Specification and Test of Real-Time Systems

Nielsen, Brian

Publication date:
2000

Document Version
Publisher's PDF, also known as Version of record

Link to publication from Aalborg University

Citation for published version (APA):
Nielsen, B. (2000). *Specification and Test of Real-Time Systems*. Aalborg Universitetsforlag. Publication : Department of Computer Science, The Faculty of Engineering and Science, Aalborg University No. 12

ISSN 1399-8145

AALBORG UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Fredrik Bajers Vej 7E, 9220 Aalborg Øst, Denmark

Specification and Test of Real-Time Systems

PhD thesis

by

Brian Nielsen

Supervisor: Arne Skou

April 2000

Abstract

Distributed real-time computer based systems are very complex and intrinsically difficult to specify and implement correctly; in part this is caused by the overwhelming number of possible interactions between system components, but especially by a lack of adequate methods and tools to deal with this complexity. This thesis proposes new specification and testing techniques.

We propose a real-time specification language which facilitates modular specification and programming of reusable components. A specification consists of a set of concurrent untimed components which describes the functional behavior of the system, and a set of constraint patterns which describes and enforces the timing and synchronization constraints among components.

We propose new techniques for automated black box conformance testing of real-time systems against densely timed specifications. A test generator tool examines a specification of the desired system behavior and generates the necessary test cases. A main problem is to construct a reasonably small test suite that can be executed within allotted resources, while having a high likelihood of detecting unknown errors. Our goal has been to treat the time dimension of this problem thoroughly.

Based on a determinizable class of timed automata, Event Recording Automata, we show how to systematically and automatically generate tests in accordance with Hennessy's classical testing theory lifted to include timed traces. We select test cases from a coarse grained state space partitioning of the specification, and cover each partition with at least one test case, possibly selecting extreme clock values. In a partition, the system behavior remains the same independently of the actual clock values.

We employ the efficient symbolic constraint solving techniques originally developed for model checking of real-time systems to compute the reachable parts of these equivalence classes, to synthesize the timed tests, and to guarantee a coverage of the equivalence class partitioning. We have implemented our techniques in the RTCAT test case generation tool.

Through a series of examples we demonstrate how Event Recording Automata can specify untrivial and practically relevant timing behavior. Despite being theoretically less expressive than timed automata, it has proven sufficiently expressive for our examples, but sometimes causing minor inconveniences. Applying RTCAT to generate tests from these specifications, including the Philips Audio Protocol, resulted in encouragingly small test suites.

We conclude that our approach is feasible and deserves further work, but also that it should be generalized and allow timing uncertainty and modeling of the environment. Some implementation improvements are also necessary.

Dansk Resumé

Distribuerede tidstro computer baserede systemer er notorisk komplekse og svære at udvikle korrekt. En væsentlig årsag hertil er den enorme og uoverskuelige mænge af mulige interaktioner imellem samtidige komponenter. Der mangler metoder og værktøjer til håndtering af denne kompleksitet. Denne afhandling foreslår nye teknikker til specifikation og test af tidstro systemer.

Det første bidrag er et specifikationssprog, der forbedrer mulighederne for at konstruere genbrugelige specifikationer og programkomponenter. En sådan specifikation består af et antal samtidige komponenter, som beskriver systemets logiske adfærd, og en mængde restriktorer, der gennemtvinger de tids- og synkroniseringskrav, der skal holde imellem komponenterne.

Afhandlingens hovedbidrag er nye teknikker til automatiseret konformitetstest med fokus på test af tidskrav. Et test genereringsværktøj analyserer en given tidsautomat specifikation af den ønskede systemadfærd og genererer automatisk de nødvendige tests. Et væsentligt problem er at generere en passende mængde af tests, der kan udføres inden for de ressourcer, der er afsat til aftestning, men som har stor sandsynlighed for at detektere ukendte fejl.

Afhandlingen viser, hvorledes det er muligt automatisk og systematisk at generere tests fra en determinisérbar klasse af tidsautomater kaldet Event Recording Automata. Testene er udledt fra Hennessy's klassiske testteori, som er udvidet til at omfatte tid. Udvælgelse af tests sker på baggrund af en grovkornet inddeling af specifikationens tilstandsrum. Specifikationens adfærd i hver del er identisk uanset specifikke urværdier. Hver del af tilstandsrummet dækkes med mindst en test, og potentielt ved valg af ekstreme urværdier.

Til beregning af specifikationens opnåelige tilstande og til udledning af tests anvendes nye effektive teknikker til symbolsk løsning og repræssentation af de lineære uligheder over urene, som forekommer i specifikationen. Disse teknikker er oprindligt udviklet til formel bevisførelse af tidstro systemer vha. modelchecking. Teknikkerne er implementeret i testgenereringsværktøjet RTCAT.

Ved specifikation af en række eksempler har det vist sig, at Event Recording Automata er egnede til specifikation af utrivielle og praktisk relevante tidskrav. Selv om udtrykskraften af denne automat model teoretisk set er mindre end generelle tidsautomater, har den været tilstrækkelig, dog sommetider med visse komplikationer. Mængden af tests genereret af RTCAT værktøjet for disse specifikationer, inklusive en Philips Audio Protokol, er kun moderat stor, hvilket er lovende for anvendeligheden af teknikkerne på større systemer.

Det konkluderes, at de foreslåede teknikker er potentielt anvendelige og bør videreudvikles. De bør genereraliseres og muliggøre specifikation af tidsusikkerhed og omgivelsesantagelser. Visse implementationsaspekter bør forbedres.

Acknowledgments

> ”One does not discover new lands without consenting to lose sight of the shore for a very long time.” (André Gide)

Doing a PhD is not only searching for new technical discoveries, but also a personal, and sometimes an alone, endeavor. But without the support of a number of people, I would most likely still be paddling in open seas with no shore lines in sight. First and foremost, I owe a lot of gratitude to my supervisor, Arne Skou, for making this project possible, and for continuously supporting and guiding me through the years (all of them).

I'm also grateful to all my colleagues in the Distributed Systems and Semantics research unit at Aalborg University for making this an agreeable place to work. Special thanks to Mikkel Christiansen, Peter K. Jensen, Kåre Kristoffersen, Paul Pettersson, Anders P. Ravn, and my office mate on more than one occasion, Yogi, for their comments and time for discussions.

My year long stay at the Open Systems Laboratory at University of Illinois in 1995 is still a recurring theme in my dreams. It has been more than a stay abroad, but an experience for life. I'm grateful to Gul Agha for hosting me, and for supervising me during this time. Special thanks to Mark Astley, Shangping Ren, Masahiko Saito, and Dan Sturman, for their friendships on and off work.

Also thanks to Jan Peleska and the members of the Distributed Systems and Operating Systems group at the University of Bremen for hosting me from November 1999 to January 2000. This gave me valuable insight into their approach to testing, and moreover, the quiescence I needed to finish the writing of this thesis.

Financial support was provided by the Danish Technical Research Foundation (STVF) and the Danish Research Academy.

Last, but not least, thanks to my family and friends for accepting my long periods of mental and physical absence.

Contents

Chapter 1

Introduction

> “The biggest problem at the moment is the timing between the electronics in the star camera that determines the satellite’s orientation in space, and the satellite’s main computer. This implies that the orientation of the satellite is not accurately known when it measures the magnetic field.
>
> According to Lars Tøffner-Clausen from the Danish Meteorologic Institute, the time stamp on the data from the star camera occasionally leaps 1 second. Therefore, the recorded time of a measured datum may be up to 50 seconds wrong. So far, the technicians have been unable to locate the cause of the error in the satellite, and the hope is therefore that the error can be corrected by post processing the data on earth.”

The above excerpt is translated from the Danish technical weekly “Ingeniøren” [86]. The news story concerns the Danish Ørsted satellite whose purpose is to make accurate and detailed measurements of the earth’s magnetic field. Although the error is not fatal, it degrades the value of the satellite if not corrected.

Unfortunately, similar reports of malfunctioning computer based systems are rather commonly reported in the technical news.

Another, widely cited case, is the Therac-25 therapeutic radiation machine in which software malfunctions caused patients to be exposed to massive radiation overdoses resulting in deaths and injuries of several patients in the period of 1985–87 [69]. One kind of errors, among Therac-25’s horribly many, is described by the Tylor incidents where unanticipated operator behavior led to wrong radiation type. Through a computer console the operator keyed in the treatment parameters such as length, strength, and type of therapy (electron vs. x-ray radiation). The machine would then prepare for the treatment by changing its physical setup, including positioning of bending magnets that would take 8 seconds. The normal treatment was x-ray radiation, so when occasionally electron radiation was required, the operator sometimes routinely set it to x-rays. The Therac-25 user interface permitted editing of the treatment parameters while the machine was changing its setup. When the operator recognized his mistake and was fast enough to change the treatment parameters before this setup was completed, the machine would fail to recognize the change from x-ray radiation to the more benign electron radiation. When the operator later prompted the machine to start the treatment, it would emit a large x-ray dose despite the user interface indicating electrons. Lack of systematic testing and other poor software engineering practices is stated to be a major contribution to the faulty behavior of Therac-25 [69].

The problem with faulty computer systems is even bigger because newspapers only report on such spectacular cases.

1.1 Distributed Real-Time Systems

The Ørsted satellite is an example of a large class of computer based systems that are distributed and that must operate according to real-time constraints. Other examples are numerous: control and monitoring of factory plants, automatic train control systems, anti break/spin systems for cars, etc. A *distributed system* consists of multiple concurrent communicating components. It is well known that the communication protocols, which describe the exact rules for how the components can and should communicate, are very complex and intrinsically difficult to specify and implement correctly. One reason for this, as the Ørsted, Therac-25, and other cases illustrate, is that the possible interactions between system components internally, and between the system and its environment, is incomprehensible even for few and small systems. Also, the lack of global state and accurate global time caused by distribution of the components to several independent processors plays an important role. In addition, processors may fail independently, which adds a wealth of fault tolerance problems and opportunities.

The components not only communicate internally, but also interact with external environment or human operators. The external environment consists of the physical equipment that is monitored and controlled by the computer system. The system can query the state of the environment through analogue sensors and analog-to-digital converters, and affect the equipment via actuators and digital-to-analogue converters. This model is depicted in Figure 1.1.

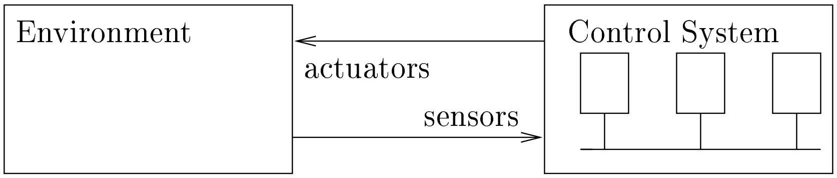


Figure 1.1. *Model of a distributed real-time system.*

The physical laws governing the environment induces a set real-time constraints which the control system must obey in order to achieve satisfactory or safe operation. Thus, a *real-time system* must not only produce the correct result or make the correct reaction to a stimuli, but must also produce the result or reaction at the correct time instant; neither too early nor too late. In other words, a timely reaction is just as important as the kind of reaction. These real-time requirements add a further layer of complexity to the already complex distributed systems. Their real-time properties must the specified, implemented, and validated.

Distributed real-time systems remain among the most challenging class of systems to develop correctly. The challenge arises in part from the inherent complexity in these systems, but in particular from the lack of adequate methods and tools to deal with this complexity. This applies to most development activities, including specification, design and implementation.

This thesis is concerned with the development of correct distributed real-time systems, in particular with their specification and test. We believe that the following items should be ingredients in a sound and effective systems development method:

- Methods should be rooted in formal methods. Formal methods not only support more stringent development, but, more importantly, permit development of sophisticated tools that can assist the developers in managing complex systems.

- Testing should be fully automatic. We propose fully automated testing from a formal specification as a way of improving correctness.

- Components should be reusable. New systems should to the widest possible extend be constructed by composing existing and previously tested components. This avoids unnecessary re-development and retesting.

Section 1.2 introduces testing, and Section 1.3 introduces our thesis.

1.2 Testing

In the following we introduce testing, and discuss how it is presently being used, and why it needs to be improved. We also relate it to other validation strategies such as formal verification.

1.2.1 Testing in Context

In essence, *testing* consists of executing a program or system with the intention of finding undiscovered errors in the system. As much as about a third of the total development time is spent on testing, and therefore constitutes a significant portion of the cost of the product. As discussed in the following, testing is performed throughout the product development cycle, and for various purposes.

Programmers usually test their own modules to check their correctness during development. Programmers also participate in the various integration tests also performed during the product development. Customers participate in the final acceptance test, where the goal is to determine whether the product satisfies the requirements listed in the original development contract negotiated between the customers and the software house.

Many companies employ dedicated test engineerers that exclusively focus on writing test cases and/or executing these. Some even use dedicated test departments that tests the products developed by another development department. The test departments may further be isolated (in clean rooms) from the developers in order to minimize 'contamination' of the test team by prejudices, presuppositions, and internal knowledge, that could biase tests.

Frequently, the products from different, and sometimes competing, companies must interoperate. To ensure this, standardization committees develop common standard specifications which must be adhered to by all vendors. A compliant product must pass a standard test suite also developed by the standardization committee. Typical examples of this are found in the found in the telecommunications world where standardization committees like ITU (International Telecommunication Union) and ISO (International Organization for Standardization) develop and manage communication protocol standards and corresponding standard test suites. For example, before a new mobile phone is mass produced, a sample must be sent to an accredited testing site that tests the product for protocol conformance and interoperability with the equipment used by mobile network providers.

Finally, safety critical systems must undergo extremely thorough testing, and the test results must convince public safety boards that the systems are sufficiently safe, e.g., the United States Federal Aviation Administration, FAA.

Despite of the many man-hours and resources that are spent on testing, and despite the many errors found and corrected, significant errors remain. Because testing is the most dominating validation activity used by industry today, there is an urgent need for improving its effectiveness, both with respect to the use of time and resources, and the number of uncovered errors. A potential solution that is being examined by researchers is to formalize testing, and to provide tools that automates test case generation and execution. This approach has experienced some level of success: Formal specification and automatic test generation are being applied in practice [19, 72, 80, 98], and commercial test generations tools are emerging [59, 82]. However, little research deals systematically with the special needs of real-time systems.

1.2.2 Automated Testing

The type of testing examined in this thesis is conformance testing. The goal is to check by means of testing whether the behavior of the implementation under test conforms to a specification. The implementation under test is viewed as a black box whose internal structure and behavior is unknown to the tester. In general, only its interface to the external world is known. Exactly when a implementation conforms to a specification is defined by a formal implementation relation. There are several possible definitions, and we shall return to these throughout the thesis.

In conventional manual conformance testing the programmer or test engineer is responsible for constructing and executing test cases, see Figure 1.2.

He typically scrutinizes the specification given as informal prose, and identifies a set of test purposes. A *test purpose* is a specific goal or action deemed necessary to test. For example, in a communications protocol, a typical test purpose is to check that a connection can be established. In a steam boiler controller, a test purpose could be to check that the steam valve opens when the pressure in a water tank reaches a given threshold. For each test purpose, the test engineerer constructs at least one test that checks for that purpose. A collection of tests that concerns the same specification is called a *test suite*. The final step is to construct a program that implements the test, and to execute it.

In its simplest form, a *test case* is a sequence of input events, expected outputs, and a pass/fail verdict for each step of the sequence. The system passes a test if its execution reaches a state with a pass verdict. A test consists of three sub-sequences: A preamble brings the implementation to a state where the test purpose can be examined. Then follows the central sequence for the test purpose. A postamble brings the system to a known (or safe idle) state.

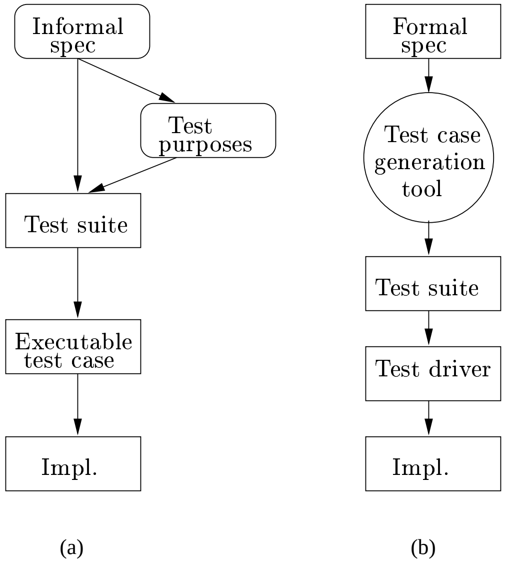


Figure 1.2. *Manual testing (a), and fully automatic testing (b).*

It is important to realize that the test engineerer is responsible for two critical steps: identifying what should be tested, and constructing the sequences. He performs two levels of test selection. One is identifying and selecting the test purposes to be tested. The other is the selection of the specific sequences needed for each purpose. It is not difficult to imagine that subtle scenarios are overlooked, or that the preamble contains errors or are too simple. Further, constructing test sequences is a time consuming and often tedious job.

Fully automatic testing is the extreme opposition to manual testing. The vision is to automatize testing completely, as illustrated by the schematics in Figure 1.2b. A test case generation tool takes as input a formal specification of the required behavior and systematically generates all relevant tests. Because exhaustive testing is impractical, it must select the most important ones for execution. A test execution tool takes the produced test cases as input, interprets them, and exercises the implementation under test accordingly and writes the test results to a log file. Additionally, the test execution tool may generate diagnostic information that can help the system developers in locating the cause of a detected discrepancy.

Between the completely manual testing and fully automatic testing approaches outlined above, there is plenty of room for various degrees of automatization and engineerer (expert user) intervention.

1.2.3 Test Selection

Even modest sized systems are so large that it is generally impossible to test them exhaustively, i.e., under all possible sequences of inputs. It is therefore necessary to select and execute only a subset of these. The goal of this procedure, known as *test selection*, is to find the 'best' test suite that can be executed within the allotted time frame and resources. A good test suite is usually one that has a high probability of finding so far unknown errors, or of finding the most serious errors. Rather than stimulating the system at random, it may prove better to systematically check as many truly different parts of the system as possible.

To make this point clear, consider the 'maxPositive' function specified and implemented in Figure 1.3. The function returns the maximum positive value of two arbitrary integers, x and y. If both arguments are less than zero the function is to return zero. Obviously, this function cannot be tested with all possible pairs of integers as inputs. A systematic strategy for dealing with this problem is to partition the input variables into input *domains*, i.e., sets of inputs that the program is expected to treat "identically" (e.g., pass through the same program path), and choose only a few representatives from each domain.

In the 'maxPositive' example there are four domains, one for each of the four cases in the specification. The resulting domains, tabulated in Table 1.4, are described as inequations of x and y. Test input data can be derived from the inequations that define each domain. Thus, at least four test cases should be generated, but usually several interior and extreme values are chosen.

domain	condition	expected output
1	$x < 0 \wedge x > y$	0
2	$y < 0 \wedge x \leq y$	0
3	$x \geq 0 \wedge x > y$	x
4	$y \geq 0 \wedge x \leq y$	y

Table 1.4. *Domains for the maxPositive specification.*

Returning our attention to real-time systems, it should be clear that the implementation cannot be stimulated at all possible time instances with all possible input actions. Thus, we are faced with a choice of *when* to stimulate the system. This choice need not be arbitrary, but can and should be made based on some notion of specification partitioning. This thesis will analyze the values of clocks in real-time specifications, and partition them, and use this as basis for time instance selection.

$$\text{maxPositive}(x, y) =_{\text{def}} max(0, max(x, y))$$

$$max =_{\text{def}} \begin{cases} x, & \text{if } x > y \\ y, & \text{otherwize} \end{cases}$$

(a)

```
int MaxPositive(int x,y) {
  int max;
  if (x > y)
    max=x;
  else
    max=y;
  if (max<0) //x < 0 ∧ y < 0
    max=0;
  return max;
}
```

(b)

Figure 1.3. *Specification of the maxPositive function (a), and C-program implementation (b) ?*

1.2.4 Testing and Verification

A legitimate concern is whether testing will ever be an effective validation technique. It is well known that testing can only prove the presence of errors, not their absence. A proof would require exhaustive testing which is only possible for small systems, and under a particular set of assumptions. It therefore appears futile and irrational to exercise the system more or less at random. It would be much better to prove that the system behaves correct under all possible conditions!

This rationale has dominated the formal methods research community, and most researches focus on proving system correctness, i.e., on *verification*. The major benefit from this priority is that verification technology has matured significantly. Efficient algorithms and data structures have been developed to an extend that permits verification of systems of industrial relevance. One kind of verification technology is *model checking* where a (fully) automatic tool examines whether the states of a behavioral description of a system satisfies a given property. Another approach is *theorem proving* where a proof assistant tool helps the system developer to prove that the system has a given property.

The downside is that testing has been treated somewhat stepmotherly: It is viewed an academically unsound activity carried out by pragmatic people who are not true believers in formal methods. Testing technology has therefore been unable to keep up with the verification technology. Fortunately, many of the developed verification techniques also seem applicable to testing, as we shall try to demonstrate in this thesis.

From the preceding discussion one might get the impression that one should decide on using either testing or verification as the prefered validation technique. However, in our view testing and verification are complementary techniques solving different problems. Therefore both should be used. The relation between different validation techniques and the (idealized) phases of software development is shown in Figure 1.5. Based on this observation we propose a development methodology where formal verification and testing are integral and complementary activities.

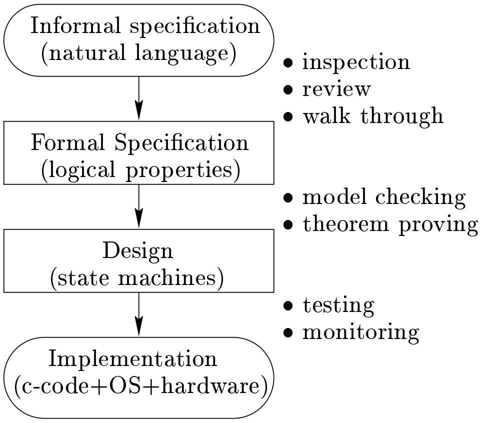


Figure 1.5. *Testing and verification are complementary techniques (inspired by Rushby [95]).*

The system analysts capture the informal requirements by interviewing customers, users, and domain experts about the requirements and expectations they have to the system. The formal requirements can then be formulated as logical properties or propositions. Completeness and soundness are achieved by reviews, inspections, and walk troughs.

The analysis and design activities result in an abstract model of the system to be implemented: its components, the behavior of these components, and their interaction. The central parts of the model can be specified in detail using a formal behavioral description language such as state machines or process algebra. Model checking can then be used to ensure that this formal model satisfies the desired properties.

Finally, the programmers take over and implement the design using a particular programming language, operating system, and hardware components. Automatic code generation from the formal model can be used in special cases. However, with the current state of the art, system construction is largely an informal step carried out by humans. Therefore, the final physical system may be faulty even if its design is correct. It should also be mentioned here that many public safety boards require extensive testing of safety critical systems. It is insufficient, although important, to show that the design is correct: It must be demonstrated that the actual physical system is safe. However, the (verified) design or specification can now be used as a basis for generating test cases.

An interesting but open question is whether it ever will be possible to formalize and automatize the implementation process entirely such that implementations can be verified. This would require faithful formal models of the operating system and hardware components, formal semantics of programming languages, verified compilers, and translators from the design specification. At least, the resulting model will be so large that it cannot be verified with the technology available in the near future.

