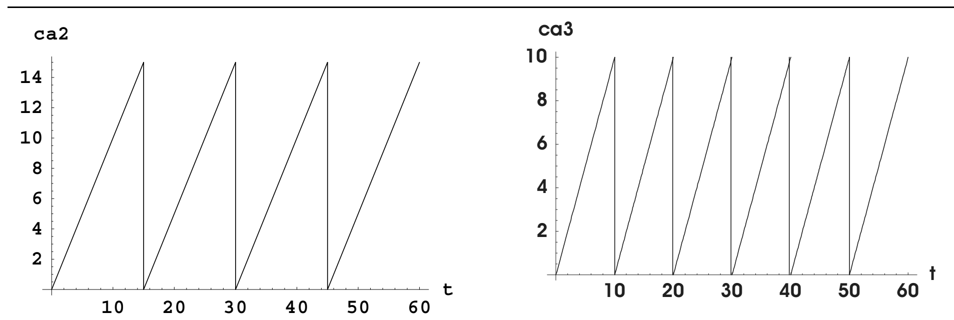


Figure 6.8: The graphs of $ca2$, $ca3$.

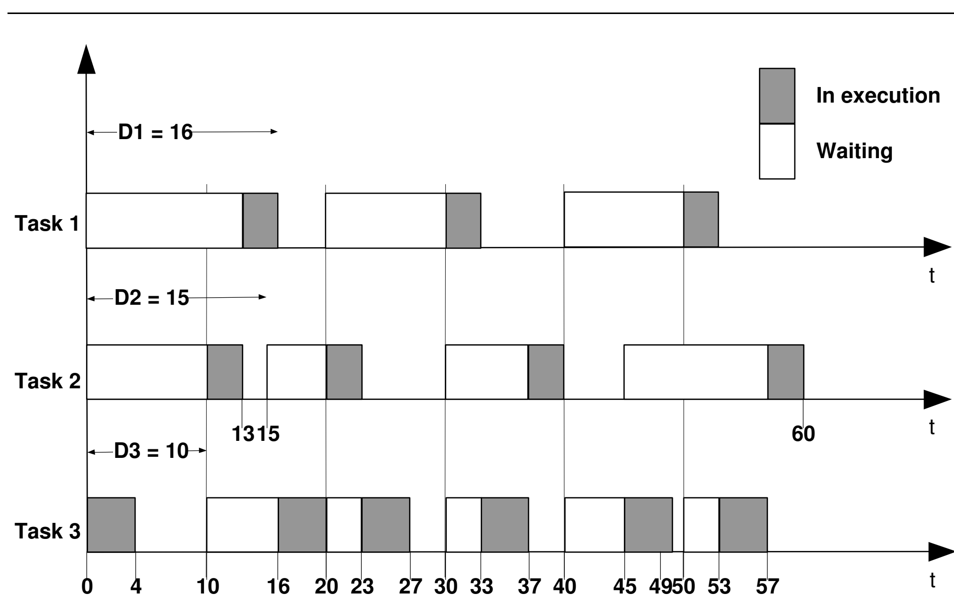


Figure 6.9: The three tasks' schedule.

computed, graphically, to aid visualization of the solution.

Because all the imposed timing constraints hold during the feasibility interval, assuming worst-case execution times of participating tasks, we can declare the task set of the example, schedulable.

6.7 Summary and Related Work

We have presented a new method for the incremental construction of scheduled systems, within the correctness-preserving framework of continuous action systems and refinement calculus. Unlike most of the previous related work, our development process starts with a nondeterministic conjunctive specification, and applies refinement rules of propagating context assertions, in order to enforce the required schedulability, mutual exclusion and scheduling policy conditions.

The main schedulability goal of a real-time task set is to provide a guarantee that the deadlines are never missed at run-time. Consequently, we model such a condition as a property of the model, more concretely, an *always* temporal property. This can be enforced into the initial incomplete task model, by means of Lemma 6, which we have introduced to show that enforcing safety properties on CAS models reduces to being able to prove certain invariants.

For the DM scheduling algorithm, we carry out successive refinements, until we reach an efficient implementation of the correct-by-construction, yet abstract model. Next, we continue the refinement process by applying the decomposition theorem of Sekerinski and Sere [141], which provides us with a two-module implementation. Thus, we have managed to extract the real-time scheduler as a separate module, which can turn into an advantage if further transformations are needed.

Next, we have applied our technique also to the EDF scheme, still under the umbrella of Lemma 6, which gives us sufficient conditions for enforcing the timeliness and policy predicates. By simulating the EDF model of a simple set of three real-time tasks, in Mathematica, we get symbolic lists of transition time moments, and of variable values. As a plus, one can visualize the timed behavior of all the discrete and continuous functions of the model, as graphs. Based on the computed data, we have actually constructed the scheduling solution, as a timing diagram.

Related Work. Formal approaches for constructing real-time schedulers have been recently applied by Kwak et al. [113], and Altisen et al. [4]. Kwak et al. propose symbolic bisimulation algorithms that can be used for deciding the schedulability of a collection of real-time tasks. For the same purpose, Altisen et al. use algorithms based on the controller synthesis paradigm. A major disadvantage of these approaches is the practical high complexity of the algorithms. Furthermore, in both papers, the authors use two different formal frameworks in order to perform the schedulability analysis of the task set. One formalism is used for modeling purposes, whereas a different one is employed for giving semantics to the respective model.

Although the semantics of CAS is given in terms of ordinary action systems, we also need at some point to employ two languages. In order to use automation, for the EDF algorithm, we are forced to translate the real-time action system into the language of Mathematica.

As an alternative solution to model-checking algorithms, we have solved the problem of scheduler construction, generally. The analysis of any particular set of real-time tasks comes down to an instantiation of our generic correct-by-construction models.

The closest work to ours is due to Altisen, Gößler, and Sifakis [5], where fixed point computation algorithms are combined with the incremental application of priority rules that restrict the initial behavior of the real-time model. The approach is very general; it provides a rich priority model, and can handle task sets with complex timing characteristics. However, the construction method does not allow the separation of the scheduler as an actual component. Initially, the scheduler is being represented by uncontrollable transitions in timed automata models. Nevertheless, the concept of task *urgency* used by the authors is non-existing in our approach.

Timed automata [9] have also been used for solving non-preemptive job-shop scheduling problems [2, 100]. Schedules are computed based on the traces resulted out of reachability analysis of locations that specify the schedulability property.

Alur and Henzinger have specified preemptive schedulers as hybrid automata [14]. Their methods use model checking algorithms for verifying safety and liveness properties.

Braberman and Felder have proposed timed automata based schedulability verification of preemptive schedulers [50]. The authors have specified the timing constraints as minimum or maximum distances between events. As such, they embrace a conservative perspective, as we also do in this chapter.

The integration of real-time scheduling theory with program refinement has also been studied by Fidge et al. [76]. Moreover, Fidge and Wellings model real-time systems by combining Z specifications with action systems [77]. In comparison, we handle the task within the action systems framework alone. The authors use a global variable that measures absolute time, and a "tick action" for advancing time. A step further is taken by Fidge, Hayes and Watson, who introduce a *deadline* command, which allows for the specification of arbitrary deadlines that must be met at run-time [75]. A summary on the advances made in developing a *refinement calculus for real-time programs* is presented by Hayes [85]. Real-time refinement laws targeting timing constructs such as delays and deadlines, as well as rules for handling infinite iterations are proposed.

Real-time scheduling of distributed action systems has been analyzed by Kurki-Suonio et al. [112]. The idea of constructing the real-time concurrent system by enforcing specific real-time properties via refinement has been investigated. Due to inherent limitations in specifying the real-time requirements, the scheduling problem has been revisited later by Kurki-Suonio and Katara [111]. This time, real-

time properties have been specified in TLA [1]. Safety conditions are imposed by strengthening action guards, and deadlines are introduced through superposition refinement. However, no actual schedules are computed.

As far as automation is regarded, TAXYS [64] and TIMES [17] are two mature tools that support the design and analysis of real-time embedded systems. Also, the theorem prover STeP has been applied to the deductive verification of real-time systems [46]. Nonetheless, to our knowledge, checking EDF schedulability by simulating a formal model that is guaranteed to be correct, provided that the tasks are schedulable, has not been applied before.

We do not claim that our refinement-based method is a panacea, we just believe it might serve as a useful alternative approach to model-checking algorithms. A combination of the two well-known methods of theorem-proving and model-checking could perhaps provide the designer with the best developing means.

Let us analyze next how can we use both angelic and demonic behavior of statements, in order to construct sequential game-like reactive programs.

Chapter 7

Controller Synthesis for Discrete Systems

A control system is usually a reactive system, made of a computer-based *controller* that observes a *plant*. We have already seen in previous examples that the plant's behavior is adjusted according to the control actions issued by the logic controller, as a result of its reaction to certain changes in the environment. The control events thus modify the system execution, such that some desired property holds. The construction of a correct controller is thus crucial.

A viable controller construction method is known in literature as *controller synthesis*. Synthesis is equivalent to computing the most general model of a controller that satisfies the respective requirements. Therefore, it is helpful to start with a nondeterministic, high-level model of the controller that abstracts away from control implementation details. The result of carrying out synthesis steps is a decrease in controller's nondeterminism, such that in the end all possible transitions that could lead to unsafe states are eliminated.

We are interested in synthesizing controllers for *invariance* and *reachability*. The invariance controller has to keep the discrete system within a so called "good" set of states, whereas the reachability controller has to guide the system into an intended set of states, in finite time. In the latter case, we focus on the synthesis of a particular case of reachability controllers, with possible application to the design of *fault-tolerant* systems that accept k faults ($k \geq 1$), and especially *fail-safe* systems. The former maintain their integrity, for a limited time, while accepting one or more errors in their operation. On the other hand, fail-safe systems are able to terminate in a safe state, if they have suffered a serious damage (a *failure*) and need to halt their operation [52].

Many studies deal with controller synthesis by applying symbolic model checking algorithms to restrict the inputs of the controlled system so as to eliminate undesired computations [18, 99]. Backward analysis is often employed, as being most useful. Unfortunately, it is also an expensive algorithm, in terms of memory

resources. Moreover, there are situations in which the systems are infinite state, thus they need to be abstracted in such a way that the properties of the abstracted finite model hold in the concrete model too [106, 133].

Being motivated by the importance and usefulness of controller synthesis, and challenged by its difficulty, we have tried to find an alternative deductive solution to this problem [26, 59]. Our solution could, for instance, be employed when the traditional model-checking approach fails. Furthermore, the method is applicable as such for controller construction of either finite or infinite state discrete systems.

The behavior of a discrete control system can be adequately modeled by a repetitive sequence of actions carried out by the controller, and the disturbance, respectively. We take advantage of the dual nondeterminism in our modeling language, and identify the control system with a two-player game having the *angel* and the *demon* as players. By identifying the angel with the controller and the demon with the disturbance, we are able to exploit further the mentioned dualism, while playing the angel against the demon. Having defined the system model, we specify the requirement, next. It is a temporal property, which should be enforced by the controller (that is, the angel) during system execution, in spite of the hostile actions of the disturbance.

The dilemma that rises within this context is whether the angel can face the challenges of the demon, and win the game by ensuring the intended overall system behavior.

We focus, in the current chapter, upon solving this dilemma. To check whether the temporal property that specifies the reachability control can be enforced by the angel, we propose an inference rule that reduces enforcement to correctness reasoning. Moreover, in case of a positive outcome, we show how to extract the angelic winning strategy. We do this through a correctness-preserving transformation, that is, assertion propagation. Two illustrative examples, a memory buffer controller and a data-processing system serve as the case-studies.

7.1 Game Tree Semantics of Action Systems

Back and von Wright have developed a semantics for action systems that generalizes the traditional behavioral action systems semantics [37]. We present here the basic notions, which we use in the rest of the chapter.

An (infinite) *action game* is a pair (p, T) where p is the *initial predicate* and T is the *action* (a monotonic predicate transformer). An execution of the action game starts from some (demonically chosen) initial state in p, and then the action T is repeatedly executed atomically, as a game between the angel and the demon.

By a *game tree* we mean a tree with *predicate nodes* (labeled with predicates, that is, sets of states, from which the demon is to select a state) and state nodes (labeled with states, from which the angel is to decide what the next predicate is), in strict alternation, where

- the root node is a predicate node,
- a predicate node is labeled with a predicate p if and only if for each $\sigma \in p$, there is exactly one child (a state node) labeled with σ and
- leaf nodes occur as follows:
  - A leaf node labeled $\top$ follows after any predicate node containing the empty predicate false (this models miraculous termination), and
  - A leaf node labeled $\bot$ can occur as the only node following any state node (this models abortion).

A game tree may be infinitely branching and it may contain infinite paths. Figure 7.1 illustrates a game tree. Here, state nodes (states) are shown as bullets (•) and predicate nodes (predicates) as circles (∘).

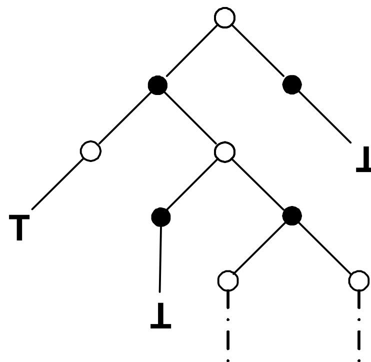


Figure 7.1: Game Tree.

We will now describe how an action game $\mathcal{A} = (p, T)$ generates a game tree, when the action T is thought of as executed as a two-step game, according to its normal form. This means that in any given state σ

1. the angel first chooses a predicate p such that $T.p.\sigma$ holds (if there is no such predicate then the execution has aborted), and
2. the demon then chooses a state σ' in p (if p is empty, then the execution has become miraculous)

and then step 1 is repeated for state σ', etc. Thus, the possible executions of the action system $\mathcal{A} = (p, T)$ give rise to a game tree which is exactly the tree generated by T with root p. This tree is the *game tree* $\mathcal{G}(\mathcal{A})$ generated by the *action game* $\mathcal{A}$.

We recall the fact that the state space $\sum$ is extended with two elements (which we call *improper states*): $\perp$ (standing for abortion or nontermination) and $\top$ (standing for a miracle). The *augmented state space* $\sum_+ = \sum \cup \{\perp, \top\}$ is a complete lattice when ordered by the following *approximation relation*:

$$\sigma \preceq \sigma' \ \triangleq\ \sigma = \perp \vee \sigma = \sigma' \vee \sigma' = \top$$

A *behavior* is a sequence $t = (t_0, t_1, \ldots)$ of states (including $\perp$ and $\top$). Behaviors can be

- *infinite*, containing only proper states,
- *aborting*, that is, ending with an infinite repetition of $\perp$, or
- *miraculous*, that is, ending with an infinite repetition of $\top$.

A *strategy f (for the angel) for game tree G* is a rule that resolves all the angelic choices in the game tree G. Thus, $f.G$ is a tree where every state node has exactly one child.

Every path in a game tree gives rise to a behavior; we simply remove every predicate node in the path, and if the path ends in $\perp$ or $\top$, we repeat that final improper state indefinitely. Thus, the tree $f.G$ can also be interpreted as a set of behaviors (state sequences): $t \in f.G$ holds if and only if t is the result of removing all predicate nodes from a complete path in the tree $f.G$ and repeating the final (improper) state indefinitely if the path is finite.

Temporal Properties. Let t be a behavior and q a predicate. Back and von Wright [37] define the *always* ($\Box$) temporal property over discrete behaviors t, as follows:

$$t \models \Box\, q \qquad \text{iff} \qquad (\forall i \cdot t_i \in q)$$

The above formula describes what it means for the behavior t to *satisfy* the property $\Box\, q$.

By extending the satisfaction of temporal formulas to game trees, we have that $G \models \phi$ holds if and only if there is a strategy f over G for the angel such that $t \models \phi$, for every $t \in f.G$. Thus, $G \models \phi$ means that the angel can *enforce* the property ϕ, that is, make sure that it becomes true.

An action game $\mathcal{A} = (p, T)$ is said to satisfy a temporal property ϕ if $\mathcal{G}(\mathcal{A}) \models \phi$. Assuming that predicate p and monotonic predicate transformer T are given, the following formula characterizes *weakest temporal preconditions* (the weakest predicate such that $\mathcal{G}(p, T)$ satisfies ϕ) for *always* properties:

$$wtp(T,\, \Box\, q) \ \triangleq\ (\nu x \bullet q \cap T.x)$$

7.2 Generic Approach to Synthesis

Given the structure of a control system, it follows naturally that it can be modeled as a game between two players, the controller and some disturbance. We assume that the behavior of the disturbance is hostile, thus we would like the controller to guarantee the requirements despite the action of the disturbance. We call the controller *the angel*, and the disturbance *the demon*. During the game, the goal of the angel is to force the system to remain inside a certain desired subset of the state space, whereas the demon's goal is to force the system to leave this same subset.

Let us assume that the *discrete system* is modeled by an action system of the form

$$Sys(y : T_y) \;\triangleq\; \mathbf{begin\ var}\ x : T_x \bullet\ Init\,;\ \mathbf{do}\, g \rightarrow A\,;D\,\mathbf{od}\ \mathbf{end} \tag{7.1}$$

Here, statement A contains angelic choices and D demonic ones. The values of the variables are chosen sequentially, first by the controller and then by the disturbance. Consequently, the composition $A\,;D$ models a two-player game where each player has complete information about the moves of its rival.

Before proceeding to synthesis, one needs to specify the control system and its requirements. Here, the *controller* is defined by an angelic nondeterministic assignment, as follows:

$$A = \{x := x' \mid b_a\}$$

Dually, the behavior of the *disturbance* is described by a demonic nondeterministic assignment:

$$D = [x := x' \mid b_d]$$

As previously mentioned, we focus on constructing controllers for invariance and reachability. Hence, in the first case the *requirement* is encoded as a *safety* property, whereas in the second case it is a *liveness* property.

Safety is an "*always*" property and has the form $\Box\, q$, where q is a set of states (predicate). The goal of the controller is to continually observe the plant, and force control events at appropriate times, such that the plant always remains within the safe set of states, q. Such controllers are called *invariance* controllers.

Traditional liveness properties are modeled as "*eventually*" ($\Diamond q$) temporal properties. However, here we look at a variant of liveness property, which is defined later in this chapter. Controllers that can enforce liveness properties are called *reachability* controllers. Their duty is to guide the system towards the intended reachable set, q, in finite time.

Assuming that at each round of the game, sequential angelic and demonic choices determine the next state of the game, we can intuitively split the synthesis problem into two sub-problems:

(a) Checking whether the angel can enforce the safety or liveness property. In our case, this process reduces to correctness proofs.

(b) In case the correctness conditions hold, it follows that the angel has a winning strategy. Therefore, extracting the angelic winning strategy is the next step. This second step is equivalent to the actual controller construction.

7.3 Synthesis of Controllers for Invariance

7.3.1 Enforcing the Safety Property

As pointed out, designing a controller for invariance implies the specification of some safety property that should be enforceable by the angel during system execution. We express this property as an "*always*" ($\Box$) temporal property.

In the following, we show how we can compute the precondition for the angel to enforce the property $\Box\, q$ in the action system Sys, given by (7.1). Applying the result proved by Back and von Wright [38], to our case, we get the following:

$$p \,\{\!| \mathbf{do}\; g \rightarrow A \,;\, D \;\mathbf{od} \,|\!\}\, \Box\, q \;\;\equiv\;\; p \subseteq (\nu X \bullet \{q\} \,;\, [g] \,;\, A \,;\, D \,;\, X).\mathsf{false} \qquad (7.2)$$

Formula (7.2) shows that we can reduce the question of whether the "*always*" temporal property can be enforced for an action system, to the question of whether a certain goal can be achieved. In this case, the goal false cannot really be established, so success can only be achieved by miraculous termination, or by non-termination caused by the demon.

As an immediate consequence of equation (7.2), we get the correctness rule for safety, at the predicate level:

$$p \,\{\!| \mathbf{do}\; g \rightarrow A \,;\, D \;\mathbf{od} \,|\!\}\, \Box\, q \;\;\equiv\;\; p \subseteq (\nu x \bullet q \cap (\neg g \cup A.(D.x))) \qquad (7.3)$$

For generalized action systems, Back and von Wright show how to prove enforcement of temporal properties by using usual invariant-based methods [38], rather than employing costly fixed-point computation algorithms. By adapting their result to our case, we get the rule shown in Lemma 7.

Lemma 7 *Assume the following action system*:

$$Sys(y : T_y) \;\;=\;\; \mathbf{begin}\;\; \mathbf{var}\;\; x : T_x \bullet\; Init \,;\; \mathbf{do}\; g \rightarrow A \,;\, D \;\mathbf{od}\;\; \mathbf{end}$$

Then, always-properties can be proved using invariants, as shown by the inference rule below:

$$\frac{p \subseteq I \qquad g \cap I \,\{\!|\, A \,;\, D \,|\!\}\, I \qquad I \subseteq q}{p \,\{\!|\; \mathbf{do}\; g \rightarrow A \,;\, D \;\mathbf{od} \;|\!\}\, \Box\, q}$$

where p, q *are predicates.*

Proof. *Similar to the proof of Lemma 6.* ■

The inference rule of Lemma 7 states that proving the "*always*" property for the loop of the action system Sys is in fact equivalent to showing that a predicate $I \subseteq q$ is an invariant of Sys. Even if this is an expected result, the presence of the angelic nondeterminism makes the process slightly more complicated. We will dwell on this issue in the next section.

Therefore, proving the safety property q by proving an invariance property subsumes the following obligations:

1. Find a predicate I, such that $I \subseteq q$ holds.
2. Prove that I is established by the initialization $Init$, that is, $p \subseteq I$, where p is the predicate that holds after $Init$.
3. Prove that I is preserved by the action $g \rightarrow A\,;D$, that is

$$\begin{aligned} & g \cap I \subseteq A.(D.I) \\ = \; & \{\text{substitute } A,\, D, \text{ rule(2.7), rule(2.12), rule(2.11)}\} \qquad (7.4) \\ & g \cap I \subseteq (\exists x' \cdot b_a \cap (D.I)[x := x']) \end{aligned}$$

It then follows that, if the above conditions hold, the angel has a winning strategy, A, thus a controller for invariance can be synthesized.

7.3.2 Extracting the Control Strategy for Invariance

After having established that the angel can enforce the required behavior, one needs to extract its respective winning strategy.

In principle, the process of controller synthesis should constrain the angelic behavior; on the other hand, usual refinement seems not to do the job, since the refinement rule of angelic assignments (2.16) says that only by adding choices can we get a refinement of such statements. However, in the following, we show how to reduce the angelic nondeterminism, with respect to the enforced property, via a correctness-preserving transformation.

Given the fact that we have proved I to be an invariant of the action system Sys defined in Lemma 7, we know, due to the correctness rule for sequential composition, recalled in chapter 2, that the new statement

$$\begin{aligned} & S \\ = \; & g \rightarrow \{I\}\,; A\,; \{D.I\}\,; D\,; \{I\} \\ = \; & \{\text{rule (2.19)}\} \\ & g \rightarrow \{g \cap I\}\,; A\,; \{D.I\}\,; D\,; \{I\} \end{aligned}$$

can replace the statement $g \rightarrow A; D$ of the $\mathbf{do} - \mathbf{od}$ loop. This is correct, since S preserves the invariant, trivially. Within this context, we can rewrite A by using the information supplied by assertion $\{D.I\}$, such that the angel is forced to restrict its choices to the ones that establish I.

In our case, $A = \{x := x' \mid b_a\}$, thus, we can propagate $\{D.I\}$ backwards. In this way, we strengthen the boolean condition of the angelic nondeterministic assignment. As a result, the angelic choices are reduced, according to the propagated information. The actual transformation is as follows:

$$\begin{array}{ll} & \{x := x' \mid b_a\} \,;\, \{D.I\} \\ = & \{(2.18)\} \\ & \{x := x' \mid b_a \cap D.I[x := x']\} \\ = & A_f \end{array} \tag{7.5}$$

At first glance, this seems a transformation that does not actually favor our agent; it rather serves the interests of the demon, since it decreases the set of final states that the angel can choose from.

However, rewriting in the specified context makes the behavior of the angel implementable. To support this claim, we prove that the angelic nondeterminism within A_f is in fact *equivalent* to demonic nondeterminism, within the analyzed context.