The fundamental *goal of testing*, which cannot be done by model checking, is to check if an actual running physical system of which we have incomplete knowledge conforms to a specification.

1.3 The Thesis

Having presented the problem domain and motivated our work, this section introduces our contributions. We outline the main problems that should be solved, and present our approach to this. We summarize our contributions and present the structure of the remainder of the thesis.

1.3.1 Specification of Real-Time Systems

The starting point for automatic testing of real-time system, and indeed for most automated tool support for their analysis, is a clear and unambiguous specification of the systems desired behavior. Such specifications tend to be complex and non-trivial to write. The particular specification language used, and the methodology used to derive specifications, is therefore of great importance. It should be possible to specify systems with a reasonable effort. Further, the generated tests or other analysis results will only be as good as the specifications on which they are based. Therefore, the correctness of these models is therefore also a concern.

Specification languages are presently given as somewhat low level and mathematically oriented formalisms such as state machines or automata specifications, or as lengthy process algebraic expressions. However, in the long run, as specifications grow in size and complexity, it becomes increasingly important how to give and manage them. In particular we believe that making *modular* and *reusable* specifications becomes essential.

It is our thesis that reuse of specifications and implementations can be facilitated by a component based approach that separates specification of time constraints and functional behavior.

We propose a real-time specification and modeling language in which a specification consists of two parts. The first part is a set of concurrent *untimed components* describing the functional behavior of the system. The second part is a set of *constraint patterns* describing and enforcing the timing and synchronization constraints that should exist between components. Both components and constraint patterns are reusable. This enables key properties to be easily maintained in one system and re-established in future ones.

Another issue is how to derive the implementation from a specification. In some cases, it may be possible to execute a specification directly with little or no extra programming effort. We show how this sometimes is feasible from our modular specifications for soft real-time systems.

1.3.2 Testing of Real-Time Systems

While a significant body of research exist on languages, theories, algorithms and tools for automatic testing of untimed systems, relatively little work deals explicitly with real-time systems. It is the aim of this thesis to develop techniques for automatically generating test cases for real-time systems.

The emphasis on real-time implies that the entire automatic testing setup must be extended to accommodate real-time. The specification language must permit *specification of real-time constrains*, and must be suited for the type of analysis required for generating test cases. The underlying *testing theory* which deals with observation of events, the precise definition of conformance, and the necessary structure of the test cases etc. must be revised and be extended to include real-time. Test cases would no longer only be sequences of inputs and expected outputs, but would also need to include *when* input is to be delivered and *when* output is expected. Because it is generally impossible to test the system exhaustively, *test selection* becomes even more imperative. Effective *algorithms and tools* for generating and selecting timed test cases must therefore be devised. Effective here means ability to handle systems with a size of practical relevance. Finally, the *execution* of real-time test cases must also be changed such that it precisely exercises the implementation at the time instances dictated by the test case. This requires special care, particular in a distributed system where clocks cannot be perfectly synchronized.

In the following we describe our approach to each of these challenges. We propose timed automata as specification language. This automata based formalism is very expressive, and has become a popular modeling language for real-time systems. In essence, a *timed automata* is a classical state machine augmented with clocks, enabling conditions on actions, and resets of clocks when actions occur.

We employ a testing theory that is a timed extension of Hennessy's classical testing theory [75]. In this theory, two systems are deemed equivalent if no test case exist that one system passes, but the other fails. The theory also defines the exact structure of the test cases required to determine testing equivalence.

We address the important test selection issue by proposing a coarse grained state space partitioning of the specification. In each partition the specification behaves the same independently of the actual clock values. This approach is related to the domain test selection strategy commonly employed in the testing of sequential programs [116]. Our basic hypothesis is that it is more important to systematically test many different situations as represented by the partitions, than it is to blindly select time instances within the same narrow part of the state space.

To represent the state space of the specification and to compute the reachable parts of the state partitions, we propose to employ the symbolic techniques that has resulted from the last decade's research in verification of real-time systems; in particular we have the UppAal approach [119, 13, 64] in mind. This tool performs reachability analysis to prove bounded liveness properties of a collection of concurrent time automata. It has been successfully applied to verification of industrially relevant systems. Because test case generation also involves exploration of the state space, the data structures and algorithms behind this success could potentially also prove advantageous to testing.

It is our thesis that the symbolic techniques developed for reachability analysis of real-time systems also can be used to generate test suites, and that application thereof results in a practical technique for systematic and automatic testing of real-time systems.

We evaluate this thesis through the construction of a prototype tool that implements our ideas. The tool will be applied to a number of specifications. Our evaluation also includes thoughts on the specification language, the algorithms, the number and kinds of generated tests. The many and important practical problems of executing tests, in particular on a distributed system shall not be our main concern here.

1.3.3 Contributions

The following summarizes our main contributions:

1. We propose a set of basic algorithms for testing of *untimed* communicating state machines, and implement these algorithms in a test case generator.
2. We propose a framework for selecting real-time test cases. The framework is based on a partitioning of the state space. We make several instantiations of this framework, and choose one for further study.
3. We design algorithms that generate tests using the above state space partitioning. The proposed algorithms are applicable to a restricted, but determinizable, class of timed automata termed *event recording automata.*
4. The test generation algorithms are implemented in a tool capable of generating test cases automatically from an event recording automata specification.
5. We show how our approach can be applied to generate test cases from a set of practically relevant specifications.
6. Our final contribution concerns reuse of program components operating under real-time constraints. We propose a specification language that supports reuse through separate and modular specification of functional behavior and time constraints.

1.3.4 Structure of the Thesis

The remainder of the thesis is organized as follows.

In Chapter 2 we present the testing fundamentals necessary to understand our approach. We lay down the classical untimed testing theory. Also, to understand how a test generation tool could operate, we study test generation algorithms in the simpler untimed setup before addressing the much more challenging real-time case. This work results in a test case generation tool for untimed communicating state machines.

Chapter 3 introduces the real-time dimension. We propose a specification language based on timed automata, and define its semantics as a timed labeled transition system. We show how to derive tests from such a timed transition system. This approach is however unideal because it gives no guidance for systematically choosing the time instance where the implementation should be tested. We also present a framework for test selection based on partitioning the state space of the specification.

Our main contribution is contained in Chapter 4 where we describe our symbolic approach to selection and generation of test cases. We present our prototype tool and describe the relevant implementation details.

In Chapter 5 we apply our tool to a set of small cases and one larger, and evaluate the applicability of our approach. We relate and compare our work to other research in the field of automated test generation in Chapter 6. Finally, Chapter 7 concludes and discuss topics for future work.

To give a natural flow of topics in the thesis, our contribution regarding reuse of real-time components is contained in Appendix A.

Chapter 2

Untimed Testing

The goal of this chapter is to lay down the fundamental definitions and test generation techniques before addressing the more general and challenging real-time case. We apply the techniques to an automata based specification language termed communicating state machines. However, this particular choice is not essential for the goal of this chapter since the essential definitions and algorithms are stated in terms of the underlying semantic model, labeled transition systems, which applies to a large family of languages. Specification and semantics of concurrent systems is discussed in Section 2.1.

An important issue is to define exactly when a system is a correct implementation of a specification, and when it is not. The adopted testing theory defines this by a formal relation between two labeled transitions system. The definition of this, the so called implementation relation, also depends on the type of observations that the tester assumes can be made about the implementation, and these must be stated explicitly. Because implementations are concurrent and non-deterministic, it is especially important that the theory checks that the deadlock properties of the implementation comply with those of the specification. Non-determinism also requires a delicate assumption to be made about the execution of tests to achieve exhaustiveness. However, we postpone the general discussion of observations and non-determinism until Chapter 6. The testing theory is presented in Section 2.2.

Given a well defined testing theory, the next issue becomes how to automatically generate tests which can be used to determine whether the implementation is correct. This involves defining precisely which tests should be generated, and finding good algorithms and data structures. We develop and present two techniques in Section 2.3. The first is based on a direct interpretation of the specification, and the second is based on a data structure called a success graph.

The success graph has several nice properties for automatic test generation, but is potentially costly to construct. To study how this data structure can be implemented, and to evaluate its cost, we have implemented it in an prototype tool described in Section 2.4. Finally, Section 2.5 discuss the advantages and disadvantages of the success graph, and the lessons learned.

2.1 Specification of Concurrent Systems

One broadly distinguishes between two different specification language paradigms, the logical languages, and the behavioral description languages. Logical languages such as temporal logics specify the requirements by a set of properties that should always or eventually hold. Behavioral languages such as process algebras and state machines specify requirements by a model that defines how the system changes state depending on current state and event.

According to the system development methodology outlined in Section 1.2.4, tests should be derived from an abstract model of the system under test, and henceforth, we shall focus on behavioral languages.

2.1.1 Communicating State Machines

We propose an automata based specification language, Communicating State Machines, or $\mathcal{L}_{\mathrm{csm}}$ for short. In the forthcoming Chapter 3 we shall propose a generalized timed version, timed automata, that has become popular for specifying real-time systems. We first present $\mathcal{L}_{\mathrm{csm}}$ informally here, and postpone the formal semantics until Section 2.1.3.

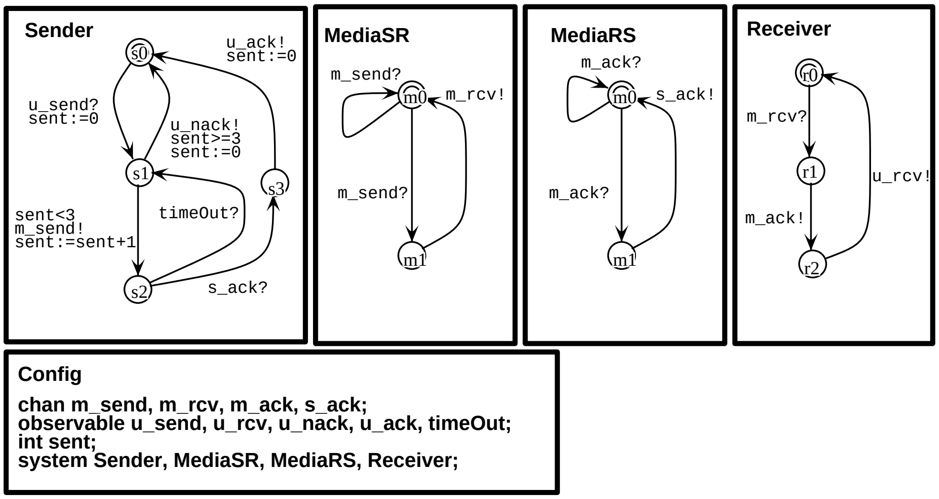
We use a graphical notation of state machines. A state machine is a directed graph whose nodes represent the different locations the machine can occupy, and whose edges represent the possible transitions between locations.

An edge is labeled with one of three kinds of actions: The machine can perform an *internal action* where it internally changes from one location to another. It can also enable an *observable action* to permit synchronization with the environment. Finally, two state machines can synchronize on complementary actions without involving the environment. When a *synchronization* occurs, both state machines change location. A pair of actions with the same name, but one suffixed with an exclamation mark, and the other with a question mark, indicates complementary actions which may synchronize. Although there is no semantic difference the exclamation mark typically indicates output and the question mark input[1].

[1] In languages with value passing they designate actual and formal parameters.

A specification consist of a *network* of state machines. A network represents a collection of concurrently executing and synchronously communicating machines. The concurrent execution is based on an interleaving semantics. A state machine may also access a number of shared integer variables. Its transitions can be guarded by a predicate over these variables such that a false guard disables the transition. When an action is executed, the variables can be updated using assignment expressions. Other finite discrete variable types could be permitted for convenience, but are adequately modeled by integeres.

A final feature is *committed* locations. If a machine enters a committed location, the next action performed by the network must be an action leaving that location, i.e., committed states must be left immediately. Committed locations are useful for modeling atomic sequences of actions, e.g., atomic broadcast or multi-way synchronization. We use the term *location* of an automaton to refer to the node that it currently occupies, and reserve the term state to denote a semantic state configuration consisting of locations and variable values.



```
chan m_send, m_rcv, m_ack, s_ack;
observable u_send, u_rcv, u_nack, u_ack, timeOut;
int sent;
system Sender, MediaSR, MediaRS, Receiver;
```

Figure 2.1. *Specification of a communication protocol.*

The example shown in Figure 2.1 demonstrates the most important features of $\mathcal{L}_{\mathrm{csm}}$. The example is a model of a simple stop and wait transport layer communication protocol with bounded retransmission. The purpose is to establish reliable communication. The network consists of four machines; a sender (`Sender`), a receiver (`Receiver`), and two machines modeling the communication media: `MediaSR` models the communication of data from sender to receiver, and `MediaRS` the communication of acknowledgments from the receiver to the sender. Table 2.2 summarizes the actions used by the protocol.

Action	Description
`u_send`	data from upper layer
`u_ack`	ack to upper layer: data has been received
`u_nack`	failure indication to upper layer: data unacknowledged
`u_rcv`	data to upper layer
`timeOut`	give up wait for acknowledgment
`m_send`	data to media from sender
`m_rcv`	data from media to receiver
`m_ack`	ack from receiver to media
`s_ack`	ack from media to sender

Table 2.2. *Protocol actions.*

The configuration box in Figure 2.1 declares the network of state machines (the `system` line), the actions used for communication internally (the `chan` line) and externally (the `observable` line), and the used integer variables (the `int` line). We use the AutoGraph [92] developed at INRIA, France, to edit state machine networks; AutoGraph is a graph editing tool with a graphical user interface. The protocol specification presented here can be fed to the automatic test generation tool to be presented later in Section 2.4.

Because the media may drop data packets and acknowledgments, the sender is prepared to time out and attempt retransmission three times. If the data is still unacknowledged, the protocol gives up and signals error to the user. The number of transmission attempts is modeled using an integer variable (`sent`), and by guarding the `m_send` and `u_nack` edges appropriately.

A final aspect of the specification worth pointing out is the modeling of timeouts. Because $\mathcal{L}_{\mathrm{csm}}$ is untimed, it does not permit specification of concrete timeout values. Instead, the possibility of timeouts is modeled by the observable `timeOut` action. The resulting model can be viewed as an over-approximation of the real protocol behavior since the timeout action is *always* enabled after a message has been sent, not only after a certain delay. An alternative is the `timeout` statement in the Promela protocol specification language [53] which only executes when no other statements are possible. This can thus be seen as an under approximation. However, such approximations are not always possible or sufficient. We therefore need a specification formalism with explicit timing.

2.1.2 Labeled Transition Systems

We use labeled transition systems (LTS) as the underlying semantic model. Sometimes it is also convenient to give small specifications directly as LTSs.

Definition 2.3 *Labeled Transition System* ($\mathcal{L}_{\mathrm{lts}}$):
An LTS is a 4-tuple $\langle S, s_0, Act_\tau, \rightarrow \rangle$, where

1. S is the set of states,

2. $s_0 \in S$ is the initial state,

3. Act is the set of observable actions, and $Act_\tau = Act \cup \{\tau\}$ the actions including the distinguished internal action τ,

4. $\rightarrow \subseteq S \times Act_\tau \times S$ is the transition relation.

5. We assume that Act is equipped with a mapping $\bar{}: Act \mapsto Act$ such that for all actions $\bar{\bar{a}} = a$. $\bar{a}$ is said to be the complementary action of a.

□

Intuitively, S represents the possible states of a computation. An edge from state s to s' labeled with action $\alpha \in Act_\tau$ represents that s' is a state that can result by executing action α in state s. Some convenient notation is summarized in Definition 2.4.

Definition 2.4 *LTS notation. Let $a \in Act$ and $\alpha \in Act_\tau$:*

Notation	Definition
$s \xrightarrow{\alpha} s'$	$(s, \alpha, s') \in \rightarrow$
$s \xrightarrow{\alpha}$	$\exists s'.\ s \xrightarrow{\alpha} s'$
$s \xrightarrow{\sigma} s'$	$\exists s_1, s_2 \ldots s_n.\ s \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} s_2 \cdots \xrightarrow{\alpha_n} s_n$ and $s_n = s'$, where $\sigma = \alpha_1 \cdot \ldots \cdot \alpha_n$
$s \xrightarrow{\sigma}$	$\exists s'.\ s \xrightarrow{\sigma} s'$
$\xrightarrow{\tau}^*$	the reflexive and transitive closure of $\xrightarrow{\tau}$
$s \overset{\epsilon}{\Rightarrow} s'$	$s = s'$ or $s \xrightarrow{\tau}^* s'$
$s \overset{a}{\Rightarrow} s'$	$\exists s_1, s_2.\ s \overset{\epsilon}{\Rightarrow} s_1 \xrightarrow{a} s_2 \overset{\epsilon}{\Rightarrow} s'$
$s \overset{\sigma}{\Rightarrow} s'$	$\exists s_1, s_2 \ldots s_n.\ s \xRightarrow{a_1} s_1 \xRightarrow{a_2} s_2 \cdots \xRightarrow{a_n} s_n$ and $s_n = s'$, where $\sigma = a_1 \cdot \ldots \cdot a_n$
$s \overset{\sigma}{\Rightarrow}$	$\exists s'.\ s \overset{\sigma}{\Rightarrow} s'$
sort(s)	$\{a \in Act \mid s \overset{a}{\Rightarrow}\}$
$Tr(s)$	$\{\sigma \in Act^* \mid s \overset{\sigma}{\Rightarrow}\}$
$Tr(\mathcal{S})$	$Tr(s_0)$
s **after** σ	$\{s' \mid s \overset{\sigma}{\Rightarrow} s'\}$
$\mathcal{S}$ **after** σ	s_0 **after** σ

□

We shall furthermore use a set of CCS [74] inspired operators on LTSs to enable their construction using algebraic notation, see Definition 2.5.

Definition 2.5 *Construction of LTSs*:
Let $\mathcal{S}_1 = \langle S_1, s_{01}, Act_1, \rightarrow_1 \rangle \in \mathcal{L}_{\text{lts}}$, $\mathcal{S}_2 = \langle S_2, s_{02}, Act_2, \rightarrow_2 \rangle \in \mathcal{L}_{\text{lts}}$, $a \in Act$, and $\alpha \in Act_\tau$.

1. The LTS *nil* $=\langle \{s_0\}, s_o, \{\tau\}, \emptyset \rangle$
2. The *action prefix* $\alpha; \mathcal{S}_1 = \langle S_1 \cup \{s'\}, s', Act_1 \cup \{\alpha\}, \rightarrow_1 \cup \{s' \xrightarrow{\alpha} s_{01}\} \rangle$, $s' \notin S_1$.
3. The *summation* $\mathcal{S}_1 + \mathcal{S}_2 = \langle S_1 \cup S_2 \cup \{s'\}, s', Act_1 \cup Act_2,$ $\rightarrow_1 \cup \rightarrow_2 \cup \{s' \xrightarrow{\alpha} s'' \mid s_{01} \xrightarrow{\alpha}_1 s'' \ \vee \ s_{02} \xrightarrow{\alpha}_2 s''\} \ \rangle, s' \notin S_1 \cup S_2$
4. The *parallel composition* $\mathcal{S}_1 \parallel \mathcal{S}_2 = \langle S_1 \times S_2, (s_{01}, s_{02}), Act_1 \cup Act_2, \rightarrow \rangle$ where the transition relation $\rightarrow$ is defined as:
$$\frac{s_1 \xrightarrow{\alpha} s'_1}{(s_1, s_2) \xrightarrow{\alpha} (s'_1, s_2)} \quad \frac{s_2 \xrightarrow{\alpha} s'_2}{(s_1, s_2) \xrightarrow{\alpha} (s_1, s'_2)} \quad \frac{s_1 \xrightarrow{a} s'_1 \qquad s_2 \xrightarrow{\bar{a}} s'_2}{(s_1, s_2) \xrightarrow{\tau} (s'_1, s'_2)}$$

□

An LTS is *deterministic* iff $s \stackrel{\tau}{\not\rightarrow}$ for any s, and whenever $s \xrightarrow{a} s'$ and $s \xrightarrow{a} s''$ then $s' = s''$. Thus, in a deterministic LTS, the execution of an action in a given state results in a unique successor state; in a non-deterministic LTS there may be several such successor states. Figure 2.6 illustrates the difference between deterministic and non-deterministic LTSs.

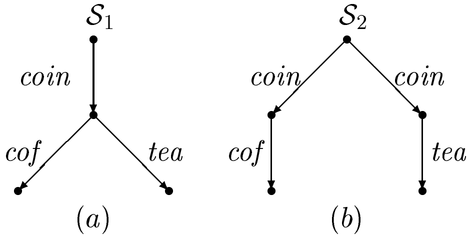


Figure 2.6. *A deterministic (a) and a non-deterministic (b) labeled transition system.*

2.1.3 Semantics of Communicating State Machines

Next we provide the definitions for a network of communicating state machines. The formal structure of a state machine over a set of variables V and actions $\mathcal{A}$ is given in Definition 2.7. The formal semantics given as an LTS follows in Definition 2.8.

Definition 2.7 *State machine*:

1. The guards $G(V)$ over a set of integer variables V is generated by the syntax $g ::= \gamma \mid g \wedge g$ where γ is a constraint of the form $v \sim c$ with $\sim \in \{\leq, <, =, >, \geq\}$ and c an integer constant, and $v \in V$.

2. $R(V)$ is the set of variable assignments of the following forms: $v := v'\ op\ c$; or $v := c$;, where $op \in \{+, -, *\}$, and $v, v' \in V$.

3. A state machine $\mathcal{M}$ over actions $\mathcal{A}_\tau$ and integer variables V is a tuple $\langle N, l_0, E, N_C \rangle$ where

   (a) N is a (finite) set of locations,

   (b) l_0 is the initial location,

   (c) $N_C \subseteq N$ the set of committed locations, and

   (d) $E \subseteq N \times G(V) \times \mathcal{A}_\tau \times 2^{R(V)} \times N$ is the set of edges where $G(X)$ is the set guards, and $R(X)$ is the set of assignments. We write $l \xrightarrow{g,a,r} l'$ if $\langle l, g, a, r, l' \rangle \in E$ to represent a transition from location l to location l' with guard g, action a, and set of assignments r.

   (e) Let $\overline{a}$ denote the complement of action $a \in \mathcal{A}$ such that $\overline{a!} = a?$ and $\overline{a?} = a!$.

□

A *network* of communicating state machines $\overline{\mathcal{M}} = (\mathcal{M}_1 \parallel \cdots \parallel \mathcal{M}_n)$ is a collection of machines $\mathcal{M}_1 \ldots \mathcal{M}_n$ operating concurrently. The actions are classified in two categories. The network synchronizes with the environment via a set of distinguished *observable actions* $\mathcal{O} \subseteq \mathcal{A}$, and can synchronize internally only via the *hidden actions* $\mathcal{A} - \mathcal{O}$. That is, no internal communication is permitted over observable actions. Because we offer no explicit restriction operator, this distinction allows a simple means of hiding the required actions from the environment.

Semantically, the state of a network is modeled by a *state configuration*: $\langle \bar{l}, \bar{v} \rangle$. The *location vector* $\bar{l}$ is a vector of locations that represent the joint control location of the network; l_i is the location of machine $\mathcal{M}_i$. We write $\bar{l}[l'_i/l_i]$ to denote the vector where the ith element of $\bar{l}$ has been replaced by l'_i. The second component $\bar{v}$ represents the current variable valuation. Let $\bar{v}(v)$ denote the value of variable v, $r(\bar{v})$ the valuation resulting by simultaneously updating $\bar{v}$ with the assignments in r, and finally, $g(\bar{v})$ the outcome of evaluating guard g in variable valuation $\bar{v}$. In the initial state $\langle \bar{l}_0, \bar{0} \rangle$ all machines are in their respective initial location, and all variables are zero. Let N_{C_i} be the set of committed locations for machine $\mathcal{M}_i$.

Definition 2.8 *Transition rules for $\mathcal{L}_{\mathrm{csm}}$:*

$$\frac{l_i \xrightarrow{g,a,r} l'_i \qquad g(\bar{v}) \qquad a \in \mathcal{O} \cup \{\tau\}}{\langle \bar{l}, \bar{v} \rangle \xrightarrow{a} \langle \bar{l}[l'_i/l_i], r(\bar{v}) \rangle}, \forall k \in [1,n].\ \text{if } l_k \in N_{C_k} \text{ then } i = k$$

$$\frac{l_i \xrightarrow{g_1,a,r_1} l'_i \qquad l_j \xrightarrow{g_2,\bar{a},r_2} l'_j \qquad (g_1 \wedge g_2)(\bar{v}) \quad i \neq j \quad a \in \mathcal{A} - \mathcal{O}}{\langle \bar{l}, \bar{v} \rangle \xrightarrow{\tau} \langle \bar{l}[l'_i/l_i, l'_j/l_j], (r_1 \cup r_2)(\bar{v}) \rangle},$$

$$\forall k \in [1,n].\ \text{if } l_k \in N_{C_k} \text{ then } k = i \vee k = j$$

□

The transition rules are defined in Definition 2.8. The first rule concerns observable and internal actions. The second rule defines internal synchronization between to automata. In both cases, the side condition ensures that when an automaton is in a committed location, a transition away from this location will be taken next.

2.2 Testing Theory for Non-deterministic Systems

In this section we review the theory underlying our testing approach. We adopt the classical testing methodology developed by de Nicola and Hennessy in [75, 49]. Their proposal states that a specification and an implementation are to be considered equivalent if no external observer can distinguish between them by performing "button" pressing experiments, i.e., if they pass exactly the same tests. Moreover, their work also characterizes the precise kind of tests needed to determine whether two systems are testing equivalent. This is obviously of great importance for automatic testing. These ideas and their application to conformance testing of communication protocols were further developed by Brinksma [21]. Our presentation of the testing theory and its definitions is derived from [75, 49].

2.2.1 Goals and Assumptions

Systematic and well founded test generation presumes a formalized implementation relation **implements** which defines when a system is a correct implementation of the specification. It also presumes a definition of tests, and a relation **passes** which defines when a system passes a test. The goal of (automated) testing is to construct a suite of tests Π from the specification $\mathcal{S}$ such that the correctness of the implementation $\mathcal{I}$ can be determined precisely from the results of executing these tests:

$$\mathcal{I}\ \textbf{implements}\ \mathcal{S} \quad \textit{iff} \quad \forall \mathcal{T} \in \Pi.\ \mathcal{I}\ \textbf{passes}\ \mathcal{T}$$

This requirement has two implications. The test suite should reject all incorrect implementations (*soundness*), and it should never reject correct implementations (*completeness*). These definitions of soundness and completeness, stated formally in Definition 2.9, are inspired by the work by Peleska and Siegel in [81] that is based on a proof theoretic view on testing, i.e, testing is viewed as a (very) special proof technique. In this setting, soundness means that no false conclusions about the relationship between implementations and specifications can be deduced. Completeness means that all true conclusions can be deduced.

Definition 2.9 *Soundness and completeness of test suites*:

1. Π is *sound iff* $\mathcal{I}\ \neg\textbf{implements}\ \mathcal{S}\quad \textit{implies}\quad \exists\mathcal{T} \in \Pi.\ \mathcal{I}\ \neg\textbf{passes}\ \mathcal{T}$
2. Π is *complete iff* $\mathcal{I}\ \textbf{implements}\ \mathcal{S}\quad \textit{implies}\quad \forall\mathcal{T} \in \Pi.\ \mathcal{I}\ \textbf{passes}\ \mathcal{T}$
3. Π is *exhaustive iff* Π is sound and complete.

□

Exhaustive testing is normally infeasible because a huge number of tests are required. One therefore risks concluding that an implementation is correct, when it is not. The passing of a test suite should consequently be interpreted more cautiously: No evidence of errors were found.

The goal of the remainder of this section is to formalize the **implements** and **passes** relations. However, these definitions hinges on a set of basic assumptions:

1. The specification is given as an LTS.
2. The implementation can be described by an LTS.
3. The observable actions of the implementation are known.
4. The implementation under test and its environment (and tester) use synchronous rendezvous style communication.
5. The specification and implementation has a finite number of states.
6. The specification converges strongly, i.e., it has no infinite sequences of τ actions.

We argue for each in turn. 1) The semantics of most specification languages can be expressed as LTSs. It therefore suffices to develop the theory for this basic language. 2) To formulate the implementation relation as a mathematical relation between two formal objects it is necessary to assume that the implementation can be described as an LTS. It is important to note that we only assume existence of such an LTS model, not explicit knowledge of one, i.e., the implementation remains unknown. This assumption is refered to as the test hypothesis [111]. 3) Knowing the observable actions of the implementation is necessary to claim or prove algorithms to be sound and complete, even in theory. 4) The necessary theory is well established for synchronously communicating systems. Further, other styles can be modeled with reasonable effort. 5) Infinite state systems causes algorithmic problems that are best disregarded at first. 6) The convergence assumption is not strictly necessary as the presented theory can be extended to also handle divergence, but absence thereof eases the technical presentation.

2.2.2 Tests and Test Execution

Tests can be modeled by using an extended type of LTSs where each state has been assigned a **pass** or a **fail** verdict. Intuitively, if the execution of a test stops in a state with a **fail** verdict, the implementation will be declared to fail that test, and is consequently considered incorrect.

Definition 2.10 *Test LTS ($\mathcal{L}_{\text{tlts}}$):*
A test $\mathcal{T}$ is an extended LTS $\langle S, s_0, Act_\tau, \rightarrow, \mathcal{V}\rangle$, where S is the set of states, s_0 the initial state, Act_τ the set of actions, $\rightarrow$ the transition relation—all as previously defined in Definition 2.3, and $\mathcal{V}$ is the verdict assignment function: $\mathcal{V} : S \mapsto \{\mathbf{pass}, \mathbf{fail}\}$.

□

In situations where the test has a specific purpose, it is common to introduce a third test verdict **inconc** to indicate that the test did not fail, but also did not reveal the desired behavior. This distinction is necessary because non-determinism may cause the implementation to take an execution path different from the one required to observe the test purpose.

The execution of a test against the implementation can be modeled by putting the tester in parallel with the implementation in place of its environment, and let the composed system evolve via synchronous communication as defined for LTSs in Definition 2.5 (4). The success of a test can now be defined as the composed system's ability to drive the tester to one of the designated success states.

Non-determinism, either internally in the environment or the implementation, or in the resolution of their synchronization actions, makes several different computations from the same start configuration possible. Thus, the same test may be both successful and unsuccessful in different computations, see Definition 2.11.

Definition 2.11 *Computation*:
A configuration of a system $\mathcal{T} \parallel \mathcal{I}$ composed of a test $\mathcal{T}$ and implementation $\mathcal{I}$ is a pair of states $(s_\mathrm{t}, s_\mathrm{i})$ such that s_t is a state in the test, and s_i a state in the implementation.

1. A computation is a maximal sequence of test configurations: $(s_{\mathrm{t}0}, s_{\mathrm{i}0}) \xrightarrow{\mathcal{T}} (s_{\mathrm{t}1}, s_{\mathrm{i}1}) \xrightarrow{\mathcal{T}} (s_{\mathrm{t}2}, s_{\mathrm{i}2}) \xrightarrow{\mathcal{T}} \cdots (s_{\mathrm{t}k}, s_{\mathrm{i}k}) \xrightarrow{\mathcal{T}} \cdots$.

2. A configuration is *deadlocked* if it cannot be extended, i.e., there is some n with $(s_{\mathrm{t}n}, s_{\mathrm{i}n}) \overset{\mathcal{T}}{\not\to}$.

3. A sequence of configurations is *maximal* if it is infinite, or there is some deadlocked terminal configuration.

4. A computation is *successful* if there exists some $s_{\mathrm{t}k}$ such that $k \leq n$ and $\mathcal{V}(s_{\mathrm{t}k}) = \mathbf{pass}$.

5. Let $Comp(\mathcal{T} \parallel \mathcal{I})$ denote the set of possible computations from the initial configuration $(s_{\mathrm{t}0}, s_{\mathrm{i}0})$.

□

This observation turns out to be important in both theory and in practice. It affects our definition of the implementation relation (whether all or only one computation should report success). Our observational power in the practical execution of a test is reduced because a single execution only reveals one computation. The implications of this will be further discussed in Section 6.1.1.

2.2.3 Implementation Relations

The relation between a specification and the implementation is either an equivalence or a preorder. An *equivalence* requires that the implementation has exactly the behavior required by the specification. In many cases an equivalence is more than what is desired, and the comparison should be based on a preorder instead. A *preorder* intuitively requires that the implementation has at least the behavior required by the specification.

The basic idea in Hennessy's testing theory is to compare systems based on the tests they pass: Two systems are regarded as equivalent if no experimenter can distinguish them by executing tests. Hennessy's testing equivalence thus requires that the specification and implementation exactly passes the same tests. Similarly, the testing preorder requires that the implementation passes at least the same tests as the specification.

Because some computations of a test may yield success and some may not, there are two possible definitions of passing a test. One is that an implementation passes a test if the possible executions *may* report success, i.e., only one of the possible computations is required to report success. The other choice is that an implementation passes a test if the possible executions *must* report success, i.e., all computations are required to report success. The may, must, and testing preorder is stated formally in Definition 2.12.

Definition 2.12 *The Testing Preorder* $\sqsubseteq_{\mathbf{te}}$:

1. $\mathcal{S}$ **must** $\mathcal{T}$ *iff* $\forall \Sigma \in \mathit{Comp}(\mathcal{T} \parallel \mathcal{S})$. Σ is successful.
2. $\mathcal{S}$ **may** $\mathcal{T}$ *iff* $\exists \Sigma \in \mathit{Comp}(\mathcal{T} \parallel \mathcal{S})$. Σ is successful.
3. $\mathcal{S} \sqsubseteq_{\mathbf{must}} \mathcal{I}$ *iff* $\forall \mathcal{T} \in \mathcal{L}_{\mathrm{tlts}}$. $\mathcal{S}$ **must** $\mathcal{T}$ *implies* $\mathcal{I}$ **must** $\mathcal{T}$
4. $\mathcal{S} \sqsubseteq_{\mathbf{may}} \mathcal{I}$ *iff* $\forall \mathcal{T} \in \mathcal{L}_{\mathrm{tlts}}$. $\mathcal{S}$ **may** $\mathcal{T}$ *implies* $\mathcal{I}$ **may** $\mathcal{T}$
5. $\mathcal{S} \sqsubseteq_{\mathbf{te}} \mathcal{I}$ *iff* $\mathcal{S} \sqsubseteq_{\mathbf{must}} \mathcal{I}$ and $\mathcal{S} \sqsubseteq_{\mathbf{may}} \mathcal{I}$

□

Testing equivalence can be obtained as $\mathcal{S} =_{\mathbf{te}} \mathcal{I}$ *iff* $\mathcal{S} \sqsubseteq_{\mathbf{te}} \mathcal{I}$ and $\mathcal{I} \sqsubseteq_{\mathbf{te}} \mathcal{S}$.

Unfortunately, the above definition of the testing preorder $\sqsubseteq_{\mathbf{te}}$ does not provide a practical means of testing an implementation: One must consider all possible tests that can be formulated as test LTSs. To enable direct comparison and automatic generation of tests, an alternative characterization is required. It has been proven that it suffices to consider properties in the two small logical languages $\mathcal{L}_{\mathbf{must}}$ and $\mathcal{L}_{\mathbf{may}}$ given in Definition 2.13.

Definition 2.13 *May and Must Properties*:

1. $\mathcal{L}_{\mathbf{must}} =_{\mathrm{def}} \{\textbf{after } \sigma \textbf{ must } A \mid \sigma \in \mathit{Act}^*, A \subseteq \mathit{Act}\}$
2. $\mathcal{S} \models \textbf{after } \sigma \textbf{ must } A$ *iff* $\forall s \in \mathcal{S}$ **after** σ. $\exists a \in A$. $s \overset{a}{\Rightarrow}$
3. $\mathcal{L}_{\mathbf{may}} =_{\mathrm{def}} \{\textbf{can } \sigma \mid \sigma \in \mathit{Act}^*\}$
4. $\mathcal{S} \models \textbf{can } \sigma$ *iff* $\sigma \in \mathit{Tr}(\mathcal{S})$

□

A *must property* requires that if the system can perform the trace σ, then no matter what state it has entered thereafter, it can also engage in one of the actions in A. Consequently, the system is not permitted to refuse synchronization with all of the actions in A. A system not satisfying the property could possibly deadlock when offered synchronization only with the actions in A. We refer to the set of actions A as a *must set*. $\mathcal{L}_{\mathbf{must}}$ thus characterizes systems based on their deadlock properties. Observe the special case **after** σ **must** $\emptyset$ that can only be satisfied by a system, if its set of reachable states after σ is empty, and in consequence, if the system cannot perform the trace σ. A *may property* requires that the system is possibly able to execute the trace σ. In non-deterministic systems, it may not always be possible to execute the entire trace.

An implementation can be rendered incorrect by finding a may or must property satisfied by the specification, but not by the implementation, as captured by Proposition 2.14.

Proposition 2.14 *Alternative Characterization [75]:*

1. $\mathcal{S} \sqsubseteq_{\mathbf{must}} \mathcal{I}$ *iff* $\forall \sigma \in Act^*, \forall A \subseteq Act.$
   $\mathcal{S} \models \mathbf{after}\ \sigma\ \mathbf{must}\ A$ *implies* $\mathcal{I} \models \mathbf{after}\ \sigma\ \mathbf{must}\ A$

2. $\mathcal{S} \sqsubseteq_{\mathbf{may}} \mathcal{I}$ *iff* $\forall \sigma \in Act^*.\ \mathcal{S} \models \mathbf{can}\ \sigma$ *implies* $\mathcal{I} \models \mathbf{can}\ \sigma$
   iff $Tr(\mathcal{S}) \subseteq Tr(\mathcal{I})$

proof: See [75]. □

It is, as will be shown in Section 2.2.5, possible to transform a must (or may) property t into a test $\mathcal{T}(t)$ such that an LTS **must** (or **may**) pass that test iff it satisfies the property:

$$\mathcal{S}\ \mathbf{must}\ \mathcal{T}(t) \quad \textit{iff} \quad \mathcal{S}\ \models\ t,\ t \in \mathcal{L}_{\mathbf{must}}, \text{ and}$$

$$\mathcal{S}\ \mathbf{may}\ \mathcal{T}(t) \quad \textit{iff} \quad \mathcal{S}\ \models\ t,\ t \in \mathcal{L}_{\mathbf{may}}$$

The main practical implication of the alternative characterization is that, in order to decide the testing preorder, it is only necessary to use and generate certain fixed structured testers, namely those corresponding to the logical properties. This transformation and the structure of tests are given in Section 2.2.5.

A further relation called **conf** (Definition 2.15) proposed by Brinksma [21] is commonly used in the field of protocol conformance testing. It is similar to the must preorder except that it only considers must properties with traces in the specification.

Definition 2.15 *Conformance*:

1. $\mathcal{I}$ **conf** $\mathcal{S}$ *iff* $\forall \sigma \in Tr(\mathcal{S}), \forall A \subseteq Act.$
   $\mathcal{S} \models \textbf{after } \sigma \textbf{ must } A$ *implies* $\mathcal{I} \models \textbf{after } \sigma \textbf{ must } A$

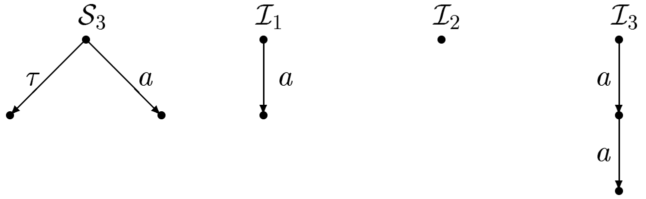
□

We compare and discuss the merits and the strengths and weaknesses of the different preorders in the following Section 2.2.4.

2.2.4 Interpretation of Implementation Relations

The preceeding section introduced four choices of preorders for comparison of specifications and implementations: The must preorder $\sqsubseteq_{\mathbf{must}}$, the may preorder $\sqsubseteq_{\mathbf{may}}$, the testing preorder $\sqsubseteq_{\mathbf{te}}$, and the conformance preorder **conf**. This means that for the same specification different implementations could be accepted depending on the actual choice of implementation relation. This choice is not only of academic or theoretical interest, but has severe practical implications. It is therefore important to understand the differences between these relations and their relative merits. The following exemplifies and characterizes the preorders.

Figure 2.16 gives a speficication and a set of possible implementations. $\mathcal{S}_3$ can optionally do an a. I_1, that necessarily performs an a, is accepted by all four preorders. Conversely, $\mathcal{S}_3$ would be a non-conforming implementation of the specification $\mathcal{I}_1$ because a is necessary in $\mathcal{I}_1$ but only optional in $\mathcal{S}_3$.



(a) $\mathcal{S}_3 \sqsubseteq_{\mathbf{must}} \mathcal{I}_1$

(b) $\mathcal{S}_3 \sqsubseteq_{\mathbf{may}} \mathcal{I}_1$

(c) $\mathcal{I}_3$ **conf** $\mathcal{S}_1$

(d) $\mathcal{S}_3$ **c$\not{o}$nf** $\mathcal{I}_1$

(e) $\mathcal{I}_2$ **conf** $\mathcal{S}_3$

(f) $\mathcal{S}_3 \sqsubseteq_{\mathbf{must}} \mathcal{I}_2$

(g) $\mathcal{S}_3 \not\sqsubseteq_{\mathbf{may}} \mathcal{I}_2$

(h) $\mathcal{I}_3$ **conf** $\mathcal{I}_1$

(i) $\mathcal{I}_1 \not\sqsubseteq_{\mathbf{must}} \mathcal{I}_3$

(j) $\mathcal{I}_1 \sqsubseteq_{\mathbf{may}} \mathcal{I}_3$

Figure 2.16. *Comparison of preorders.*

Implementation $\mathcal{I}_2$ is a deadlocked implementation, which can do absolutely nothing. Because a is only optional, $\mathcal{I}_2$ is accepted by the conformance and must preorder. However, it is not accepted by the may preorder, because the a possible in the specification should also be possible in the implementation; therefore $\mathcal{S}_3 \not\sqsubseteq_{\mathbf{may}} \mathcal{I}_2$ and $\mathcal{S}_3 \not\sqsubseteq_{\mathbf{te}} \mathcal{I}_2$.

$\mathcal{I}_1$ can be interpreted as a specification requiring an a. The implementation $\mathcal{I}_3$ does two consecutive a's. Thus, we can say that the implementation has an extended functionality or behavior compared to the specification. Such extended behavior is permitted by **conf** and $\sqsubseteq_{\mathbf{may}}$, but not by the must preorder:

$$\mathcal{I}_1 \models \mathbf{after}\ a \cdot a\ \mathbf{must}\ \emptyset, \text{but}\ \mathcal{I}_3 \not\models \mathbf{after}\ a \cdot a\ \mathbf{must}\ \emptyset$$

The may preorder ensures that the behavior in the specification is also in the implementation. However, it also allows the implementation to have unspecified deadlocks, and is therefore rarely used alone when testing non-deterministic systems.

The **conf** relation permits such extended functionality, but the must preorder does not. One would think that extended behavior is positive, and should always be permitted. Therefore, **conf** should suffice. **conf** also has the advantage over the must preorder that only the traces in the specifications must be considered, and consequently, fewer tests need to be executed. However, the next series of examples demonstrates that acceptance of extended functionality depends on the application.

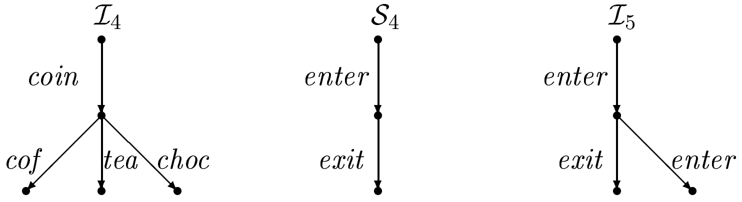


Figure 2.17. *The pros and cons of extended behavior.*

First compare the vending machine specifications $\mathcal{S}_1$ and $\mathcal{S}_2$ previously given in Figure 2.6. The specification $\mathcal{S}_1$ requires that an implementation after accepting a coin enables the environment to choose between tea and coffee. $\mathcal{S}_2$ would be a faulty implementation using either the conformance or the must preorder, because it chooses internally whether to enable tea or coffee. $\mathcal{S}_1$ and $\mathcal{S}_2$ are distinguishable by the property **after** *coin* **must** $\{cof\}$ that $\mathcal{S}_1$ satisfies, but $\mathcal{S}_2$ does not.

$\mathcal{I}_4$ from Figure 2.17 is a deterministic alternative that does not have the same problem as $\mathcal{S}_2$. However, it is able to sell hot chocolate in addition to tea and coffee. If the goal is to ensure that we can buy coffee or tea, thus using the conformance relation, we can safely accept $\mathcal{I}_4$ as an implementation. However, if the vending machine offered alcoholic drinks instead of hot chocolate or splashed coffee on the floor when no coins and cups have been inserted, we might be more reluctant to accept it. The must preorder does not permit such added behavior, as demonstrated by the property **after** $coin \cdot choc$ **must** $\emptyset$ that $\mathcal{S}_1$ satisfies, but not $\mathcal{I}_4$.

For some applications it may even be disastrous to accept extended behavior. The mutual exclusion property specified by $\mathcal{S}_4$ requires that the critical section is entered and exited in strict sequence. $\mathcal{I}_5$ is a faulty implementation because it permits two processes to enter the critical section simultaneously. $\mathcal{I}_5$ **conf** $\mathcal{S}_4$, but $\mathcal{S}_4 \not\sqsubseteq_{\mathbf{must}} \mathcal{I}_5$.

It is useful to observe that the testing preorder can be obtained by combining **conf** with two trace inclusions, as captured by Proposition 2.18.

Proposition 2.18 *Decomposition of the Testing Preorder*:

1. $\mathcal{S} \sqsubseteq_{\mathbf{must}} \mathcal{I} \quad \textit{implies} \quad Tr(\mathcal{I}) \subseteq Tr(\mathcal{S})$
2. $\mathcal{S} \sqsubseteq_{\mathbf{may}} \mathcal{I} \quad \textit{implies} \quad Tr(\mathcal{S}) \subseteq Tr(\mathcal{I})$
3. $\mathcal{S} \sqsubseteq_{\mathbf{te}} \mathcal{I} \quad \textit{iff} \quad \mathcal{I}\ \mathbf{conf}\ \mathcal{S} \quad \wedge \quad Tr(\mathcal{I}) \subseteq Tr(\mathcal{S}) \quad \wedge \quad Tr(\mathcal{S}) \subseteq Tr(\mathcal{I})$

Proof:
1 and 2 is shown in [75], and 3 is shown in [111]. □

Following Peleska [79] each component of the testing preorder contributes with different aspects of system correctness:

- Checking that the implementation does not have extra unspecified behavior ($Tr(\mathcal{I}) \subseteq Tr(\mathcal{S})$) corresponds to checking the *safety* of the implementation.