The correctness rule of the sequence $\{g \cap I\} \,;\, A_f \,;\, D \,;\, \{I\}$ with respect to precondition $g \cap I$ and postcondition I is

$$\frac{g \cap I \;\{\!|\{g \cap I\} \,;\, A_f|\!\}\; D.I \qquad D.I \;\{\!|D \,;\, \{I\}|\!\}\; I}{g \cap I \;\{\!|\{g \cap I\} \,;\, A_f \,;\, D \,;\, \{I\}|\!\}\; I}$$

We have that

$$\begin{array}{ll} & (\{g \cap I\} \,;\, A_f).(D.I) \\ = & \{\text{substitute } A_f\} \\ & (\{g \cap I\} \,;\, \{x := x' \mid b_a \cap (D.I)[x := x']\}).(D.I) \\ = & \{\text{wp rules for angelic assignment, sequential composition, and assertion}\} \\ & g \cap I \cap (\exists x' \cdot b_a \cap (D.I)[x := x']) \\ \subseteq & \{\text{logic}\} \\ & g \cap I \cap \mathsf{true} \\ = & \{\text{rewrite } \mathsf{true}\} \\ & g \cap I \cap (\forall x' \cdot b_a \cap (D.I)[x := x'] \subseteq (D.I)[x := x']) \\ = & \{\text{wp rules for demonic assignment, seq. composition, and assertion}\} \\ & (\{g \cap I\} \,;\, [\, x := x' \mid b_a \cap (D.I)[x := x']\,]).(D.I) \end{array}$$

We also have that

$$\begin{array}{ll} & (\{g \cap I\} \,;\, A_f).(D.I) \\ = & \{\text{substitute } A_f\} \\ & (\{g \cap I\} \,;\, \{x := x' \mid b_a \cap (D.I)[x := x']\}).(D.I) \end{array}$$

$$
\begin{array}{ll}
= & \{\text{wp rules for angelic assignment, sequential composition, and assertion}\} \\
 & g \cap I \cap (\exists x' \cdot b_a \cap (D.I)[x := x']) \\
\supseteq & \{(7.4)\text{: } g \cap I \subseteq (\exists x' \cdot b_a \cap (D.I)[x := x'])\} \\
 & g \cap I \cap \mathsf{true} \\
= & \{\text{rewrite } \mathsf{true}\} \\
 & g \cap I \cap (\forall x' \cdot b_a \cap (D.I)[x := x'] \subseteq (D.I)[x := x']) \\
= & \{\text{wp rules for demonic assignment, seq. composition, and assertion}\} \\
 & (\{g \cap I\}\, ;\, [\, x := x' \mid b_a \cap (D.I)[x := x']\,]).(D.I)
\end{array}
$$

■

Hence, for whatever choice the angel picks, such that it stays within the pool of states denoted by $b_a \cap D.I[x := x']$, the final predicate I holds, no matter how the demon chooses to play.

The idea of implementing angelic nondeterminism by systematically transforming angelic nondeterministic statements into demonic nondeterministic statements, or deterministic statements is advocated by Celiku and von Wright [57]. Rules that guarantee the correctness of such transformations are proposed, yet without targeting synthesis of controllers that need to meet (various) temporal requirements.

7.3.3 Example: A Producer-Consumer Application

Let us assume that we are given the task of designing a controller for a *First-In-First-Out* (FIFO) memory buffer (or a *Last-In-First-Out* buffer, for that matter). A specific *producer* process adds data to the buffer, while a particular *consumer* takes away data from the buffer, with respect to predefined rules. This kind of pipelined controller could be useful, for instance, in the design of hardware devices.

Our goal is to ensure that the producer can always provide at least one new input to the buffer, that is, the buffer is never full after the consumer has finished its round. We choose to show our proposed methodology on a *parameterized* model, where the parameter is the capacity of the buffer.

In the example that we present, we suppose that the producer places items at one end of the buffer, and the consumer removes items at the other end (Figure 7.2 a)). However, this is just a modeling point of view, since the methodology applies also if they operate at the same end of the buffer (Figure 7.2 b)).

System Modeling. We start by imagining a game between the controller (producer), represented by the controllable variables, and the plant (consumer), modeled by the uncontrollable ones. The players take turns and make moves with respect to the following rules:

- each time the system executes, the producer has to add one or two items into the virtual buffer (it can not add zero);

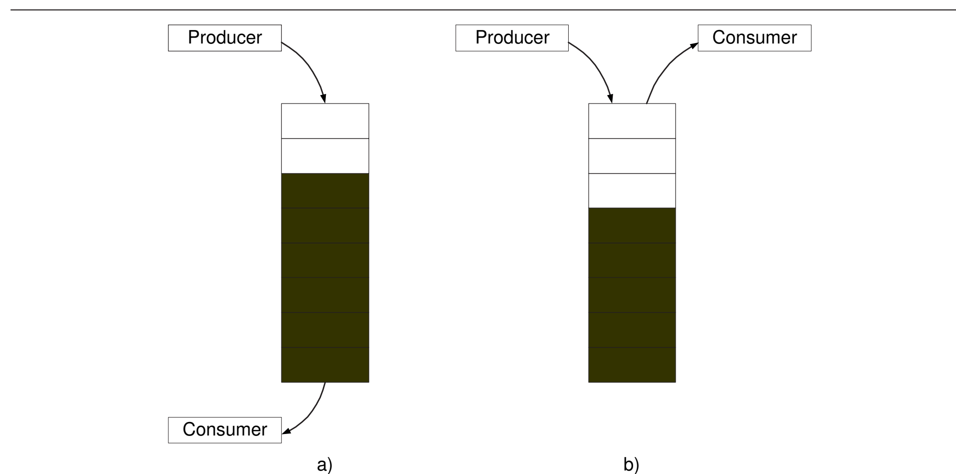


Figure 7.2: The producer-consumer example: a) FIFO, b) Stack (LIFO)

- the consumer may choose to remove at most two items at a time, or leave the number of items unchanged, depending on the sizes of the respective data packages:

  - the consumer is allowed to remove zero items, if it has removed one item from the buffer, in the immediate previous step;

  - if the consumer has removed zero in the previous round, it has to remove two items, in the current one;

  - if the consumer has removed two items, it is mandatory that it removes only one item during the current round.

Note that an external observer sees the start of each round and the end of it, without noticing the intermediate states.

The goal of the controller is to find a way to enforce the required property, during the execution of the system. For instance, such property is the requirement to *"never exceed the capacity of the buffer"*, or *"if an error has occurred, keep the buffer within certain limits"*.

The variables that describe the system state are as follows:

- C : Nat - models the number of items in the buffer, as updated by the consumer, at the end of each round of the game; it represents the observable value;

- r : $\{0,1,2\}$ - represents the number of items removed by the consumer, from the buffer;

- max : Nat- models the capacity of the buffer (maximum number of items), yet not less than 4 locations ($max \geq 4$), for the buffer to be sufficiently large.

The goal of the producer is a postcondition formalized as an "always" temporal property:

$$\Box\, q = \Box(0 \leq C < max),$$

The property can be interpreted in terms of the game: the producer loses the game if the consumer manages to leave the buffer full, after its respective update. By enforcing $\Box\, q$, we ensure that there is a continuous activity at the producer end of the buffer.

In spite of the partly nondeterministic moves of the consumer, the producer should be able to enforce $\Box\, q$. Having a way to keep it true during the entire execution of the system is equivalent to synthesizing a controller for invariance.

Hence, we focus on synthesizing such a controller, within the mentioned setup.

7.3.4 The Producer-Consumer Model as an Action System

The process of controller synthesis is gradual; it starts with a nondeterministic model of the controller, which has to be further adjusted correctly, in order to be brought closer to the implementable level. This justifies our decision to specify the actions of the producer, as an angelic nondeterministic assignment. Thus, the behavior of the controller is described as follows:

$$Prod = \{C := C' \,|C < C' \leq C + 2\} \tag{7.6}$$

The boolean condition of the assignment ensures that the producer adds one or two items to the buffer. Should we not require this condition to hold, the basic angelic behavior is not enforced.

As the consumer is partly uncontrollable, it behaves demonically. Consequently, it is modeled by a demonic nondeterministic assignment:

$$\begin{aligned} Cons \;&=\; [r, C := r', C' \,|(r = 0 \subseteq r' = 2) \\ &\qquad\qquad\qquad \cap (r = 1 \subseteq r' \in \{0, 1, 2\}) \\ &\qquad\qquad\qquad \cap (r = 2 \subseteq r' = 1) \\ &\qquad\qquad\qquad \cap C' = C - r'] \\ &=\; [r := r' \,|(r = 0 \subseteq r' = 2) \\ &\qquad\qquad \cap (r = 1 \subseteq r' \in \{0, 1, 2\}) \\ &\qquad\qquad \cap (r = 2 \subseteq r' = 1)]\,; \\ &\quad\; C := C - r \end{aligned} \tag{7.7}$$

The statement $Cons$ regulates the moves of the consumer, according to the rules mentioned in section 7.3.3.

The producer is responsible for enforcing the safety property $\Box\, q$, formalized previously. At each turn, it should choose an appropriate number of items to add to the buffer, such that the latter can never be left fully occupied, by the consumer.

The property should be guaranteed, regardless of the demonic nondeterministic moves.

Further, we model the producer and the consumer, together, as the action system below, where we substitute statement (7.6) for $Prod$, and (7.7) for $Cons$. The system terminates upon the completion of the process. This is decided by an external device, modeled by some statement Dev. For simplicity, we choose here to model a non-terminating loop. At will, the guard true can be replaced by a non-trivial one.

$$
\begin{array}{lcl}
\mathcal{B}uf & \triangleq & \textbf{begin var } r : \{0,1,2\}, C : \mathsf{Nat} \bullet \\
& & \quad r := 0 \,;\, C := 0 \,;\, [max \geq 4]; \\
& & \quad \textbf{do } \mathsf{true} \;\rightarrow Prod \,;\, Cons \,;\, Dev \textbf{ od} \\
& & \textbf{end} : max
\end{array}
\tag{7.8}
$$

7.3.5 Applying the Synthesis Method

According to the theory, we first check whether the safety property $\Box\, q$, given as $\Box(0 \leq C < max)$, can be enforced by the producer. In case this is possible, we move along the line established in section 7.3.2, to extract the winning strategy.

Given the system model as the action system $\mathcal{B}uf$ defined by (7.8), we proceed as follows:

A1) Firstly, we need to find a predicate $I \subseteq q$. We choose I as being

$$
\begin{array}{lcl}
I & = & (r = 0 \cap 0 \leq C < max) \cup \\
& & (r \neq 0 \cap 0 \leq C < max - 1)
\end{array}
$$

Proving that $I \subseteq q$ is straightforward. ■

A2) Next, I has to be an invariant of the action system $\mathcal{B}uf$. We assume that the statement Dev preserves the invariant ($I\{\!|Dev|\!\}I$ holds), since it does not interfere with the variables mentioned in I. The invariant is established by the initialization statement:

$$
\begin{array}{ll}
& p \\
\triangleq & r = 0 \cap C = 0 \cap max \geq 4 \\
\subseteq & \{\text{logic}\} \\
& I
\end{array}
$$
■

Then, we prove that I is preserved by the action of the loop, that is,

$$I \subseteq Prod.(Cons.I)$$

Since statement *Dev* preserves the invariant trivially, the relation

$$I \subseteq Prod.(Cons.I) \Rightarrow I \subseteq Prod.(Cons.(Dev.I))$$

holds. Hence, if we are able to prove the left-hand side of the above implication, the right-hand side follows. The proof is shown below.

$$
\begin{array}{ll}
 & Prod.(Cons.I) \\
= & \{\text{substitute statement Cons}\} \\
 & Prod.(([r := r' \mid \\
 & \quad (r = 0 \subseteq r' = 2) \cap \\
 & \quad (r = 1 \subseteq r' \in \{0, 1, 2\}) \cap \\
 & \quad (r = 2 \subseteq r' = 1)] \,;\, C := C - r).I) \\
= & \{\text{rules (2.7), (2.4), (2.12)}\} \\
 & Prod.(\forall r' \bullet (r = 0 \subseteq r' = 2) \cap \\
 & \quad (r = 1 \subseteq r' \in \{0, 1, 2\}) \cap \\
 & \quad (r = 2 \subseteq r' = 1)) \\
 & \subseteq (I[C := C - r])[r := r'] \\
= & \{\text{substitute statement Prod, simplify}\} \\
 & \{C := C' \mid C < C' \leq C + 2\}. \\
 & \quad ((r = 0 \subseteq 2 \leq C \leq max) \cap \\
 & \quad (r = 1 \subseteq 2 \leq C \leq max - 1) \cap \\
 & \quad (r = 2 \subseteq 1 \leq C \leq max - 1)) \\
= & \{\text{rule (2.11)}\} \\
 & (\exists\, C' \bullet C < C' \leq C + 2 \cap \\
 & \quad ((r = 0 \subseteq 2 \leq C' \leq max) \cap \\
 & \quad (r = 1 \subseteq 2 \leq C' \leq max - 1) \cap \\
 & \quad (r = 2 \subseteq 1 \leq C' \leq max - 1))) \\
= & \{\text{case analysis}\} \\
 & \bullet \{\text{witness } C' = C + 1\} \\
 & \quad (C' = C + 1) \cap \\
 & \quad (r = 0 \subseteq 2 \leq C' \leq max) \cap \\
 & \quad (r = 1 \subseteq 2 \leq C' \leq max - 1) \cap \\
 & \quad (r = 2 \subseteq 1 \leq C' \leq max - 1)) \\
 & \supseteq \{\text{logic}\} \\
 & \quad I \cap (C \geq 2) \\
 & \bullet \{\text{witness } C' = C + 2\} \\
 & \quad (C' = C + 2) \cap \\
 & \quad ((r = 0 \subseteq 2 \leq C' \leq max) \cap \\
 & \quad (r = 1 \subseteq 2 \leq C' \leq max - 1) \cap \\
 & \quad (r = 2 \subseteq 1 \leq C' \leq max - 1)) \\
 & \supseteq \{\text{logic, assumption } max \geq 4\} \\
 & \quad I \cap (0 \leq C \leq 1) \\
\supseteq & \{\text{logic}\} \\
 & (I \cap (0 \leq C \leq 1)) \cup (I \cap (C \geq 2)) \\
= & \{\text{logic}\} \\
 & I
\end{array}
$$

We have also proved the invariant with the *Prototype Verification System* (PVS) [65]. The PVS specification of *Prod*, *Cons*, and the mechanized proof of invariance are shown in Figures 7.3 and 7.4, respectively. Consequently, irrespective of the chosen value of r, the producer has a way of enforcing $\Box\, q$.

B) Within this step, we apply rule (7.5) for rewriting the statement $Prod$, given by (7.6). The rewriting procedure uses backward propagation of the assertion

$$
\begin{aligned}
& \{Cons.I\} \\
= & \\
& \{\ (r = 0 \subseteq 2 \le C \le max) \\
& \cap (r = 1 \subseteq 2 \le C \le max - 1) \\
& \cap (r = 2 \subseteq 1 \le C \le max - 1)\ \}
\end{aligned}
$$

Below, we show the one-step derivation that leads to the winning strategy of the producer:

$$
\begin{aligned}
& \{C := C' \mid C < C' \le C + 2\}\ ;\ \{Cons.I\} \\
= \quad & \{\text{rule (7.5)}\} \\
& \{C := C' \mid C < C' \le C + 2 \\
& \qquad \cap (r = 0 \subseteq 2 \le C' \le max) \\
& \qquad \cap (r = 1 \subseteq 2 \le C' \le max - 1) \\
& \qquad \cap (r = 2 \subseteq 1 \le C' \le max - 1)\} \\
= \quad & \{\text{notation}\} \\
& Prod_f \qquad\qquad (7.9) \\
= \quad & \{\text{ result proved in section 7.3.2, fixed postcondition } Cons.I\ \} \\
& [C := C' \mid C < C' \le C + 2 \\
& \qquad \cap (r = 0 \subseteq 2 \le C' \le max) \\
& \qquad \cap (r = 1 \subseteq 2 \le C' \le max - 1) \\
& \qquad \cap (r = 2 \subseteq 1 \le C' \le max - 1)]
\end{aligned}
$$

The statement $Prod_f$, given by (7.9), represents the winning strategy of the producer, which guarantees that q is `true`, for all possible executions. Depending on the current value of C, as updated by the consumer, the lower and upper bounds on C' may force the angel to add either one item strictly, $C' = C + 1$, or two items only ($C' = C + 2$), or may allow for both $C + 1$ and $C + 2$ as valid alternatives. The strategy ensures a win for the angel, for whatever demonic choices.

Consequently, as discussed in section 7.3.2, the final model is implementable. By strengthening the boolean condition of the angelic nondeterministic assignment

$Prod$, given by (7.6), with the information available as the assertion $Cons.I$, we have eliminated the angelic choices that would not establish I. The producer can blindly select its moves, yet satisfying $\Box\, q$, which has been our design target. Now, we can safely replace $Prod$ by $Prod_f$ in the action system $\mathcal{B}uf$.

An interesting extension of the analyzed example is to try to keep the content of the buffer within certain specified limits. In this case, additional information, which describes the conditions at the other end of the buffer, should be considered also.

For example, let us assume that a safety property that tolerates neither a full, nor an empty buffer, should be enforced on $\mathcal{B}uf$:

$$\Box\, q_{new} = \Box\,(1 \leq C < max)$$

The condition q_{new} requires an ongoing activity at both ends of the buffer.

Applying the same technique as for synthesizing a controller for q, we need to find an invariant that implies q_{new}. The respective predicate is as follows:

$$\begin{aligned} & I_{new} \\ = \;& (r = 0 \cap 1 \leq C < max) \cup \\ & (r \neq 0 \cap 1 \leq C < max - 1) \end{aligned}$$

Note that only the lower bound of C has been modified, accordingly. It can be proved that the winning strategy of the producer, which guarantees q_{new}, for any possible choice, is

$$\begin{aligned} \{C := C' \mid & C < C' \leq C + 2 \\ & \cap\,(r = 0 \subseteq 3 \leq C' \leq max) \\ & \cap\,(r = 1 \subseteq 3 \leq C' \leq max - 1) \\ & \cap\,(r = 2 \subseteq 2 \leq C' \leq max - 1)\} \end{aligned}$$

Should we decide to always maintain the buffer filled at least n locations, and at most $max - N$ locations, we then have, by induction, that:

$$\begin{aligned} & I_{gen} \\ = \;& (r = 0 \cap n \leq C < max - N) \cup \\ & (r \neq 0 \cap n \leq C < (max - N) - 1) \end{aligned}$$

and the winning strategy:

$$\begin{aligned} \{C := C' \mid & C < C' \leq C + 2 \\ & \cap\,(r = 0 \subseteq n + 2 \leq C' \leq max - N) \\ & \cap\,(r = 1 \subseteq n + 2 \leq C' \leq (max - N) - 1) \\ & \cap\,(r = 2 \subseteq n + 1 \leq C' \leq (max - N) - 1)\} \end{aligned}$$

The ideas introduced here might be applied to a more general producer-consumer problem. This would indeed lead to the construction of a correct and reliable template for such a class of systems; then, not only the capacity, but also the number of inputs and outputs, that is, the choices of the producer and the consumer, respectively, should be parameterized.

Producer-Consumer Specification in the Language of the Prototype Verification System. The *Prototype Verification System* (PVS, in short) [65, 144] offers mechanized support for formal program specification and verification. It has been developed at SRI International, and it comprises a *specification language* and a rich built-in *prelude*, made of theories that contain useful definitions and theorems. The PVS specification language builds on classical typed higher-order logic.

PVS specifications are packaged as *theories* that can be parametric in types and constants. The built-in prelude and loadable libraries provide standard specifications and proved facts for a large number of theories.

```
fifo: theory

begin

nat_4 : TYPE+ = {n: nat | 4 <= n} CONTAINING 4
 N: nat_4
 C, C0: var nat
 r, r0: var {n: nat | n < 3}

 I(r, (C: int)): bool = (r = 0 and 0 <= C and C < N) or
                        (r /= 0 and 0 <= C and C <= N - 2)

prod(C, C0): bool = C < C0 and C0 <= C + 2

cons(r, r0): bool =
        (r = 0 => r0 = 2) and
        (r = 1 => r0 < 3) and
        (r = 2 => r0 = 1)


invariant: lemma I(r, C) => exists C0: prod(C, C0) and
                  (forall r0: cons(r, r0) => I(r0, C0 - r0))

wp: lemma (forall r0: cons(r, r0) => I(r0, C - r0)) =
                ((r = 0 => 2 <= C and C <= N) and
                 (r = 1 => 2 <= C and C <= N - 1) and
                 (r = 2 => 1 <= C and C <= N - 1))

end fifo
```

Figure 7.3: PVS specification of *Prod, Cons*

Proofs in PVS are presented in a sequent calculus. The atomic commands of the PVS prover include *quantifier instantiation, automatic conditional rewriting, sim-*

plification using arithmetic and equality decision procedures and type information, and propositional simplification using binary decision diagrams. The SKOSIMP command, for example, introduces constants of the form x!i for universal quantifiers, and ASSERT combines rewriting with decision procedures.

Finally, PVS has a strategy language for combining atomic inference steps into more powerful proof strategies. The strategy GRIND, for example, combines rewriting with propositional simplification using *binary decision diagrams* (BDDs) and decision procedures.

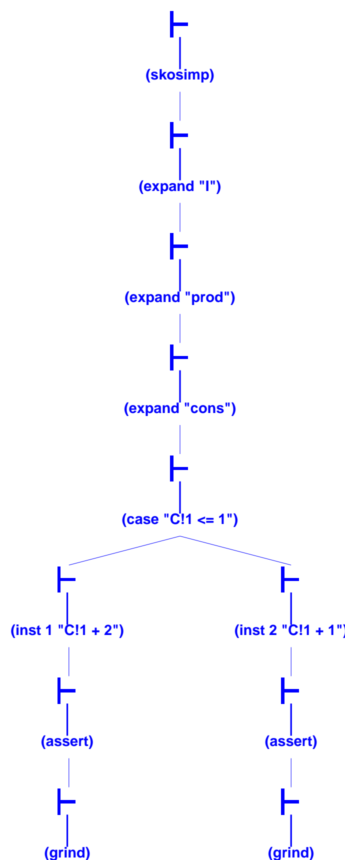


Figure 7.4: PVS proof of invariance.

Since all we need to do is to prove an invariance property, we have reduced the predicate transformer model of the producer and the consumer, to a boolean specification in PVS. This lighter model, as can be seen in Figure 7.3, offers better premises for a more intuitive, less intricate mechanized proof.

At first, we have proved the *wp lemma*, which verifies the correctness of the computed weakest precondition $Cons.I$. The proof is carried out in 4 steps: invoke

the quantifier rule SKOLEM!, expand the definition of $Cons$, then of I, and last, use the rule for simplification with decision procedures, GRIND.

The main lemma called *invariant* checks for the validity of the invariant, after executing the sequence $Prod\,; Cons$. We have verified that predicate I implies the existence of a $C0$ (C' in (7.6)), such that for any number of items removed by the consumer $cons(r, r0)$, the invariant is preserved in the end. The proof is given in Figure 7.4 and follows a similar pattern to the pen-and-paper proof.

7.4 Synthesis of Controllers for Reachability

Reachability controllers have to guide the system into a specified set of states, from any initial state, or provided that some nontrivial condition is fulfilled. One of the possible applications of the latter is in *fault-tolerant* systems.

Even if we agree with the point of view that the best way to deal with faults is not to have them, it is not possible to make *any* system 100 % fault-free. Therefore, *full* fault-tolerance is desirable. However, most systems can perform reliably up to a limited number of faults, with a possible decrease in performance.

For example, in order to guarantee reliability of fault-tolerant systems, one may wish to encode the requirement that a system fault (or an accepted number of faults) should not be followed by other faults, during the system's life-time.

As distinct from the previously mentioned fault-tolerant systems, fail-safe systems should be able to terminate in a safe state, if they suffer a serious damage and need to halt their operation. Failures result from many causes, such as degradation, overloading, design errors etc. Besides minimizing the chances of a failure, one should also strive to reduce its effect. This means that a system has to be designed such that, if a failure occurs, its controller issues commands that lead to restoring safety and termination.

In the following, we address the design of reachability controllers suited for classes of fault-tolerant systems as described above.

7.4.1 Characterizing Enforcement of Response Properties in Action Systems

Formal Definition of Weak Response. Let us consider a reactive system described by an action system given by (7.1). In principle, for designing correct reachability controllers, the angel has to guarantee liveness properties, modeled as "*eventually*" ($\Diamond$) properties. Here, we focus on an eventuality property that we denote by the temporal operator $\Diamond_w(p, q)$, p, q predicates. We call this property *weak response*.

The weak response property holds if:

- the system has reached a state in the set of reachable states p and the angel has a way of ensuring that the system execution terminates in a state of q, or

- if the angel can keep the system in a state of $\neg p$, forever.