- Checking that the implementation has all the behavior of the specification ($Tr(\mathcal{S}) \subseteq Tr(\mathcal{I})$) corresponds to checking the *robustness* of the implementation. The term robustness is used here because the implementation responds the way required by the specification to rare and unexpected environment behaviors. In alternative wording, this aspect checks the *liveness* of the implementation with respect to the specification.

- Checking conformance ($\mathcal{I}$ **conf** $\mathcal{S}$) corresponds to checking that all explicit requirements of the specification are satisfied, i.e., *requirements coverage*. The mandatory behavior of the specification is thus implemented.

Hence, if all these aspects must be checked, the testing preorder should be used.

2.2.5 Test Languages

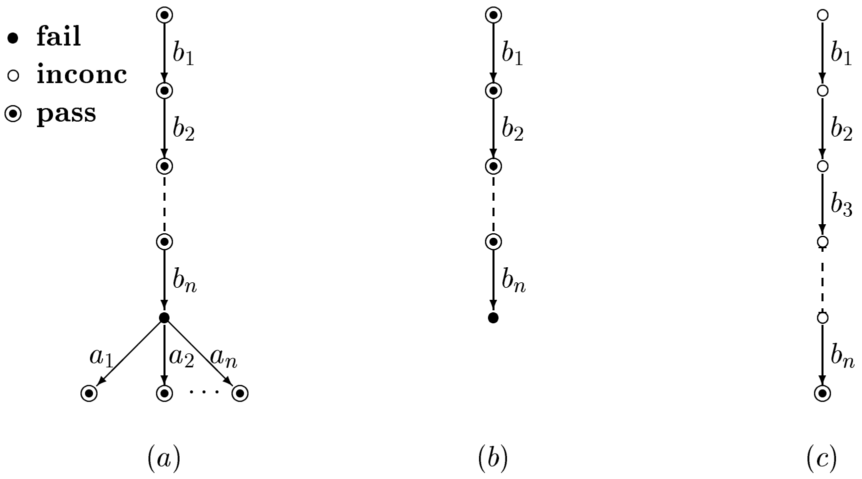
The translation of may and must properties to test LTSs is relatively straightforward. However, for technical reasons a slight redefinition of successful computations will be necessary to permit deterministic tests with state based verdicts. In Hennessy's presentation [75, 49] of the theory, testers are nondeterministic and may spontaneously abort test execution and flag success by executing a special success action ω. However, in a practical setting this is disadvantageous—it is preferable to attempt as much communication as possible, and therefore test as much behavior as possible, before flagging success or failure, and thus priority should be given to further synchronizations.

> A (finite) computation $(s_{\mathrm{t}0}, s_{\mathrm{i}0}) \xrightarrow{\tau} (s_{\mathrm{t}1}, s_{\mathrm{i}1}) \xrightarrow{\tau} \cdots \xrightarrow{\tau} (s_{\mathrm{t}n}, s_{\mathrm{i}n})$ is *successful iff* $\mathcal{V}(s_{\mathrm{t}n}) = \mathbf{pass}$.

Similarly, a test *fails* if its execution deadlocks in a state with a fail verdict. The change should also be seen in the light that testing in practice involves observation of only *finite* length synchronization sequences—one cannot wait indefinitely for the outcome.

The main idea in the translation of a must property **after** σ **must** A is illustrated in Figure 2.19a. First, the tester tries to perform the trace $b_1 \cdot \ldots \cdot b_n$ and then to offer the actions in $\{a_1, \ldots, a_n\}$. If the system refuses these actions, the tester remains deadlocked in a fail state. For clarity, Figure 2.19b shows the tester for the special case where $A = \emptyset$. Satisfaction of this property implies that the system cannot perform the trace $b_1 \cdot \ldots \cdot b_n$. The tester therefore enters a fail state after having performed the last action b_n in the trace.

The translation of a may property consist of a sequence of actions with all states labeled with fail (or inconclusive) verdicts except the last successful state: To satisfy the may property, there must exist at least one computation that is able to drive the tester to its terminal state, thus executing all actions in the trace. Provided that the purpose of this test is to show the existence of the trace, it is more natural to use inconclusive verdicts along the trace. This is shown in Figure 2.19c.



(a) (b) (c)

Figure 2.19. *The Hennessy testers with state based verdicts.*

The translations are given in Definition 2.20. Some examples are depicted in Figure 2.21.

Definition 2.20 *Translation of test properties to test LTSs*:

1. Let $t =$ **after** $b_1 \cdot b_2 \ldots b_n$ **must** $A \in \mathcal{L}_{\mathbf{must}}$ (Figure 2.19a and 2.19b).

   (a) $\mathcal{T}(t) = \langle \{s_1 \ldots s_{n+1}\} \cup \{s_a \mid a \in A\}, s_1, \rightarrow, \mathcal{V} \rangle$.

   (b) $\mathcal{V}(s_i) = \mathcal{V}(s_a) =$ **pass** if $1 \leq i \leq n$. $\mathcal{V}(s_{n+1}) =$ **fail**.

   (c) The transitions $\rightarrow$ are defined as:

   $$\forall i \in 1 \ldots n.\ s_i \xrightarrow{b_i} s_{i+1}.$$
   $$\forall a \in A.\ s_{n+1} \xrightarrow{a} s_a$$

2. Let $t =$ **can** $b_1 \cdot b_2 \ldots b_n \in \mathcal{L}_{\mathbf{may}}$ (Figure 2.19c.)

   (a) $\mathcal{T}(t) = \langle \{s_1 \ldots s_{n+1}\}, s_1, \rightarrow, \mathcal{V} \rangle$.

   (b) $\mathcal{V}(s_i) =$ **inconc** if $1 \leq i \leq n$. $\mathcal{V}(s_{n+1}) =$ **pass**.

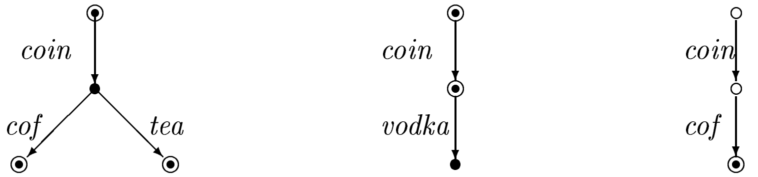
   (c) The transitions $\rightarrow$ are defined as:

   $$\forall i \in 1 \ldots n.\ s_i \xrightarrow{b_i} s_{i+1}.$$

□

We finally remark that several of these basic tests can be composed into a single more complex test that tests for several properties at once. Consider a must test **after** σ **must** A. There is normally no reason to stop test execution when one of the actions a in A has been executed. It is better to continue execution with a test that exploits the fact that the implementation has already executed the trace $\sigma \cdot a$. Figure 2.22 illustrates a composed test.

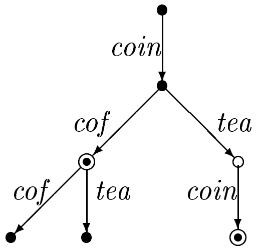
In practice, this composition is extremely important because it reduces the number of tests to be executed. Also, it reduces the number of re-executions of the same test, because the trace most recently executed is reused when a test requiring that trace as prefix follows immediately. The required prefix trace may not be accepted by a non-deterministic implementation after it has been reset.



after *coin* **must**$\{cof, tea\}$ **after** *coin* $\cdot$ *vodka* **must** $\emptyset$ **can** *coin* $\cdot$ *cof*

Figure 2.21. *Example tests.*

Not all properties can be composed because they require incompatible verdicts in intersecting states. We shall not give the exact composition rules here, but note that when composition is possible, the verdict assignment to intersecting states must be done carefully to ensure that success of the composed test implies success of all properties visited along the path to the success state.



(a)

after ϵ **must**$\{coin\}$
after *coin* **must**$\{cof, tea\}$
after *coin* $\cdot$ *cof* $\cdot$ *cof* **must** $\emptyset$
after *coin* $\cdot$ *cof* $\cdot$ *tea* **must** $\emptyset$
can *coin* $\cdot$ *tea* $\cdot$ *coin*

(b)

Figure 2.22. *Composition of tests. A composite test (a), and the tested properties (b).*

2.3 Test Generation

Given the formalized testing setup presented in the preceding sections, the next question is how to automate test generation. This includes identifying the tests to be generated and developing data structures and algorithms for their actual construction. Tests can be generated either by interpreting the transition rules of the specification directly, or by transforming the specification to a data structure better suited for test generation. We describe the direct approach in Section 2.3.2 and the proposed success graph data structure thereafter in Section 2.3.3.

2.3.1 Relevant Hennessy Testers

Although the class of Hennessy testers fixed the structure of the necessary tests, they are still infinitely many. To make the setup operational, it is necessary to narrow the class of tests further. Only tests that contributes with new knowledge about the implementation relation should be considered for execution. A first implication of this requirement is that only tests that the specification itself passes should be executed; only then must the implementation also pass them. A second implication is that it is futile to execute tests that are doomed to fail because their failure are implied by other tests. These observations results in the reduced, but still sufficient, class of *relevant* Hennessy testers in Definition 2.23.

Definition 2.23 *The relevant Hennessy testers*:

1. If $\sigma \in Tr(\mathcal{S})$ and $\mathcal{S} \models \textbf{after}\ \sigma\ \textbf{must}\ A$ then $\mathcal{T}(\textbf{after}\ \sigma\ \textbf{must}\ A)$ is a relevant must test.
2. If $\sigma \in Tr(\mathcal{S})$ and $\sigma \cdot a \notin Tr(\mathcal{S})$ then $\mathcal{T}(\textbf{after}\ \sigma \cdot a\ \textbf{must}\ \emptyset)$ is a relevant must test.
3. If $\sigma \in Tr(\mathcal{S})$ then $\mathcal{T}(\textbf{can}\ \sigma)$ is a relevant may test.

□

2.3.2 A Direct Test Generation Algorithm

A starting point for developing algorithms for test generation is to inspect the steps needed to find a may or must property of a specification. To construct must properties it will frequently be convenient with the additional notation summarized in Definition 2.24 for operating on sets of states.

Definition 2.24 *Operations on sets of states*:
Let $\mathcal{S} = \langle S, s_0, Act, \rightarrow\rangle \in \mathcal{L}_{\text{lts}}$ and $B \subseteq S$.

1. B **after** $\sigma =_{\text{def}} \bigcup_{s \in B} s$ **after** σ
2. $\text{Sort}(B) =_{\text{def}} \bigcup_{s \in B} \text{sort}(s)$
3. B **must** A *iff* $\forall s \in B.\ \exists a \in A.\ s \overset{a}{\Rightarrow}$
4. $St(B) =_{\text{def}} \{s \in B \mid s \overset{\tau}{\not\rightarrow}\}$

□

The following steps compute a must property **after** σ **must** A satisfied by a specification:

1. find a trace $\sigma \in Tr(\mathcal{S})$
2. compute the possible states after σ: $B = \mathcal{S}$ **after** σ
3. find a set of actions A such that B **must** A

Similarly, **can** $\sigma \cdot a$ properties can be found by concatenating σ with an action a in the sort B of the states reachable after σ. **after** $\sigma \cdot a$ **must** $\emptyset$ properties can be found by choosing an action a not in the sort after σ. Thus, tests can be found by inspecting the actions that *must* be accepted in the reachable states B after a given trace, the set of actions that *can* be accepted, and the actions that must be *refused*, as captured by Definition 2.25.

Definition 2.25 *Must, can, and refusal sets*:

1. $\text{Must}(B) =_{\text{def}} \{A \subseteq Act \mid B \text{ **must** } A\}$
2. $\text{Can}(B) =_{\text{def}} \text{Sort}(B)$
3. $\text{Ref}(B) =_{\text{def}} \{a \in Act \mid \forall s \in B.\ s \overset{a}{\not\Rightarrow}\} = Act - \text{Sort}(B)$

□

It is possible to reduce the number of must sets to be executed after a given trace. One reduction is to only choose the *minimal* must sets, see Definition 2.26. The result is a significant reduction of number of generated tests. The reduction does not weaken the testing strength of the test suite since it follows from the definition of satisfaction of a must property that a 'smaller' must property logically implies satisfaction of a 'larger' one:

$$\mathcal{S} \models \textbf{after } \sigma \textbf{ must } A \ \wedge\ A \subseteq A' \ \textit{implies}\ \mathcal{S} \models \textbf{after } \sigma \textbf{ must } A'$$

Definition 2.26 *Minimal must sets*:

Let M be a set of must sets.

1. A is a minimal element in M *iff* $\neg\exists A' \in M . A' \subset A$.
2. $min_{\subseteq}(M) = \{A \in M \mid A \text{ is minimal}\}$

□

A second reduction is possible by noting that the actions that the specification *must* do, it also *can* do. Therefore, if an action is *observed* as part of a must set, there is no need to also observe it using a separate may test, and consequently, the Can(B) actions can be reduced by removing the actions observed in a Must(B) test. It should be noted that it is insufficient to merely ensure that an action is included in a must test; the test execution system must ensure that it is observed during execution.

So far we have only considered how to produce one property. Our testing theory requires that all relevant properties are contained in some generated test. Whether or not one decides to execute all tests is a matter of test selection which logically is the next phase of test generation. An algorithm that constructs and composes all relevant and reduced tests is outlined in Algorithm 2.27.

Algorithm 2.27 *Generation of a test*:
Let $\mathcal{S} \in \mathcal{L}_{\text{lts}}$, $B \subseteq S$ be a subset of states, and $\mathcal{T} \in \mathcal{L}_{\text{tlts}}$.
TestGen($\mathcal{S}$) $=_{\text{def}}$ TestGen($\mathcal{S}$ **after** ϵ)
TestGen(B) $=_{\text{def}}$

1. Compute $M = min_{\subseteq}(\text{Must}(B))$, $C = \text{Sort}(B) - \bigcup_{A \in M} A$, and $R = Act - \text{Sort}(B)$
2. Construct one of
   (a) $\mathcal{T}_A = \sum_{a \in A} a; \mathcal{T}_a, \quad A \in M, \quad \mathcal{V}(\mathcal{T}_A) = \textbf{fail}$
   (b) $\mathcal{T}_A = \sum_{a \in A} a; \mathcal{T}_a, \quad A = C, \mathcal{V}(\mathcal{T}_A) = \textbf{inconc}$
   (c) $\mathcal{T}_A = \sum_{a \in A} a; nil, \quad A = R, \mathcal{V}(\mathcal{T}_A) = \textbf{pass}, \mathcal{V}(\mathcal{T}_A \textbf{ after } a) = \textbf{fail}$
   (d) $\mathcal{T}_A = nil, \quad \mathcal{V}(\mathcal{T}_A) = \textbf{pass}, \quad \textit{if } \text{Sort}(B) = \emptyset$
3. Construct a $\mathcal{T}_a$ in case (2a) or (2b) by calling TestGen(B **after** a)

The algorithm recursively constructs a *tree* structured test that tests for several properties. If the implementation accepts one of the must actions offered by the test, execution continues with a sub test for the possible states reached after synchronization with the action. The algorithm composes a branch (summation of actions) at the current level with tests constructed from the set of states reachable after each of the actions in the summation. The verdict of the branch is assigned appropriately depending on the property tested.

Case 2a generates a branch in the test corresponding to the actions in a minimized *must* set. One of these actions must be accepted. Otherwise, the **fail** verdict must be given. We adopt the notation $\mathcal{V}(\mathcal{T})$ to assign the verdict to the initial state in $\mathcal{T}$.

To ensure that all actions after a given trace can be observed, and to ensure that all traces in the specification are covered, case 2b generates a test branch containing the actions that are *enabled* in the current state but not contained in a minimal must set. Because it may not be possible to observe these actions in all executions, a **inconc** verdict is given.

Case 2c generates a branch containing actions *not* enabled in the current state, and that must be refused by the implementation. The verdict is thus **pass** if it deadlocks in the state before the offered action, and **fail** thereafter.

Case 2d covers the situation where no further actions are possible. This ensures that (possible) traces ends in a success state. This rule can also be applied to terminate test generation when a sufficiently long test has been generated.

Compared to all possible may and must properties satisfied by the specification the algorithm only generates a subset of these. The first reduction, implied by our definition of relevant tests, is that absence of extended behavior is checked by refusing the disabled action in the specification after a trace *in* the specification, and not after all traces. By assuming synchronous communication, and by observing that the behavior is not extended by one step, it can be concluded that the implementation has no extended behavior. Hence this reduction is safe. The second reduction is the minimization of must sets that discards logically implied properties. However, the reduction may also discard some actions that the algorithm instead observes as part of the may properties. The third and final reduction is that actions used in a minimized must property is not included in the can actions. For this reduction to be safe, as previously stated, it is necessary to require that every action in a must set will be observed during possibly repeated test execution.

Exhaustive test generation can be achieved by using all possible choices in steps 2 and 3, and by ensuring that every edge of each test is executed. The latter requirement ensures that every property is executed.

A problem with the algorithm is its lack of termination criterion. Obviously the algorithm can be terminated when a specific trace length is reached. But this is unsatisfactory because it neither produces an exhaustive test suite nor relates to a good coverage criterion of the specification.

An interesting question is therefore whether the trace length of the testers needs to be infinite in order to decide the implementation relation, or whether an upper bound exists. Under some restrictive assumptions such a bound can in fact be established. See the discussion about checking experiments in Section 6.1.3. From a theoretical perspective, this bound is the ideal termination criterion. From a practical perspective, one might be concerned about the assumptions this result requires, and whether the bound is so high for large specifications that test selection becomes necessary.

2.3.3 Success Graphs

An alternative data structure that provides further insights in the practice and theory of generating tests is the *success graph* defined in Definition 2.28. Once constructed, it has several advantages over the direct algorithm. These are summarized in Section 2.5.

Definition 2.28 *Success Graph*:
Let $\mathcal{S} = \langle S, s_0, Act_\tau, \rightarrow \rangle$. The success graph $SG(\mathcal{S})$ for $\mathcal{S}$ is an extended LTS $SG(\mathcal{S}) = \mathcal{S}_d = \langle S_d, s_{0d}, Act, \rightarrow_d, M, C, R \rangle$ satisfying:

1. $\mathcal{S}_d$ is deterministic.
2. $Tr(\mathcal{S}) = Tr(\mathcal{S}_d)$. That is, $\mathcal{S}$ and $\mathcal{S}_d$ are trace equivalent.
3. (a) $M(\mathcal{S}_d \textbf{ after } \sigma) = min_\subseteq(\text{Must}(\mathcal{S} \textbf{ after } \sigma))$ is called the success set for state $\mathcal{S}_d$ **after** σ, and contains the minimized set of must sets (see Definition 2.26) holding after σ.

   (b) $C(\mathcal{S}_d \textbf{ after } \sigma) = \text{Sort}(\mathcal{S}_d \textbf{ after } \sigma) - \bigcup_{A \in M(\mathcal{S}_d \textbf{ after } \sigma)} A$, is the reduced enabled actions after σ, and

   (c) $R(\mathcal{S}_d \textbf{ after } \sigma) = Act - \text{Sort}(\mathcal{S}_d \textbf{ after } \sigma)$ is the actions that must be refused after σ.

□

We use the term *success graph* because a communication attempt with a set of actions from M must be successful; specifically when $A \in M(s_{0d} \textbf{ after } \sigma)$ then no matter how the system $\mathcal{S}$ performs the trace σ a synchronization with some action in A is possible. To avoid ambiguity we shall refer to states in the specification LTS as 'states' and to states in the success graph as *branches*. A branch represents a branching point where a test would branch out.

Success graphs are similar to Hennessy's *acceptance graphs*, except that success graphs label branches with must sets, whereas acceptance graphs label them with acceptance sets [32]. An acceptance set is the set of actions possible from a stable state, i.e., the acceptance sets of $\mathcal{S}$ **after** σ equals $\{\text{sort}(s') \mid s' \in \mathcal{S}\ \textbf{after}\ \sigma \ \wedge \ s' \overset{\tau}{\not\rightarrow}\}$.

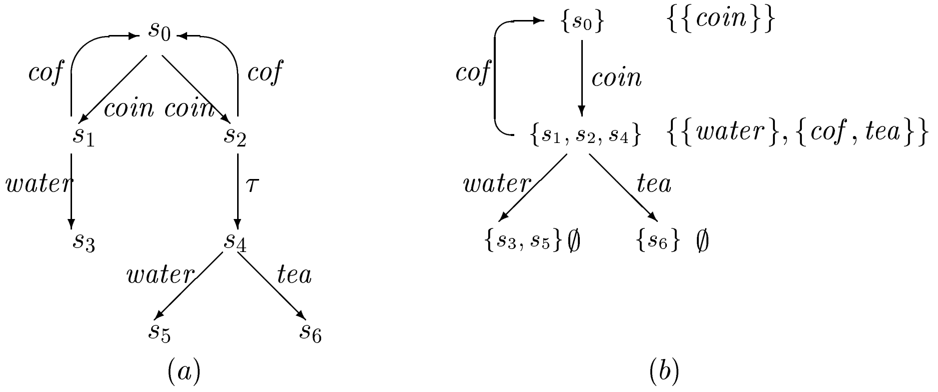


Figure 2.29. *A labeled transition system (a), the corresponding success graph (depicts must sets only) (b).*

An example of a specification and its success graph is shown in Figure 2.29. Construction of the success graph is conceptually easy. The main idea is to view a non-deterministic LTS not as occupying one state at a time, but as occupying the set of states reachable after having performed a given trace. From this set of states, the deadlock properties, M, C and R, of the original LTS after that trace can be infered.

If two sets of states are identical, they share the same deadlock properties and possible future behaviors, even if they were reached by different traces. To capture the deadlock properties of a non-deterministic LTS, it therefore suffices to construct a deterministic LTS whose nodes are sets of states, and whose transitions represents the possibility of performing an action from one set of states and thereby occupying a (possibly) different set of states. The deterministic property of a success graph makes construction of test cases straightforward since the deadlock properties and the verdicts holding after a given trace are unique and directly available from the unique node reached by executing the trace in the success graph.

Such a trace equivalent deterministic LTS can be computed using an algorithm essentially identical with the determinization algorithms well known from automata theory [70]. The inductive definition of determinization in Definition 2.30 nearly gives an implementable algorithm.

Definition 2.30 *Determinization of LTSs*:
Let $\mathcal{S} = \langle S, s_0, Act_\tau, \rightarrow \rangle$. $\mathcal{S}_d = \langle S_d, s_{0d}, Act, \rightarrow_d, M, C, R\rangle$ is deterministic and trace equivalent to $\mathcal{S}$ if

> The states $S_d \subseteq 2^S$ and transitions $\rightarrow_d$ are defined inductively as
>
> (a) $s_{0d} = \mathcal{S}$ **after** $\epsilon \ \in S_d$
>
> (b) if $B \in S_d \wedge a \in \mathrm{Sort}(B)$ then $B' = B$ **after** $a \ \in S_d$ and $B \xrightarrow{a}_d B'$

□

A transition $B \xrightarrow{a}_d B'$ represents the following: B is the set of states reached after a given trace, and B' is all states that can be reached from B by performing an a action followed by a sequence of τ actions. Exactly which specific state is reached is unknown because of the uncertainty arising from non-determinism.

To make the definition implementable, the induction rule must be applied systematically to ensure that all reachable states and transitions are added. A termination check is also needed. These steps can be done as follows: starting from the initial branch s_{0d} transitions are added either depth first or breadth first by applying the induction rule for all actions in the sort of the current branch. Before a new branch B' is added, it is checked whether there already exists a $B \in S_d$ equal to B'. If such a branch exists, it is unnecessary to explore the target branch further since all transitions and branches reachable from that either have been or will be added, depending on traversal order. Our test tool presented in Section 2.4 implements an instance of this algorithm.

It is important to note that the transformation of a non-deterministic finite automata to a deterministic one is known to be PSPACE-complete [32]. It is therefore questionable whether it will be possible to create the entire success graph for very large specifications. The problem arises because the branches in the determinized graph consists of *subsets* of states of the non-deterministic LTS.

The success graph contains the information necessary to generate tests, and it is relatively straightforward to traverse it, combine its must, may, and refusal properties, and output the tests. In fact, Algorithm 2.27 can be used essentially without modification: It should be executed on the success graph rather than the LTS. An algorithm using the success graph is easier to implement and is potentially faster, but suffers from the potentially expensive constructing of the complete success graph.

2.3.4 Test Selection and Coverage

A specification usually contains a huge number of relevant may and must properties, and it is generally infeasible to generate and execute them all. It is therefore necessary to select only a small subset of the possible tests. The general aim of *test selection* is to produce a test suite that has a high likelihood of detecting non-conformance, but at the same time has a cost (e.g., number of tests, total execution time) that lies within the resources allocated to testing, cf. Brinksma et al. [22].

A test *selection criterion* (or coverage criterion) is a rule describing what behavior or requirements should be tested. In general, test selection cannot be done solely on the basis of the specification but requires some external knowledge.

One kind of external knowledge is *test purposes* which are specific objectives or requirements that the developer has declared important to test. At least one test is (automatically) generated per purpose. With *fault model* based test selection, only tests that may detect certain specific implementation errors are generated[2]. One could also aim at covering the specification or implementation *structurally* by requiring transition or statement coverage. Further, the input domains of a system can be partitioned into *equivalence classes* in which the system is expected to behave identically, and then select at least one input for each such class.

Coverage is a metric of completeness with respect to a test selection criterion [12], e.g., what fraction of the test objectives have been tested. Such a quantified metric can be used in several ways. One use is to compare the strengths of test suites: if the cost of two test suites are comparable, the suite with highest coverage is preferable. Another use is to evaluate how much effort has been put into testing so far, and how much more resources need to be allocated to achieve a required level of coverage. Further, it can be used to indicate how many errors that might remain in the implementation under test. For these reasons, explicit selection and coverage metrics are required in any professional engineering approach to testing.

Because the number of test cases that can be generated is enormous and even infinite, it is usually inefficient to generate all possible test cases and then select the required subset. In our view, a good test generation tool applies the test selection criterion constructively during test generation to obtain a

[2] Although test purposes and faults are two sides of the same coin, we find it useful to distinguish them such that a test purpose means a manually stated, abstract, specification dependent property, and such that a fault model means implementation errors like (syntactical) mutations of a specification, e.g., transfer faults in finite state machines, see Section 6.1.3.

covering test suite. It should also minimize the amount of redundancy in the test suite with respect to the selection criterion to reduce its cost.

We shall not propose detailed coverage criteria here, except to observe that the success graph constitute a semantic model of the specification, namely its may and must properties, and that a set of potentially important criteria can be formulated as coverage of the success graph, e.g., when all edges (or nodes) combined with the properties contained in the nodes have been used in some test at least once. A remaining challenge is to devise traversal algorithms that minimize the size of the generated test suites. Other criteria for untimed concurrent systems are proposed by Taylor in [110]. We return to test selection in Section 3.3 where we propose a specific set of coverage criteria for timed systems.

2.4 A Test Generation Tool: TestGen

Based on the testing theory outlined in the previous section, we have developed a prototype tool that assists in the generation of tests from $\mathcal{L}_{\text{csm}}$ specifications. In Section 2.4.2 we present the concrete combined determinization and reachability algorithm implemented by TestGen. Then follows a few examples demonstrating the capabilities of the tool in Sections 2.4.5 to 2.4.7.

2.4.1 Tool Features

Given a $\mathcal{L}_{\text{csm}}$ specification TestGen uses reachability analysis to construct the success graph. Its essential features are:

- Construction of the success graph.
- Detection and handling of diverging specifications.
- Optional termination when a maximum observable trace depth has been reached.
- On-the-fly generation of the state space.
- Output of the success graph in 'dot'-format.

The state space of the specification is constructed on-the-fly. Using on-the-fly techniques has the advantage of only spending memory and time for exploring the state space that will actually be needed to construct the potentially partial success graph.

Dot [62] is a tool capable of performing automatic graph layout and generating postscript files thereof. That is, the output of TestGen can be fed to Dot to produce illustrations of the success graph. As previously mentioned, the AutoGraph tool [92] is used to draw the input to TestGen.

TestGen is implemented as approximately 15K lines of C++ code. The implementation is based on code from a timed automata simulator part of an old version of the UppAal toolkit [65]. The existing AutoGraph file format parser could thus be readily reused, and the abstract interpreter was easy to down grade to an untimed version.

The purpose of the prototype implementation is to gain concrete insights into the realization of the fundamental algorithms outlined in Section 2.3 before addressing the more challenging real-time case.

Not all facilities required in a fully functioning test generation tool are present in the prototype. In particular, traversal of the success graph and generation of concrete tests are not implemented. Test selection beyond limiting the trace depth is also not implemented. Further, the implemented algorithms are based rather directly on the formal definitions, which from a algorithmic point of view may not be optimal. Similarly, not much time has been spent on implementation level optimization to reduce time and space consumption.

2.4.2 Construction of the Success Graph

The implementation constructs two LTSs, the reachable transition graph and the success graph. To construct the reachability graph on-the-fly, the state exploration procedure is invoked from within the success graph construction algorithm as this progresses branch by branch.

A state is represented the same way as it were in the semantics, namely as a location vector and variable valuation pair: $\langle \bar{l}, \bar{v} \rangle$. In addition, it contains a number of flags that is used when the reachability graph is traversed. The transition rules defined in Definition 2.8 are used to compute the successor states from a given state. The branches contain sets of states as previously discussed, edges to its successors, and the minimized must sets.

In Algorithm 2.31 we present some abstract code that attempts to mirror the order in which the actual C++ code implements the abstract test generation algorithms previously defined. The algorithm consists of four procedures. Reach(B) computes the states reachable via a sequence of τ actions or one observable action. The next two procedures are provided to emphasize the separation of reachability analysis and construction of the success graph. After$\tau(B)$ returns the set of states reachable via a sequence of τ actions that has just been computed by Reach(B). Similarly, After(B, a) returns the set of states reachable from B via an a action.

The main procedure $SG(B)$ constructs the transitions from the partial branch B. First, B must be fully constructed by adding the τ reachable states. The algorithm then checks if an identical branch with the same set of states already has been constructed. If a matching branch exists, no further exploration of this B is necessary. As an aside, it should be noted that the previously added transitions with B as destination is modified to point to the matching state in order to "close" the graph. If no matching state existed, B is added to the set of branches, and the transitions with B as source are then constructed. This procedure continues recursively depth first.

The must sets $M(B)$ for branch B can be computed as soon as B is fully constructed, or as is currently implemented, in separate iteration of the branches when determinization has terminated.

Algorithm 2.31 *Computation of Success Graph*:

```
input: A network of communicating state machines $\overline{\mathcal{M}}$ with initial state $s_0 = \langle \bar{l}_0, \bar{0} \rangle$
output: The corresponding success graph $\langle S_d, s_{0d}, Act, \rightarrow_d, M, C, R \rangle$.
Let $\langle S, s_0, Act_\tau, \rightarrow \rangle$ be the LTS to be constructed from $\overline{\mathcal{M}}$
global variables:
$s_d = \rightarrow_d = \rightarrow = \emptyset; S = \{s_0\}$

Reach($B$) $=_{\mathrm{def}}$  for $s \in B$ do
                    whenever $s \xrightarrow{a} s' \ \wedge \ s' \notin S$        //one observable step
                       $S := S \cup \{s'\}$
                    $B_\tau = \emptyset$
                    whenever $s \xrightarrow{\tau} s' \ \wedge \ s' \notin S$        //one $\tau$ step
                       $S := S \cup \{s'\}$; $B_\tau := B_\tau \cup \{s'\}$
                    Reach($B_\tau$)                                  //depth first traversal

After$\tau$($B$) $=_{\mathrm{def}}$  Reach($B$)                                 //invoke reachability
                 collect $B_\tau = \{s' \mid \exists s \in B.\ s \Rightarrow s'\}$
                 return $B_\tau$                                  //closure of $\tau$'s

After($B, a$) $=_{\mathrm{def}}$ return $\{s' \mid \exists s \in B.\ s \xrightarrow{a} s'\}$

SG(var $B$) $=_{\mathrm{def}}$ $B :=$ After$\tau$($B$)                       //states after $\tau^*$
                 if $B \notin S_d$ then                           //unconstructed state
                    $S_d := S_d \cup \{B\}$
                    for $a \in$ Sort($B$) do                      //states after an a
                       $B_a :=$ After($B, a$)
                       add $B \xrightarrow{a}_d B_a$                     //new transition
                       SG($B_a$)                                 //depth first

$M(B) =_{\mathrm{def}}$         See Section 2.4.4
$C(B) =_{\mathrm{def}}$         See Definition 2.28
$R(B) =_{\mathrm{def}}$         See Definition 2.28

init $=_{\mathrm{def}}$          $SG(\{s_0\})$
```

The reached states are stored in the set S which is expanded as the state space is constructed. Whenever a state is constructed, it is checked whether an identical state already is in S and therefore has been reached from another state. Further exploration is thus unnecessary. Because this check is done frequently it is essential that it can be done fast. S is therefore implemented as a hash table where the hash key is computed from the location vector of the state; variable values are not used in the prototype.

Similarly, a hash table is used to hold constructed branches. The set of states in a branch is represented as a quick sorted array of pointers to states. Two branches are equal if their sorted arrays are identical. The hash key is currenty computed by combining the hash key from the first and last state of the branch.

2.4.3 Divergency Check

It is possible to write $\mathcal{L}_{\mathrm{csm}}$ specifications that do not strongly converge. TestGen is able to detect and correctly handle specifications that diverge through τ loops, i.e., it has a state s such that $s \xrightarrow{\tau}^{*} s$.

Detection of divergence is necessary for two reasons. First, the correctness of our algorithm for computing must sets depends on convergence, and second, divergence in the specification is likely to be unintended. If a specification diverges, the implementation is permitted to compute internally indefinitely without being required to accept inputs or deliver outputs. If a branch B contains diverging states, it can in principle execute the internal actions indefinitely without accepting external communications, and it is therefore our view, that it can have no must sets. We therefore set $M(B) = \emptyset$. Moreover, it is highly questionable if such behavior is at all acceptable. The tool therefore outputs a warning if it finds any τ cycles.

TestGen detects divergence by invoking a cycle detection algorithm on the reachability graph when a new sub graph has been added by the Reach function. Diverging states are marked with a flag to avoid rechecking states that has already been checked.

2.4.4 Construction of Must Sets

One way to compute the must sets is to simply enumerate all subsets of observable actions and then check if each subset is a must set. However, this procedure is obviously inefficient for large sets of observable actions. Instead the minimal success set should be computed from the states contained in the branch.

Let B denote the set of states in a branch. The first observation is that only stable states can contribute with actions to a must set. An unstable state can always decide to perform an internal transition to another state, and hence refuse the must set. The same argument can be made to the destination state.

By our assumption of strong convergence the argument stop at a stable state, which then must be able to engage in one of the actions of the must set. If a stable state of B can perform no actions (its sort is empty) the specification contains a deadlock after the trace leading to B, and the success set of B is consequently empty. The second observation is that if a set of actions is to be a must set for B it must contain at least one action from the sort of each stable state.

Our algorithm starts by constructing the must sets M_i^0 for each stable state s_i. This equals a must set for each observable action in s_i, i.e., $M_i^0 = \{\{a\} \mid a \in \mathrm{sort}(s_i)\}$. A must set over two stable states can now be computed by uniting a must set from each state. All must sets are found by forming all such pairs $M_j^1 = M_{2j}^0 \bowtie M_{2j+1}^0, j \in 0 \ldots i/2$, where $X \bowtie Y =_{\mathrm{def}} \ \{x \cup y \mid x \in X, y \in Y\}$. The must set over four states (level 2) can be computed from the must sets of level 1: $M_k^2 = M_{2k}^1 \bowtie M_{2k+1}^1, k \in 0 \ldots j/2$. This is now continued recursively until one set remains M^n, which terminates the algorithm. M^n then contains must sets over all the stable states in B. When the number of elements at a given level is odd, the unpaired element is moved to the next level.

A minimal must set can now be computed by minimizing M^n. However, it is much more efficient to minimize the must sets computed at each level *before* joining them, because fewest possible elements must then be joined. That is, $min_{\subseteq}(M_1 \bowtie M_2) = min_{\subseteq}(min_{\subseteq}(M_1) \bowtie min_{\subseteq}(M_2))$.

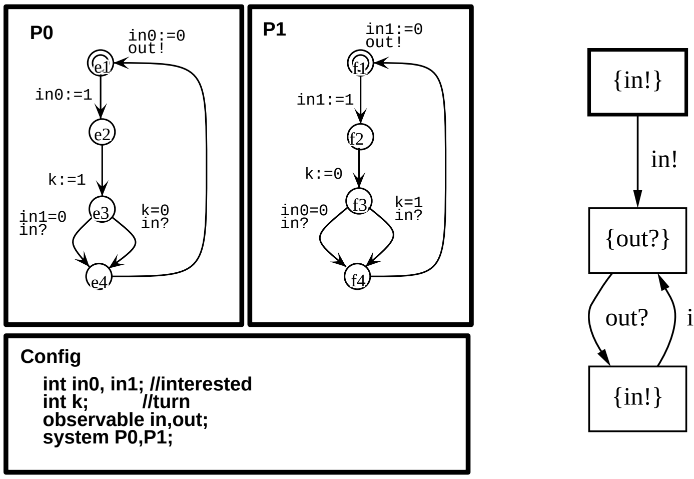
The costly operations in the algorithm are the joining and minimization operations which both are quadratic in the number of actions in the sets.

2.4.5 Example: Peterson's Mutual Exclusion Algorithm

Figure 2.32a shows the mutual exclusion property specified using Peterson's algorithm [106]. The configuration box defines the structure of the system. In this case, two processes p_1 and p_2 sharing three integer variables, in_0, in_1, and k, and offering the observable actions `in!` and `out?` to the environment.

In Peterson's algorithm each process P_i announces its interest to enter the critical section by raising a flag in_i. If two or more processes are interested simultaneously, the protocol uses a turn variable k to resolve the conflict. A process graciously gives the turn to the other process. A process may enter the critical section if no other processes have declared their interests, or if the turn variable declares that process as a winner.

The success graph generated by TestGen is shown in Figure 2.32b. Each rectangle corresponds to a branch in the success graph. The initial state has been bold faced. Each rectangle contains the must sets for that state. As expected, the success graph shows that the legal behavior consists of sequences of strictly alternating `in!` and `out?` actions.



(a) (b)

Figure 2.32. *Communicating State Machine version of Peterson's mutual exclusion algorithm (a), and its success graph (b).*

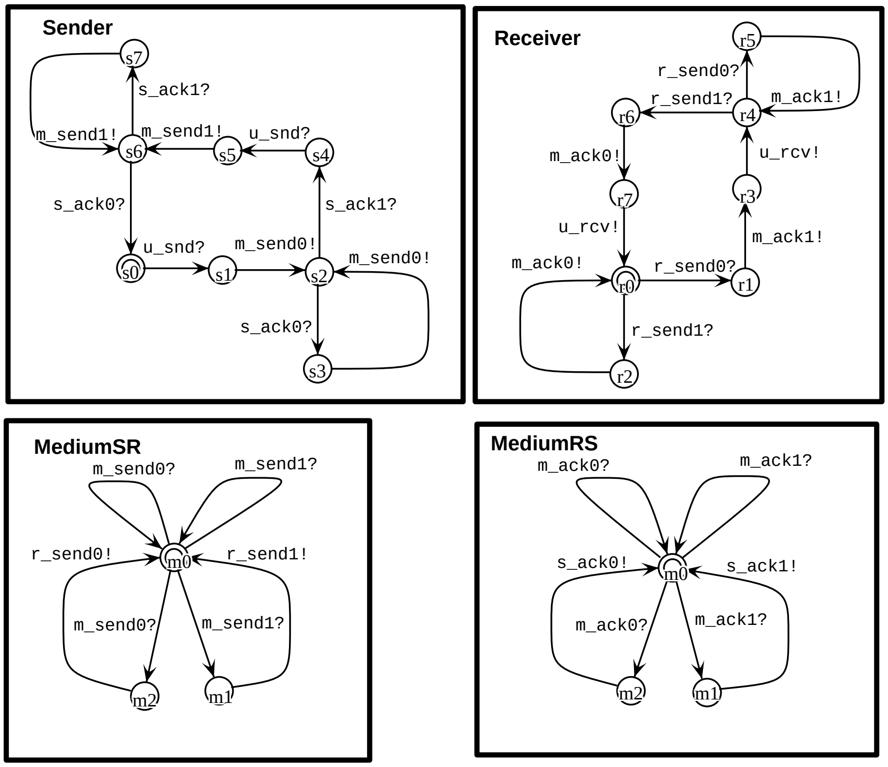
2.4.6 Example: The Alternating Bit Protocol

Figure 2.34 depicts the specification of a system communicating using the alternating bit protocol. The sender and receiver communicate via the same lossy communication medium as in Figure 2.1, so messages may be lost. The basic principle is to stamp messages with a one bit sequence number. When a protocol entity sends a message (either a data or an acknowledgment) with sequence number b, the next message it receives should be $\neg b$. If the sequence number is not as expected, the protocol entity concludes that a message has been lost, and retransmits. The protocol actions are summarized in Table 2.33.

Action	Description
u_snd	data from upper layer: data to be sent
u_rcv	data to upper layer: data has been received
m_send$b$	data to media from sender with seq. nr. $b \in \{0, 1\}$
r_send$b$	data from media to receiver
m_ack$b$	ack from receiver to media
s_ack$b$	ack from media to sender

Table 2.33. *Alternating bit protocol actions.*

The generated success graph is depicted in Figure 2.35. An example of interesting protocol behavior is the trace u_snd · u_snd · u_rcv. After the first two



Config

```
chan m_send1, m_send0, r_send0, r_send1, m_ack1, m_ack0, s_ack1, s_ack0;
observable u_rcv, u_snd, TO;
system Sender, MediumSR, MediumRS, Receiver;
```

Figure 2.34. *Specification of a protocol using alternating bits.*

send actions, the `u_rcv` action is *mandatory* at the receiver. It would at a first glance seem illogical that reception is guaranteed when the two protocol entities communicate via a media without delivery guarantees. The explanation is however logical: First note that the second send is only a possibility since the success set is empty. But if the second send succeeds, the sender must have received an acknowledgment with correct sequence number from the receiver earlier. Otherwise it could not have entered location s_4. This implies that the receiver successfully received the data. Therefore, the receiver entity must also be able to deliver it to the upper layer.

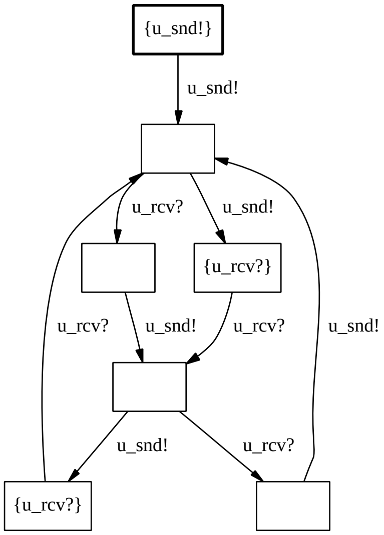


Figure 2.35. *Success graph for the protocol using alternating bits.*

Real-life protocol specifications are usually considerable more complex than this example, but with only slightly more complex protocols systematic *manual* test generation becomes overwhelming.

Finally, observe that the above model does not use timeouts to trigger retransmission. This means that not all protocol behaviors usually found in a alternating bit protocol implementation are reflected by the success graph. Only 26 states were reachable. If explicit observable timeout transitions were added from location s_2 to s_1, and from location s_6 to s_5, the state space would increase to 168 states, and the success graph would contain 37 branches, and would thus be too large to show here. The success graph for the protocol presented in Figure 2.1 first in this chapter contains 37 branches, and is also to large to present here.

2.4.7 Size Matters

The next examples do not demonstrate new functionality of TestGen per se. They are used to provide some rudimentary performance figures when the tool is subjected to larger models. These figures give insight into the behavior of the implemented algorithms, and can point out some of their limitations.

Interesting figures include the size of the state space, the size of the success graph, and the time and space used to construct it. The size of the success graph is interesting because it gives an indirect measure of the number of tests that need to be executed. Because the implemented algorithms have a high degree of complexity, the actual time and space consumption should be monitored, because these easily become bottlenecks.

We have designed two extreme specifications, *parServer* and *nonSense*. *parServer* models a simple cluster computing system consisting of a number of parallel servers and an interface to clients. A job offered by the environment is accepted by the interface (`requestHandler` process) when a free server exists. It forwards the job non-deterministally to one of the free `server` processes. The server, able to process one job at a time, receives the job, computes the result, and forwards it to the interface (`replyHandler` process) which agains formulates a reply to the environment.

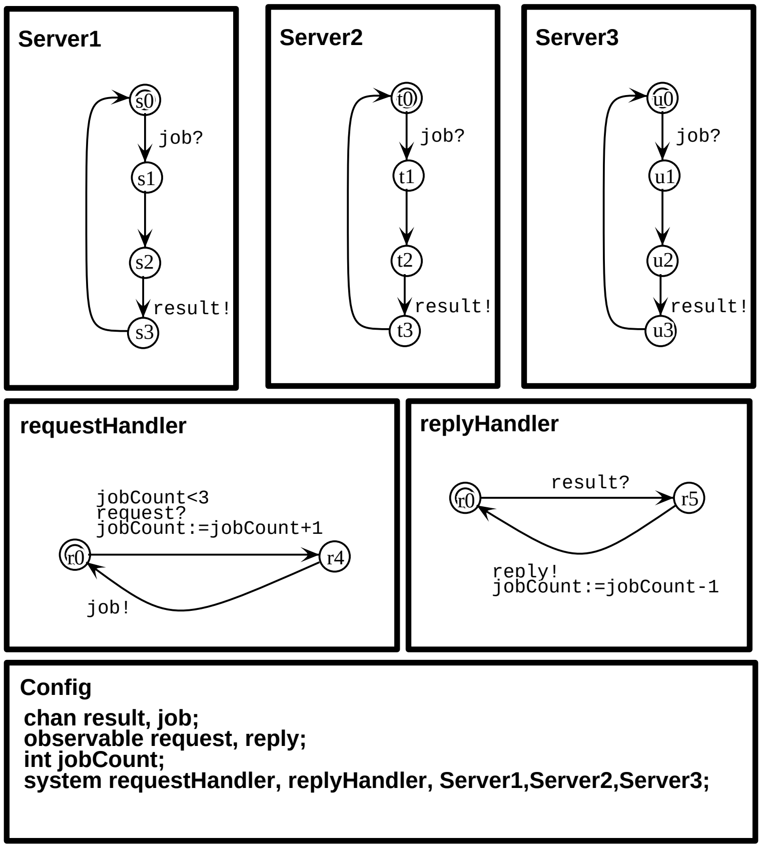
An easy but effective way of increasing the size of the model is to add more server components. A *parServer* with n servers is denoted $parServer^n$. Figure 2.36 depicts $parServer^3$.