As proved by Back and von Wright [38], for "*always*" and "*until*" properties, enforcement of temporal properties is reduced to traditional correctness properties of special fixed-point statements. To be able to further characterize $\Diamond_w(p,q)$, we define the following recursive statement, which subsumes the existence of two loops:

$$\begin{aligned} & \mathsf{WRes.p.q} \\ \triangleq \quad & (\nu X \cdot [\neg p] \,;\, [g] \,;\, A \,;\, D \,;\, X \sqcap [p] \,;\, (\mu Y \cdot [\neg q] \,;\, \{g\} \,;\, A \,;\, D \,;\, Y)) \end{aligned} \tag{7.10}$$

In fact, WRes.p.q is a weak iteration that is necessarily terminating, if the particular condition p holds.

We can say that the statement WRes.p.q is an *interpreter* for $\Diamond_w(p,q)$; it executes the constituent statements for determining whether the weak response property is valid. The behavior of WRes.p.q is shown in Figure 7.5.

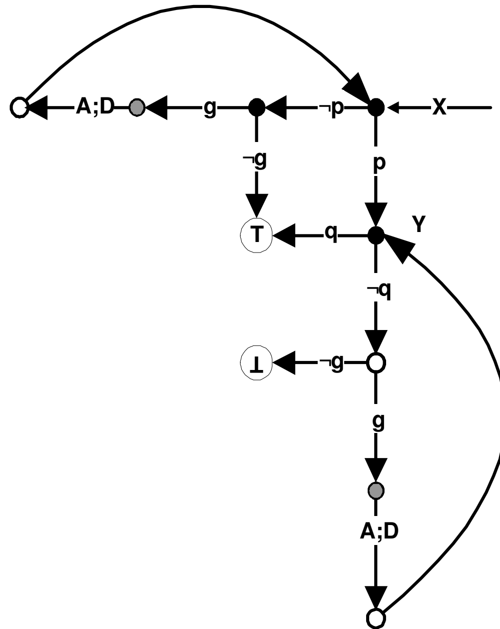


Figure 7.5: An interpreter for WRes.p.q

The diagram shows the angelic choices as empty circles and the demonic choices as filled circles. A grey circle means that we do not know whether the choice is angelic or demonic.

For mission-critical systems (e.g., in avionics), p can model a failure that, once encountered, has to be followed by the restoration of the safe state, and termination in such a state (where q holds). The fail-safe systems attempt to limit the

amount of damage caused by a failure [52]. Termination when q holds is compulsory. To achieve this, we need to specifically add a guarded action that disables the respective action system whenever $p \cap q$ holds. The action should have the form $p \wedge q \rightarrow S$, where S is an assignment that makes the guard of the system false.

To help intuition, we give a practical example described by Burns and Wellings, in their book on real-time systems [52]: "The A310 Airbus's slat and flap control computers, on detecting an error on landing, restore the system to a safe state and then shut down. In this situation, a safe state is having both wings with the same settings; only asymmetric settings are hazardous in landing". Informally, if p = (error on landing), then the airplane controller should enforce q = (equal settings for both wings), at landing. Otherwise, if p does not hold (no error on landing has occurred), q need not necessarily hold.

We translate the informal descriptions presented above into a formal lemma, which characterizes the weakest precondition of a statement with respect to the weak-response property.

Lemma 8 *Assume that predicates p, q, and the monotonic predicate transformer S are given. Then, the weakest predicate such that the game tree generated by S satisfies $\diamondsuit_w(p, q)$ is:*

$$S.\diamondsuit_w(p, q) \quad = \quad (\nu x \bullet (p \cup S.x) \cap (\neg p \cup (\mu y \bullet q \cup S.y)))$$

Proof. We follow the proof line of Lemma 2, in [37]. Let us consider $q' = (\nu x \bullet (p \cup S.x) \cap (\neg p \cup (\mu y \bullet q \cup S.y)))$. Consider also the game tree with root $p_0 = S.\diamondsuit_w(p, q)$ generated by S.

The angel must be able to choose a predicate p' such that

$$(\forall \sigma \bullet (p.\sigma \vee S.p'.\sigma) \wedge (\neg p.\sigma \vee (p'.\sigma = \lim_{\alpha} q_{\alpha+1}.\sigma)))$$

holds and (p', S) satisfies $\diamondsuit_w(p, q)$.

In the above, the approximations q_α are defined by:

$$\begin{aligned} q_0 &= \mathsf{false} \\ q_{\alpha+1} &= q \cup S.q_\alpha \\ q_\alpha &= (\cup \beta < \alpha \cdot q_\beta), \text{for limit ordinals } \alpha \end{aligned}$$

The approximation q_α characterizes those states for which the angel can guarantee that, if we set a "counter" at α and decrease it after each time the demon has made a choice, then q will be established before the counter runs out. Thus, the limit of these approximations is exactly the set of states that guarantee $\diamondsuit q$.

Since p_0 is the greatest predicate with this property, we have:

$$\begin{aligned} \sigma \in p_0 \quad \equiv \quad & (\sigma \in p \vee (\exists p' \bullet S.p'.\sigma \wedge p' \subseteq p_0)) \\ & \wedge (\sigma \in \neg p \vee (\exists p' \bullet p' \subseteq p_0 \wedge p'.\sigma = \lim_{\alpha}(q.\sigma \vee S.q_\alpha.\sigma))) \end{aligned} \tag{7.11}$$

Further, we have:

$$
\begin{array}{ll}
 & \sigma \in p_0 \\
\equiv & \{ \text{ argument (7.11) } \} \\
 & \quad (\sigma \in p \vee (\exists p' \bullet S.p'.\sigma \wedge p' \subseteq p_0)) \\
 & \wedge\, (\sigma \in \neg p \vee (\exists p' \bullet p' \subseteq p_0 \wedge p'.\sigma = \lim_{\alpha}(q.\sigma \vee S.q_\alpha.\sigma))) \\
\equiv & \{ \text{ } S \text{ monotonic } \} \\
 & \quad (\sigma \in p \vee S.p_0.\sigma) \wedge\ (\sigma \in \neg p \vee (p_0.\sigma = \lim_{\alpha}(q.\sigma \vee S.q_\alpha.\sigma))) \\
\equiv & \{ \text{ set notation } \} \\
 & \quad (\sigma \in p \cup S.p_0) \wedge\ (\sigma \in \neg p \cup (p_0 = \lim_{\alpha}(q \cup S.q_\alpha)))
\end{array}
$$

Since this holds for all σ, we get:

$$p_0 = (p \cup S.p_0) \cap (\neg p \cup \lim_{\alpha}(q \cup S.q_\alpha))$$

Thus, $p_0 = S.\diamondsuit_w(p, q)$ is a fixed point of

$$(\lambda x \bullet (p \cup S.x) \cap (\neg p \cup \mu.(\lambda y \bullet q \cup S.y)))$$

To see that p_0 is the greatest fixed point, it is sufficient to show that $q' \subseteq p_0$. The latter follows from proving that $\mathcal{G}(q', S) \models \diamondsuit_w(p, q)$.

For this, we note that the root of $\mathcal{G}(q', S)$ is $q' = (p \cup S.q') \cap (\neg p \cup (\mu y \cdot q \cup S.y))$, which means that every state that the demon can choose is either in p, or in $\neg p$. After that, the angel can choose q' as the next predicate. If the demon chooses a state in p, given the fact that the angel chooses q', $(\mu y \cdot q \cup S.y)$ holds. If the demon chooses a state of $\neg p$, it follows that $S.q'$ holds. Thus, by induction, $(p \cup S.q') \cap (\neg p \cup (\mu y \cdot q \cup S.y))$ will hold in every state if the angel follows the strategy "always choose the predicate q'". ∎

The characterization in Lemma 8 of the weakest precondition gives us the following result for action game satisfaction of weak response.

Theorem 4 *Assume that S is a monotonic predicate transformer and p_0, p, q are predicates. Then*

$$\mathcal{G}(p_0, S) \models \diamondsuit_w(p, q) \quad \text{iff} \quad p_0 \subseteq (\nu x \bullet (p \cup S.x) \cap (\neg p \cup (\mu y \bullet q \cup S.y)))$$

Proof. Follows from Lemma 8. ∎

This shows how reasoning about weak response is reduced to fixed point reasoning about predicate transformers.

Correctness Lemma for Weak Response. Here, we formulate enforcement of weak response, as a correctness property.

Since

$$\forall \sigma \bullet \sigma \; \{\!|\mathbf{do}\; g \rightarrow A \,;\, D \;\mathbf{od}\;|\!\} \; \Diamond_w(p,q) \equiv (\mathsf{WRes.p.q}).\mathsf{false}.\sigma,$$

we can further claim the following result.

Lemma 9 *Let statements A, D, and predicates p, q be the same as above. Then*

$$\begin{aligned} & p_0 \; \{\!|\mathbf{do}\; g \rightarrow A \,;\, D \;\mathbf{od}\;|\!\} \; \Diamond_w(p,q) \\ \equiv\; & p_0 \subseteq (\nu X \bullet [\neg p] \,;\, [g] \,;\, A \,;\, D \,;\, X \sqcap [p] \,;\, (\mu Y \bullet [\neg q] \,;\, \{g\} \,;\, A \,;\, D \,;\, Y)).\mathsf{false} \end{aligned}$$

■

7.4.2 Proving Enforcement of Weak Response

Invariant-based Proof Rule. Below, we propose an inference rule for checking whether the angel can enforce weak response properties in the particular case of an action system of the form given in the following lemma.

Lemma 10 *Assume the following action system*:

$$Sys(y : T_y) \;\; = \;\; \mathbf{begin}\;\; \mathbf{var}\;\; x : T_x \bullet \;\; Init \,;\; \mathbf{do}\; g \rightarrow A \,;\, D \;\mathbf{od}\;\; \mathbf{end}$$

Then, weak-response properties can be proved using invariants, and termination arguments, as follows:

$$\frac{\begin{array}{rcl} p_0 & \subseteq & I \\ g \cap I & \{|A \,;\, D|\} & I \\ p \cap I & \subseteq & q \cup g \\ \neg q \cap g \cap (t = w) & \{|A \,;\, D|\} & (q \cup g) \cap (t < w) \end{array}}{p_0 \{|\mathbf{do}\; g \rightarrow A \,;\, D \;\;\mathbf{od}\;|\} \; \Diamond_w(p,q)}$$

Here, p, q are predicates, and the state function t ranges over some well-founded set.

Proof.

$$\begin{aligned} & p_0 \; \{|\mathbf{do}\; g \rightarrow A \,;\, D \;\;\mathbf{od}\;|\} \; \Diamond_w(p,q) \\ \equiv\;\; & \{\text{correctness rule - Lemma 9}\} \\ & p_0 \subseteq (\nu X \cdot [\neg p] \,;\, [g] \,;\, A \,;\, D \,;\, X \sqcap [p] \,;\, (\mu Y \cdot [\neg q] \,;\, \{g\} \,;\, A \,;\, D \,;\, Y)).\mathsf{false} \end{aligned}$$

$\equiv$ {successive application of rules (2.7), (2.6), (2.10), (2.5)}
$p_0 \subseteq (\nu x \cdot (p \cup \neg g \cup A.(D.x)) \cap (\neg p \cup (\mu y \cdot q \cup (g \cap A.(D.y)))))$
$\Leftarrow$ {assumption $p_0 \subseteq I$}
$I \subseteq (\nu x \cdot (p \cup \neg g \cup A.(D.x)) \cap (\neg p \cup (\mu y \cdot q \cup (g \cap A.(D.y)))))$
$\Leftarrow$ {greatest fixed point induction rule (2.14)}
$I \subseteq (p \cup \neg g \cup A.(D.I)) \cap (\neg p \cup (\mu y \cdot q \cup (g \cap A.(D.y))))$
$\Leftarrow$ {assumption $g \cap I \subseteq A.(D.I)$}
$I \subseteq (p \cup \neg g \cup (g \cap I)) \cap (\neg p \cup (\mu y \cdot q \cup (g \cap A.(D.y))))$
$\equiv$ {logic}
$I \subseteq (p \cup \neg g \cup I) \cap (\neg p \cup (\mu y \cdot q \cup (g \cap A.(D.y))))$
$\Leftarrow$ {logic}
$I \subseteq \neg p \cup (\mu y \cdot q \cup (g \cap A.(D.y)))$
$\equiv$ {logic}
$p \cap I \subseteq (\mu y \cdot q \cup (g \cap A.(D.y)))$
$\Leftarrow$ {assumption $p \cap I \subseteq q \cup g$}
$q \cup g \subseteq (\mu y \cdot q \cup (g \cap A.(D.y)))$
$\Leftarrow$ {least fixed point rule (2.13)}
$\forall w \cdot (q \cup g) \cap (t = w) \subseteq q \cup (g \cap A.(D.((q \cup g) \cap (t < w))))$
$\Leftarrow$ {shunting, simplification}
$\forall w \cdot \neg q \cap g \cap (t = w) \subseteq A.(D.((q \cup g) \cap (t < w)))$
$\Leftarrow$ {last assumption}
true

■

The rule of Lemma 10 shows the proof obligations when carrying out controller synthesis for weak response. The predicate I might include states of q, or states of $\neg p$. Provided that the angel has started from a state of $\neg q \cap g$, one can prove that it has a winning strategy if it can find a way to keep the system in I, or lead the system into a state of $q \cup g$, trying to decrease t. This means that the controller is forced to make appropriate moves, such that at the end of the game, $q \cup g$ gets established, after a finite number of iterations ($t < w$).

Let us see, next, how these theoretical results apply to a concrete example.

7.4.3 Example: A Data Processing System

In this section, we analyze the operation of an abstract, distributed data processing system. The input data is produced by one unit (**PU**), and transferred, via a limited capacity channel, to a collection of collector devices (**CD**) that process it

further. The channel is similar to a buffer that contains *max* locations. A graphical representation of the system is given in Figure 7.6.

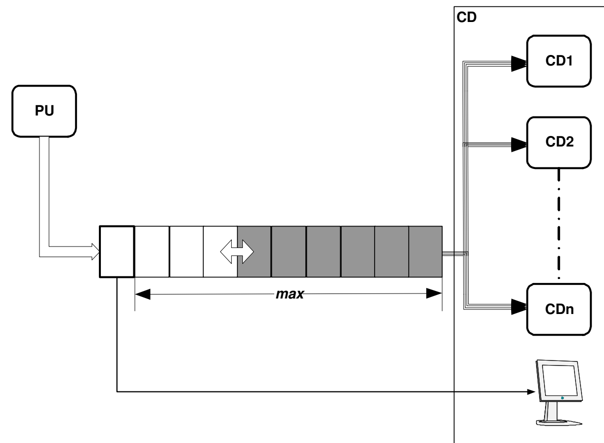


Figure 7.6: The data processing system

The example resembles, in some sense, the one introduced and analyzed in section 7.3.2: the **PU** is the producer, the **CD** replaces the consumer of the respective example. The methodology described in section 7.3 allows us to build a safe system where the margins of the buffer are not exceeded, in either sense.

As distinct from the specification of the mentioned example, now, the buffer may be in service only for a limited period of time (however large), measured by the number of transactions between the **PU** and **CD**. In addition, the **PU** also specifies which collector device should process the current data item at the **CD** end of the buffer. Even if we are not modeling this feature here, we nevertheless need to take it into consideration. A monitor / controller identifies this address and consequently directs the data towards the appropriate **CD**. A costly, from certain points of view, implementation of the system would add, on top of the necessary data locations in the buffer, the number of locations required to store the target address as issued by **PU**[1]. Hence, instead of *max* buffer locations, one needs to employ *max* + *m* locations (where $m \in \{1, 2, \ldots\}$ is the number of slots storing the address, assuming that each slot is of length 1). For simplicity, in the following we consider that each address occupies one slot only, that is, $m = 1$.

From a safety point of view, it would be enough if the designer of such a system integrates m extra locations into the buffer, thus increasing its capacity; then, one could apply the solution found in section 7.3.2 (with *new_max* = *max* + *m*). However, one could think of optimizing the usage of the buffer, such that its dimensions remain the same (*max*). We are helped in this quest by an additional assumption: built with its own safety considerations, the monitor / controller that receives the

[1] An also somewhat costly implementation would use an extra variable to store the address.

target address from the **PU**, may reconstruct it, once, during the life time of the system, even if the necessary data is missing or corrupted.

Let us see how this additional assumption can be satisfied in practice. Based on the fault-tolerant feature, one may think of a system that works "normally" until it falls into the undesired erroneous state (from where the address of the next processing **CD** may not be extracted by the monitor / controller). If this is the first time that the event occurs, the error is detected and repaired by the monitoring procedures. After this, the system must be protected from reaching the same situation again, during its life time. The resulting system is illustrated in Figure 7.7.

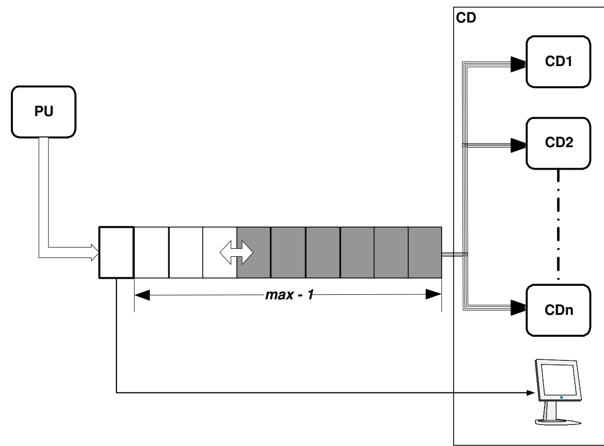


Figure 7.7: The data processing system, with shorter buffer

System Modeling. Here, **PU** is the angelic program, while **CD** behaves demonically. Hence, the incoming signals are modeled as angelic updates, whereas the removal of the buffer content is viewed as a disturbance. As in the example of section 7.3.2, the angel, **PU**, adds one or two data elements to the buffer (apart from the newly required target address). The capacity of the buffer between the two components is represented by the parameter max ($max \geq 4$).

The variables in the system are as follows:

- $life$: Nat - records the functioning time of the system. It is incremented at the end of an execution round (after both angel and demon have played their turns).
- C : Nat - models the number of packages in the buffer. Whenever its value goes over the value $max - 1$, thus violating the requirement of reserving the last location for the target address, an error message is set. Concurrently, some part (k locations) of the buffer content are automatically cleared (bringing the buffer to the level of $C - k$). The valid range of values for parameter k will result while discharging the corresponding proof obligations.

- r : {0,1,2} - represents the number of items removed by **CD**.
- err : Nat - models the error message; err is initialized to 0 and incremented by one every time the conflicting situation appears.

The behavior of the demon is similar to the behavior of the consumer in the producer-consumer model presented in section 7.3.2. The removal of data packages is arbitrary, within some given rules. For simplicity, we choose the rules to be exactly the same as the ones in statement $Cons$ (see (7.7)).

The angel wins in two situations:

1. if the system reaches the end of its life time (expressed in the model by the constant $LimFunc$) and there has not been any conflict between placing the data packages and the target address. Concretely, the angel wins if $err = 0 \cap life = LimFunc$ holds, or
2. if after one error has been signaled, the angel is able to keep the buffer filled within the limits 0 and $max - 1$, until $life = LimFunc$. Hence, the angel avoids the occurrence of a second, similar error. Consequently, $err = 1 \cap life = LimFunc$ is enforced.

The angel loses if, after one error has been signaled, it does not manage to maintain the buffer occupied between 0 and $max - 1$. This means that a second conflict target address - data input can not be avoided, thus $err > 1$ holds.

The behavior of the angel is represented by the statement below:

$$\mathbf{PU} \triangleq \{C := C' \mid C < C' \leq C + 2\} \tag{7.12}$$

The demonic behavior is captured as follows:

$$\begin{aligned}
\mathbf{CD} \triangleq\ & (C \geq max \wedge err = 0 \rightarrow err := err + 1\,;\, C := C - k \\
& \sqcap C \geq max \wedge err > 0 \rightarrow err := err + 1 \\
& \sqcap C < max \rightarrow \mathsf{skip}); \\
& [\, r := r' \mid (r = 0 \subseteq r' = 2) \\
& \qquad \cap (r = 1 \subseteq r' \in \{0, 1, 2\}) \\
& \qquad \cap (r = 2 \subseteq r' = 1)\,]; \\
& C := C - r\,;\, life := life + 1
\end{aligned} \tag{7.13}$$

The additional information that we need is given as:

$$\begin{aligned}
q &\triangleq (err = 1) \cap (life = LimFunc) \\
g &\triangleq life \leq LimFunc - 1 \\
p &\triangleq (err > 0) \cap (life < LimFunc) \\
I &= \quad (err = 0) \cap ((r = 0) \cap (0 \leq C \leq max - 1) \\
&\qquad\quad \cup (r \neq 0) \cap (0 \leq C \leq max - 2)) \\
&\quad \cup (err > 0) \cap ((r = 0) \cap (0 \leq C \leq max - 2) \\
&\qquad\quad \cup (r \neq 0) \cap (0 \leq C \leq max - 3))
\end{aligned} \tag{7.14}$$

One can notice that the above definition of I includes the invariant used in section 7.3.2. The old invariant ensures that, in case there exists an angelic winning strategy, we get a model in which the capacity of the buffer is not exceeded. The choice is also motivated by reusability of previous designs. The current invariant, however, takes into account the presence of the error messages, too.

The overall **PU**-**CD** system is then described as the following action system:

$$
\begin{array}{lll}
\mathcal{DPS} & \triangleq & \textbf{begin var } r \in \{0,1,2\}, C, \mathit{life}, \mathit{err} : \mathsf{Nat} \bullet \\
& & \quad r, C, \mathit{life}, \mathit{err} := 0,0,0,0 \,;\, [max \geq 4]; \\
& & \quad \textbf{do } (\mathit{life} \leq LimFunc - 1) \rightarrow \mathbf{PU} \,;\, \mathbf{CD} \textbf{ od} \\
& & \textbf{end} : max
\end{array}
$$

Applying the Synthesis Method for Weak Response. In order to find a winning strategy for the angel (if any), we have to first check the validity of the following relations.