The *nonSense* specification shown in Figure 2.37 contrasts the *parServer*. *nonSense* uses almost exclusively observable actions that are triggered by the environment, and relatively little internal communication, whereas *parServer* has few observable actions, but a substantial amount of internal communication.

The experiment measures the number of branches in the success graph, the number of reachable states, the execution time to construct the success graph, and the memory consumption as function of the number of replicated components n. The platform used in the experiment is a Sun Ultra-250 workstation running Solaris 5.7. The machine is equiped with 1 GB RAM and 2×400 MHz CPU's. No extra compiler optimizations was done on the code.

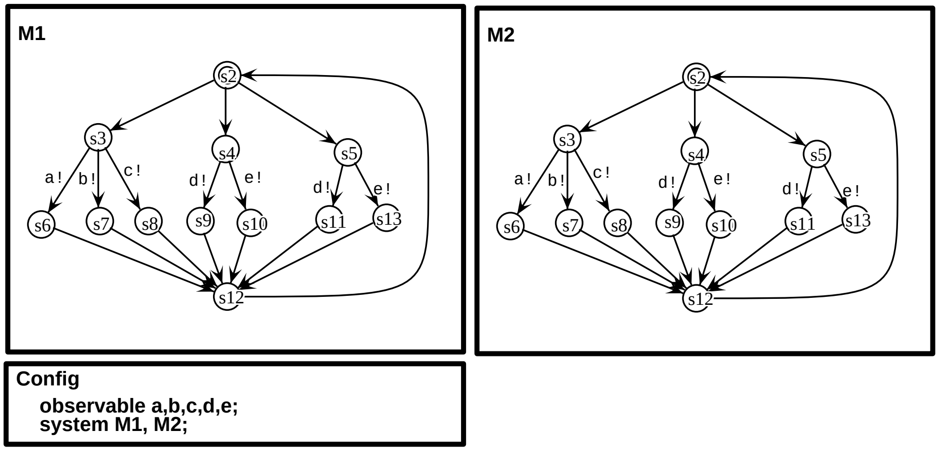
The measured values are summarized in Table 2.38. Although one should be very careful in drawing general conclusions from a simple and small experiment like this, we believe that the following observations can be made:

- As should normally be expected, the reachable state space grows exponentially with the number of parallel components (the state explosion problem).



```
chan result, job;
observable request, reply;
int jobCount;
system requestHandler, replyHandler, Server1,Server2,Server3;
```

Figure 2.36. *parServer*^{3} *specification.*



```
observable a,b,c,d,e;
system M1, M2;
```

Figure 2.37. *nonSense*^{2} *specification.*

	$parServer^n$					
n	3	6	7	8	9	10
Reachable States	208	15808	64256	259328	1042432	4180992
Branches	13	34	43	53	64	76
Memory (MB)	49	56	69	118	328	1165
Execution Time (s)	0.3	7	34	160	790	22608

	$nonSense^n$			
n	1	2	3	4
Reachable States	12	144	1728	20736
Branches	6	151	1156	4906
Memory (MB)	50	53	58	285
Execution Time (s)	0.3	1	42	2318

Table 2.38. *Results of the measurements.*

- In both cases the number of branches is considerable smaller than the number of reachable states. This is caused by the collapsing of τ transitions. This finding is consistent with [32].

- The growth of the success graph should also be expected to be exponential. This is however only confirmed by the *nonSense* specification. Although there exists known automata with n locations that expand to the full set of 2^n locations when determinized [93], they appear rather pathological. These observations indicate that determinization will be feasible in many practical cases.

- It seems difficult to predict the number of branches in the success graph from either the number of parallel components or the size of the state space. The *parServer* has a large state space but a very small success graph. In contrast, *nonSense* produces a large success graph from a moderately sized state space. Thus, the size of the success graph seems to depend on the mix of internal and observable actions, and on how 'malicious' the non-determinism is.

- The tool uses a lot of memory. This is not only used to store the state space, but especially the branches. Recall that a branch contains a set of pointers to reachable states. This direct representation appears to be very memory demanding.

- The number of parallel components for which TESTGEN can construct the full success graph is somewhat limited. Both specifications were extreme in their own way, and one could hope that realistic specifications are better behaved.

As previously pointed out, the implemented algorithms are quite naive. Significantly better algorithms for computation of success graph like data structures have been developed in the context of model checking with refinement relations similar to the testing preorder, e.g, failures divergence refinement [93, 94]. These algorithms reduce the number of states contained in the branches, and are sometimes able to reduce and determinize each component before combining them to the global success graph. Some care is required to preserve the information needed to compute must sets and divergence information.

We would finally like to stress that constructing very large success graphs is not a goal by itself. In the end, the success graph must be traversed to construct tests that are to be executed. It is usually the case that the number of tests that can be executed in practice is much less than the number that can be generated from a large success graph, even when the tests are executed automatically.

2.5 Summary

In this chapter, we have examined how tests can be generated for untimed systems. Our behavioral specification language specify system behavior by a network of concurrent finite state machines communicating synchronously on complementary actions and sharing integer variables. Formally based testing requires a precise definition of the implementation relation, tests, and test execution. In our work these are based on Hennessy's classical testing theory that characterizes systems based on the tests, modeled as labeled transition systems (LTS), they may and must pass. We showed during our discussion of the possible implementation relations that the choice is not arbitrary and depends on the application.

Based on a LTS semantics of the communicating state machine languages and the formalized implementation relation, we have proposed two algorithms for generating tests, one based on a direct interpretation of the LTS, and one based on the success graph of the specification. The success graph is a condensed version of the specification containing the information necessary for test generation.

The main advantage of the direct interpretation is that it avoids the state explosion problem by only requiring storage of the set of states that the specification could possibly occupy after the prefix trace of the test generated so far. Thus, very large specification can be handled.

However, we believe that the success graph approach is generally advantageous for the following reasons:

1. Once constructed, tests can be generated from the success graph by an *easy* graph traversal.

2. A test can be constructed very *fast* in an online fashion (as it is being executed). This is potentially important when a stress test is performed where events must be offered as fast as possible. The direct approach has the overhead of computing and collecting the reachable states after each step, and computing the associated must, can and refusal action sets. Alternatively, very long tests must be generated and written to a file off line.

3. It allows the formulation of a *coverage criterion* based on a semantic model of the specification, namely its may and must properties.

4. *Systematically* generating different tests becomes easier by keeping track of which properties have been generated and which have hitherto not been included in some test.

The potential disadvantages are the size of the resulting graph, and the space and time used to construct it. Thus, the space explosion problem may become a limiting factor.

The two algorithms can be viewed as two extreme approaches: The direct interpretation uses no helping data structure at all, and the success graph is the ideal data structure. Other data structures between these extremes are possible, such as performing only τ reduction but no determinization, either from the composed system or componentwise. These approaches then form different compromises between space and time usage early in the process versus the amount of computation and effort needed to construct tests later in the process.

We have implemented a prototype test generator that constructs the success graphs for communicating state machine specifications. Our experiences indicate that the success graph can be feasibly constructed in its entirety in many cases, but also that it can become unmanageable in certain cases when the specification is very large, or when the internal non-determinism cannot be reduced. The implemented naive algorithms should also be improved.

Turning the attention towards real-time systems, it can be noted that the success graph data structure cannot be re-used as is. It does not include the necessary timing information, and, since the state space of a real-time system is infinite, a finite success graph cannot be constructed using existing data structures and algorithms. The next chapters address the problem of generating tests from real-time specifications with the additional goal of finding a finite data structure similar to success graphs.

Chapter 3

Timed Testing

The previous chapter introduced the necessary ingredients in a setup for automated test generation: a specification language, an implementation relation and corresponding test notation language, an algorithm for test generation, and a strategy for selecting the test case to be executed. The introduction of time requires a revision of these ingredients.

During the last decade the *timed automata* model due to Alur and Dill [9] has become popular for specifying real-time systems, and we adopt this as our timed specification language. A timed automaton adds clock variables to an ordinary state machine, and provides conditions on the enabling of its transitions. Timed automata are introduced and formally defined in Section 3.1.

Our implementation relation is based on a timed version of the Hennessy tests developed in the untimed setting. A timed test describes at what time instances the actions in the test must be offered and observed. A timed test can also be expressed as a timed automaton that deadlocks in a fail labeled location when the test is not passed. Real-time tests are introduced in Section 3.2.

A test generation algorithm is given in Section 3.3. This generates relevant timed tests based on a direct interpretation of the specification. However, we find this algorithm imperfect because the generated tests cannot easily be related to a coverage criterion of the specification. Selection of the tests to be produced is an even more pertinent issue when time is added because there is an enormous number of possible time instances that could be relevant to examine. To deal with this problem, we propose to partition the state space of the specification, and cover each of these. Such a partitioning can be done in numerous ways, and therefore Section 3.4 proposes a framework for partitioning and covering the state space of the specification.

Furthermore, it would be advantageous to use a data structure similar to the success graph used in the untimed case to support test generation. However, because of the infinite state space of timed systems, the success graph cannot be directly re-used. The goal of succeeding Chapter 4 will be to employ the selected criterion to generate timed tests using an appropriate data structure, and a symbolic analysis technique. Ideally, coverage according to the chosen criterion should follow by covering the support data structure.

Finally, Section 3.5 summarizes this chapter.

3.1 Timed Automata

The *timed automata* model, originally proposed by Alur and Dill in [8, 9], has become a popular formalism in the model checking community for specifying and modeling real-time systems. A considerable amount of work has been done on both their theoretical properties and on tools for their analysis.

Timed automata augment the automata well known from the study of formal languages with a set of (dense) clock variables which are used to express enabling conditions on the transitions in the automaton. Timed automata is not a single well defined model, but is rather a family of models sharing the idea of automata, clock variables, and enabling conditions. The variations differ in their expressiveness, decidability properties, and, from a practical point of view the important aspects of how easy systems can be modeled, and how efficiently they can be analyzed.

3.1.1 Informal Description of Timed Automata

We define a timed automata variant based on the communicating state machines model defined in Section 2.1.1. The model consists of a set of concurrently executing timed automata sharing a set of clocks. This model is designed to be general and expressive but still analyzable. However, it will be necessary to restrict the model depending on how tests are to be derived. We first describe the model informally, and then its formal syntax and semantics.

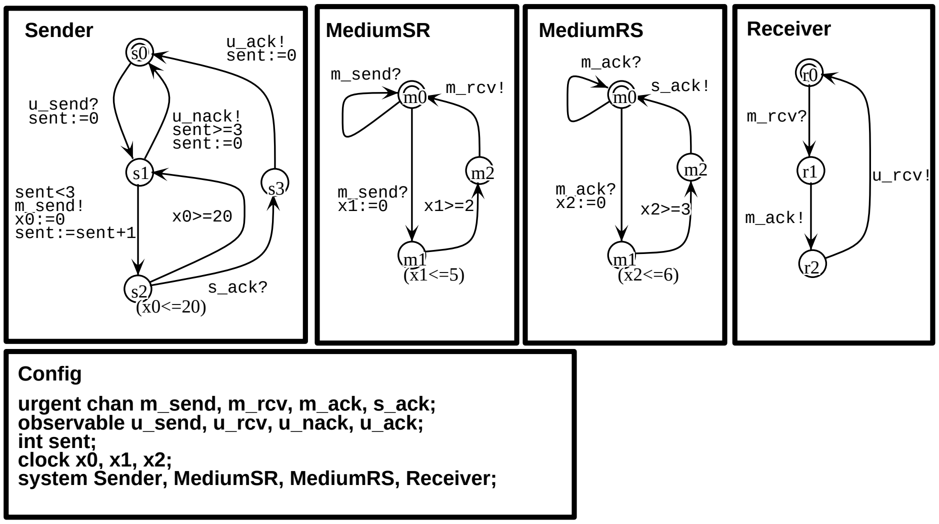
We use the term locations to denote nodes in the automaton, and reserve the term states to denote semantic automata configurations consisting of a location vector combined with a clock valuation.

Clock values are modeled by the set of positive reals $\mathbb{R}_{\geq 0}$. A clock can be viewed as a piecewise continuous function of time with the derivative one. As time passes, the values of all clocks increase by the same amount. When an action is executed, clock assignments may cause discrete jumps in the clock values.

An edge in a timed automaton is labeled with three pieces of information g, a, and r. The enabling condition or *guard* g is the conditions over clocks that must be satisfied before the action a is enabled. When action a is executed, the set of *clock assignments* in r are simultaneously executed. The execution of an action is instantaneous, i.e., takes no time. Locations are further labeled with *invariant conditions* which states under which clock conditions the automata may remain in that location. It must change location before the invariant condition becomes false.

Actions are classified as either *observable* or *hidden*. Only observable actions are visible to the environment, and only synchronizations on hidden actions are permitted within the network. Moreover, actions are further classified as *urgent* or *non-urgent*. Intuitively, whenever two automata are ready to synchronize over urgent actions, the synchronization takes place immediately. In contrast, it is unpredictable when synchronization on non-urgent actions take place, i.e., time may pass even if they both are enabled. Invariants can be used to give an upper bound on this synchronization delay.

At the syntactic level, a non-urgent τ action is specified by an edge without action labeling. There is no syntactic construction for the urgent variant since this can always be achieved by synchronizing with an auxiliary τ server automaton that is always prepared to synchronize over a dedicated urgent action.



```
urgent chan m_send, m_rcv, m_ack, s_ack;
observable u_send, u_rcv, u_nack, u_ack;
int sent;
clock x0, x1, x2;
system Sender, MediumSR, MediumRS, Receiver;
```

Figure 3.1. *Timed automata model of a transmission protocol.*

A timed automata variant of the communication protocol specified in Section 2.1.1 is shown in Figure 3.1. It uses the same actions as the original specification; these are listed in Table 2.2. In addition, it uses three clocks. One is used by the sender to time out waiting for an acknowledgment, and two are used by the transmission medium (one in each direction) to express the permitted transmission delay.

The sender waits in location $s2$ for 20 time units for an acknowledgment. If no acknowledgment is received within this time bound, the sender makes an internal transition to location $s1$ to initiate retransmission. The invariant condition on location s_2 and guard on the time out edge ensures that the sender moves away precisely when the 20 time units have elapsed. Transmission from sender to receiver takes between 2 and 5 time units. Transmission in the reverse direction is slightly slower.

3.1.2 Dense Time Semantics of Timed Automata

The semantics of a network of timed automata can be given as an infinite state timed LTS, see Definition 3.2. The progress of time can be modeled by adding a set of special *delay actions* $\{\varepsilon(d) \mid d \in \mathbb{R}_{\geq 0}\}$ to the set of actions. Execution of a delay action $\varepsilon(d)$ represents the passage of d time units, where d is a finite positive real-valued number. By adopting the *two-phase functioning principle* [76] we observe system execution by alternating between observing a set of instantaneous actions and observing a delay.

Definition 3.2 *Timed Labeled Transition System*:
An timed LTS is a 4 tuple $\langle S, s_0, Act_{\tau\varepsilon}, \rightarrow \rangle$, where

1. S is the set of states,
2. $s_0 \in S$ is the initial state,
3. Act is the observable actions, and $Act_{\tau\varepsilon} = Act \cup \{\tau\} \cup \{\varepsilon(d) \mid d \in \mathbb{R}_{\geq 0}\}$ the actions including the internal action τ and delay actions $\varepsilon(d)$.
4. $\rightarrow \subseteq S \times Act_{\tau\varepsilon} \times S$ is the transition relation satisfying the following consistency constraints:

   Time Determinism: Whenever $s \xrightarrow{\varepsilon(d)} s'$ and $s \xrightarrow{\varepsilon(d)} s''$ then $s' = s''$.

   Time Additivity: $\forall s, s'' \in S.\ \exists s' \in S.\ s \xrightarrow{\varepsilon(d_1)} s' \xrightarrow{\varepsilon(d_2)} s''$ *iff* $s \xrightarrow{\varepsilon(d_1+d_2)} s''$

   Null delay: $\forall s, s' \in S.\ s \xrightarrow{\varepsilon(0)} s'$ *iff* $s = s'$
5. We assume that Act is equipped with a mapping^{-}: $Act \mapsto Act$ such that for all actions $\bar{\bar{a}} = a$. $\bar{a}$ is said to be the complementary action of a.
6. We lift the notation given in Definition 2.4 for LTS to apply to timed LTS, with the notable addition:

   $s \overset{\varepsilon(d)}{\Longrightarrow} s'$ *iff* $s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} s_n$ such that $s_n = s'$,
   $\forall i \in [1, n].\ a_i = \tau \ \vee a_i = \varepsilon(d)$, and $d = \sum\{d_i \mid a_i = \varepsilon(d_i)\}$.

□

Next we provide the formal definitions for our variant of timed automata. The formal structure of a timed automaton over a set of clocks X and actions $\mathcal{A}$ is given in Definition 3.3. The formal semantics of a network of timed automaton is given as a timed LTS in Definition 3.4.

Definition 3.3 *Timed automaton*:

1. The guards $G(X)$ over a set of clocks X is generated by the syntax $g ::= \gamma \mid g \wedge g$ where γ is a constraint of the form $x_1 \sim c$ or $x_1 - x_2 \sim c$ with $\sim \in \{\leq, <, =, >, \geq\}$, c a non-negative integer constant, and $x_1, x_2 \in X$.
2. $R(X)$ is the set of clock assignments of the form $x := c$;.
3. A timed automaton $\mathcal{M}$ over actions $\mathcal{A}_\tau$ and clocks X is a tuple $\langle N, l_0, I, E \rangle$ where
   (a) N is a (finite) set of locations,
   (b) $l_0 \in N$ is the initial location,
   (c) $I : l \mapsto G(X)$ is the location invariants,
   (d) $E \subseteq N \times G(X) \times \mathcal{A}_\tau \times 2^{R(X)} \times N$ is the set of edges where $G(X)$ is the set of guards, and $R(X)$ is the set of assignment operations. We write $l \xrightarrow{g,a,r} l'$ if $\langle l, g, a, r, l' \rangle \in E$ to represent a transition from location l to location l' with guard g, action a, and assignments $r \subseteq R(X)$.
   (e) Let $\overline{a}$ denote the complementary action of action $a \in \mathcal{A}$ such that $\overline{a!} = a?$ and $\overline{a?} = a!$.

□

A *network* of timed automata $\overline{\mathcal{M}} = (\mathcal{M}_1 \parallel \cdots \parallel \mathcal{M}_n)$ is a collection of concurrent timed automata. Let $\mathcal{L}_{ta}$ be the class of timed automata networks. The network synchronizes with the environment via a set of distinguished observable actions $\mathcal{O} \subseteq \mathcal{A}$. The network can synchronize internally only via the hidden actions $\mathcal{A} - \mathcal{O}$. That is, no internal communication is permitted over external, observable actions. As we offer no explicit restriction operation, this distinction allows a simple means of hiding the required actions from the environment. $\mathcal{U} \subseteq \mathcal{A}$ is the set of urgent actions.

Semantically, a state of a network is modeled by a configuration $\langle \bar{l}, \bar{u} \rangle$. The first component, the location vector $\bar{l}$, is a vector of locations that represent the joint control location of the network; l_i is the location of automaton $\mathcal{M}_i$. We write $\bar{l}[l'_i/l_i]$ to denote the vector where the ith element of $\bar{l}$ has been replaced by l'_i. The second component $\bar{u} \in \mathbb{R}_{\geq 0}^{|X|}$ is the current clock valuation. Let $\bar{u}(x)$ denote the value of clock x, $r(\bar{u})$ the clock valuation identical to $\bar{u}$ except for $\bar{u}(x_i)$ which equals c_i for all $x_i := c_i \in r$, and let $\bar{u} + d$ denote the clock valuation where all clocks are advanced by d time units. Let $g(\bar{u})$ denote the evaluation of guard g given clock valuation $\bar{u}$. The invariant of a location vector is the conjunction of the invariants of the individual locations, i.e., $I(\bar{l}) = \bigwedge_{1 \leq i \leq n} I(l_i)$. The evaluation of a location vector invariant with clock valuation $\bar{u}$ is written $I(\bar{l})(\bar{u})$. The initial state of the network is $\langle \bar{l}_0, \bar{0} \rangle$, where $\bar{l}_0$ is the vector of initial locations, and $\bar{0}$ is the clock valuation with all clocks being zero.

We allow both urgent and non-urgent actions. Therefore, there will implicitly be two variants of the internal τ action. No syntax is given for urgent internal actions, but it arises as the result of synchronization on urgent actions. When we need to distinguish, τ_u denotes the urgent variant, and τ_n the non-urgent variant. τ denotes either.

Our interpretation of network of timed automata is given by Definition 3.4. The first transition rule concerns internal or observable actions of a single automaton. When the guard of the action evaluates to true, the action is enabled, and the effect of its execution is an updated location vector and clock valuation. The location invariant in the target state must also be satisfied. The second rule states that two automata may synchronize over complementary hidden actions when both guards are true. The effect is a new state where both automata have changed locations, and where both sets of resets have been applied. Again, the invariant in the target location must also be satisfied.

Definition 3.4 *Transition rules for networks of timed automata*:

$$\frac{l_i \xrightarrow{g,a,r} l_i' \qquad g(\bar{u}) \quad I(\bar{l}')(\bar{u}') \quad a \in \mathcal{O} \cup \{\tau_n\}}{\langle \bar{l}, \bar{u} \rangle \xrightarrow{a} \langle \bar{l}', \bar{u}' \rangle}, \text{where } \bar{l}' = \bar{l}[l_i'/l_i], \bar{u}' = r(\bar{u})$$

$$\frac{l_i \xrightarrow{g_1,a,r_1} l_i' \qquad l_j \xrightarrow{g_2,\bar{a},r_2} l_j' \qquad (g_1 \wedge g_2)(\bar{u}) \quad I(\bar{l}')(\bar{u}') \quad i \neq j}{\langle \bar{l}, \bar{u} \rangle \xrightarrow{\tau'} \langle l', \bar{u}' \rangle}, \text{ where}$$

$a \in \mathcal{A} - \mathcal{O}, \tau' = \tau_u$ if $a \in \mathcal{U}, \tau' = \tau_n$ if $a \notin \mathcal{U}, \bar{l}' = \bar{l}[l_i'/l_i, l_j'/l_j],\ \bar{u}' = (r_1 \cup r_2)(\bar{u})$

$$\frac{\forall d' < d.\ (\langle \bar{l}, \bar{u} + d' \rangle \overset{\tau_u}{\not\longrightarrow} \quad I(\bar{l})(\bar{u} + d))}{\langle \bar{l}, \bar{u} \rangle \xrightarrow{\varepsilon(d)} \langle \bar{l}, \bar{u} + d \rangle}$$

□

The *progress condition* in the third rule describes how the network behaves with respect to time. Time may progress by some amount d if the invariant remains true, and if no synchronization on hidden urgent actions are possible in the interim. Thus, the network must change location when an urgent synchronization becomes possible, or when the location invariant becomes false.

It would be trivial to also permit shared integer variables and their usage in guards like it was done in the communicating state machine model in Section 2.1.1. Likewise, committed locations could be included. Not including these into the semantic description enables a more convenient notation.

We say that a timed automata network is *progressive* if all reachable states either can let time pass, or perform a finite sequence of internal actions to one that can. In essence, this prevents an invariant condition from blocking progress of time until the environment synchronizes on an observable action. A network is *Zeno-free* if for any bounded time interval, the network can only perform a finite number of actions.

It is also valuable to observe that there are two sources of non-determinism in the given semantics. The first source, *action non-determinism*, results from a choice between two actions, i.e, $s \xrightarrow{\tau} s'$ or $s \xrightarrow{\tau} s''$. The second source, *timing uncertainty*, results from a choice between executing an action or letting time pass, i.e., $s \xrightarrow{\tau_n} s'$ or $s \xrightarrow{\varepsilon(d)} s''$. Timing uncertainty is useful where the duration of an operation is unknown or only known within a bound. This cannot be accurately modeled in timed automata variants with only urgent actions.

3.1.3 Discrete versus Dense Time

The two dominating models of time are the *dense time* model, where clock valuations are drawn from a dense domain such as positive reals, and the *discrete time* model, where clock valuations are drawn from the set of positive integers. A discrete time interpretation of $\mathcal{L}_{ta}$ can easiliy be obtained by using only the timed action $\varepsilon(1)$ in Definition 3.4.

There are two main advantages of the discrete time model. First, it accurately models most digital systems. Secondly, the state space of discrete time system is finite, and can consequently be analysed using the same state space exploration techniques as in the untimed case. A minor complication is that clock values may sometimes progress towards infinity. Obtaining a finite state space therefore requires the observation that for each clock x, there is a largest constant c_x that is relevant for the enabling conditions involving x. The values of x exceeding c_x can therefore be equalized with the designated symbol ∞.

We use the dense model for the following reasons: First and foremost, it is the most general and the most realistic model of *physical time*. Secondly, when the resolution of the discrete clock is very high, it may be more appropriate to view it as being dense. For example, if the processor clock on a 500 MHz PC were used as time base for measuring mili-second range durations (500.000 clock tics per mili-second), the time domain is for all practical purposes dense. Finally, the test selection techniques to be developed which are mandatory in dense time generation, could also benefit discrete time models.

The main problem is that the state space cannot be represented and explored explicitly. Instead, infinite sets of states must be represented symbolically.

3.2 Timed Must Tests

In this section we propose the class of *timed may* and *must* properties as a reasonable way of characterizing the behavior of real-time specifications and implementations. Our starting point will be Hennessy's may and must properties defined in Section 2.2 which we shall lift to include time. The properties will now include timed traces consisting of observable actions and a labeling with the absolute or relative time at which they occur.

Because the LTS of a timed automaton with dense clocks has an infinite number of states, it is in generally preferable to describe tests as timed automata as well. We therefore construct *testing automata* that test for the satisfaction of timed may or must properties. We define an implementation relation based on the ability to pass timed may and must tests, and offer an interpretation of the resulting preorder.

3.2.1 Assumptions

We shall assume the following:

1. The specification is given as an network of timed automata, and the implementation can be described by some network of timed automata. Both are progressive and Zeno-free.

2. The observable actions that the implementation uses to communicate with its environment are known.

3. The implementation under test communicates with its environment using *urgent* and synchronous rendezvous style communication, i.e., $\mathcal{O} \subseteq \mathcal{U}$.

4. The specification converges strongly, i.e., an infinite sequence of τ actions is never possible.

Note in particular assumption 3 which has a significant bearing on how observations can and should be made. This assumption is made for two reasons. First, we find it most intuitive that two *ready* components synchronize immediately; the desired uncertainty about when a component becomes ready can still be expressed using non-urgent internal and hidden actions. Secondly, the progressivity assumption requires that the environment and the system always permit time to progress. This implies that they cannot use invariant conditions to stop time, and thereby force the other component to produce an awaited action. However, without invariant conditions, there is no upper bound on the synchronization delay on non-urgent actions. Effectively, the synchronization delay over non-urgent actions between the system and its environment cannot be bounded, and may not even take place. Thus, this communication mode used *externally* does not appear to be very important. On the other hand, it is still very convenient *internally* in the specification.

3.2.2 The Implementation Relation

The timed may and must properties are defined in Definition 3.5. The set of observable actions and delays is denoted by $\mathcal{O}_\varepsilon = \mathcal{O} \cup \{\varepsilon(d) \mid d \in \mathbb{R}_{\geq 0}\}$.

Definition 3.5 *Timed must properties*:
Let $\mathcal{S} \in \mathcal{L}_{ta}$.

1. $\mathcal{L}_{\mathbf{tmust}} =_{\mathrm{def}} \{\mathbf{after}\ \sigma\ \mathbf{must}\ A \mid \sigma \in \mathcal{O}_\varepsilon^*, A \subseteq \mathcal{O}\}$
2. $\mathcal{S} \models\ \mathbf{after}\ \sigma\ \mathbf{must}\ A$ *iff* $\forall s \in \mathcal{S}\ \mathbf{after}\ \sigma.\ \exists a \in A.\ s \overset{a}{\Rightarrow} \wedge$
   $(s \overset{a}{\not\rightarrow}$ *implies* $s \not\xrightarrow{\varepsilon(d)})$
3. $\mathcal{L}_{\mathbf{tmay}} =_{\mathrm{def}} \{\mathbf{can}\ \sigma \mid \sigma \in \mathcal{O}_\varepsilon^*\}$
4. $\mathcal{S} \models \mathbf{can}\ \sigma$ *iff* $\sigma \in Tr(\mathcal{S})$

□

A timed must property requires that the system after having performed the timed trace σ is ready in to engage in a synchronization with one of the actions in A. The synchronization is required at the time instant reached after the trace: The system is not permitted to wait and then become ready later. This means that, if the system in a given state is unable to accept an a action in must set A, then time is not permitted to pass. The system may however execute an internal action to a state that is able to accept an action in the must set. Similarly, a timed may property requires that the system is able to perform the corresponding timed trace.

It can be argued that it is impossible to test these properties in practice because it cannot be ensured that an action is enabled in precisely the required moment. However, it is our belief that a synchronization in a small interval around the time instant is sufficient in most practical situations. The absolute size of the interval naturally depends on the magnitude of the delays in the trace, of the required precision, and also of the precision of the physical timers in the test system.

Our proposed timed may and must implementation relations, formally defined in Definition 3.6, require that every timed must (may) property satisfied by the specification, is also satisfied by the implementation. Or equivalently, that every test automaton corresponding to a timed must (may) property that the specification must (may) pass, the implementation must (may) also pass.

Definition 3.6 *Timed must preorder*:
Let $\mathcal{S}, \mathcal{I} \in \mathcal{L}_{ta}$.

1. $\mathcal{S} \sqsubseteq_{\mathbf{tmust}} \mathcal{I}$ *iff* $\forall \sigma \in \mathcal{O}_{\varepsilon}, \forall A \subseteq \mathcal{O}.$
   $\mathcal{S} \models \mathbf{after}\ \sigma\ \mathbf{must}\ A$ *implies* $\mathcal{I} \models \mathbf{after}\ \sigma\ \mathbf{must}\ A$
2. $\mathcal{S} \sqsubseteq_{\mathbf{tmay}} \mathcal{I}$ *iff* $\forall \sigma \in \mathcal{O}^*_{\varepsilon}.\ \mathcal{S} \models \mathbf{can}\ \sigma$ *implies* $\mathcal{I} \models \mathbf{can}\ \sigma$
3. $\mathcal{S} \sqsubseteq_{\mathbf{tte}} \mathcal{I}$ *iff* $\mathcal{S} \sqsubseteq_{\mathbf{tmust}} \mathcal{I} \wedge \mathcal{S} \sqsubseteq_{\mathbf{tmay}} \mathcal{I}$

□

3.2.3 Test Automata

A test automaton is a timed automaton whose locations have been labeled with either a **pass**, **inconc**, or **fail** verdict, i.e., $\mathcal{V} : N \mapsto \{\mathbf{fail}, \mathbf{inconc}, \mathbf{pass}\}$ is the verdict assignment function.

Test execution is modeled by a parallel composition of the tester and the implementation using urgent communication. The semantics of this parallel composition was given in Definition 3.4 with the twist that observable actions now become internalized.

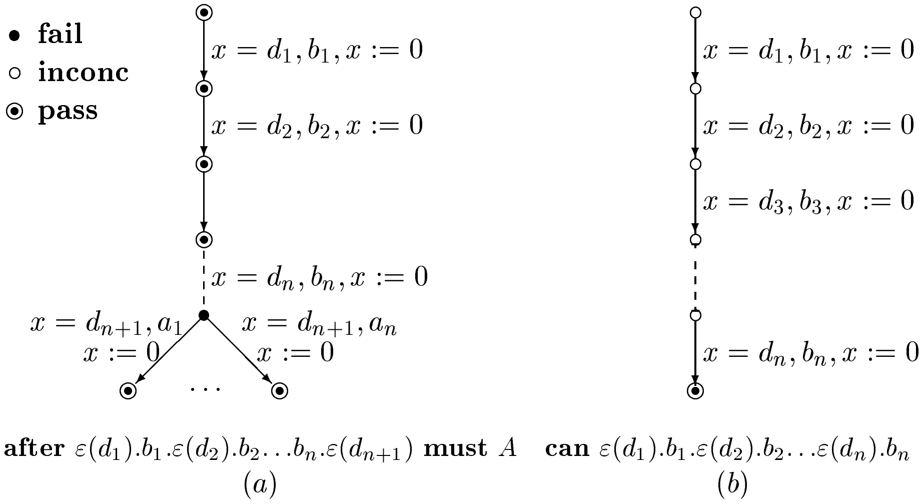
Definition 3.7 *Timed computations*:
Let $\mathcal{T} \parallel \mathcal{I}$ denote the parallel composition of a test $\mathcal{T}$ and implementation $\mathcal{I}$. A test configuration in the composed system is a pair of states $(s_{\mathrm{t}}, s_{\mathrm{i}})$ such that s_{t} is a state in the test, and s_{i} a state in the implementation.

1. A *timed computation* is a maximal sequence of configurations:
   $(s_{\mathrm{t}0}, s_{\mathrm{i}0}) \xrightarrow{\varepsilon(d_1)} (s_{\mathrm{t}1}, s_{\mathrm{i}1}) \xrightarrow{\tau} (s_{\mathrm{t}2}, s_{\mathrm{i}2}) \xrightarrow{\varepsilon(d_2)} \cdots (s_{\mathrm{t}k}, s_{\mathrm{i}k}) \xrightarrow{\tau} (s_{\mathrm{t}k+1}, s_{\mathrm{i}k+1}) \xrightarrow{\varepsilon(d_{k+1})} \cdots$
2. A test configuration is *deadlocked* if no further synchronizations are possible, i.e., there is some n such that for all $k > n$. $(s_{\mathrm{t}k}, s_{\mathrm{i}k}) \stackrel{\tau}{\not\to}$.
3. A computation is *successful* if there exists some $\langle \bar{l}, \bar{u} \rangle_{\mathrm{t}k}$ such that $\mathcal{V}(\bar{l}) =$ **pass**.

□

To obtain a more practical testing language using deterministic test automata with state based verdicts we, analogous to the untimed case in Section 2.2.5, redefine a successful computation to mean a computation that deadlocks in a location with a **pass** verdict. Similarly, a computation fails if it deadlocks in a **fail** location.

The structure of a test automaton $\mathcal{T}(t)$ for checking a timed must property t is illustrated in Figure 3.8a, and the automaton for a may test is shown in Figure 3.8b. The required timed trace is enforced by using a clock x that is reset with every action, and used in guards of the form $x = d$, where d is the delay after which the action is to be enabled.



after $\varepsilon(d_1).b_1.\varepsilon(d_2).b_2 \ldots b_n.\varepsilon(d_{n+1})$ **must** A
(a)

can $\varepsilon(d_1).b_1.\varepsilon(d_2).b_2 \ldots \varepsilon(d_n).b_n$
(b)

Figure 3.8. *Structure of timed test automata. Must test (a), and may test (b).*

3.2.4 Interpretation

This section exemplifies how the timed may and must preorders discriminate systems based on their timing behavior. Consider the series of small timed automata depicted in Figure 3.9.

$\mathcal{S}_1$ is able to do an a at all times. $\mathcal{I}_1$ is able to do an a at all times between one and two time units from the start, but is not able to do anything outside this interval. $\mathcal{S}_2$ can possibly perform an a at all times, but is definitely ready after two time units (note the non-urgent τ action and the invariant condition in location l_1). $\mathcal{I}_2$ is only ready after two time units. A comparison yields the following relations:

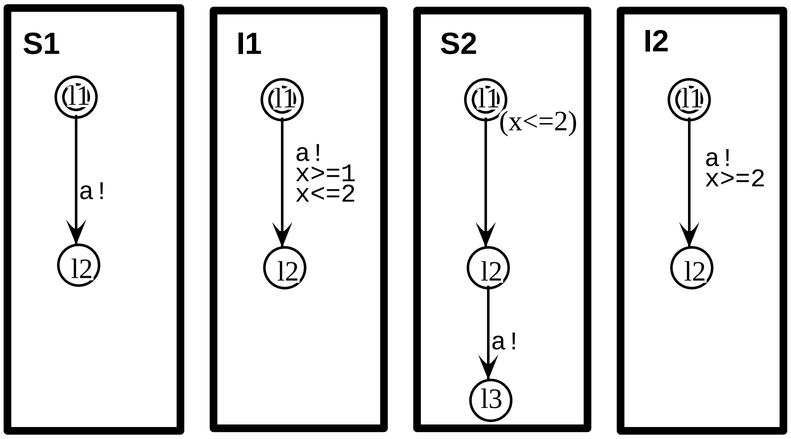


Figure 3.9. *Example timed automata.*

Relation	Distinguishing property
$\mathcal{S}_1 \not\sqsubseteq_{\mathbf{tmust}} \mathcal{I}_1$	**after** $\varepsilon(\frac{1}{2})$ **must** $\{a\}$
	after $\varepsilon(2\frac{1}{2})$ **must** $\{a\}$
$\mathcal{I}_1 \not\sqsubseteq_{\mathbf{tmust}} \mathcal{S}_1$	**after** $\varepsilon(\frac{1}{2}) \cdot a$ **must** $\emptyset$
	after $\varepsilon(2\frac{1}{2}) \cdot a$ **must** $\emptyset$
$\mathcal{I}_1 \sqsubseteq_{\mathbf{tmay}} \mathcal{S}_1$	
$\mathcal{S}_2 \not\sqsubseteq_{\mathbf{tmay}} \mathcal{I}_2$	**can** $\varepsilon(\frac{1}{2}) \cdot a$
$\mathcal{S}_2 \sqsubseteq_{\mathbf{tmust}} \mathcal{I}_2$	
$\mathcal{I}_2 \not\sqsubseteq_{\mathbf{tmust}} \mathcal{S}_2$	**after** $\varepsilon(\frac{1}{2})$ **must** $\emptyset$
$\mathcal{S}_2 \sqsubseteq_{\mathbf{tte}} \mathcal{S}_1$	

Thus, the timed must preorder requires that at all times when the specification (must) have some actions enabled (or disabled) the implementation must also have them enabled (or disabled). Similarly, the may preorder requires that all traces that are possible in the specification also are possible in the implementation. These requirements may be too strong in some cases. Consider for example the relation $\mathcal{S}_2 \not\sqsubseteq_{\mathbf{tmay}} \mathcal{I}_2$. One could think of $\mathcal{S}_2$ as a specification of a communication medium with a transmission delay of at most two time units. This requirement is satisfied by $\mathcal{I}_2$ which is rejected by the may preorder. Because S_2 is faster than I_2, we may consider using S_2 as an implementation of I_2. However this is rejected by the must preorder because S_2 (being faster) contains extended functionality that, as discussed in Section 2.2.4, may not be safe.

This observation suggests that the implementation relation should be chosen with great care. More significantly, it questions what timed preorder should be used in practice. Although the preorders defined here are obvious generalizations of the untimed testing preorders, many others could be suggested, e.g., time abstracted and faster-than relations. The specific choice depends on the application and the goal of testing. We discuss other proposed timed preorders in Section 6.2.1.

3.3 Timed Test Generation

One approach to timed testing is to generalize the direct untimed algorithm. Algorithm 3.11 gives such a procedure which is nearly identical to the untimed algorithm presented in Algorithm 2.27, except for the choice of delay in step 1. This delay decides when the tester should attempt to synchronize with the implementation. Like the untimed algorithm, the timed version uses three reduced sets of actions characterizing respectively the communications that must succeed (must sets), communications that possibly succeeds (can sets), and communications that must not succeed (refusals) after a give trace. These are defined in Definition 3.10.

Definition 3.10 *Timed must, can, and refusal sets*:
Let B be a set of states.

1. B **must** A *iff* $\forall s \in B$ **after** $\epsilon.\ \exists a \in A.\ s \overset{a}{\Rightarrow}\ \wedge\ (s \overset{a}{\not\to}$ *implies* $s \overset{\varepsilon(d)}{\not\longrightarrow})$
2. $\mathrm{Must}(B) =_{\mathrm{def}} \{A \subseteq \mathcal{O} \mid B\ \textbf{must}\ A\}$
3. $\mathrm{Can}(B) =_{\mathrm{def}} \{a \in \mathcal{O} \mid \exists s \in B.\ s \overset{a}{\Rightarrow}\} = \mathrm{Sort}(B)$
4. $\mathrm{Ref}(B) =_{\mathrm{def}} \{a \in \mathcal{O} \mid \forall s \in B.\ s \overset{a}{\not\Rightarrow}\} = \mathcal{O} - \mathrm{Sort}(B)$

□

Note that we abuse the notation for the summation and prefix of labeled transition systems earlier defined in Section 2.1.2 and use it to construct test automata. We justify this by observing that a test automata can be viewed as an LTS with edges labeled with triples (g, a, r) consisting of a guard, an action, and a set of clock resets. Also, we use the notation $\mathcal{V}(\mathcal{T})$ to assign the verdict to the initial state in $\mathcal{T}$.

It is important to realize that it is non-trivial to compute the sets of states B **after** $\varepsilon(d)$ and B **after** a with a dense time interpretation. The timing uncertainty arising from non-urgent actions causes these sets to be infinite, and can therefore not be computed directly by applying the semantic transition rules for timed automata. Instead they can by computed by symbolic execution using a non-trivial application of the symbolic methods that will be given in Section 4.3. Section 4.3.7 shows how to compute these sets.

Algorithm 3.11 *Generation of a timed test*:
Let B range over sets of states.
input: $\mathcal{S} \in \mathcal{L}_{ta}$
output: A test case $\mathcal{T} \in \mathcal{L}_{tta}$
init: TestGen($\mathcal{S}$ **after** ϵ)
TestGen$(B) =_{\mathrm{def}}$

1. Choose some delay $d \in \mathbb{R}_{\geq 0}$
2. Compute $B' = B$ **after** $\varepsilon(d)$
3. Compute $M = min_{\subseteq}(\mathrm{Must}(B'))$, $C = \mathrm{Sort}(B') - \bigcup_{A \in M} A$, and $R = \mathcal{O} - \mathrm{Sort}(B')$
4. Construct one of
   (a) $\mathcal{T}_A = \sum_{a \in A} (x = d, a, x := 0); \mathcal{T}_a, \quad A \in M, \quad \mathcal{V}(\mathcal{T}_A) = \textbf{fail}$
   (b) $\mathcal{T}_A = \sum_{a \in A} (x = d, a, x := 0); \mathcal{T}_a, \quad A = C, \quad \mathcal{V}(\mathcal{T}_A) = \textbf{inconc}$
   (c) $\mathcal{T}_A = \sum_{a \in A} (x = d, a, x := 0); nil, \quad A = R, \quad \mathcal{V}(\mathcal{T}_A) = \textbf{pass},$
   $\mathcal{V}(\mathcal{T}_A$ **after** $(x = d, a, x := 0)) = \textbf{fail}$
   (d) $\mathcal{T}_A = nil, \quad \mathcal{V}(\mathcal{T}_A) = \textbf{pass}, \quad$ *if* $\mathrm{Sort}(B') = \emptyset$
5. Construct a $\mathcal{T}_a$ in case (4a) or (4b) by calling TestGen(B' **after** a)

Another problem of the algorithm is that it does not provide a strategy for choosing delays. Some possibilities are:

Random Sampling: The delay between two synchronization attempts are chosen randomly, or according to some specific probability distribution.

All Delays: All possible delays are chosen. This is clearly possible for discrete time interpretations, and it may even be feasible if the number of time instances to be checked is reasonably small. Handling dense time in this way is more problematic. One would expect this to be impossible because there is an infinite number of delays to choose from, but recent research [105] has shown that under certain assumptions only a finite number of tests is required to achieve exhaustive testing. It is therefore possible, at least in principle, to choose a sufficiently small delay to cover all relevant synchronization times. We shall discuss the foundation of this observation in Section 6.2.

Minimum Delays: When a tester applies this strategy, it offers the actions as fast as possible, i.e., the tester only delays when this is required by the specification, or when this is required to enable an untested edge or action. This strategy will stress the implementation and thereby check whether it is fast enough under different input sequences.

Periodic Sampling: A fixed delay is chosen every time. The tester can therefore be viewed as a process that periodically samples the behavior of the implementation. A big advantage of periodic sampling is that it suggests a method of incremental testing. Initially, a very coarse sampling period is chosen. When a sufficient number of tests with sufficient duration have been executed successfully, a smaller interval can be tried. The sampling interval can be refined for as long as there are resources available for testing. Further, it may be possible to find a strategy for adjusting the sampling frequency dynamically such that e.g., a higher frequence is chosen when many things happen in the specification or implementation in a short amount of time.

However, we believe that these strategies are imperfect because tests are selected in an *ad hoc* fashion, and not systematically from a coverage criterion. The resulting coverage can only be measured, and by generating a lot of test one may hope to obtain sufficient coverage, but this is likely to include a lot of redundancy.

3.4 State Based Selection

It is infeasible to generate and execute a test suite that covers every timed may and must property satisfied by the specification. It is therefore necessary to select only a small subset of the possible tests. This selection should be based on a coverage criterion. In the following we shall propose to base this selection on certain properties of the reachable states.

3.4.1 State Space Partitioning

Intuitively, we shall say that a state has been *covered* if some observations of the system's behavior have been made in that state. We shall also say that a set of states are covered if at least one of them is, and if the states enjoys a common property that makes us believe that only one needs to be tested.

The hypothesis is that it is more important to test inequivalent states than it is to test equivalent states multiple times.

A test selection strategy can now be formed by inspecting the states of the specification and only make observations of one or a few representatives from each set of equivalent states.

Formally, we propose to partition the state space and ensure that the generated test suite contains a set of tests that makes the required observations of a representative from each partition. There are many possible ways of partitioning the same state space, and we therefore propose a common framework that captures our notion of state based selection.

Non-determinism makes it is impossible to predict and control which state the specification occupies after having performed a given trace. We therefore find it convenient to view the system state not as a single state, but as the set of *possible* states the specification can occupy after having performed a given trace. In consequence, a *partition* $Q \subseteq 2^S$ becomes a *set of sets of possible states*. A *state partitioning* $\mathcal{Q}$ is a set of partitions, see Figure 3.12.