1. $p_0 \subseteq I$. Immediate proof, after replacing I with its definition (7.14) and also considering $p_0 = (err = 0 \cap r = 0 \cap C = 0 \cap \mathit{life} = 0 \cap max \geq 4)$. ∎

2. $p \cap I \subseteq q \cup g$. From definitions (7.14) we have, at first, that

$$
\begin{array}{ll}
& q \cup g \\
= & \{ \text{ definitions (7.14)} \} \\
& (err = 1 \cap \mathit{life} = LimFunc) \cup (\mathit{life} \leq LimFunc - 1)
\end{array}
$$

   The proof of the required relation is immediate. ∎

3. $g \cap I \subseteq \mathbf{PU}.(\mathbf{CD}.I)$. We start by denoting:

$$
\begin{array}{lll}
\mathbf{CD}1 & \triangleq & C \geq max \wedge err = 0 \rightarrow err := err + 1 \,;\, C := C - k \\
& & \sqcap\, C \geq max \wedge err > 0 \rightarrow err := err + 1 \\
& & \sqcap\, C < max \rightarrow skip \\
\mathbf{CD}2 & \triangleq & [\, r := r' \mid (r = 0 \subseteq r' = 2) \\
& & \qquad \cap\, (r = 1 \subseteq r' \in \{0,1,2\}) \\
& & \qquad \cap\, (r = 2 \subseteq r' = 1) \,]; \\
& & C := C - r \,;\, \mathit{life} := \mathit{life} + 1
\end{array}
$$

   We compute:

$$
\begin{array}{ll}
& \mathbf{CD}.I \\
= & \{ \text{ rule (2.7), } \mathbf{CD} \triangleq \mathbf{CD}1 \,;\, \mathbf{CD}2 \ \} \\
& \mathbf{CD}1.(\mathbf{CD}2.I)
\end{array}
$$

$$
\begin{aligned}
= \quad & \{\text{rules (2.4),(2.7), (2.12), logic}\} \\
& \mathbf{CD}1.(((r = 0 \subseteq err = 0 \cap (2 \leq C \leq max)) \\
& \qquad\qquad \cap (r = 1 \subseteq err = 0 \cap (2 \leq C \leq max - 1)) \\
& \qquad\qquad \cap (r = 2 \subseteq err = 0 \cap (1 \leq C \leq max - 1))) \\
& \qquad\quad \cup ((r = 0 \subseteq err = 0 \cap (2 \leq C \leq max - 1)) \\
& \qquad\qquad \cap (r = 1 \subseteq err = 0 \cap (2 \leq C \leq max - 2)) \\
& \qquad\qquad \cap (r = 2 \subseteq err = 0 \cap (1 \leq C \leq max - 2)))) \\
= \quad & \{\text{logic}\} \\
& \mathbf{CD}1.(2 \leq C \leq max - 2 \cup (err = 0 \cap C = max - 1) \\
& \qquad \cup (err = 0 \cap r = 0 \cap C = max) \\
& \qquad \cup (r = 0 \cap C = max - 1) \\
& \qquad \cup (r = 2 \cap C = 1)) \\
= \quad & \{\text{logic}\} \\
& \mathbf{CD}1.(2 \leq C \leq max - 2 \cup (err = 0 \cap C = max - 1) \\
& \qquad \cup (r = 0 \cap err = 0 \cap C = max) \\
& \qquad \cup (r = 0 \cap C = max - 1) \\
& \qquad \cup (r = 2 \cap C = 1)) \\
= \quad & \{\text{notation}\} \\
& \mathbf{CD}1.I_2 \\
= \quad & \{wp \text{ rules}\} \\
& \ (C \geq max \cap err = 0) \subseteq I_2[err := err + 1, C := C - k]) \\
& \cap ((C \geq max \cap err > 0) \subseteq I_2[err := err + 1]) \\
& \cap (C < max \subseteq I_2) \\
= \quad & \{\text{logic}\} \\
& \ 2 \leq C \leq max - 2 \cup (r = 0 \cap C = max - 1) \\
& \cup r = 2 \cap C = 1 \cup (err = 0 \cap C = max - 1) \\
& \cup err = 0 \cap C = max \cap (k + 2 \leq C \leq max + k - 2)
\end{aligned}
$$

We can write further that:

$$
\begin{aligned}
& \mathbf{PU}.(\mathbf{CD}1.I_2) \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (7.15) \\
= \quad & \{\text{definition}\} \\
& \exists C' \cdot (C < C' \leq C + 2) \cap \mathbf{CD}1.I_2[C := C'] \\
\supseteq \quad & \{\text{case split}\} \\
& \bullet \{\text{witness } C' = C + 1\} \\
& \quad (C' = C + 1 \cap 2 \leq C' \leq max - 2) \\
& \quad \cup (r = 0 \cap C' = max - 1) \cup (r = 2 \cap C' = 1) \\
& \quad \cup (err = 0 \cap C' = max - 1) \\
& \quad \cup (err = 0 \cap C' = max \cap k + 2 \leq C' \leq max + k - 2) \\
& \supseteq \{2 \leq k \leq max - 2\} \\
& \quad I \cap C \geq 2
\end{aligned}
$$

$\bullet$ {witness $C' = C + 2$}

$$
\begin{aligned}
&(C' = C + 2 \cap\ 2 \leq C' \leq max - 2) \\
&\cup (r = 0 \cap C' = max - 1) \\
&\cup (r = 2 \cap C' = 1)\ \cup\ (err = 0\ \cap\ C' = max - 1) \\
&\cup (err = 0\ \cap\ C' = max)\ \cap\ (k + 2 \leq C' \leq max + k - 2)
\end{aligned}
$$

$\supseteq$ $\{max \geq 4\}$

$$
\begin{aligned}
& I\ \cap\ (0 \leq C \leq 1) \\
= \;& \{\text{logic}\} \\
& (I\ \cap\ 0 \leq C \leq 1)\ \cup\ (I\ \cap\ C \geq 2) \\
= \;& \{\text{logic}\} \\
& I \\
\supseteq \;& \{\text{logic}\} \\
& g\ \cap\ I
\end{aligned}
$$

Notice the extraction of parameter k: its range of values is limited to the interval $[2..max - 2]$. By showing that $g \cap I\ \subseteq\ \mathbf{PU}.(\mathbf{CD}.I)$, the above derivation (7.15) completes the invariance proof. ■

4. $\neg q \cap g \cap (t\ =\ w)\ \subseteq\ \mathbf{PU}.(\mathbf{CD}.((q \cup g) \cap (t\ <\ w)))$. We consider that $t\ \triangleq\ LimFunc - life$. We denote:

$$
\begin{aligned}
& (q \cup g) \cap (t < w) \qquad (7.16) \\
= \;& \{\text{definitions } (7.14)\} \\
& ((err = 1 \cap life = LimFunc) \cup\ (life \leq LimFunc - 1)) \cap\ (t < w) \\
= \;& \{\text{definition of } t\} \\
& ((err = 1 \cap life = LimFunc) \cup\ (life \leq LimFunc - 1)) \\
& \qquad \cap (life \geq LimFunc - w + 1) \\
= \;& \{\text{notation}\} \\
& X
\end{aligned}
$$

We then have that:

$$
\begin{aligned}
& \mathbf{PU}.(\mathbf{CD}.((q \cup g) \cap (t < w))) \\
= \;& \{\text{ definitions } (7.14), \text{notation } (7.16)\} \\
& \mathbf{PU}.(\mathbf{CD}.X) \\
= \;& \{\text{ rule } (2.7)\ \} \\
& \mathbf{PU}.(\mathbf{CD}1.(\mathbf{CD}2.X)) \\
= \;& \{\ wp \text{ rules }\} \\
& \mathbf{PU}.(\mathbf{CD}1.(((err = 1 \cap\ life = LimFunc - 1) \\
& \qquad\qquad \cup\, life \leq LimFunc - 2) \\
& \qquad \cap\, life \geq LimFunc - w)
\end{aligned}
$$

$$
\begin{array}{ll}
= & \{\ wp \text{ rules}\ \} \\
 & \mathbf{PU}.((err = 1 \cap \ life = LimFunc - 1 \\
 & \qquad\quad \cap\, life \geq LimFunc - w) \\
 & \qquad \cup\, (err = 0 \cap C \geq max \cap \ life = LimFunc - 1 \\
 & \qquad\quad \cap\, life \geq LimFunc - w) \\
 & \qquad \cup\, (life \leq LimFunc - 2 \cap \ life \geq LimFunc - w)) \\
= & \{\ \text{rule (2.11)}\ \} \\
 & \ (\exists\, C'.(C < C' \leq C + 2) \\
 & \qquad \cap\, (err = 1 \cap \ life = LimFunc - 1 \\
 & \qquad\quad \cap\, life \geq LimFunc - w) \\
 & \qquad \cup\, (err = 0 \cap C' \geq max \cap \ life = LimFunc - 1 \\
 & \qquad\quad \cap\, life \geq LimFunc - w) \\
 & \qquad \cup\, (life \leq LimFunc - 2 \cap \ life \geq LimFunc - w)) \\
= & \{\ \text{predicate calculus}\ \} \\
 & ((err = 1 \cap \ life = LimFunc - 1 \\
 & \qquad\quad \cap\, life \geq LimFunc - w) \\
 & \cup\, (life \leq LimFunc - 2 \cap \ life \geq LimFunc - w)) \\
 & \cup\, (\exists\, C'.(C < C' \leq C + 2) \\
 & \qquad \cap\, (err = 0 \cap C' \geq max \cap \ life = LimFunc - 1 \\
 & \qquad\quad \cap\, life \geq LimFunc - w) \\
\supseteq & \{\ \text{logic}\ \} \\
 & (err = 1 \cap \ life = LimFunc - 1 \cap \ life \geq LimFunc - w) \\
 & \cup\, (life \leq LimFunc - 2 \cap \ life \geq LimFunc - w) \\
\supseteq & \{\ \text{logic}, w \geq 2\ \} \\
 & \neg(err = 1 \cap \ life = LimFunc \\
 & \cap\, life \leq LimFunc - 1 \\
 & \cap\, life = LimFunc - w) \\
= & \{\ \text{identification}\ \} \\
 & \neg q \cap \ g \cap \ (t = w) \qquad\qquad\qquad\qquad \blacksquare
\end{array}
$$

This proves that, whenever the system starts in a state of $\neg q \cap g$, it terminates, establishing $q \cup g$. In our case, termination is triggered by the invalidation of g ($life = LimFunc$), and therefore, at that moment, q holds.

Extracting the Control Strategy. This step is similar to the one in section 7.3.2. We have proved that I is an invariant of the action system $\mathcal{DPS}$, defined by (7.4.3). Consequently, the statement

$$\{lim \leq LimFunc \ \cap \ I\}\,;\mathbf{PU}\,;\{\mathbf{CD}.I\}\,;\mathbf{CD}\,;\{I\}$$

can substitute $\mathbf{PU}\,;\mathbf{CD}$ and become the new body of the loop of $\mathcal{DPS}$. Hence, we can rewrite $\mathbf{PU}$, by propagating the assertion $\{\mathbf{CD}.I\}$ in $\mathbf{PU}$. If we assume $k = 2$,

we get the winning strategy of the angel, with all the unsafe moves eliminated:

$$
\begin{aligned}
\mathbf{PU}' \quad = \quad & \{C := C' \mid (C < C' \leq C+2) \\
& \qquad\qquad \cap (2 \leq C' \leq max - 2 \\
& \qquad\qquad\quad \cup (r = 0 \ \cap \ C' = max - 1) \\
& \qquad\qquad\quad \cup (r = 2 \ \cap \ C' = 1) \\
& \qquad\qquad\quad \cup (err = 0 \ \cap \ max - 1 \leq C' \leq max))\} \\
= \quad & \{\text{simplify, result proved in section 7.3.2, fixed postcondition } \mathbf{CD}.I\} \\
& [C := C' \mid (C < C' \leq C+2) \\
& \qquad\qquad \cap (2 \leq C' \leq max - 2 \\
& \qquad\qquad\quad \cup ((r = 0 \cup err = 0) \cap C' = max - 1) \\
& \qquad\qquad\quad \cup (r = 2 \cap C' = 1) \\
& \qquad\qquad\quad \cup (err = 0 \cap C' = max))]
\end{aligned}
$$

This strategy does not require any angelic intelligence in deciding the "good" moves, yet it does not rule out the non-harmful nondeterminism. Therefore, whatever choice we select for implementation, such that the boolean condition of the assignment is satisfied, the correctness of the controller is guaranteed.

The proof of $\neg q \cap g \cap (t = w) \subseteq \mathbf{PU}.(\mathbf{CD}.((q \cup g) \cap (t < w)))$ shows only termination and establishment of q in finite time. We do not need to propagate the information of assertion $\{\mathbf{CD}.((q \cup g) \cap (t < w))\}$, into statement **PU**, since it does not mention C. Hence, $\{\mathbf{CD}.I\}$ is sufficient for extracting the angelic winning strategy.

Discussion. As mentioned in the beginning of this section, the additional requirements of the presented example could have been easily satisfied if one considered modifying the buffer capacity. Still, we have proved that the response properties are met without increasing max.

Observe that along the deductive procedures, we have also synthesized the value of the parameter k, and we concluded that the property is satisfied at least two steps prior to termination, that is, $w \geq 2$. This low bound of the possible range of values for w comes from the fact that the information about p is not considered by the fourth correctness assertion of Lemma 10. That assertion should hold under any execution scenario. For example, we can imagine that, when $life = LimFunc - 1$, p holds, $\neg q$ holds and after one step a new error is encountered (angel loses). This means that q does not hold but the system stops. So, the postcondition $q \cup g$ is not established, hence the correctness assertion is false.

Thus, we need at least two iterations prior to termination ($w \geq 2$), for the fourth correctness rule to always hold.

7.5 Summary and Related Work

We have tackled the problem of discrete controller synthesis, by modeling the system as an action system, and the synthesis process as a zero-sum two-player game.

The players are the controller, called the angel, and the plant, called the demon, which make moves sequentially, each according to some statement, respectively. The goal of the angel, which models the requirement specification, is a safety or a liveness temporal property. In the first case, the method ends up with a controlling strategy for invariance, whereas in the second case we synthesize a control strategy for reachability. In the latter case, we apply the theory for designing controllers for *fault-tolerant* systems, under one-fault scenarios. Full fault-tolerance is desirable, however, it is rarely achieved in practice. It is often so that, for safety reasons, the system can not accommodate more than one error during its lifetime; it is then required that the system maintains its integrity after the error has occurred, which is equivalent to avoiding another error until termination.

To express this sort of properties in our framework, we have defined a new temporal operator $\diamondsuit_w(p, q)$. We have characterized it formally, by defining it over game trees. Proving enforcement of $\diamondsuit_w(p, q)$ reduces to the proof rule that we have proposed as Lemma 10.

Our work relies on the angel-demon game formalization within the dually nondeterministic weakest precondition framework [34], and on its later extensions [35, 37, 38].

In general, relationships between players may involve both cooperation and competition. To make the synthesis possible, in our case, the angel competes with the demon.

We have started with an angelic nondeterministic assignment as the model of the controller, and a demonic update for the behavior of the plant. Any angelic or demonic nondeterministic behavior can be cast into a nondeterministic assignment, respectively. Therefore, this way of modeling is comprehensive in expressing any kind of nondeterministic choice.

In either invariance or reachability case, the synthesis subsumes two main steps. Firstly, we check whether the angel can enforce the required behavior (A1, A2 of section 7.3.5, and also the second paragraph of section 7.4.3 show how the first step is applied in practice). If this first step holds, we extract the angelic winning strategy next (step B in section 7.3.5, and last paragraph of section 7.4.3).

In order to restrict the angelic choices to the ones that establish the safety property, by a correctness-preserving transformation, we have used backward propagation of assertions. The assertion is the computed weakest precondition for the demonic assignment to establish the invariant. This precondition is used to rewrite the angelic update. The end-result is a correct-by-construction controller, tailored to the required behavior.

Two illustrative case-studies have shown the application of the proposed approach, in practice. Due to Lemma 7 and the method described in section 7.3.2, we have synthesized an invariance controller for a producer - consumer - like system. Next, by applying the proof rule of Lemma 10, we have constructed a control strategy to win games intended to model *fault-tolerant* systems.

Related Work. Viewing a reactive system as a two-player game is not a new idea, it can be traced back to Ramadge and Wonham [139], and Pnueli and Rosner [137]. The authors developed synthesis algorithms for finite-state discrete systems, and showed that finding a winning strategy for the game was equivalent to synthesizing a controller that satisfied the requirements.

Recently, on-the-fly algorithms have been developed, by Tripakis and Altisen, for solving the issue of controller synthesis for discrete and dense-time systems [154]. The method is restricted to finite-state systems. In the discrete case, the algorithms are fully on-the-fly; a strategy is returned as soon as it is found, thus the state space does not necessarily have to be entirely generated.

Asarin, Maler, and Pnueli also apply concurrent game techniques to construct discrete controllers. The system is viewed as a timed automaton with trivial continuous dynamics [18]. The authors develop fixed-point algorithms in order to compute the maximal strategy. The method uses a "predecessor" operator that might imply a resource-consuming implementation, and also the exploration of possibly unreachable states. Similar algorithms suited for model-checking are proposed by Maler, Pnueli and Sifakis, who solve the problem of infinite-state controller synthesis for timed games, symbolically [122].

A deductive approach to controller synthesis is also proposed by Manna and Sipma [123]. The authors follow an incremental controller design pattern, applied to hybrid systems. The system and its environment are modeled as *actor phase transition systems* [146]. Verification rules are used to determine whether the system together with a control strategy meets the specification. The method is applied only to the synthesis of invariance controllers.

Slanina [147] develops proof rules for safety and response linear temporal logic properties of reactive system games. However, the equivalent of our second synthesis step, that is, extracting the (angelic) winning strategy, is not apparent. Moreover, the author does not use a two-fold nondeterminism, neither does he apply his theoretical results on detailed examples.

The idea of refining an initial non-implementable specification towards a correct implementation, by making successively more transitions explicit, is also applied by Henzinger, Manna and Pnueli on hybrid control systems modeled as *Phase Transition Systems* [91]. The requirement is modeled as a *hybrid temporal logic* property. Even if the notion of game is not explicitly used, a stringent form of refinement is defined such that the controller "wins" no matter how demonically the environment behaves. However, our angelic-demonic dualism that makes the two parties (controller - environment) distinguishable is suggested by the authors through partitioning variables into *controlled variables* and *environment variables*.

Our synthesis method is general and can be applied as such, to both infinite and finite systems. This is an immediate consequence of the deductive-based construction. An elegant way to handle the control of infinite-state concurrent (multi-player) game structures, based on abstraction, has been proposed by Henzinger et al. [90]. Properties of interest (of the abstracted system) are specified

in the *alternating-time temporal logic* [15]; the ”worst-case“ created abstraction (less powerful controller, more powerful environment) is then model-checked with MOCHA [16]. In comparison, our games are sequential. Nevertheless, they have proved to be a simple, well-suited model for reactive systems. An advantage is the fact that, even if infinite, the reactive game that we consider need not be abstracted to a finite instance. Deductive rules apply directly on the (possibly infinite-state) original model.

As distinct from the fixed-point symbolic synthesis algorithms developed by various researchers [18, 99, 122], our game-based method is fit for interactive theorem proving (PVS [65, 144], HOL etc.). To support this claim, we have proved the invariance property of the producer-consumer system, in PVS.

A very useful tool that can increase the level of automation is the *weakest precondition and correctness calculator* [56]. One can use this tool for proving the total correctness conditions needed in both synthesis cases.

Chapter 8

Conclusions and Discussion

We have presented a methodology for building correct-by-construction reactive systems, modeled by action systems or their continuous counterparts. The development process has served several viewpoints, yet all supported by the refinement calculus dedicated correctness-preserving transformations.

Since we have been analyzing reactive systems, we could not avoid the problems regarding their *behavior control* and *compositionality*. We have agreed to meet the challenges by solving the associated problems within our framework by resorting just to the standard demonic behavior of action systems.

Multiple, *simultaneously active* computing agents that interact with one another are sine-qua-non parts of any complex reactive system. A feasible, library-based bottom-up design methodology requires that the designer composes the system from parallel concurrent components called *modules*. The research presented on this topic has been motivated by an analysis of control aspects and of modular design techniques, as supported by the current action systems formal framework. We have exemplified that the interleaving concurrency paradigm might require modeling of cumbersome protocols among parallel reactive modules, in order to guarantee correctness. Our solution for avoiding the overload on modules comes as a synchronization mechanism. It implies a new virtual execution model of action systems, applicable to both discrete and hybrid designs.

Synchronized action systems are suitable for designing reactive systems that have to present a simultaneous global response to sets of input stimuli. To achieve this, we have introduced a new parallel composition operator (sharp, $\sharp$) that ensures correct outputs to all sets of inputs, without employing communication channels between modules. Consequently, our mechanism bears the advantage of less coding effort, in practice. The new execution model requires a certain type of action systems that we call *partitioned action systems*, which separate local actions from global actions. We recall the important remark that the synchronization operator increases the external system determinism, while preserving an internal nondeterministic execution of modules.

The proofs on the usefulness of our synchronized parallel environment, with

respect to modular design have showed that the capabilities of the action systems framework, for modularity, are improved. This translates into being able to carry out (trace) refinements of modules, modeled as action systems or continuous action systems, in isolation, without knowledge about the invariants of the other modules of the parallel environment. However, the invariants needed for trace refinement should also be proper (meaning that they depend only on the system's own write variables). Theorem 3, and Corollary 1 of chapter 3 demonstrate these claims.

As a precursor to full formal analysis, simulation of hybrid system models can be used effectively, especially if the state space is represented *symbolically*. This allows for the modeling of a potentially infinite number of states, and for the simulation of a potentially infinite number of trajectories in one symbolic simulation.

We have built such a tool using Mathematica, a commercial symbolic manipulation program [156]. The tool takes a description of any CAS as input, and provides automatically a symbolic simulation of the system, up to a given maximum time. The restrictions on the simulation are essentially those of Mathematica. Nevertheless, more efficient algorithms for solving the satisfiability problem for action systems guards, which are boolean conditions, need to be implemented. Further on, we have used the tool for validation purposes, in chapter 6, while developing *Earliest-Deadline-First* scheduled systems. An important aspect of our tool is the fact that it does not require any semantic changes of the model, which is a CAS. Moreover, being symbolic, it does not use a fixed- or variable-step numerical solver, at least for the analysis of linear hybrid systems. Another advantage is that it does not require abstraction of the continuous component either.

Many reactive systems are defined using parameters. They are intended to work correctly under specific parametric conditions. These relationships may be hard to find by following an intuitive approach alone, especially if we treat *hybrid systems*, which exhibit a continuous behavior interleaved with discrete control decisions. A combination of analysis tools can therefore be beneficial, at first to help intuition and rule out bad candidates, and then to verify exhaustively all reachable configurations. The latter would eventually lead to finding constraints on parameters, defining the set of all possible values for which the parametric system satisfies a property.

In this endeavor, one is helped by model-checkers like UPPAAL [117], HYTECH [87, 88] or TREX [48]. The former does not support synthesis of parameters as such, it rather verifies one's guess with respect to their relationship or range of values. In contrast, both HYTECH and TREX are suitable, in some cases, for extracting parametric conditions, automatically. For example, as stated by Henzinger et al. [93], systems with complex relationships between multiple parameters and timing constants can quickly lead to arithmetic overflow, when analyzed with HYTECH, whereas analysis with a single parameter is often successful.

We have given a general mathematical proof to the parametric reachability problem, based on traditional forward analysis, applicable even in those cases where relationships between parameters can not be guessed. The salient point of

our approach is its capacity to deal with big numbers of parameters and clocks. The method is based on iterative invariance checking, by superposition of lemmas, each stating a new predicate as the new hybrid system property. Thus, the process would tremendously benefit from mechanization of CAS theory. One could use, for instance, PVS and its powerful decision procedures to automate invariance checking, and also the refinements performed in chapter 6, in the real-time case. We expect such automation to be instrumental in the assessment of the applicability of our method to larger case-studies.

When timing requirements are set on top of the functional ones, for any type of reactive system, be it discrete or hybrid, we need to find a means to cope with them, in design. This should be done somewhat regardless of the respective functional behavior. Being faithful to this viewpoint too, we have presented a top-down method for the incremental construction of scheduled systems, within the same refinement calculus framework. We have applied the existing techniques of the latter, in an innovative way. Our development process starts with a nondeterministic conjunctive specification, and applies refinement rules of propagating context assertions, in order to enforce the required schedulability, mutual exclusion and scheduling policy conditions. Next, we take a step further and provide a monolithic implementation of the constructed system, via trace refinement. After a series of correctness-preserving transformations, we end up with a two-module implementation that clearly separate functionality between the scheduler and the tasks.

The last viewpoint is rooted in the opinion that it is beneficial to start with a nondeterministic high-level model, when approaching the design of a reactive system. This gives flexibility in modeling and frees the designer from the burden of taking into account implementation details, from the beginning. We have proposed a game-based method for the synthesis of invariance and certain reachability controllers. The usual approaches to the control problem and the synthesis of controlling strategies are *algorithmic* and can only be applied to finite-state systems [114]. Our solution fits both infinite state sequences, as well as terminating ones, without requiring *abstraction*, as proposed by Henzinger et al., to handle the control of infinite-state systems [90].

The idea of synthesis carried out by playing games with dually nondeterministic statements has been appealing to us, since our framework supports two kinds of nondeterminism, angelic and demonic. It came then naturally, to identify the behavior of the controller with an angelic statement, and the plant's actions with a demonic one.

We have aimed at reaching correct controllers for invariance, and reliable controllers for reachability. In each case, the angel is supposed to enforce a temporal property, thus guaranteeing a win under any scenario proposed by the demon. Our approach relies heavily on the recent work of Back and von Wright, who have defined temporal properties in the extended predicate transformer framework [37, 38]. Their work made it possible for us to introduce the correctness rule for proving enforcement of *weak-response* properties. These are defined by means of

extended trace semantics of action systems, namely, *game tree semantics*.

Our method starts by checking whether the angel has a way to win the game with respect to the specified requirement. The latter is either an *always* property, when we tackle synthesis of controllers for *invariance*, or a special form of *eventually* property, which we call *weak response* and denote by $\Diamond_w(p, q)$, when addressing synthesis of reliable controllers for reachability. In the first case, the controller has to be able to keep the system within a safe set of states. To enforce weak response, the angel has to find a way to terminate in a state of q if condition p holds. Alternatively, if $\neg p$ is true, then the angel wins if it can keep the system within $\neg p$ until the end of system execution.

If we succeed in proving that there exists such a winning strategy for the angel, we extract it, next, by rewriting the respective angelic statement, in a certain context resulted from the correctness proofs carried out in the first step. This transformation reveals the actual choices of the angel. The result is a correct, implementable model, which is guaranteed to preserve the required temporal property.

Most interesting with respect to this method is the fact that, while applying it on practical case-studies, new information is extracted on the way. For example, lower / upper limits on parameters, or even number of iterations needed in order to establish the requirement result while discharging the proof obligations. This is an important point that adds to the fact that we avoid using *backward fixed-point iterations* of *symbolic predecessor* operators, as many of the approaches targeting the same result do [18, 99, 122]. The experience gathered out of applying our game-based method on several case-studies reinforced the opinion that the level of generality and insight provided by a deductive analysis method can not be attained with model-checking.

We came to the conclusion that deductive methods, even more, if supported by interactive theorem proving, are viable alternatives to algorithmic methods. Timed automata [9], hybrid automata [86] and state-exploration techniques are invaluable for quickly analyzing a specific system. On the other hand, as pointed out by Dutertre, if one is interested in the analysis of certain classes of systems, or infinite-state systems, deductive methods prove more powerful [73]. Nevertheless, they also involve more human effort than their algorithmic counterparts.

Our work on discrete reactive systems, hybrid systems analysis and real-time scheduler construction proved our initial claim that the action systems framework, together with the refinement calculus methods are good candidates for a unified, effective and rigorous development environment. Diagrammatic reasoning and model-checking tools targeting this formal framework are believed to be a must for future applications of our methodology to emergent topics regarding systems research.

Limitations. Each of our construction and analysis approaches has certain limitations, which we underline in the following.

Invariants are essential for the trace refinements carried out in chapter 3, parameter synthesis method of chapter 5, and stepwise refinements of the real-time models of chapter 6. Also, proving the existence of angelic winning strategies during the process of controller synthesis for discrete systems (chapter 7) involves coming up with adequate system invariants. This step might not be trivial, especially for games with complex rules. For the process of finding the right invariants not to become a stumbling-block, one could for instance consider programming invariants first, rather than the behavioral specification. Among others, Back advocated *invariant based programming* (supported by *invariant diagrams*) as being an easier way to construct both the program and the respective conditions that need to hold [23]. Alternatively, tools like SAL (*Symbolic Analysis Laboratory*) [43] can help in finding a first approximation of the invariant of a state-transition system, and its further strengthened versions. Given the fact that action systems are indeed state-transition systems, one could employ SAL in the design process. In SAL, the underlying technique of invariant construction is based on a combination of least and greatest fixed-point computation of reachable states [150].

The applicability of the Mathematica-based symbolic simulator introduced in chapter 5 could possibly benefit from more efficient guard solving algorithms. Such improvement might consequently speed up the computation of the minimum time point when some action of the hybrid model under analysis becomes enabled. Also, in order to properly assess the utility, and discover the limitations of our tool, one needs to extensively simulate hybrid systems characterized by involved nonlinear continuous evolutions. Last but not least, the simulation tool awaits the design of a suitable graphical user interface that would use Mathematica as a back-end.

Throughout this study we have discussed and proposed solutions to some of the issues of reactive system design, from a formal perspective. Unfortunately, the relatively small case-studies can not answer questions about size and complexity of our methods when applied to real-world systems. Scalability is clearly not exercised within this context. We can just hope that the presented examples have added some merit to the theoretical results.

Future Work. The research carried out in this thesis can be extended in several directions. It could be interesting and helpful to improve on the mechanism of synchronized composition of action systems, such that the requirement of disjoint sets of global write variables is removed. This would imply the modeling, within the global action *Update*, of a concurrency protocol that regulates the modules' access to common resources. Preliminarily, such an approach has been studied by Seceleanu and Garlan [143], for modeling self-adaptive systems, in multimedia environments. A hierarchy of synchronized partitioned action systems is employed in order to accommodate various user requirements, while specifying a system that delivers multimedia services.

Resource sharing (in a real-time environment) and multiprocessor cost-efficient scheduling are also among prospective lines of research. The refinement techniques

niques presented in chapter 6 may be adapted to real-time interdependent tasks, possibly executing on distinct processors. Deductive rules could help in finding the optimized strategy to produce, for example, a low-power, distributed schedule.

One can think of developing *controller synthesis algorithms* for action systems. These could then be implemented in a model-checker for action systems, to provide automated support. In this way, by combining the method introduced in chapter 7, with the algorithmic one, one might be able to perform controller construction for finite-state systems, automatically, and for infinite-state systems, interactively, within the same formal framework.

Another possible direction targets game-based synthesis of controllers for real-time and hybrid systems. In these cases, the game techniques need to take into consideration the time-advancing statement, too. Therefore, the simple sequence of nondeterministic assignments might not suffice anymore.

Chapter 9

Appendix

Preliminaries

Before presenting the proofs (in a calculational format) of some of the statements of chapter 3, we introduce those particular results that help us achieve our goals.
• We will make use of Corollary 27 proved in [36]:

Corollary 2 *Assume that G and H are conjunctive predicate transformers and that $g \wedge h \equiv \mathsf{false}$. Then*

$$\mathbf{do}\ g \rightarrow G\ [\!]\ h \rightarrow H\ \mathbf{od}\ = \mathbf{do}\ h \rightarrow H\ \mathbf{od}\ ; \mathbf{do}\ g \rightarrow (G\ ; \mathbf{do}\ h \rightarrow H\ \mathbf{od}\ )\ \mathbf{od}$$

In the above, intuitively speaking, the condition $g \wedge h \equiv \mathsf{false}$ states that the statements G and H exclude each other, that is, they cannot be enabled simultaneously.
• We will also make use of the Theorem 31 proved in [36]:

Theorem 5 *Assume that G and H are conjunctive predicate transformers. Assume further that $H.\mathsf{true} \equiv \mathsf{true} \wedge g \notin wH$, which means that statement H terminates and preserves g. Then*

$$\mathbf{do}\ g \rightarrow G\ [\!]\ \neg g \wedge h \rightarrow H\ \mathbf{od}\ = \mathbf{do}\ g \rightarrow G\ \mathbf{od}\ ; \mathbf{do}\ h \rightarrow H\ \mathbf{od} \qquad \text{(A-1)}$$

• We recall some of the weakest precondition rules [71], which we apply:

1. wp rule for guarded action: $(g \rightarrow S).Q \ \triangleq\ g \Rightarrow S.Q$
2. wp rule for choice: $(S_1\ [\!]\ S_2).Q \ \triangleq\ S_1.Q \wedge S_2.Q$
3. wp rule for assignment statement: $(x := e).Q \ \triangleq\ Q[x := e]$
4. wp rule for sequential composition: $(S_1\ ; S_2).Q \ \triangleq\ S_1.(S_2, Q)$

• We additionally recall the definition of a loop as the least fixed point of the unfolding function [35]:

$$\mathsf{while}\ g\ \mathbf{do}\ S\ \mathbf{od}\ \triangleq\ (\mu X \cdot\ \mathbf{if}\ g\ \mathbf{then}\ S\,;X\ \mathbf{else}\ \mathsf{skip}\ \mathbf{fi}\ ) \qquad \text{(A-2)}$$

We also have $\mathbf{do}\ g \rightarrow S\ \mathbf{od}\ = \mathsf{while}\ g\ \mathbf{do}\ S\ \mathbf{od}$.
• We state here another helpful theorem, and three loop transformation rules (g, α predicates), as follows.

Theorem 6 *Assume that G, H and W are conjunctive predicate transformers. Then*

$$(G \parallel H)\,; W = (G\,; W) \parallel (H\,; W)$$

• Loop elimination rule [35].

$$\{\neg g\}\,; \mathbf{do}\ g \rightarrow S\ \mathbf{od}\ = \{\neg g\} \qquad \text{(A-3)}$$

• Remove one iteration loop.

$$\{g\}\,; \mathbf{do}\ g \rightarrow S\,; \{\neg g\}\ \mathbf{od}\ = S\,; \{\neg g\} \qquad \text{(A-4)}$$

Proof.

$$\begin{array}{cl} & \{g\}\,; \mathbf{do}\ g \rightarrow S\,; \{\neg g\}\ \mathbf{od} \\ = & \{\ \text{definition}\ (A-2),\ \text{unfolding}\} \\ & \{g\}\,; \mathbf{if}\ g\ \mathbf{then}\ S\,; \{\neg g\}\,; \mathbf{do}\ g \rightarrow S\,; \{\neg g\}\ \mathbf{od}\ \mathbf{else}\ \mathsf{skip}\mathbf{fi} \\ = & \{\text{logic}\} \\ & S\,; \{\neg g\}\,; \mathbf{do}\ g \rightarrow S\,; \{\neg g\}\ \mathbf{od} \\ = & \{\text{loop elimination rule}\ (A-3)\} \\ & S\,; \{\neg g\} \end{array}$$

• Propagation of assertion inside loop [35]:

$$\{\alpha\}\,; \mathbf{do}\ g \rightarrow S\ \mathbf{od}\ = \{\alpha\}\,; \mathbf{do}\ g \rightarrow \{\alpha\}\,; S\ \mathbf{od} \qquad \text{(A-5)}$$

A-1 Proof of Theorem 2 (chapter 3)

(a) By Definition 3.6, the synchronized parallel composition of the partitioned action systems

$$\begin{array}{lcl} \mathcal{A}_1(z_1) & \triangleq & \mathbf{begin\ \ var}\ x_1 \bullet\ Init_1\,; \mathbf{do}\ g_L^1 \rightarrow L_1\ \parallel\ g_S^1 \rightarrow S_1\ \mathbf{od\ \ end} \\ \mathcal{A}_2(z_2) & \triangleq & \mathbf{begin\ \ var}\ x_2 \bullet\ Init_2\,; \mathbf{do}\ g_L^2 \rightarrow L_2\ \parallel\ g_S^2 \rightarrow S_2\ \mathbf{od\ \ end} \end{array}$$

is given by the system

$$
\begin{array}{ll}
 & \mathcal{P}(z) \\
\triangleq & \textbf{begin var } x \,;\, sel[1..2] : \mathsf{Bool} \,;\, run : \mathsf{Nat} \bullet Init; \\
 & \textbf{do } gg_A \rightarrow \\
 & \quad\quad run = 0 \wedge \neg sel[1] \rightarrow sel[1] := \mathsf{true} \,;\, run := 1 \\
 & \quad [\!] \; run = 0 \wedge \neg sel[2] \rightarrow sel[2] := \mathsf{true} \,;\, run := 2 \\
 & \quad [\!] \; run = 1 \wedge g_L^1 \rightarrow L_1 \\
 & \quad [\!] \; run = 1 \wedge \neg g_L^1 \wedge g_S^1 \rightarrow wS_1c := wS_1 \,;\, S_1' \,;\, run := 0 \\
 & \quad [\!] \; run = 1 \wedge \neg gg_{A_1} \rightarrow run := 0 \\
 & \quad [\!] \; run = 2 \wedge g_L^2 \rightarrow L_2 \\
 & \quad [\!] \; run = 2 \wedge \neg g_L^2 \wedge g_S^2 \rightarrow wS_2c := wS_2 \,;\, S_2' \,;\, run := 0 \\
 & \quad [\!] \; run = 2 \wedge \neg gg_{A_2} \rightarrow run := 0 \\
 & \quad [\!] \; sel \wedge run = 0 \rightarrow Update \,;\, sel := \mathsf{false} \\
 & \textbf{od} \\
 & \textbf{end}
\end{array}
$$

We denote the actions of $\mathcal{P}$ as (where $j \in [1..2]$):

$$
\begin{array}{lcl}
Sel & \triangleq & run = 0 \rightarrow Sel_1 \; [\!] \; Sel_2 \\
Sel_1 & \triangleq & \neg sel[1] \rightarrow sel[1] := \mathsf{true} \,;\, run := 1 \\
Sel_2 & \triangleq & \neg sel[2] \rightarrow sel[2] := \mathsf{true} \,;\, run := 2 \\
A_j & \triangleq & A_j^1 \; [\!] \; A_j^2 \; [\!] \; A_j^3 \\
A_j^1 & \triangleq & run = j \wedge g_L^j \rightarrow L_j \\
A_j^2 & \triangleq & run = j \wedge \neg g_L^j \wedge g_S^j \rightarrow C_j \,;\, S_j' \,;\, run := 0 \\
A_j^3 & \triangleq & run = j \wedge \neg gg_{A_j} \rightarrow run := 0 \\
C_j & \triangleq & wS_jc := wS_j \\
U & \triangleq & sel \wedge run = 0 \rightarrow Update \,;\, sel := \mathsf{false}
\end{array}
$$

Notice that the composition $L \triangleq gg_A \rightarrow Sel \; [\!] \; A_1 \; [\!] \; A_2$ forms the local action of $\mathcal{P}$, while the action U is its global action. The first two requirements for showing that $\mathcal{P}$ is a partitioned action system are immediate. We just have to analyze the third one, that is, to prove that $(\textbf{do } L \textbf{ od }).(\neg gL \wedge gU) \equiv \mathsf{true}$.

We start by observing that only the situation when $gg_A \equiv \mathsf{true}$ is of interest, otherwise the whole system $\mathcal{P}$ is disabled. Therefore, we only have to show that

$$
(\textbf{do } Sel \; [\!] \; A_1 \; [\!] \; A_2 \textbf{ od }).(\neg gL \wedge gU) \equiv \mathsf{true}
$$

We proceed as follows:

$$
\begin{array}{ll}
 & \textbf{do } Sel \; [\!] \; A_1 \; [\!] \; A_2 \textbf{ od} \\
= & \{ \text{ Corollary 2 } \} \\
 & \textbf{do } Sel \textbf{ od } ; \textbf{do } A_1 \; [\!] \; A_2 \,;\, \textbf{do } Sel \textbf{ od } \textbf{ od}
\end{array}
\tag{A-6}
$$

$$
\begin{aligned}
\sqsupseteq\ & \{\text{Init or } A_j^2 \,[\!]\, A_j^3 \text{ establishes } run = 0\ , \{p\} \sqsubseteq \mathsf{skip} \\
& \quad\quad \text{introduce assertions}\ \} \\
& \{run = 0\}\ ; \mathbf{do}\ run = 0 \rightarrow Sel_1 \,[\!]\, Sel_2\ ; \{run \neq 0\}\ \mathbf{od}\ ; \\
& \quad\quad \mathbf{do}\ A_1 \,[\!]\, A_2\ ; \mathbf{do}\ Sel\ \mathbf{od}\ \ \mathbf{od} \\
=\ & \{\ \text{rule (A-4), drop assertion}\ \} \\
& (Sel_1 \,[\!]\, Sel_2)\ ; \mathbf{do}\ A_1 \,[\!]\, A_2\ ; \mathbf{do}\ Sel\ \mathbf{od}\ \ \mathbf{od} \\
=\ & \{\ \text{Theorem 6}\ \} \\
& (Sel_1\ ; \mathbf{do}\ A_1 \,[\!]\, A_2\ ; \mathbf{do}\ Sel\ \mathbf{od}\ \ \mathbf{od}\ ) \\
& \quad\quad [\!]\ (Sel_2\ ; \mathbf{do}\ A_1 \,[\!]\, A_2\ ; \mathbf{do}\ Sel\ \mathbf{od}\ \ \mathbf{od}\ ) \\
=\ & \{\ \text{Theorem 6,} \\
& \text{notation: } Choice_2 \ \hat{=}\ (Sel_2\ ; \mathbf{do}\ A_1 \,[\!]\, A_2\ ; \mathbf{do}\ Sel\ \mathbf{od}\ \ \mathbf{od}\ )\} \\
& (Sel_1\ ; \mathbf{do}\ A_1\ ; \mathbf{do}\ Sel\ \mathbf{od}\ \ [\!]\ A_2\ ; \mathbf{do}\ Sel\ \mathbf{od}\ \ \mathbf{od}\ )\ [\!]\ Choice_2 \\
=\ & \{\ \text{Corollary 2}\ \} \\
& (Sel_1\ ; \mathbf{do}\ A_1\ ; \mathbf{do}\ Sel\ \mathbf{od}\ \ \mathbf{od}\ ; \\
& \quad \mathbf{do}\ A_2\ ; \mathbf{do}\ Sel\ \mathbf{od}\ \ ; \mathbf{do}\ A_1\ ; \mathbf{do}\ Sel\ \mathbf{od}\ \ \mathbf{od}\ \ \mathbf{od}\ )\ [\!]\ Choice_2
\end{aligned}
$$

We continue by focusing on the sequence $Sel_1\ ; \mathbf{do}\ A_1\ ; \mathbf{do}\ Sel\ \mathbf{od}\ \ \mathbf{od}$:

$$
\begin{aligned}
& Sel_1\ ; \mathbf{do}\ A_1\ ; \mathbf{do}\ Sel\ \mathbf{od}\ \ \mathbf{od} \\
=\ & \{\ \text{definition of } A_1\text{, Theorem 6}\ \} \\
& Sel_1\ ; \mathbf{do}\ A_1^1\ ; \mathbf{do}\ Sel\ \mathbf{od}\ \ [\!]\ A_1^2\ ; \mathbf{do}\ Sel\ \mathbf{od}\ \ [\!]\ A_1^3\ ; \mathbf{do}\ Sel\ \mathbf{od}\ \ \mathbf{od} \\
=\ & \{\ \text{Theorem 5 w.r.t. } A_1^1\ ; \mathbf{do}\ Sel\ \mathbf{od}\ \ [\!]\ A_1^2\ ; \mathbf{do}\ Sel\ \mathbf{od}\ \text{ and } A_1^3\} \\
& Sel_1\ ; \mathbf{do}\ A_1^1\ ; \mathbf{do}\ Sel\ \mathbf{od}\ \ [\!]\ A_1^2\ ; \mathbf{do}\ Sel\ \mathbf{od}\ \ \mathbf{od}\ ; \qquad \text{(A-7)} \\
& \quad\quad \mathbf{do}\ A_1^3\ ; \mathbf{do}\ Sel\ \mathbf{od}\ \ \mathbf{od} \\
=\ & \{\ \text{Theorem 5 w.r.t. } A_1^1\ ; \mathbf{do}\ Sel\ \mathbf{od}\ \text{ and } A_1^2\ ; \mathbf{do}\ Sel\ \mathbf{od}\ \ \} \\
& Sel_1\ ; \mathbf{do}\ A_1^1\ ; \mathbf{do}\ Sel\ \mathbf{od}\ \ \mathbf{od}\ \ ; \mathbf{do}\ A_1^2\ ; \mathbf{do}\ Sel\ \mathbf{od}\ \ \mathbf{od}\ ; \\
& \quad\quad \mathbf{do}\ A_1^3\ ; \mathbf{do}\ Sel\ \mathbf{od}\ \ \mathbf{od}
\end{aligned}
$$

Next, since $gg_A \equiv \mathsf{true}$, the last element of the sequence $\mathbf{do}\ A_1^3\ ; \mathbf{do}\ Sel\ \mathbf{od}\ \ \mathbf{od}$ can be replaced by skip. We focus on the first two terms of the sequence:

$$
\begin{aligned}
& Sel_1\ ; \mathbf{do}\ A_1^1\ ; \mathbf{do}\ Sel\ \mathbf{od}\ \ \mathbf{od} \\
=\ & \{\ run \notin wA_1^1\ \} \\
& Sel_1\ ; \mathbf{do}\ A_1^1\ ; \{run = 1\}\ ; \mathbf{do}\ Sel\ \mathbf{od}\ \ \mathbf{od} \qquad \text{(A-8)} \\
=\ & \{\ \text{definition of } Sel\text{, rule (A-3), drop assertion}\ \} \\
& Sel_1\ ; \mathbf{do}\ A_1^1\ \mathbf{od}
\end{aligned}
$$

We already know ($\mathcal{A}_1$ is partitioned) that $\mathbf{do}\ g_L^1 \ \rightarrow\ L_1\ \mathbf{od}$ terminates and establishes $\neg g_L^1 \wedge g_S^1$. Hence, $(run\ =\ 1\ \rightarrow\ (\mathbf{do}\ g_L^1\ \rightarrow\ L_1\ \mathbf{od}\ ))\ =\ \mathbf{do}\ A_1^1\ \mathbf{od}$ terminates and establishes $run\ =\ 1 \wedge \neg g_L^1 \wedge g_S^1$. As $sel[1] \notin wA_1^1$, we actually have that, after executing $\mathbf{do}\ A_1^1\ \mathbf{od}$, $sel[1] \wedge run = 1 \wedge \neg g_L^1 \wedge g_S^1$ holds.

We continue with the analysis of $\mathbf{do}\ A_1^2\ ; \mathbf{do}\ Sel\ \mathbf{od}\ \ \mathbf{od}$. Considering the

above, we have

$$
\begin{array}{ll}
 & \{sel[1] \wedge run = 1 \wedge \neg g_L^1 \wedge g_S^1\} \,;\, \mathbf{do}\ A_1^2 \,;\, \mathbf{do}\ Sel\ \mathbf{od}\ \ \mathbf{od} \\
= & \{\text{general rule: } \{p \wedge q\} = \{p\} \,;\, \{q\}\} \\
 & \{sel[1]\} \,;\, \{run = 1 \wedge \neg g_L^1 \wedge g_S^1\} \,;\, \mathbf{do}\ A_1^2 \,;\, \mathbf{do}\ Sel\ \mathbf{od}\ \ \mathbf{od} \\
= & \{\text{ rule (A-5), as } sel[1] \notin wA_1^2\ \} \\
 & \{sel[1]\} \,;\, \{run = 1 \wedge \neg g_L^1 \wedge g_S^1\}; \\
 & \quad \mathbf{do}\ A_1^2 \,;\, \{sel[1]\} \,;\, \mathbf{do}\ Sel\ \mathbf{od}\ \ \mathbf{od} \\
= & \{\ A_1^2 \text{ establishes } run = 0, \text{ strengthen assertion }\} \\
 & \{sel[1]\} \,;\, \{run = 1 \wedge \neg g_L^1 \wedge g_S^1\} \,;\, \mathbf{do}\ A_1^2; \\
 & \quad \{sel[1] \wedge run = 0\} \,;\, \mathbf{do}\ Sel\ \mathbf{od}\ \ \mathbf{od} \\
= & \{\text{ definition of } Sel, \text{ rewrite using context} \\
 & \quad \text{information } (\{sel[1] \wedge run = 0\})\ \} \\
 & \{sel[1]\} \,;\, \{run = 1 \wedge \neg g_L^1 \wedge g_S^1\} \,;\, \mathbf{do}\ A_1^2 \,;\, \{sel[1] \wedge run = 0\}; \\
 & \quad \mathbf{do}\ run = 0 \rightarrow Sel_2\ \mathbf{od}\ \ \mathbf{od} \\
= & \{\text{ introduce assertion } \{run \neq 0\}\} \\
 & \{sel[1]\} \,;\, \{run = 1 \wedge \neg g_L^1 \wedge g_S^1\} \,;\, \mathbf{do}\ A_1^2 \,;\, \{sel[1] \wedge run = 0\}; \\
 & \quad \mathbf{do}\ run = 0 \rightarrow Sel_2 \,;\, \{run \neq 0\}\ \mathbf{od}\ \ \mathbf{od} \\
= & \{\text{ rule (A-4), drop assertions }\} \\
 & \mathbf{do}\ A_1^2 \,;\, Sel_2\ \mathbf{od}
\end{array}
\tag{A-9}
$$

The above loop terminates, since both A_1^2 and Sel_2 terminate. Moreover, $(\mathbf{do}\ A_1^2\ ;\ Sel_2\ \mathbf{od}\ ).(run\ =\ 2 \wedge sel[2]) \equiv \mathsf{true}$. Applying a similar reasoning for the action $Choice_2$, we eventually come to the conclusion that:

$$(\mathbf{do}\ Sel \ [\!]\ A_1\ [\!]\ A_2\ \mathbf{od}\ ).(sel \wedge run = 0) \equiv \mathsf{true}$$

which means that the local action terminates, and it enables the execution of the global action of the system $\mathcal{P}$.It is easy to check that the other requirements of Definition 1 are also satisfied, thus, $\mathcal{P}$ is a partitioned action system. ■

(b) Follows from the commutativity of the choice operator. ■

A-2 Proof of Theorem 3 (chapter 3)

We assume the system $\mathcal{P}$ as being the synchronized composition of two action systems:

$$
\begin{array}{rl}
& \mathcal{P}(z) \\
\triangleq & \textbf{begin var } x \,;\, sel[1..2] : \mathsf{Bool} \,;\, run : \mathsf{Nat} \bullet Init; \\
& \textbf{do } gg_A \rightarrow \\
& \quad\quad run = 0 \land \neg sel[1] \rightarrow sel[1] := \mathsf{true} \,;\, run := 1 \\
& \quad [\!] \; run = 0 \land \neg sel[2] \rightarrow sel[2] := \mathsf{true} \,;\, run := 2 \\
& \quad [\!] \; run = 1 \land g_L^1 \rightarrow L_1 \\
& \quad [\!] \; run = 1 \land \neg g_L^1 \land g_S^1 \rightarrow wS_1c := wS_1 \,;\, S_1' \,;\, run := 0 \\
& \quad [\!] \; run = 1 \land \neg gg_{A_1} \rightarrow run := 0 \\
& \quad [\!] \; run = 2 \land g_L^2 \rightarrow L_2 \\
& \quad [\!] \; run = 2 \land \neg g_L^2 \land g_S^2 \rightarrow wS_2c := wS_2 \,;\, S_2' \,;\, run := 0 \\
& \quad [\!] \; run = 2 \land \neg gg_{A_2} \rightarrow run := 0 \\
& \quad [\!] \; sel \land run = 0 \rightarrow Update \,;\, sel := \mathsf{false} \\
& \textbf{od end}
\end{array}
$$

First, we give, without proof, one simple invariant of system $\mathcal{P}$:

$$(\textstyle\bigvee_{j\in[0..2]}(run = j)) \equiv \mathsf{true} \tag{A-10}$$

We state further that

$$I_0^1 \;\triangleq\; I_1 \;\land\; (sel[1] \land run \neq 1 \Rightarrow I_1') \tag{A-11}$$

is an invariant of the system $\mathcal{P}$, where

- I_1 is the proper invariant respected by the system $\mathcal{A}_1$. Therefore, we also have that $I_1[wS_1 := w'S_1, v := z] \equiv I_1[wS_1 := w'S_1, v := z'])$.

- $I_1' = I_1[wS_1 := wS_1^c]$

In the following, we show that I_0^1 is an invariant of every action of $\mathcal{P}$.

1. I_0^1 is preserved by action $A_1 \triangleq \neg sel[1] \land (run = 0) \rightarrow sel[1] := \mathsf{true}\,;$ $run := 1$. We have:

$$
\begin{array}{rl}
& A_1.I_0^1 \\
\equiv & \{\text{definition of } A_1,\ wp \text{ rules for guarded action,} \\
& \quad\quad \text{sequential comp., assignment}\} \\
& sel[1] \lor run \neq 0 \lor I_0^1[sel[1] := \mathsf{true}, run := 1]) \\
\equiv & \{\text{definition of } I_0^1,\ sel, run \text{ do not appear in } I_1 \text{ or in } I_1', \text{ logic}\} \\
& sel[1] \lor run \neq 0 \lor I_1 \\
\Leftarrow & \{\text{definition (A-11), logic}\} \\
& I_0^1
\end{array}
$$

2. I_0^1 is preserved by action $A_2 \triangleq \neg sel[2] \wedge (run = 0) \rightarrow sel[2] := \mathsf{true}$; $run := 2$. We have:

$$
\begin{array}{ll}
 & A_2.I_0^1 \\
\equiv & \{\text{definition of } A_2,\ wp \text{ rules for guarded action,} \\
 & \qquad\qquad \text{sequential comp., assignment}\} \\
 & sel[2] \vee run \neq 0 \vee I_0^1[sel[2] := \mathsf{true}, run := 2] \\
\equiv & \{\text{definition of } I_0^1,\ sel, run \text{ do not appear in } I_1 \text{ or in } I_1', \text{ logic}\} \\
 & sel[k] \vee run \neq 0 \vee (I_1 \wedge (sel[2] \Rightarrow I_1')) \\
\Leftarrow & \{\text{logic, relation (A-10)}\} \\
 & I_0^1
\end{array}
$$

3. That I_0^1 is preserved by the action $A_3 \triangleq (run = 2) \wedge g_L^2 \rightarrow L_2$, follows from the fact that A_3 does not write any of the variables mentioned by I_0^1, therefore the latter is an invariant, trivially.

4. I_0^1 is preserved by the action $A_4 \triangleq (run = 1) \wedge g_L^1 \rightarrow L_1$ comes from the fact that I is an invariant of $g_L^1 \rightarrow L_1$.

5. I_0^1 is preserved by the action $A_5 \triangleq (run = 1) \wedge \neg g_L^1 \wedge g_S^1 \rightarrow wS_1c := wS_1 ; S_1' ; run := 0,\ where\ S_1' = S_1[wS_1 := wS_1c]$. We first have that:

$$
\begin{array}{ll}
 & (x := y ; S[y := x]).Q[y := x] \\
\equiv & (x := y).(S[y := x].Q[y := x]) \\
\equiv & (x := y).(S.Q)[y := x]) \\
\equiv & ((S.Q)[x/y])[x := y] \\
\equiv & S.Q
\end{array}
\qquad \text{(A-12)}
$$

Next, we get:

$$
\begin{array}{ll}
 & A_5.I_0^1 \\
\equiv & \{wp \text{ rules for guarded action, sequential composition, assignment}\} \\
 & run \neq 1 \vee g_L^1 \vee \neg g_S^1 \vee (wS_1c := wS_1 ; S_1').I_0^1[run := 0] \\
\equiv & \{\text{definition of } I_0^1,\ run \text{ does not appear in } I_1, \text{ or in } I_1'\} \\
 & run \neq 1 \vee g_L^1 \vee \neg g_S^1 \vee (wS_1c := wS_1 ; S_1').(I_1 \wedge (sel[1] \Rightarrow I_1')) \\
\equiv & \{wp \text{ rule for conjunctive statements, } wS_1c \text{ does not appear in } I_1\} \\
 & run \neq 1 \vee g_L^1 \vee \neg g_S^1 \vee (I_1 \wedge (wS_1c := wS_1 ; S_1').(sel[1] \Rightarrow I_1')) \\
\equiv & \{wp \text{ rule for sequential composition}\} \\
 & run \neq 1 \vee g_L^1 \vee \neg g_S^1 \vee (I_1 \wedge (wS_1c := wS_1).(S_1'.(sel[1] \Rightarrow I_1'))) \\
\equiv & \{\text{definition of } I_1', \text{ relation (A-12), } sel \text{ does not mention } wS_1, wS_1c\} \\
 & run \neq 1 \vee g_L^1 \vee \neg g_S^1 \vee (I_1 \wedge S_1.(sel[1] \Rightarrow I_1)) \\
\Leftarrow & \{S_1.(\neg sel[1] \vee I_1) \Leftarrow (S_1.I_1 \vee S_1.(\neg sel[1]))\} \\
 & run \neq 1 \vee g_L^1 \vee \neg g_S^1 \vee (I_1 \wedge (S_1.I_1 \vee S_1.\neg sel[1]))
\end{array}
$$

$$\Leftarrow \quad \{\text{logic}\}$$
$$run \neq 1 \vee g_L^1 \vee \neg g_S^1 \vee (I_1 \wedge S_1.I_1)$$
$$\Leftarrow \quad \{I_1 \text{ is invariant of the original system: } I_1 \Rightarrow S_1.I_1, \text{ logic}\}$$
$$I_1$$
$$\Leftarrow \quad \{\text{definition of } I_0^1, \text{ logic}\}$$
$$I_0^1$$

6. The fact that I_0^1 is preserved by the action $A_6 \triangleq (run = 1) \wedge \neg g_L^1 \wedge \neg g_S^1 \rightarrow run := 0$ follows the lines of the previous proof.

7. I_0^1 is preserved by the action $A_7 \triangleq (run = 2) \wedge \neg g_L^2 \wedge g_S^2 \rightarrow wS_2c := wS_2 \,;\, S_2' \,;\, run := 0$, where $S_2' = S_2[wS_2 := wS_2c]$:

$$A_6.I_0^1$$
$$\equiv \quad \{wp \text{ rules for guarded action, sequential comp., assignment}\}$$
$$run \neq 2 \vee g_L^2 \vee \neg g_S^2 \vee (wS_2c := wS_2 \,;\, S_2').I_0^1[run := 0]$$
$$\equiv \quad \{\text{definition of } I_0^1,\ run, wS_2c, wS_2 \text{ do not appear in } I_1,\ \text{or in } I_1'\}$$
$$run \neq 2 \vee g_L^2 \vee \neg g_S^2 \vee (I_1 \wedge (sel[1] \Rightarrow I_1'))$$
$$\Leftarrow \quad \{\text{logic, relation (A-10)}\}$$
$$I_0^1$$

8. The fact that I_0^1 is an invariant of the action $A_8 \triangleq (run = 2) \wedge \neg g_L^2 \wedge \neg g_S^2 \rightarrow run := 0$ has a similar proof to the one for action A_7.

9. Proof of the fact that I_0^1 is preserved by the action $U \triangleq sel \wedge (run = 0) \rightarrow wS_1 := wS_1c \,;\, wS_2 := wS_2c \,;\, sel := \mathsf{false}$:

$$U.I_0^1$$
$$\equiv \quad \{wp \text{ rules for guarded action, sequential comp., assignment}\}$$
$$\neg sel \vee run \neq 0 \vee (wS_1 := wS_1c \,;\, wS_2 := wS_2c).I_0^1[sel := \mathsf{false}]$$
$$\equiv \quad \{\text{definition of } I_0^1, \text{ succesive application of } wp \text{ rules}\}$$
$$\neg sel \vee run \neq 0 \vee (I_1[wS_1, wS_2 := wS_1c, wS_2c] \wedge run \neq 1)$$
$$\equiv \quad \{\text{relation (A-10): } run \neq 0 \equiv (run = 1 \vee run = 2)\}$$
$$\neg sel \vee run \neq 0 \vee I_1[wS_1, wS_2 := wS_1c, wS_2c]$$
$$\equiv \quad \{I_1 \text{ is proper}\}$$
$$\neg sel \vee run \neq 0 \vee I_1[wS_1, wS_2 := wS_1c, wS_2]$$
$$\equiv \quad \{\text{drop assignment } wS_2 := wS_2\}$$
$$\neg sel \vee run \neq 0 \vee I_1[wS_1 := wS_1c]$$
$$\equiv \quad \{\text{notation}\}$$
$$\neg sel \vee run \neq 0 \vee I_1'$$
$$\Leftarrow \quad \{\text{logic}, \neg sel \equiv \neg sel[1] \vee \neg sel[2],\ run \neq 0 \equiv (run = 1 \vee run = 2)\}$$
$$I_0^1$$

The above show that I_0^1 is an invariant of the system $\mathcal{P}$. In a similar manner, we can show that

$$I_0^2 \triangleq I_2 \wedge (sel[2] \wedge \neg(run = 2) \Rightarrow I_2')$$

is also an invariant of $\mathcal{P}$. Hence, $I \triangleq I_0^1 \wedge I_0^2$ is an invariant of $\mathcal{P}$.
Properness. Notice further that:

$$\begin{array}{ll} & I_0^1[wS_1, wS_2 := wS_1c, wS_2c, sel := \mathsf{false}, v := v'] \\ \equiv & \{\text{definition } I_0^1\} \\ & I_1[wS_1, wS_2 := wS_1c, wS_2c, v := v'] \\ \equiv & \{v, wS_2 \notin w\mathcal{A}_1,\ I_1 \text{ is proper}\} \\ & I_1[wS_1, wS_2 := wS_1c, wS_2, v := v] \\ \equiv & \{I_1 \text{ is proper}\} \\ & I_1[wS_1 := wS_1c] \\ \equiv & \{sel, wS_2 \notin w\mathcal{A}_1,\ I_1 \text{ is proper}\} \\ & I_1[wS_1, wS_2 := wS_1c, wS_2c, sel := \mathsf{false}] \\ \equiv & \{\text{logic}\} \\ & (I_1 \wedge (sel[1] \wedge run \neq 1 \Rightarrow I_1'))[wS_1, wS_2 := wS_1c, wS_2c, sel := \mathsf{false}] \\ \equiv & \{\text{notation}\} \\ & I_0^1[wS_1, wS_2 := wS_1c, wS_2c, sel := \mathsf{false}] \end{array} \quad \text{(A-13)}$$

Hence, I_0^1 is a proper invariant of $\mathcal{P}$.

Repeating the above proof for the other invariant I_0^2 and summing up, we reach the conclusion that $I \triangleq I_0^1 \wedge I_0^2$ is a proper invariant of $\mathcal{P}$.

The results can be generalized to the synchronized composition of $k, k > 2$ partitioned action systems. ■

A-3 Proof of Corollary 1 (chapter 3)

Suppose that we have the partitioned action system $\mathcal{A}_j$, as part of the synchronized composition $\mathcal{P} = \mathcal{A}_1 \sharp \ldots \sharp \mathcal{A}_n$. Additionally, I_j is some invariant respected by $\mathcal{A}_j$, and we also have that $\mathcal{A}_j \sqsubseteq_{R_j, I_j} \mathcal{A}_j'$, following the requirements of Lemma 2. Thus, $\mathcal{A}_j'$ is a partitioned action system, too. Consequently [39], $I_j' \triangleq R_j \wedge I_j$ is an invariant of $\mathcal{A}_j'$. As R_j does not introduce any new global variables, it is independent of other variables than those of $\mathcal{A}_j, \mathcal{A}_j'$, and using a similar line of proof as in (A-13), one can prove that I_j' is proper.

We do not insist here on the (trivial, given the above assumptions) task of showing that $\mathcal{P} \sqsubseteq \mathcal{P}'$ ($\mathcal{P}' \triangleq \mathcal{A}_1 \sharp \ldots \sharp \mathcal{A}_j' \sharp \ldots \sharp \mathcal{A}_n$). Relevant is to show that, using the notations of Appendix A-2, $I' \triangleq I_0^1 \wedge \ldots \wedge I_0^{\prime j} \wedge \ldots \wedge I_0^n$ is an invariant of

$\mathcal{P}'$. This is solved by following the same steps as in Appendix A-2. Moreover, we have

$$
\begin{array}{ll}
& I'_j \triangleq R_j \wedge I_j \\
\Rightarrow & \{ \text{ notation: } Q_j \triangleq (sel[j] \wedge run \neq j) \Rightarrow I''_j, \\
& \quad I''_j \triangleq R \wedge I'_j[wS_1 := wS_1 c], \\
& \quad \text{definition (A-11): } I'^j_0 \triangleq R_j \wedge I_j \wedge Q_j, \text{ logic}\} \\
& I'^j_0 \Rightarrow I^j_0 \\
\Rightarrow & \{\text{logic}\} \\
& \quad I^1_0 \wedge \ldots \wedge I'^j_0 \wedge \ldots \wedge I^n_0 \\
& \quad \Rightarrow I^1_0 \wedge \ldots \wedge I^j_0 \wedge \ldots \wedge I^n_0 \\
\Rightarrow & \{\text{notation}\} \\
& \quad I_1 \wedge Q_1 \wedge \ldots \wedge I'_j \wedge Q_j \wedge \ldots \wedge I_n \wedge Q_n \\
& \quad \Rightarrow I_1 \wedge Q_1 \wedge \ldots \wedge I_j \wedge Q_j \wedge \ldots \wedge I_n \wedge Q_n \\
\Rightarrow & \{\text{logic}\} \\
& \quad I_1 \wedge \ldots \wedge I'_j \wedge \ldots \wedge I_n \\
& \quad \Rightarrow I_1 \wedge \ldots \wedge I_j \wedge \ldots \wedge I_n
\end{array}
$$

The above illustrate the fact that $\mathcal{P} \sqsubseteq_{R_j,I_j} \mathcal{P}'$, and that $\mathcal{P}'$ still preserves the properties of each of the original action system $\mathcal{A}_k,\ k \in [1..n]$, as expressed by the conjunction $I_1 \wedge \ldots \wedge I_j \wedge \ldots \wedge I_n$. ■

A-4 Refinement of *sys* $\mathcal{F}$ (chapter 3)

We consider the systems:

$$
\begin{array}{ll}
& \mathcal{F}(X, Z[0,..N-2], Y : T) \\
= & \textbf{begin } X, Z[0,..N-2], Y := x_0, z_0, y_0; \\
& \quad \textbf{do } Y := h[0] \times X + \sum_{k=1}^{N-1} h[k] \times Z[k-1] \textbf{ od} \\
& \textbf{end} : h[0,..N-1]
\end{array}
$$

and

$$
\begin{array}{ll}
& \mathcal{F}_S(X, Z[0,..N-2], Y : T) \\
= & \textbf{begin var } temp : T \,;\, step : [0,..N] \bullet \\
& \quad X, Z, Y := x_0, z_0, y_0 \,;\, temp := 0 \,;\, step := 0; \\
& \quad \textbf{do } step = 0 \rightarrow temp := 0 \,;\, step := step + 1 \\
& \quad\quad [\!] \ step \in [1,..N-1] \rightarrow temp := temp + h[step] \times Z[step-1]; \\
& \quad\quad\quad\quad step := step + 1 \\
& \quad\quad [\!] \ step = N \rightarrow Y := temp + X \times h[0] \,;\, step := 0 \\
& \quad \textbf{od} \\
& \textbf{end} : h[0,..N-1]
\end{array}
$$

We denote the actions of the above systems as:

$$
\begin{aligned}
A &\triangleq Y := X \times h[0] + \sum_{k=1}^{N-1} h[k] \times Z[k-1] \\
C &\triangleq C_1 \,;\, C_2, \\
C_1 &\triangleq step = N \rightarrow Y := temp + X \times h[0], \\
C_2 &\triangleq step := 0 \\
A_1 &\triangleq step = 0 \rightarrow temp := 0 \,;\, step := step + 1, \\
A_2 &\triangleq step \in [1, ..N-1] \rightarrow temp := temp + h[step] \times Z[step-1] \,; \\
&\qquad\qquad step := step + 1,
\end{aligned}
$$

We first show that

$$
I \triangleq (step = 1 \Rightarrow temp = 0) \wedge \bigwedge_{p=2}^{N} (step = p \Rightarrow temp = \sum_{k=1}^{p-1} h[k] \times Z[k-1])
$$

is an invariant of $\mathcal{F}_S$.

Observe first, that $I[step := 0] \equiv true$. Hence, $C.I \equiv true$, therefore $I \Rightarrow C.I$ holds, trivially. The same is valid for the action A_1. We analyze next the situation that concerns the action A_2.

$$
\begin{aligned}
& A_2.I \\
\equiv\ & \{wp \text{ rules for guarded action, sequential composition, assignment}\} \\
& step \in [1, ..N-1] \Rightarrow I[step := step + 1, temp := temp_{new}] \\
\equiv\ & \{\text{assume } step + 1 = p + 1 \in [2, ..N],\ \text{substitution}\} \\
& step \in [1, ..N-1] \Rightarrow \\
& \quad (step = p \Rightarrow temp = \sum_{k=1}^{p-1} h[k] \times Z[k-1]) \\
\Leftarrow\ & \{\ \text{logic}\} \\
& step = p \Rightarrow temp = \sum_{k=1}^{p-1} h[k] \times Z[k-1]) \\
\Leftarrow\ & \{\text{logic}\} \\
& (step = 1 \Rightarrow temp = 0) \\
& \quad \wedge \bigwedge_{p=2}^{N} (step = p \Rightarrow temp = \sum_{k=1}^{p-1} h[k] \times Z[k-1] \\
\equiv\ & \{\text{definition}\} \\
& I
\end{aligned}
$$

In order to prove the above specified refinement, we go through the requirements of Lemma 1:

1. Initialization: $I[step := 0, temp := 0, Z := z_0] \equiv \mathsf{true}$

2. Main action: We have to prove that $A \sqsubseteq_I C$. For this, we have to show that: $I \wedge A.Q \Rightarrow C.(I \wedge Q), \forall Q$. We proceed as follows.

$$
\begin{array}{ll}
& C.(I \wedge Q) \\
\Leftarrow & \{\text{logic, } wp \text{ rule for sequential composition: } C \triangleq C_1 \, ; C_2\} \\
& I \wedge C_1.(C_2.(I \wedge Q)) \\
\equiv & \{wp \text{ rule for assignment}\} \\
& I \wedge C_1.(I[step := 0] \wedge Q[step := 0]) \\
\equiv & \{Q \text{ does not mention } step,\ I[step := 0] \equiv \mathsf{true}\} \\
& I \wedge C_1.Q \\
\equiv & \{wp \text{ rules for guarded action, assignment}\} \\
& I \wedge (step = N \Rightarrow Q[Y := temp + X \times h[0]]) \\
\equiv & \{\text{notation: } I_0^{N-1} \triangleq (step = 1 \Rightarrow temp = 0) \\
& \qquad \wedge \displaystyle\bigwedge_{p=2}^{N-1} (step = p \Rightarrow temp = \sum_{k=1}^{p-1} h[k] \times Z[k-1])\} \\
& I_0^{N-1} \wedge \ (step = N \Rightarrow temp = \displaystyle\sum_{k=1}^{N-1} h[k] \times Z[k-1]) \\
& \qquad \wedge (step = N \Rightarrow Q[Y := \displaystyle\sum_{k=1}^{N-1} h[k] \times Z[k-1] + X \times h[0]]) \\
\Leftarrow & \{\text{replacement of } temp\text{, logic}\} \\
& I_0^{N-1} \wedge \ (step = N \Rightarrow temp = \displaystyle\sum_{k=1}^{N-1} h[k] \times Z[k-1]) \\
& \wedge (step = N \Rightarrow Q[Y := \displaystyle\sum_{k=1}^{N-1} h[k] \times Z[k-1] + X \times h[0]]) \\
\Leftarrow & \{\text{logic}\} \\
& I_0^{N-1} \wedge \ (step = N \Rightarrow temp = \displaystyle\sum_{k=1}^{N-1} h[k] \times Z[k-1]) \\
& \wedge Q[Y := \displaystyle\sum_{k=1}^{N-1} h[k] \times Z[k-1] + X \times h[0]]
\end{array}
$$

$$\equiv \{\text{definitions of } I,\ I_0^{N-1}\}$$

$$I \ \wedge\ Q[Y := X \times h[0] + \sum_{k=1}^{N-1} h[k] \times Z[k-1]]$$

$$\equiv \{wp \text{ rule for assignment}\}$$

$$I \wedge A.Q$$

3. Auxiliary action. For the auxiliary actions A_1, A_2, we have that $wA_1, wA_2 \in \{step, temp\}$, therefore they behave like skip with respect to the global variables. Hence, skip $\sqsubseteq_I A_1 \wedge$ skip $\sqsubseteq_I A_2$.

4. Continuation condition:

$$I \wedge gA \Rightarrow gC \vee gA_1 \vee gA_2$$

$$\equiv \{gA_1 \vee gA_2 \vee gC \equiv \mathsf{true}\}$$

$$I \wedge gA \Rightarrow \mathsf{true}$$

$$\equiv \{\text{logic}\}$$

$$\mathsf{true}$$

5. Internal convergence. It is easy to observe that A_1 terminates after one execution as it disables itself, while A_2 disables itself after $N-1$ executions.

From the above we have that, in *isolation*, the system $\mathcal{F}_S$ is a refinement of $\mathcal{F}$, under the invariant I: $\mathcal{F} \sqsubseteq_I \mathcal{F}_S$. ■

In addition, even if not necessary in this context, yet needed when dealing with the same refinement in a synchronized environment, we also show that I is a proper invariant. For this, we only check what happens when the global action (C) is executed.

$$I[wC := wC', v := z]$$

$$\equiv \{\text{computation}\}$$

$$I[step := 0, Y := temp + X \times h[0], v := z]$$

$$\equiv \mathsf{true}$$

$$\equiv I[step := 0, Y := temp + X \times h[0], v := z']$$

$$\equiv I[wC := wC', v := z']$$

Hence, I is a proper invariant of the system $\mathcal{F}_S$.

A-5 Computation of weakest precondition $\mathcal{T}(i).I_t$ (chapter 6)

We give the detailed computation of the weakest precondition for a preemptible real-time task to establish the timeliness condition. The task model is as follows:

$$
\begin{array}{lll}
\mathcal{T}(i) & \triangleq & \quad state[i].now = sl \wedge c_a[i].now = P[i] + ofs[i].now \\
& & \quad \rightarrow c_a[i] :- (\lambda t \cdot t - now) \, ; state[i] :- (\lambda t \cdot wt) \, ; UT \\
& & [\!] \; state[i].now = wt \\
& & \quad \rightarrow c_e[i] :- (\lambda t \cdot t - now) \, ; state[i] :- (\lambda t \cdot ex) \, ; UT \\
& & [\!] \; state[i].now = ex \wedge c_e[i].now = E[i] \\
& & \quad \rightarrow c_e[i] :- (\lambda t \cdot 0) \, ; c_p[i] :- (\lambda t \cdot 0); \\
& & \qquad [\, ofs[i] :- x' \mid \forall t \geq now \cdot x'.t \in \mathsf{Real}_+ \,]; \\
& & \qquad state[i] :- (\lambda t \cdot sl) \, ; UT \\
& & [\!] \; state[i].now = ex \wedge c_e[i].now < E[i] \\
& & \quad \rightarrow c_e[i] :- (\lambda t \cdot c_e[i].now); \\
& & \qquad c_p[i] :- (\lambda t \cdot c_p[i].now + t - now); \\
& & \qquad state[i] :- (\lambda t \cdot pt) \, ; UT \\
& & [\!] \; state[i].now = pt \\
& & \quad \rightarrow c_e[i] :- (\lambda t \cdot c_e[i].now + t - now); \\
& & \qquad c_p[i] :- (\lambda t \cdot c_p[i].now) \, ; state[i] :- (\lambda t \cdot ex) \, ; UT
\end{array}
$$

The timeliness predicate is as follows

$$
\begin{array}{lll}
I_t & \equiv & \forall i \cdot \forall t \in [start, now) \cdot \\
& & \quad (state[i].start = ex \Rightarrow \\
& & \qquad (state[i].t = ex \wedge c_a[i].t - (c_e[i].t + c_p[i].t) \leq D[i] - R[i] \ \wedge \\
& & \qquad\quad c_a[i].t = c_a[i].start + t - start)) \\
& & \wedge \, (state[i].start = ex \wedge c_e[i].start = 0 \Rightarrow c_p[i].t = 0)
\end{array}
$$

We denote:

$$
\begin{array}{lll}
I_t^j & \equiv & \forall t \in [start, now) \cdot \\
& & \quad (state[j].start = ex \Rightarrow \\
& & \qquad (state[j].t = ex \wedge c_a[j].t - (c_e[j].t + c_p[j].t) \leq D[j] - R[j] \ \wedge \\
& & \qquad\quad c_a[j].t = c_a[j].start + t - start)) \\
& & \wedge \, (state[j].start = ex \wedge c_e[j].start = 0 \Rightarrow c_p[j].t = 0)
\end{array}
$$

The actual wp computation is

$$\mathcal{T}(i).I_t$$

$\equiv$ { substitute $\mathcal{T}(i), I_t,$ rule (2.10),
actions $sl \rightarrow wt, ex \rightarrow sl, ex \rightarrow pt$ satisfy the invariant, trivially }

$$
\begin{array}{l}
(\forall j \neq i \cdot \mathcal{T}(i).I_t^j) \wedge \\
(state[i].now = wt \rightarrow c_e[i] :- (\lambda t \cdot t - now)\ ; \\
\qquad\qquad\qquad state[i] :- (\lambda t \cdot ex)\ ; \\
\qquad\qquad\qquad start := now\ ; now := min\{t' \geq now \mid gg.t'\}).I_t \\
\wedge\ (state[i].now = pt \rightarrow c_e[i] :- (\lambda t \cdot c_e[i].now + t - now)\ ; \\
\qquad\qquad\qquad c_p :- (\lambda t \cdot c_p[i].now)\ ; \\
\qquad\qquad\qquad state[i] :- (\lambda t \cdot ex)\ ; \\
\qquad\qquad\qquad start := now\ ; now := min\{t' \geq now \mid gg.t'\}).I_t
\end{array}
$$

$\equiv$ { rules (2.6), (2.4), (2.7) }

$$
\begin{array}{l}
(\forall j \neq i \cdot \mathcal{T}(i).I_t^j) \wedge \\
(state[i].now = wt \Rightarrow \\
\quad (\forall t \in [start, min\{t' \geq now \mid gg.t'\}) \cdot \\
\qquad (c_e[i] :- (\lambda t \cdot t - now)\ ; state[i] :- (\lambda t \cdot ex)\ ; start := now).I_t)) \\
\wedge\ (state[i].now = pt \Rightarrow \\
\quad (\forall t \in [start, now = min\{t' \geq now \mid gg.t'\}] \cdot \\
\qquad (c_e[i] :- (\lambda t \cdot c_e[i].now + t - now)\ ; c_p :- (\lambda t \cdot c_p[i].now)\ ; \\
\qquad state[i] :- (\lambda t \cdot ex)\ ; start := now).I_t))
\end{array}
$$

$\equiv$ { rules (2.7), (2.4), compute $gg.t' \equiv (c_e[i].t' \leq E[i])$ }

$$
\begin{array}{l}
(\forall j \neq i \cdot \mathcal{T}(i).I_t^j) \wedge \\
(state[i].now = wt \Rightarrow \\
\quad (\forall\ now' \cdot now' = min\{t' \geq now \mid c_e[i].t' \leq E[i]\} \Rightarrow \\
\qquad (\forall t \in [start, now') \cdot (c_e[i] :- (\lambda t \cdot t - now)\ ; \\
\qquad\qquad\qquad state[i] :- (\lambda t \cdot ex)).I_t[start := now]))) \\
\wedge\ (state[i].now = pt \Rightarrow \\
\quad (\forall\ now' \cdot now' = min\{t' \geq now \mid c_e[i].t' \leq E[i]\} \Rightarrow \\
\qquad (\forall t \in [start, now') \cdot (c_e[i] :- (\lambda t \cdot c_e[i].now + t - now)\ ; \\
\qquad\qquad\qquad c_p[i] :- (\lambda t \cdot c_p[i].now)\ ; state[i] :- (\lambda t \cdot ex)). \\
\qquad\qquad\qquad\qquad I_t[start := now])))
\end{array}
$$

$\equiv$ { substitute $start = now,$ rules (2.7), (2.4),
substitute $state[i].start = ex$ in I_t}

$$
\begin{array}{l}
(\forall j \neq i \cdot \mathcal{T}(i).I_t^j) \wedge \\
(state[i].now = wt \Rightarrow
\end{array}
$$

$$
\begin{array}{l}
(\forall\, now' \cdot now' = min\{t' \geq now \mid c_e[i].t' \leq E[i]\} \Rightarrow \\
\quad (\forall t \in [now, now') \cdot (c_e[i] :- (\lambda t \cdot t - now)). \\
\quad\quad (c_a[i].t - (c_e[i].t + c_p[i].t) \leq D[i] - R[i] \\
\quad\quad\quad\quad \wedge\, c_a[i].t = c_a[i].now + t - now \\
\quad \wedge\, c_e[i].now = 0 \Rightarrow c_p[i].t = 0))) \\
\wedge\, (state[i].now = pt \Rightarrow \\
\quad (\forall\, now' \cdot now' = min\{t' \geq now \mid c_e[i].t' \leq E[i]\} \Rightarrow \\
\quad\quad (\forall t \in [now, now') \cdot (c_e[i] :- (\lambda t \cdot c_e[i].now + t - now)\,; \\
\quad\quad\quad\quad c_p[i] :- (\lambda t \cdot c_p[i].now)). \\
\quad\quad\quad (c_a[i].t - (c_e[i].t + c_p[i].t) \leq D[i] - R[i] \\
\quad\quad\quad\quad \wedge\, c_a[i].t = c_a[i].now + t - now \\
\quad\quad \wedge\, c_e[i].now = 0 \Rightarrow c_p[i].t = 0)))
\end{array}
$$

$\equiv$ { successive application of rules (2.7), (2.4), substitute $c_p[i].t, c_e[i].t$}

$$
\begin{array}{l}
(\forall j \neq i \cdot \mathcal{T}(i).I_t^j) \wedge \\
(state[i].now = wt \Rightarrow \\
\quad (\forall now' \cdot now' = min\{t' \geq now \mid c_e[i].t' \leq E[i]\} \Rightarrow \\
\quad\quad (\forall t \in [now, now') \cdot c_a[i].t - (t - now) \leq D[i] - R[i] \;\wedge \\
\quad\quad\quad c_a[i].t = c_a[i].now + t - now))) \\
\wedge\; (state[i].now = pt \Rightarrow \\
\quad (\forall now' \cdot now' = min\{t' \geq now \mid c_e[i].t' \leq E[i]\} \Rightarrow \\
\quad\quad (\forall t \in [now, now') \cdot c_a[i].t = c_a[i].now + t - now \;\wedge \\
c_a[i].t - (c_e[i].now + (t - now) + c_p[i].now) \leq D[i] - R[i])))
\end{array}
$$

A-6 Proof of $\overline{\mathcal{RTS}}'^s \sqsubseteq \overline{\mathcal{RTS}}^m$ (chapter 6)

Given the predicate

$$
\begin{array}{lll}
I_1 & \equiv & \forall i \cdot \forall now \cdot \\
& & (ok[i].now \wedge state[i].now = wt \Rightarrow \\
& & \quad (pr[i].now = Max(q.now) \wedge c_a[i].now \leq D[i] - R[i] \wedge \\
& & \quad (\forall j \neq i \cdot (state[j].now = sl \wedge c_a[j].now < P[j]) \vee \\
& & \quad\quad state[j].now = wt \vee state[j].now = pt)))
\end{array}
$$

$$
\begin{array}{ll}
\wedge & (ok[i].now \wedge state[i].now = pt \Rightarrow \\
& \quad (pr[i].now = Max(q.now) \wedge \\
& \quad\quad c_a[i].now - (c_e[i].now + c_p[i].now) \leq D[i] - R[i] \wedge \\
& \quad (\forall j \neq i \cdot (state[j].now = sl \wedge c_a[j].now < P[j]) \vee \\
& \quad\quad state[j].now = wt \vee state[j].now = pt))) \\
\wedge & (\neg ok[i].now \wedge state[i].now = ex \Rightarrow \\
& \quad (pr[i].now \neq Max(q.now) \wedge c_e[i].now < E[i]))
\end{array}
$$

and $k \in [1..n]$, we have to prove $\overline{\mathcal{RTS}}'^{s} \sqsubseteq_{I_1} \overline{\mathcal{RTS}}^{m}$.

Since trace refinement conditions (6.10), (6.14) and (6.15) are immediate (the auxiliary variable ok is a local variable, and the auxiliary actions are self-disabling, thus they terminate), we concentrate on proving the remaining three.

• I_1 is preserved by the actions of the concrete system $\overline{\mathcal{RTS}}^{m}$ (requirement (6.11)).

We sketch the proof for one of the actions only, namely for

$$
\begin{array}{lll}
A^i_{21m} & = & \neg ok[i].now \wedge pr[i].now = Max(q.now) \wedge \\
& & (\forall j \neq i \cdot (state[j].now = sl \wedge c_a[j].now < P[j]) \vee \\
& & \quad state[j].now = wt \vee state[j].now = pt) \wedge \\
& & state[i].now = wt \wedge c_a[i].now \leq D[i] - R[i] \\
& & \rightarrow ok[i] :- (\lambda t \cdot \mathsf{true}) \,;\, UT
\end{array}
$$

We have that $I_1 = I_1[1] \wedge \ldots \wedge I_1[n]$. Below, we consider only $I_1[i]$.

$$
\begin{array}{ll}
& I_1[i] \\
\Rightarrow & A^i_{21m}.I_1[i] \\
\equiv & \{\text{substitute } I_1\text{, rules (2.7), (2.4), assume } now = min\{\ldots\}\} \\
& (\neg ok[i].now \wedge pr[i].now = Max(q.now) \wedge \\
& (\forall j \neq i \cdot (state[j].now = sl \wedge c_a[j].now < P[j]) \vee \\
& \quad\quad state[j].now = wt \vee state[j].now = pt) \wedge \\
& state[i].now = wt \wedge c_a[i].now \leq D[i] - R[i]) \\
\Rightarrow & ((state[i].now = wt \Rightarrow \\
& \quad (pr[i].now = Max(q.now) \wedge c_a[i].now \leq D[i] - R[i] \wedge \\
& \quad\quad (\forall j \neq i \cdot (state[j].now = sl \wedge c_a[j].now < P[j]) \vee \\
& \quad\quad state[j].now = wt \vee state[j].now = pt))) \\
\wedge & (state[i].now = pt \Rightarrow \\
& \quad (pr[i].now = Max(q.now) \wedge \\
& \quad\quad (\forall j \neq i \cdot (state[j].now = sl \wedge c_a[j].now < P[j]) \vee \\
& \quad\quad state[j].now = wt \vee state[j].now = pt) \wedge \\
& \quad c_a[i].now - (c_e[i].now + c_p[i].now) \leq D[i] - R[i]))))
\end{array}
$$

$$
\begin{aligned}
\equiv\quad & \{\text{logic}\}\\
& ok[i].now \lor pr[i].now \neq Max(q.now) \lor state[i].now \neq wt\\
& \lor (\exists j \neq i \cdot \neg(state[j].now = sl \land c_a[j].now < P[j]) \land\\
& \qquad state[j].now \neq wt \land state[j].now \neq pt))\\
& \lor c_a[i].now > D[i] - R[i] \lor state[i].now \neq pt\\
& \lor (pr[i].now = Max(q.now) \land\\
& \qquad (\forall j \neq i \cdot (state[j].now = sl \land c_a[j].now < P[j]) \lor\\
& \qquad state[j].now = wt \lor state[j].now = pt) \land\\
& \qquad c_a[i].now - (c_e[i].now + c_p[i]) \leq D[i] - R[i])\\
\equiv\quad & \{state[i].now \neq wt \equiv\\
& \quad (state[i].now = sl \lor state[i].now = pt \lor state[i].now = ex),\\
& \quad \text{logic}\}\\
& \mathsf{true}
\end{aligned}
$$

• We prove just one condition of the type (6.12), that is

$$A_2'^i \sqsubseteq_{I_1} A_{22m}^i$$

The other two refinements are similar.

The bodies of the above actions are identical. Therefore, we are left with proving refinement of guards:

$$
\begin{aligned}
& I_1[i] \land ok[i].now \land state[i].now = wt\\
& \Rightarrow (pr[i].now = Max(q.now) \land state[i].now = wt \land\\
& \qquad (\forall j \neq i \cdot (state[j].now = sl \land c_a[j].now < P[j]) \lor\\
& \qquad\quad state[j].now = wt \lor state[j].now = pt) \land\\
& \qquad c_a[i].now \leq D[i] - R[i])\\
\equiv\quad & \{\text{logic}\}\\
& \quad pr[i].now \neq Max\{q[k].now\} \lor c_a[i].now > D[i] - R[i]\\
& \lor \neg ok[i].now \lor state[i].now \neq wt\\
& \lor (\exists j \neq i \cdot \neg(state[j].now = sl \land c_a[j].now < P[j]) \land\\
& \qquad state[j].now \neq wt \land state[j].now \neq pt)\\
& \lor (pr[i].now = Max\{q[k].now\} \land c_a[i].now \leq D[i] - R[i] \land\\
& \qquad (\forall j \neq i \cdot (state[j].now = sl \land c_a[j].now < P[j]) \lor\\
& \qquad state[j].now = wt \lor state[j].now = pt))\\
\equiv\quad & \{\text{logic}\}\\
& \mathsf{true}
\end{aligned}
$$

• In order to fulfill requirement (6.13) of trace refinement of CAS, we have to prove that

$$I_1 \wedge gg_{\overline{\mathcal{RTS}}'^s} \Rightarrow gg_{\overline{\mathcal{RTS}}^m} \vee gg_X$$

We identify

$$gg_{\overline{\mathcal{RTS}}'^s} \equiv \bigvee_{i=1}^{n} \left(\begin{array}{l} \quad (state[i].now = sl \ \wedge \ c_a[i].now = P[i] + ofs[i].now) \\ \vee \ (state[i].now = ex \ \wedge \ c_e[i].now = E[i]) \\ \vee \ (state[i].now = ex \wedge c_e[i].now < E[i] \\ \qquad \wedge \ pr[i].now \neq Max(q.now)) \\ \vee \quad (state[i].now = wt \ \wedge \ c_a[i].now \geq D[i] - R[i] \\ \qquad \wedge \ pr[i].now = Max(q.now) \\ \qquad \wedge \ (\forall j \neq i \cdot (state[j].now = sl \wedge c_a[j].now < P[j]) \\ \qquad\quad \vee state[j].now = wt \vee state[j].now = pt)) \\ \vee \quad (state[i].now = pt \\ \qquad \wedge \ c_a[i].now - (c_e[i].now + c_p[i].now) \leq D[i] - R[i] \\ \qquad \wedge \ pr[i].now = Max(q.now) \\ \qquad \wedge \ (\forall j \neq i \cdot (state[j].now = sl \wedge c_a[j].now < P[j]) \\ \qquad\quad \vee state[j].now = wt \vee state[j].now = pt)) \end{array} \right)$$

$$gg_{\overline{\mathcal{RTS}}^m} \equiv \bigvee_{i=1}^{n} \left(\begin{array}{l} \quad (\neg ok[i].now \ \wedge \ state[i].now = sl \\ \qquad \wedge \ c_a[i].now = P[i] + ofs[i].now) \\ \vee \ (\neg ok[i].now \ \wedge \ state[i].now = ex) \\ \vee \ (ok[i].now \ \wedge \ state[i].now = ex \\ \qquad \wedge \ c_e[i].now = E[i]) \\ \vee \ (ok[i].now \ \wedge \ state[i].now = wt) \\ \vee \ (ok[i].now \ \wedge \ state[i].now = pt) \end{array} \right)$$

$$gg_X \equiv \bigvee_{i=1}^{n} \left(\begin{array}{l} \quad (\neg ok[i].now \ \wedge \ pr[i].now = Max(q.now) \\ \qquad \wedge \ state[i].now = wt \wedge c_a[i].now \leq D[i] - R[i] \\ \qquad \wedge \ (\forall j \neq i \cdot (state[j].now = sl \wedge c_a[j].now < P[j]) \\ \qquad\quad \vee \ state[j].now = wt \vee state[j].now = pt)) \\ \vee (ok[i].now \ \wedge \ state[i].now = sl) \\ \vee (ok[i].now \ \wedge \ pr[i].now \neq Max(q.now) \\ \qquad \wedge \ state[i].now = ex \ \wedge \ c_e[i].now < E[i]) \\ \vee (\neg ok[i].now \ \wedge \ pr[i].now = Max(q.now) \\ \qquad \wedge \ state[i].now = pt \\ \qquad \wedge \ c_a[i].now - (c_e[i].now + c_p[i].now) \leq D[i] - R[i] \\ \qquad \wedge \ (\forall j \neq i \cdot (state[j].now = sl \wedge c_a[j].now < P[j]) \\ \qquad\quad \vee \ state[j].now = wt \vee state[j].now = pt)) \end{array} \right)$$

By inspecting the above conditions one can notice that the continuation condition holds. ■

Bibliography

[1] M. Abadi and L. Lamport. An old-fashioned recipe for real-time. *ACM Transactions on Programming Languages and Systems*, 16(5), pp. 1543-1571, 1994. Also appeared in J. W. de Bakker, C. Huizing, W.-P. de Roever, and G. Rozenberg, editors, Lecture Notes in Computer Science, vol. 600, Springer-Verlag, 1992.

[2] Y. Abdeddaïm and O. Maler. Job-shop Scheduling Using Timed Automata. In *Proceedings of CAV'01*, Lecture Notes in Computer Science, vol. 2102, pp. 478-492, Springer-Verlag, 2001.

[3] J. R. A. Abrial. *The B-book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[4] K. Altisen, G. Gößler, A. Pnueli, J. Sifakis, S. Tripakis, and S. Yovine. A Framework for Scheduler Synthesis. In *Proceedings of RTSS'99*, pp. 154-163. IEEE Computer Society Press, 1999.

[5] K. Altisen, G. Gößler, and J. Sifakis. Scheduler Modeling based on the Controller Synthesis Paradigm. *Journal of RTS*, nr. 23, pp. 55-84, 2002.

[6] R. Alur, C. Courcoubetis, T.A. Henzinger, and P.-H. Ho. Hybrid Automata: An Algorithmic Approach to the Specification and Verification of Hybrid Systems. *Hybrid Systems, R. Grossman, A. Nerode, A. P. Ravn, and H. Rischel (eds.)* , Lecture Notes in Computer Science, vol. 736, pp. 209-229, Springer-Verlag, 1993.

[7] R. Alur, C. Courcoubetis, N. Halbwachs, T.A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The Algorithmic Analysis of Hybrid Systems. *Theoretical Computer Science*, 138, pp. 3-34, 1995.

[8] R. Alur, T. Dang, and F. Ivancic. Reachability Analysis of Hybrid Systems via Predicate Abstraction. *ACM Transactions on Embedded Computing Systems (TECS)*, 2004.

[9] R. Alur and D.L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, vol. 126(2), pp. 183-235, 1994.

[10] R. Alur and R. Grosu. Modular Refinement of Hierarchic Reactive Machines. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 26, issue 2, pp. 339 - 369, 2004.

[11] R. Alur, R. Grosu, Y. Hur, V. Kumar, and I. Lee. Modular Specification of Hybrid Systems in CHARON. In *Proceedings of the 3rd International Workshop on Hybrid Systems: Computation and Control (HSCC 2000)*, Lecture Notes in Computer Science, vol. 1790, pp. 6 - 19, Springer-Verlag, 2000.

[12] R. Alur and T. A. Henzinger. Modularity for Timed and Hybrid Systems. In *Proceedings of the 8th International Conference on Concurrency Theory (CONCUR 97)*, Lecture Notes in Computer Science, vol.1243, pp. 74-88, Springer-Verlag, 1997.

[13] R. Alur and T. A. Henzinger. Reactive modules. *Formal Methods in System Design*, 15, 1, pp. 7-48, 1999.

[14] R. Alur, T. A. Henzinger, and P.-H. Ho. Automatic Symbolic Verification of Embedded Systems. *IEEE Transactions on Software Engineering*, 22(3), pp. 181-201, 1996.

[15] R. Alur, T. A. Henzinger, and O. Kupferman. Alternating-time Temporal Logic. In *Journal of the ACM*, 49: 672-713, 2002. Preliminary versions appeared in *Proceedings of the 38th Annual Symposium on Foundations of Computer Science (FOCS)*, IEEE Computer Society Press, pp. 100-109, 1997, and in *Compositionality - The Significant Difference*, Lecture Notes in Computer Science, vol. 1536, pp. 23-60, Springer–Verlag, 1999.

[16] R. Alur, T. A. Henzinger, F.Y.C. Mang, S. Qadeer, S. K. Rajamani, and S. Tasiran. MOCHA: modularity in model checking. In *Proceedings of the Computer-aided Verification (CAV'98)*, Lecture Notes in Computer Science, vol. 1427, pp. 521-525, Springer-Verlag, 1998.

[17] T. Amnell, E. Fersman, L. Mokrushin, P. Petterson, and W. Yi. Times - a tool for modelling and implementation of embedded systems. In *Proceedings of ETAPS 2002, Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2002)*, Lecture Notes in Computer Science, vol. 2280, pp. 460-464, Springer-Verlag, 2002.

[18] E. Asarin, O. Maler, and A. Pnueli. Symbolic controller synthesis for discrete and timed systems. In *Hybrid Systems II*, Lecture Notes in Computer Science, vol. 999, Springer-Verlag, 1995.

[19] P. Ashenden. The Designer's Guide to VHDL - Second Edition. *Morgan Kaufmann Publishers*, 2002.

[20] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. Hard Real-Time Scheduling: The Deadline-Monotonic Approach. In *Real-Time Programming*, pp. 127-132, Pergamon Press, 1992.

[21] R. J. Back. *On the correctness of refinement in program development*. Ph.D. thesis, Department of Computer Science, University of Helsinki, Finland, 1978.

[22] R. J. Back. Refinement calculus, part II: Parallel and Reactive Programs. *Stepwise Refinement of Distributed Systems*, Lecture Notes in Computer Science, nr. 430, pp. 67-93, Springer, 1989.

[23] R. J. Back. Invariant Based Programming Revisited. *Technical Report* nr. 661, Turku Centre for Computer Science, 2005.

[24] R. J. Back and C. Cerschi. Modeling and Verifying a Temperature Control System using Continuous Action Systems. In *Proceedings of the 5th International ERCIM Workshop on Formal Methods for Industrial Critical Systems (FMICS'2000)*, GMD Report 91, pp. 265-286, *ERCIM and GMD*, 2000.

[25] R. J. Back, C. Cerschi Seceleanu, and J. Westerholm. Symbolic Simulation of Hybrid Systems. In *Proceedings of the 9th Asia-Pacific Software Engineering Conference (APSEC 2002)*, pp. 147 - 158, IEEE Computer Society Press, 2002.

[26] R. J. Back and C. Cerschi Seceleanu. Contracts and Games in Controller Synthesis for Discrete Systems. In *Proceedings of the 11th IEEE International Conference on the Engineering of Computer-Based Systems (ECBS 2004)*, pp. 307 - 315, IEEE Computer Society Press, 2004.

[27] R. J. Back, L. Petre, and I. Porres-Paltor. Continuous Action Systems as a Model for Hybrid Systems. *Nordic Journal of Computing*, vol. 8, pp. 2-21, 2001.

[28] R.-J. R. Back and K. Sere. From Action Systems to Modular Systems. In *Proceedings of Formal Methods Europe'94*, Lecture Notes in Computer Science, Springer-Verlag, 1994.

[29] R. J. R. Back and K. Sere. Action Systems with Synchronous Communication. *Programming Concepts, Methods and Calculi*. In *E.-R. Olderog, IFIP Transactions* A-56, pp. 107-126, 1994.

[30] R. J. R. Back and K. Sere. Superposition Refinement of Reactive Systems. *Formal Aspects of Computing*, vol. 8, nr. 3, pp. 324-346, Springer–Verlag, 1996.

[31] R.J.R. Back and R. Kurki-Suonio. Decentralization of Process Nets with Centralized Control. In *Proceedings of the 2nd Annual Symposium on Principles of Distributed Computing*, Lecture Notes in Computer Science 873, ACM SIGACT-SIGOPS, pp. 131-142, 1983.

[32] R.J.R. Back and R. Kurki-Suonio. Distributed Cooperation with Action Systems. *ACM Transactions on Programming Languages and Systems*, 10(4):513-554, 1988.

[33] R. J. R. Back and J. von Wright. Trace Refinement of Action Systems. In *Proceedings of CONCUR-94*, Lecture Notes in Computer Science, nr. 836, Springer–Verlag, 1994.

[34] R. J. R. Back and J. von Wright. Games and winning strategies. *Information Processing Letters*, 53(3):165-172, 1995.

[35] R. J. R. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Springer–Verlag, 1998.

[36] R. J. R. Back, J. von Wright. Reasoning algebraically about loops. *Acta Informatica* 36, pp. 295-334, Springer–Verlag, 1999.

[37] R. J. R. Back and J. von Wright. Verification and Refinement of Action Contracts. *Technical Report* nr. 374, Turku Centre for Computer Science, 2000.

[38] R. J. R. Back and J. von Wright. Enforcing Behavior with Contracts. *Programming Methodology, A. McIver, C. Morgan (Eds.)*, pp. 17-52, Springer–Verlag, 2003.

[39] R. J. R. Back and J. von Wright. Compositional Action System Refinement. *Formal Aspects of Computing* (2-3): 103-117, 2003.

[40] D.A. van Beek and J.E. Rooda. Languages and Applications in Hybrid Modelling and Simulation: Positioning of Chi. *Control Engineering Practice*, vol. 8, nr. 1, pp. 81-91, 2000.

[41] R. Behrends and R. Stirewalt. The Universe Model: An Approach for Improving the Modularity and Reliability of Concurrent Programs. In *Proceedings of ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE'2000)*, 2000.

[42] F. Bellegarde, J. Julliand, and O. Kouchnarenko. Synchronized Parallel Composition of Event Systems in B. In *Proceedings of the ZB 2002*, Lecture Notes in Computer Science, vol. 2272, pp. 436-457, Springer-Verlag, 2002.

[43] S. Bensalem, V. Ganesh, Y. Lakhnech, C. Muñoz, S. Owre, H. Rueß, J. Rushby, V. Rusu, H. Saïdi, N. Shankar, E. Singerman, and A. Tiwari. An

Overview of SAL. In *Proceedings of the 5th NASA Langley Formal Methods Workshop (LFM 2000)*, pp. 187-196, NASA Langley Research Centre, 2000.

[44] S. Bensalem and Y. Lakhnech. Automatic Generation of Invariants. *Formal Methods in System Design*, 15:75-92, 1999.

[45] G. Berry. The Foundations of Esterel. *Proof, Language and Interaction: Essays in Honour of Robin Milner*, G. Plotkin, C. Stirling and M. Tofte (eds.), MIT Press, 1998.

[46] N. S. Bjorner, Z. Manna, H. B. Sipma, and T. E. Uribe. Deductive verification of real-time systems using STeP. *Theoretical Computer Science*, 253:27-60, 2001.

[47] S. Bornot and J. Sifakis. On the composition of hybrid systems. In *First International Workshop Hybrid Systems : Computation and Control (HSCC'98)*, Lecture Notes in Computer Science, vol. 1386, pp. 49-63, Springer–Verlag, 1998.

[48] A. Bouajjani, A. Collomb-Annichini, and M. Sighireanu. TREX: A tool for reachability analysis of complex systems. In *Proceedings of CAV 2001*, Lecture Notes in Computer Science, vol. 2102, pp. 368-372, Springer-Verlag, 2001.

[49] H. Bowman. Modelling Timeout without Timelocks. In J.-P. Katoen, editor, *Formal Methods for Real-Time and Probabilistic Systems*, Lecture Notes in Computer Science 1601, Springer-Verlag, 1999.

[50] V. Braberman and M. Felder. Verification of Real-Time Designs. *LNCS 1687*, pp. 494-510, Springer–Verlag, 1999.

[51] A. Burns. Scheduling Hard Real-Time Systems: A Review. *Software Engineering Journal*, 6, pp. 116-128, 1991.

[52] A. Burns and A. Wellings. *Real-Time Systems and Programming Languages (Third Edition)*. Addison-Wesley, 2001.

[53] M. Butler. Stepwise Refinement of Communicating Systems. *Science of Computer Programming*, 27(2):139-173, 1996.

[54] M. J. Butler, J. Grundy, T. Långbacka, R. Rukšėnas, and J. von Wright. The Refinement Calculator: Proof support for program refinement. In *Proceedings of Formal Methods Pacific '97*, Springer Series in Discrete Mathematics and Theoretical Computer Science, pp. 40 - 61, Springer–Verlag, 1997.

[55] A. Cavalcanti and J. Woodcock. A Predicate Transformer Semantics for a Concurrent Language of Refinement. *Communicating Process Architectures*

- 2002, J.Pascoe, P.Welch, R.Loader and V.Sunderam (Eds.), IOS Press, pp. 147-165, 2002.

[56] O. Celiku and J. von Wright. Theorem prover support for precondition and correctness calculation. In *Proceedings of the 4th International Conference on Formal Engineering Metods (ICFEM 2002)*, Lecture Notes in Computer Science, vol. 2495, pp. 299-310, Springer–Verlag, 2002.

[57] O. Celiku and J. von Wright. Implementing Angelic Nondeterminism. In *Proceedings of the 10th Asia-Pacific Software Engineering Conference (APSEC'03)*, IEEE Computer Society Press, pp. 176 - 185, 2003.

[58] C. Cerschi Seceleanu. Formal Development of Real-Time Priority-Based Schedulers. In *Proceedings of the 12th IEEE International Conference on the Engineering of Computer-Based Systems (ECBS 2005)*, pp. 263 - 270, IEEE Computer Society Press, 2005.

[59] C. Cerschi Seceleanu. Designing Controllers for Reachability. In *Proceedings of the 29th Annual International Computer Software and Applications Conference (COMPSAC 2005)*, IEEE Computer Society Press, 2005.

[60] C. Cerschi Seceleanu and T. Seceleanu. Synchronization Can Improve Reactive Systems Control and Modularity. *Journal of Universal Computer Science (JUCS)*, 10(10): 1429 - 1468, Springer, 2004.

[61] C. Cerschi Seceleanu and T. Seceleanu. Modular Design of Reactive Systems. In *Proceedings of the 28th Annual International Computer Software and Applications Conference (COMPSAC 2004)*, IEEE Computer Society Press, 2004.

[62] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, Reading, MA, University of Texas-Austin, 1988.

[63] M. Charpentier. An Approach to Composition Motivated by wp. In *Proceedings of the 5th International Conference on Fundamental Approaches to Software Engineering*, Lecture Notes in Compter Science, vol. 2306, pp. 1-14, Springer–Verlag, 2002.

[64] E. Closse, M. Poize, J. Pulou, J. Sifakis, P. Venier, D. Weil, and S. Yovine. Taxys: a tool for the development and verification of real-time embedded systems. In *Proceedings of CAV'01*, Lecture Notes in Computer Science, vol. 2102, Springer-Verlag, 2001.

[65] J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivas. A tutorial introduction to PVS. In *WIFT'95 Workshop on Industrial-Strength Formal Specification Techniques*, 1995.

[66] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis. In *Proceedings of the* 4th *ACM Symposium on Principles of Programming Languages*, Association for Computing Machinery, 1977.

[67] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. In *Hybrid Systems III*, Lecture Notes in Computer Science, vol. 1066, pp. 208 - 219, Springer–Verlag, 1996.

[68] DES group University of Michigan. UMDES software library. *http://www.eecs.umich.edu/undes/projects.html*.

[69] E.W. Dijkstra. Notes on Structured Programming. In O.-J. Dahl, E.W. Dijkstra, and C.A.R. Hoare, editors, *Structured Programming*, Academic Press, 1972.

[70] E.W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453-457, 1975.

[71] E.W. Dijkstra. *A Discipline of Programming*. Prentice Hall Series in Automatic Computation, Prentice Hall, 1976.

[72] E.W. Dijkstra. *Predicate Calculus and Program Semantics*. Texts and Monographs in Computer Science, Springer-Verlag, 1990.

[73] B. Dutertre. Formal Analysis of the Priority Ceiling Protocol. In *Proceedings of the 21st IEEE Real-Time Systems Symposium (RTSS'00)*, pp. 151-160, IEEE Computer Society Press, 2000.

[74] H. Elmqvist. Object-Oriented Modeling and Automatic Formula Manipulation in Dymola. In *SIMS'93*, Scandinavian Simulation Society, 1993.

[75] C. Fidge, I. Hayes, and G. Watson. The Deadline Command. *IEEE Proceedings - Software*, 146(2), pp. 104-111, Special Issue on Real-Time Systems, 1999.

[76] C. Fidge, M. Utting, P. Kearney, and I. Hayes. Integrating Real-Time Scheduling Theory and Program Refinement. In *Proceedings of Formal Methods Europe 96 (FME'96)*, Lecture Notes in Computer Science, vol. 1051, pp. 327-346, Springer–Verlag, 1996.

[77] C. J. Fidge and A. J. Wellings. An Action-based Formal Model for Concurrent Real-time Systems. *Formal Aspects of Computing*, 9(2):175-207, 1997.

[78] C. Flanagan and M. Abadi. Types for Safe Locking. In *Proceedings of the 2nd European Joint Conference on Theory and Practice of Software, (ESOP)*, Lecture Notes in Computer Science, vol. 1576, pp. 91-108, Springer-Verlag, 1999.

[79] A. Göllü, M. Kourjanski, and P. Varaiya. The SHIFT Simulation Framework: Language, Model and Implementation (Extended Abstract). In *Proceedings of the 5th International Hybrid Systems Workshop*, Notre Dame, Indiana, 1997.

[80] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. Lecture Notes in Computer Science, 1254:72 - 83, 1997.

[81] V. Gupta, R. Jagadeesan, and V.A. Saraswat. Truly concurrent constraint programming. *Theoretical Computer Science* 278, pp. 223-255, 2002.

[82] V. Gupta, R. Jagadeesan, V. Saraswat, and D. Bobrow. Programming in Hybrid Constraint Languages. In *Hybrid Systems II*, Lecture Notes in Computer Science, vol. 999, Springer–Verlag, 1995.

[83] D. Harel and A. Naamad. The STATEMATE Semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology*, vol. 5, nr. 4, pp. 29333, 1996.

[84] S. J. Hartley. *Concurrent Programming, the Java Pogramming Language*. Oxford University Press, 1998.

[85] I. Hayes. The Real-Time Refinement Calculus: A Foundation for Machine-Independent Real-Time Programming. In *Proceedings of the 23rd International Conference on Applications and Theory of Petri Nets (ICATPN 2002)*, Lecture Notes in Computer Science, vol. 2360, pp. 44-58, Springer–Verlag, 2002.

[86] T. A. Henzinger. The Theory of Hybrid Automata. In *Proceedings of the 11th Annual Symposium on Logic in Computer Science (LICS)*, IEEE Computer Society Press, pp. 278-292, 1996.

[87] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. A user guide to HYTECH. In *Proceedings of TACAS 95*, Lecture Notes in Computer Science, vol. 1019, pp. 41-71, Springer-Verlag, 1995.

[88] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. HYTECH: A model-checker for hybrid systems. *Software Tools for Technology Transfer*, 1(1): 110-122, 1997.

[89] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proceedings of Symposium on Principles of Programming Languages*, pp. 58-70, 2002.

[90] T. A. Henzinger, R. Majumdar, F. Mang, and J.-F. Raskin. Abstract interpretation of game properties. In *Proceedings of the 7th International Static Analysis Symposium (SAS)*, Lecture Notes in Computer Science, vol. 1824, pp. 220-239, Springer-Verlag, 2000.

[91] T. A. Henzinger, Z. Manna, and A. Pnueli. Towards Refining Temporal Specifications into Hybrid Systems. In *Hybrid Systems*, Lecture Notes in Computer Science, vol. 736, pp. 60-76, Springer-Verlag, 1993.

[92] T. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic Model-Checking for Real-Time Systems. *Information and Computation*, 111(2):193-244, 1994.

[93] T. A. Henzinger, J. Preußig, and H. Wong-Toi. Some Lessons from the HyTech Experience. In *Proceedings of the 40th Annual IEEE Conference on Decision and Control (CDC 2001)*, , 2001.

[94] T. Henzinger and V. Rusu. Reachability Verification for Hybrid Automata. In *Proceedings of the First International Workshop on Hybrid Systems: Computation and Control (HSCC 98)*, Lecture Notes in Computer Science, vol. 1386, pp. 190-204, Springer–Verlag, 1998.

[95] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576-580, 1969.

[96] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*,1(4):271-281, 1972.

[97] C. A. R. Hoare. *Communicating Sequential Processes.* International Series in Comnuter Science, Prentice Hall, 1985.

[98] C. A. R. Hoare and J. He. *Unifying Theories of Programming*. Prentice Hall, 1998.

[99] G. Hoffmann and H. Wong-Toi. Symbolic Synthesis of Supervisory Controllers. In *Proceedings of the American Control Conference*, Chicago, IL, pp. 2789–2793, 1992.

[100] T. Hune, K. G. Larsen, and P. Petterson. Guided Synthesis of Control Programs using UPPAAL. *Nordic Journal of Computing*, 8(1):43-64, 2001.

[101] E. C. Ifeachor and B. W. Jervis. Digital Signal Processing Practical Approach. *Addison Wesley Publishing Company*, 1997.

[102] C. B. Jones. *Systematic Software Development using VDM*. Prentice Hall Series in Comnuter Science, Prentice Hall, 2 edition, 1990.

[103] M. Joseph and P. Pandaya. Finding Response Times in a Real-Time System. *The Computer Journal*, vol. 29, nr. 5, pp. 390 - 395, 1986.

[104] S. Katz and Z. Manna. A Heuristic Approach to Program Verification. In *Proceedings of the 3rd IJCAI*, pp. 500-512, 1973.

[105] J. C. King. Symbolic Execution and Program Testing. In *Communications of the ACM*, vol. 19, no. 7, pp. 385-394, 1976.

[106] Y. Kesten and A. Pnueli. Verification by Finitary Abstraction. *Information and Computation*, 163:203–243, 2000.

[107] E. Kofman. Discrete Event Simulation of Hybrid Systems. In *SIAM Journal on Scientific Computing*, Volume 25, Number 5 pp. 1771-1797, 2004.

[108] P. Kopke, T. Henzinger, A. Puri, and P. Varaiya. Whats Decidable About Hybrid Automata? In *Proceedings of the 27th Annual ACM Symposium on Theory of Computing (STOC95)*, pp. 373-382, 1995.

[109] S. Kowalevski, O. Stursberg, M. Fritz, H. Graf, I. Hoffmann, J. Preuß ig, S. Simon, M. Remelhe, and H. Treseler. A Case Study in Tool-Aided Analysis of Discretely Controlled Continuous Systems: the Two Tanks Problem. In *Proceedings of the 15th International Workshop on Hybrid Systems (HS V)*, Lecture Notes in Computer Science, Springer–Verlag, 1997.

[110] R. Kurki-Suonio. Action systems in incremental and aspect-oriented modeling. *Distributed Computing*, 16: 201 - 217, Springer–Verlag, 2003.

[111] R. Kurki-Suonio and M. Katara. Logical layers in specifications with distributed objects and real-time. *Computer Systems Science & Engineering*, 14(4): 217 - 226, 1999.

[112] R. Kurki-Suonio, K. Systä, and J. Vain. Real-time specification and modeling with joint actions. *Science of Computer Programming*, 20(1-2): 113 - 140, 1993.

[113] H. H. Kwak, I. Lee, A. Philippou, J-Y. Choi, and O. Sokolsky. Symbolic Schedulability Analysis of Real-time Systems. In *Proceedings of IEEE RTSS'98*, 1998.

[114] O. Kupferman, P. Madhusudan, P. Thiagarajan, and M. Vardi. Open systems in reactive environments: Control and synthesis. In *Proceedings of the 11th International Conference on Concurrency Theory CONCUR 2000*, Lecture Notes in Computer Science, vol. 1877, pp. 92-107, Springer–Verlag, 2000.

[115] L. Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872-923, 1994.

[116] L. Lamport. Specifying Systems - The TLA+ Language and Tools for Hardware and Software Engineers. *Addison Wesley Publishing Company*, 2002.

[117] K. Larsen, P. Petterson, and W. Yi. UPPAAL in a nutshell. *Springer International Journal of Software Tools and Technology Transfer*, vol. 1, nr. 1, 1997.

[118] J.Y.T. Leung and J. Whitehead. On the Complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks. *Performance Evaluation (Netherlands)*, 2, pp. 237-250, 1982.

[119] C.L. Liu and J.W. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *Journal of the ACM*, 20, pp. 40-61, 1973.

[120] N. Lynch, R. Segala, and F. Vaandrager. Hybrid I/O Automata Revisited. In *Proceedings of the 4th Hybrid Systems Computation and Control (HSCC 2001)*, Lecture Notes in Computer Science, vol. 2034, pp. 403-417, 2001.

[121] N. A. Lynch and M. R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2, pp. 219-246, 1989.

[122] O. Maler, A. Pnueli, and J. Sifakis. On the synthesis of discrete controllers for timed systems. In *Proceedings of STACS'95, E. W. Mayr and C. Puech (Eds.)*, Lecture Notes in Computer Science, vol. 900, pp. 229-242, Springer-Verlag, 1995.

[123] Z. Manna and H. B. Sipma. A Deductive Approach towards Controller Synthesis. In *Proceedings of the IEEE International Symposium on Intelligent Control*, pp. 35-41, IEEE Press, 1995.

[124] The MathWorks: Developers of MATLAB and Simulink for Technical Computing. *http://www.mathworks.com*.

[125] MATLAB User Guide. The MathWorks, Inc., Natwick, USA, 1997.

[126] R. Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, vol. 25, issue 3, pp. 267-310, 1983.

[127] R. Milner. *Communication and Concurrency.* International Series in Computer Science. Prentice Hall, 1989.

[128] U. Montanari, F. Rossi. Concurrency and Concurrent Constraint Programming. In *A. Podelski ed., Constraint Programming: Basics and Trends*, Lecture Notes in Computer Science 910, pp. 171-193, Springer-Verlag, 1995.

[129] C. C. Morgan. The specification statement. *ACM Transactions on Programming Languages and Systems*, 10(3):403-419, 1988.

[130] C. C. Morgan. *Programming from Specifications*. Prentice Hall, 2 edition, 1994.

[131] J. M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*,9(3):287-306, 1987.

[132] P. J. Mosterman. An Overview of Hybrid Simulation Phenomena and Their Support by Simulation Packages. In *Hybrid Systems: Computation and Control (HSCC'99)*, Lecture Notes in Computer Science, vol. 1569, pp. 165 - 177, 1999.

[133] L. de Moura, H. Rueß, and M. Sorea. Lazy Theorem Proving for Bounded Model Checking over Infinite Domains. In *Proceedings of the 18th Conference on Automated Deduction (CADE 2002)*, Lecture Notes in Computer Science, vol. 2392, pp. 438-455, Springer-Verlag, 2002.

[134] G. Nelson. A Generalization of Dijkstra's Calculus. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 11, no. 4, pp. 517-561, 1989.

[135] C. A. Petri. Communication with Automata. Tech. Rep. RADC-TR-65-377, Vol. 1, Suppl. 1, Applied Data Research, Princeton, NJ, 1966.

[136] A. Pnueli. The Temporal Logic of Programs. In *Proceedings of the 18th IEEE Symposium on the Foundations of Computer Science (FOCS-77*, pp. 46-57, IEEE Computer Society Press, 1977.

[137] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proceedings of the 16th ACM Symposium on Principles of Programming Languages*, pp. 179-190, 1989.

[138] A. Puri and P. Varaiya. Verification of Hybrid Systems using Abstractions. In *Hybrid Systems II*, Lecture Notes in Computer Science, vol. 999, pp. 359-369, Springer–Verlag, 1995.

[139] P.J. Ramadge and W.M. Wonham. Supervisory control of a class of discrete event processes. *SIAM Journal of Control and Optimization 25*, pp. 206-230, 1987.

[140] M. Rönkkö and H. Li. Linear Hybrid Action Systems. *Nordic Journal of Computing*, 8, 1, pp. 159-177, 2001.

[141] E. Sekerinski and K. Sere. A Theory of Prioritizing Composition. *The Computer Journal*, vol. 39, nr. 8, pp. 701-712, 1996.

[142] H. Saïdi and N. Shankar. Abstract and model check while you prove. In *Proceedings of Computer-Aided Verification (CAV'99)*, Lecture Notes in Compueter Science, no. 1633, pp. 443-454, 1999.

[143] T. Seceleanu, D. Garlan. Synchronized Architectures for Adaptive Systems. In *Proceedings of the 2nd International Workshop on Software Cybernetics*, Edinburgh, Scotland, 2005, pp. 146-151.

[144] N. Shankar, S. Owre, J. M. Rushby, and D. W. J. Stringer-Calvert. PVS prover guide. *Computer Science Laboratory*, SRI International, Menlo Park, CA, 2002.

[145] H. B. Sipma. Diagram-Based Verification of Discrete, Real-Time and Hybrid Systems. Ph.D. Dissertation, Computer Science Department, Stanford University, 1999.

[146] H. B. Sipma and Z. Manna. Specification and Verification of Controlled Systems. In *Proceedings of the Third International Symposium on Formal Techniques in Real Time and Fault Tolerant Systems*, Lecture Notes in Computer Science, pp. 641-659, 1994.

[147] M. Slanina. Control Rules for Reactive System Games. In *Proceedings of the 2002 AAAI Spring Symposium on Logic-Based Program Synthesis*, Stanford, California, 2002.

[148] O. Slupphaug, J. Vada, and B. A. Foss. MPC in Systems with Continuous and Discrete Control Inputs. In *Proceedings of the American Control Conference*, 1997.

[149] J. H. Taylor Rigorous Hybrid Systems Simulation with Continuous-time Discontinuities and Discrete-time Agents. Chapter 60 in *Software and Hardware Engineering for the 21st Century*, pp. 383-388, World Scientific and Engineering Society Press, 1999.

[150] A. Tiwari, H. Rueß, H. Saïdi, and N. Shankar. A Technique for Invariant Generation. In *Proceedings of the 4th European Joint Conference on Theory and Practice of Software, Tools and Algorithms (TACAS 2001)*, Lecture Notes in Computer Science, Springer-Verlag, 2001.

[151] A. Tiwari, N. Shankar, and J. Rushby. Invisible Formal Methods For Embedded Control Systems. In *Proceedings of the IEEE*, vol. 91, nr. 1, pp. 29-39, 2003.

[152] G. K. Theodoropoulos. Distributed Simulation of Asynchronous Hardware: The Program Driven Synchronization Protocol. *Journal of Parallel and Distributed Computing*, vol. 62, Issue 4, pp. 622-655, 2002.

[153] H. Treharne and S. Schneider. Using a Process Algebra to control B Operations. In *Proceedings of the 1st International Conference on Integrated Formal Methods (IFM 99)*, Lecture Notes in Computer Science, pp. 437-456, Springer-Verlag, 1999.

[154] S. Tripakis and K. Altisen. On-the-fly controller synthesis for discrete and dense-time systems. In *Proceedings of the World Congress on Formal Methods (FM'99)*, 1999.

[155] N. Wirth. Program development by stepwise refinement. *Communications of the ACM*, 14(4):221-227, 1971.

[156] S. Wolfram. *The Mathematica Book*. Fourth Edition, Wolfram Media/Cambridge University Press, 1999.

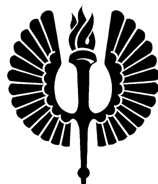
[157] W. M. Wonham. Notes on control and discrete event systems. Department of Electrical and Computer Engineering, University of Toronto, 1999.

[158] H. Zheng, E. A. Lee. Operational Semantics for Hybrid Systems. Presentation at the *Sixth Biennial Ptolemy Miniconference*, 2005.

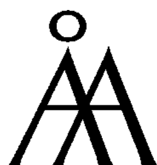
[159] Q. Zheng and K. G. Shin. On the Ability of Establishing Real-Time Channels in Point-to-Point Packet-Switched Networks. *IEEE Transactions on Communications*, vol. 42, nr. 2/3/4, FEBRUARY/MARCH/APRIL, 1994.

TURKU
CENTRE *for*
COMPUTER
SCIENCE

Lemminkäisenkatu 14 A, 20520 Turku, Finland | www.tucs.fi



University of Turku
- Department of Information Technology
- Department of Mathematical Sciences



Åbo Akademi University
- Department of Computer Science
- Institute for Advanced Management Systems Research

Turku School of Economics and Business Administration
- Institute of Information Systems Sciences

ISBN 951-29-4015-9
ISSN 1239-1883