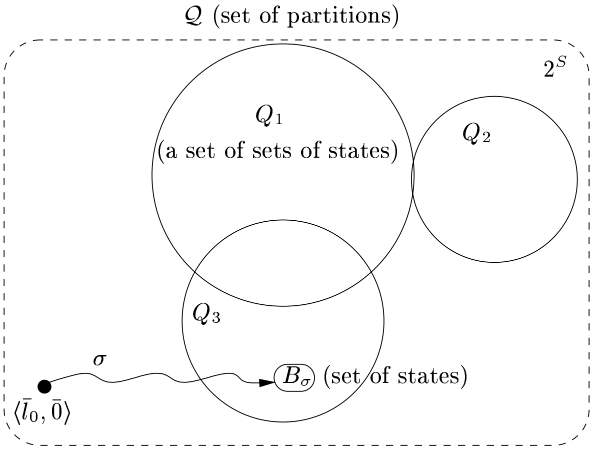

$\mathcal{Q}$ (set of partitions)
2^S
Q_1
(a set of sets of states)
Q_2
Q_3
σ
B_σ (set of states)
$\langle \bar{l}_0, \bar{0} \rangle$

Figure 3.12. *State Space Partitioning*

Let the triple $\mathcal{S}\langle \mathcal{Q}, Obs \rangle$ denote that specification $\mathcal{S}$ is covered such that the observations in Obs are made in each partition in $\mathcal{Q}$ of the specification.

Let $\sigma \in \mathcal{O}_\varepsilon^*$ be a timed trace, and let $B_\sigma = \mathcal{S}$ **after** σ denote the set of states reachable in the specification after σ. We assume that tests consists of (or can be decomposed to) a trace followed by an observation, and we let $\sigma \circ o$ denote such a test. We further assume a relation $\mathcal{S} \models \sigma \circ o$ that determines if observation $o \in Obs$ can be made in the states reached after σ, i.e., if the specification passes the test corresponding to $\sigma \circ o$.

Definition 3.13 *Specification coverage*:

$$\begin{array}{lll} \Pi \; \textit{covers}\; \mathcal{S}\langle \mathcal{Q}, \mathit{Obs}\rangle & \textit{iff} & \forall Q \in \mathcal{Q}. \\ (a) \; \exists \sigma \circ o \in \Pi. & B_\sigma \in Q \; \wedge & \\ (b) & \forall o' \in \{o'' \in \mathit{Obs} \mid \mathcal{S} \models \sigma \circ o''\}.\; \exists \sigma \circ o' \in \Pi & \end{array}$$

□

The coverage criterion in Definition 3.13 requires that a) the test suite contains a trace leading to all (reachable) partitions $Q \in \mathcal{Q}$, and b) that the test suite contains a test for each relevant observation of Obs in the representative states after the chosen trace σ. Because the criterion only requires one trace per partition, the state partitioning also equalizes all traces whose destination states are contained in the same partition.

Instantiations of the framework must define the desired partitioning $\mathcal{Q}$, the desired observations Obs, and the $\circ$ and $\models$ relations.

3.4.2 Instantiations

We shall now instantiate the framework, and propose a number of selection strategies by varying the coarseness of the state partitioning. If fact, they form a subsumption hierarchy where the coverage of the finer partitioning implies coverage of the coarser ones. The given instances and their subsumption relation are:

$$\begin{array}{c} \text{Region Partitioning} \supseteq \\ \text{Stable Edge Set Partitioning} \supseteq \\ \text{Edge Set Partitioning} \supseteq \\ \text{Action Partitioning} \end{array}$$

Let $\mathcal{S} = \langle N, l_0, I, E\rangle$ be a timed automaton. We use properties of the following type as observations: **after** ϵ **must** A, **can** a, and **after** a **must** $\emptyset$. Let $\mathbb{O}$ be the set of all such observations over the set of observable actions $\mathcal{O}$. This is a natural class of observations given the timed may and must preorders defined in Section 3.2. The decomposition of tests to traces and observations is done as follows:

$$\begin{array}{l} \sigma \circ \textbf{after}\; \epsilon\; \textbf{must}\; A = \textbf{after}\; \sigma\; \textbf{must}\; A \\ \sigma \circ \textbf{after}\; a\; \textbf{must}\; \emptyset = \textbf{after}\; \sigma \cdot a\; \textbf{must}\; \emptyset \\ \sigma \circ \textbf{can}\; a = \textbf{can}\; \sigma \cdot a \end{array}$$

This direct correspondence between the pairs $\sigma \circ o$ and tests allows us to adopt the satisfaction of timed may and must properties from Definition 3.5 as the definition of $\models$ in the selection framework.

Region Partitioning

A *region* is a symbolic representation of a set of clock valuations, or formally, an equivalence class on clock valuations induced by the equivalence relation defined in Definition 3.14. The region concept was proposed by Alur and Dill in [9, 7] as a vehicle for studying decision procedures for timed automata, and has also been applied to model checking.

Definition 3.14 *Region Equivalence [9]*:
Let X be a set of clocks, and let $\bar{u}, \bar{u}'$ be clock valuations. Two clock valuations are region equivalent, written $\bar{u} \doteq_\rho \bar{u}'$, *iff* $\forall x, y \in X$

1. $\lfloor \bar{u}(x) \rfloor = \lfloor \bar{u}'(x) \rfloor$ or $\bar{u}(x) > c_x$ and $\bar{u}'(x) > c_x$
2. if $\bar{u}(x) \leq c_x$ and $\bar{u}(y) \leq c_y$ then $(\mathit{frac}(\bar{u}(x)) \leq \mathit{frac}(\bar{u}(y))$ *iff* $\mathit{frac}(\bar{u}'(x)) \leq \mathit{frac}(\bar{u}'(y)))$
3. if $\bar{u}(x) \leq c_x$ then $(\mathit{frac}(\bar{u}(x)) = 0$ *iff* $\mathit{frac}(\bar{u}'(x)) = 0)$

□

A clock value is divided into two parts, the integral part $\lfloor \bar{u}(x) \rfloor$, and the fractional part $\mathit{frac}(\bar{u}(x))$. The integral part is the largest integer not larger than $\bar{u}(x)$. Two clock valuations are equivalent if the clocks agree on their integral parts, and if all fractional parts are 0 or if they have the same ordering on the fractional parts. A clock can be assigned the designated value ∞ when it exceeds the fixed constant c_x. Beyond c_x the precise value of x is irrelevant with respect to the evaluation of the guards in the automaton. c_x equals the largest constant used in guards on x when the only assignments to x are $x := 0$.

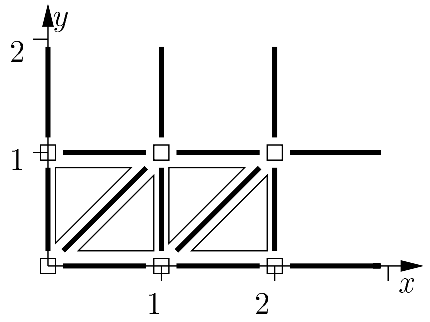


Figure 3.15. *The regions (boldfaced line segments, corner points, and interior triangles) induced by two clocks x, y and maximum domains $c_x = 2$ and $c_y = 1$. There are 28 regions in this example.*

Figure 3.15 visualizes the region concept. The key observation is that no guard (defined from the syntax in Definition 3.3) can distinguish between two clock valuations in the same region. A timed automaton can therefore be analyzed by picking a single representative clock valuation from each region. A *region state* is a pair consisting of a location vector and a region. The reachable state space of a timed automaton can be computed from the initial region state, and by recursively computing its successor regions.

The number of regions in a timed automata with $|X|$ clocks is bounded by $|X|! \cdot 2^{|X|} \cdot \Pi_{x \in X}(2c_x + 2)$. It can thus be noted that the number of regions depends exponentially on both the number of clocks and the clock constants. The number of region states is bounded by multiplying the number of regions with the number of locations.

Region coverage is defined in Definition 3.16. Because of non-determinism, the possible states after a trace will in general belong to a set of region states.

Definition 3.16 *Region Coverage* $\mathcal{S}\langle \mathcal{Q}, Obs \rangle$:

Let $\langle l, \bar{u} \rangle$ be a state, and let $\rho(\bar{u})$ be the region containing $\bar{u}$. The region state of $\langle l, \bar{u} \rangle$, written $\rho(\langle l, \bar{u} \rangle)$ is the pair $\langle l, \rho(\bar{u}) \rangle$. The region states of a set of states B, denoted $\rho(B) = \{\rho(\langle l, \bar{u} \rangle) \mid \langle l, \bar{u} \rangle \in B\}$. Similarly, let $\rho(\mathcal{S})$ be all region states of the specification.

1. $Obs = \mathbb{O}$
2. $Q_R = \{B_\sigma \mid \exists \sigma \in Tr(\mathcal{S}).\ \rho(B_\sigma) = R\}$
3. $\mathcal{Q} = \{Q_R \mid R \subseteq \rho(\mathcal{S}),\ Q_R \neq \emptyset\}$

□

Stable Edge Set Coverage

The stable edge set coverage in Definition 3.17 regards two sets of states as identical if they consists of the same locations and enable precisely the same set of edges. The idea behind this partitioning is primarily to capture different *deadlock* properties. When the set of enabled edges change, the enabled and disabled actions also potentially change, and so do the actions that must be accepted and refused. A secondary motivation is that when the enabled edges change because time progresses, some timer would have expired in the implementation, and therefore the possible states it can occupy has changed, and they should be tested by their own test case.

Definition 3.17 *Stable Edge Set Coverage* $\mathcal{S}\langle\mathcal{Q}, Obs\rangle$:
Let $E(B) = \{l \xrightarrow{g,a,r} l' \in E \mid \exists\langle l, \bar{u}\rangle \in B.\ g(\bar{u})\}$, and $L(B) = \{l \mid \exists\langle l, \bar{u}\rangle \in B\}$.

1. $Obs = \mathbb{O}$
2. $Q_{E',L} = \{B_\sigma \mid \exists\sigma \in Tr(\mathcal{S}).\ E(B_\sigma) = E' \ \wedge\ L(B_\sigma) = L\}$
3. $\mathcal{Q} = \{Q_{E',L} \mid E' \subseteq E,\ L \subseteq N,\ Q_{E',L} \neq \emptyset\}$

□

The definition also requires that the same set of locations are reached. This condition ensures that the situations, where no actions are enabled in the specification, will be tested for all location configurations, i.e., that the implementation also has no actions enabled. Checking this is important for the timed must preorder.

Edge Coverage

In the edge coverage criterion in Definition 3.18, there is a partition for each edge; sets of states are considered equivalent if they enable the same edge. This partitioning is motivated by white box testing of sequential programs, where the goal is a structural coverage of the program such that every statement is executed at least once. Edge coverage ensures a structural coverage of the specification.

Definition 3.18 *Edge Coverage* $\mathcal{S}\langle\mathcal{Q}, Obs\rangle$:
Let $enabled(e, B)$ be a predicate that is true if B enables the edge $e = l \xrightarrow{g,a,r} l'$, i.e., if $\exists\langle l, \bar{u}\rangle \in B.\ g(\bar{u})$

1. $Obs = \mathbb{O}$
2. $Q_e = \{B_\sigma \mid \exists\sigma \in Tr(\mathcal{S}).\ enabled(e, B_\sigma)\}$
3. $\mathcal{Q} = \{Q_e \mid e \in E,\ Q_e \neq \emptyset\}$

□

Action Coverage

The final partitioning we shall propose here, see Definition 3.19, has the rather modest goal of ensuring that the test suite will have the ability to observe every observable action. In *connectivity testing* [46], systems are viewed as a composition of hardware and embedded software that both are assumed to be correct, i.e., has been proven or verified. Implementation faults then consists of missing connections between software actions and the external system interface. Thus, if every action is observed it can be concluded that no connections are missing.

Definition 3.19 *Action Coverage* $\mathcal{S}\langle \mathcal{Q}, Obs\rangle$:

1. $Obs = \mathbb{O}$
2. $Q_a = \{B_\sigma \mid \exists \sigma \in Tr(\mathcal{S}).\ \exists \langle l, \bar{u}\rangle \in B_\sigma.\ \langle l, \bar{u}\rangle \xrightarrow{a}\}$
3. $\mathcal{Q} = \{Q_a \mid a \in \mathcal{O},\ Q_a \neq \emptyset\}$

□

3.4.3 Choice of Coverage Criterion

A coverage criterion can be used in two ways. One is as a posteriori metric of what has been covered after test execution. Our goal has been an a priori application to assist test generation. In this situation, there are two main factors that influence the choice of coverage metric. One factor is the type and amount of required observations. The finer partitions the higher error detection power, but also the more tests and higher cost. The second main factor is the computer resources (CPU time and memory) it will take to construct a test suite with the desired coverage, and related, the difficulty with which algorithms can be devised and implemented in tools.

We choose to examine stable edge set coverage further for the following reasons:

- Region partitioning is too fine grained for most practical specifications. It results in too many tests, and is expensive to compute in terms of space and time.
- Action coverage only requires observation of each action, and therefore essentially ignores timing and deadlock situations. For real-time blackbox conformance testing this is insufficient.
- The term stable edge set suggests that the system is in a stable state in which it enables and disables no new actions. When the edge set of the specification changes by passage of time (thereby enabling or disabling actions) or execution of internal actions, it indicates that a *timeout* could have occurred in an implementation, and that it therefore should be checked that the implementation ends up in state that conforms to the specification.

- All representatives in such a partition are equivalent with respect to our observation set $\mathbb{O}$: For any equivalent B_σ, $B_{\sigma'}$ then also $\forall o \in \mathbb{O}.\ B_\sigma \models o$ *iff* $B_{\sigma'} \models o$. See also Proposition 4.9 in Section 4.2.2. Stable edge set partitioning is therefore a good match for the timed may and must properties. Edge coverage does not have this property.

3.5 Summary

This chapter instantiated the ingredients in a formally based approach to testing. We defined our specification language timed automata formally as a dense timed LTS. timed automata can be viewed as the communicating state machine model proposed in Chapter 2 extended with shared clock variables. No proposals for implementation relations for dense timed system exist. We therefore defined a modest timed generalization of Hennessy's may and must preorders, and used this as a basis for our test generation algorithm. Our discussion on the interpretation of the implementation relation showed that the exact variant should be chosen with great care.

We stated an algorithm based on a direct interpretation of the timed automata specification. This algorithm avoids the state explosion problem, but has the serious disadvantage that it does not enable one to systematically select test cases from a selection criterion. To aid test selection we proposed to partition the spate space and cover each partition with a specific set of test cases. We examined four specific partitioning strategies, the region-, stable edge set-, edge-, and action-partitioning, and argued that stable edge set partitioning formed a good compromise between testing thoroughness and cost, and matched the needs of the timed must preorder perfectly.

Given a specification language, a test language, and a notion of coverage, the problem of developing algorithms for generating the required test suite can be addressed. In the next chapter we show how a stable edge set covering test suite can be generated for a restricted class of timed automata. This will be achieved by constructing a partition graph and generating tests from this. Once constructed, it becomes easier to generate tests and obtain the desired coverage. The resulting graph can be viewed as a timed version of the success graph.

Chapter 4

Symbolic Test Generation

This chapter develops a technique for automatically generating and selecting timed Hennessy tests. Timed Hennessy tests has the potential of detecting important timing and deadlock faults of the implementation. The technique constructs a graph of partitions based on the stable edge set partitioning, and then employs a symbolic constraint solving technique to compute the reachable parts of the partition graph. From this graph, a covering test suite can be obtained.

We demonstrate our technique on a a restricted class of timed automata, called event recording automata, which will be defined in Section 4.1. The technique is applicable to deterministic timed automata as well. The unrestricted timed automata model presented in the previous chapter turns out to be problematic to analyze for both principal and technical reasons. The principal problems are caused by undecidability of language inclusion, and more importantly by the fact that timed automata cannot be determinized, and are not closed under complement. The technical problems are related to computing the desired partitions and their reachable parts, and specifically, representing and manipulating unions of concave sets of clock valuations, and maintaining a common time base when different clocks were reset along non-deterministic choices.

Section 4.2 shows how to construct the partition graph for event recording automata, and proposes a test generation algorithm based thereon. The symbolic techniques applied in the algorithm are derived from model checking of real-time systems by means of reachability analysis. These techniques offer a tool box of efficient algorithms and compact data structures for representing and manipulating convex sets of clock valuations. Their definition and application to the partition graph is presented in Section 4.3. To ensure termination of the reachability analysis, we employ the notion of extrapolated symbolic states in Section 4.4. In the same vein, we propose a set of pragmatic termination criteria that can be used to limit the size of the reachability graph when this cannot be entirely constructed.

The above algorithm generates individual tests, one for each partition. These can be composed to obtain fewer tests. Section 4.5 outlines a procedure for composing timed tests from the reachability graph.

We have implemented these algorithms in a prototype tool, called RTCAT. Section 4.6 presentsits facilities, and some implementation remarks. Finally, Section 4.7 concludes this chapter.

4.1 Event Recording Automata

Two important undecidability results from the theoretical work on timed languages described by timed automata are that 1) a non-deterministic timed automaton cannot be converted into a deterministic (trace) equivalent timed automaton, and 2) trace (language) inclusion between two non-deterministic timed automata is undecidable [10, 118]. Thus, unlike the untimed case, deterministic and non-deterministic automata are not equally expressive. The Event Recording Automata model (ERA) was proposed by Alur, Fix, and Henzinger in [10] as a determinizable subclass of timed automata that enjoys both properties. On the other hand, surprisingly, reachability is decidable for timed automata.

Further, it is known that timed automata allowing internal actions accept more languages than timed automata without [114], and consequently, edges with internal actions cannot in general be removed, in contrast to the untimed case. Specifically, edges with an internal action that resets clocks and that lies on a cycle cannot be removed [114].

Although our concern here is to derive finite length tests, and not directly to decide language inclusion between two explicitly given timed automata, we have seen in Chapter 2 that determinization and removal of internal actions plays a central role in untimed test generation algorithms for non-deterministic specifications. This property allowed us to construct a finite success graph from which it was easy to systematically generate tests, and thereby obtain a coverage of the deadlock properties of the specification.

To obtain a similar data structure for timed systems, we have decided to steer clear of these potential problems and apply our ideas to ERA. A further benefit is that the required symbolic analysis is reasonably simple for this model. ERA thus turn out to be an excellent vehicle for developing our techniques. We comment in Chapter 5 on the impact their restrictions might have on the practical applicability of our techniques.

4.1.1 Definition of Event Recording Automata

Like a timed automaton, an ERA has a set of clocks that can be used in guards on actions and be reset when an action is taken. In ERA however, a unique clock is associated with each action. Whenever an action is executed, the associated clock is automatically reset. No additional resets are permitted. Thus, unlike ordinary timed automata where clocks can be assigned to at will, ERA maintains a strict correspondence between actions and clocks. This ensures that the environment is in control over which clocks are reset, and this in turn gives the determinizability property [10]. The *event clock* x_a associated with action a, thus *records* the amount of time passed since the last occurrence of a. The formal definition is given in Definition 4.1.

Definition 4.1 *Event Recording Automata*:
An ERA specification is a network of timed automata as defined in Definitions 3.3 and 3.4 with the following restrictions:

1. For each action a there is an associated clock x_a, called the event clock of a. Thus, the set of clocks is $X = \{x_a \mid a \in \mathcal{A}\}$.

2. The clock x_a is automatically reset when action a is executed. Thus, every edge labeled with action a is also implicitly labeled with the assignment $x_a := 0$. No further clock assignments are permitted.

3. All actions are observable and urgent. This implies that neither internal actions (τ labeled edges), nor internal synchronizations between the individual automata in the network is permitted. That is, $\mathcal{A} = \mathcal{O} = \mathcal{U}$. Note that the original definition of ERA does not distinguish between urgent and non-urgent actions. ERA was developed with the purpose of studying timed languages, and not directly with modeling and testing of real-time systems. Therefore it was less important to distinguish whether a trace or an action can definitely, or only possibly occur. But for testing this distinction is essential.

4. No location invariants is permitted, i.e., for all locations l, $I(l) = tt$. □

Our test generator also allows shared integer variables, but for simplicity we omit their description in our presentation. Semantically, the integer variables will be treated as part of the control location. Integers can be used in guards, and can be assigned during a transition. Let i, j range over integer variables. We permit guards on integers of the forms $i \sim c$ and $i - j \sim c$. We permit simple integer assignments of the forms $i := c$ and $i := j$ *op* c with *op* being addition, subtraction, multiplication, or division. See also the formal description of communicating state machines with integer variables in Section 2.1.3.

Although the parallel composition in the ERA model is very restricted, the automata need not operate completely independent: they share clocks and integer variables. Therefore, an action in one automaton may enable or disable actions in the others via assignments of shared integers and resets of shared clocks. Even this restricted parallel composition is very useful; it allows convenient specification of systems that should be understood as nearly independent concurrent components, and most required synchronizations can be achieved using shared integers.

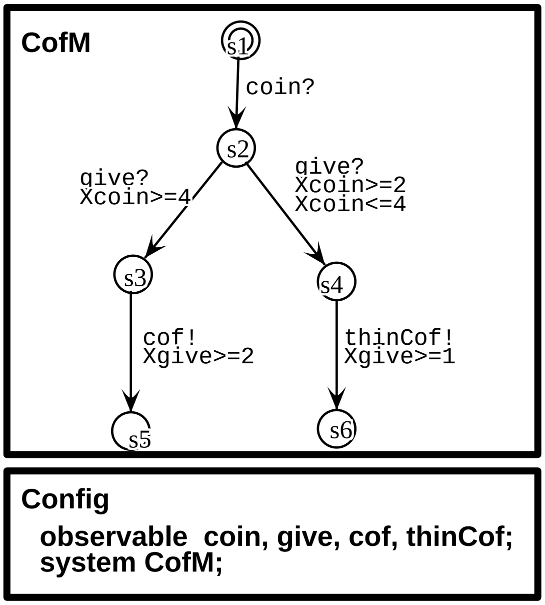


Figure 4.2. *ERA specification of a coffee vending machine.*

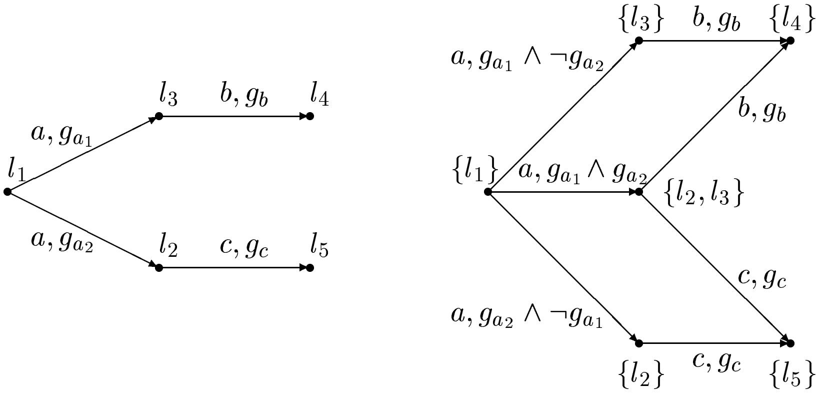
Figure 4.2 shows an example of a small ERA. It models a coffee vending machine built for impatient users such as busy researchers. When the user has inserted a coin (`coin`), he must press the give button (`give`) to indicate his eagerness to get a drink. If he is very eager, he presses `give` soon after inserting the coin, and the vending machine outputs thin coffee (`thinCof`); apparently, there is insufficient time to brew good coffee. If he waits more than four time units, he is certain to get good coffee (`cof`). If he presses `give` after exactly four time units, the outcome is non-deterministic. Note that clock resets are performed automatically, and are therefore implicit in ERA models.

4.1.2 Determinization of Event Recording Automata

An essential feature of ERA models is that they can be determinized. In a *deterministic* automaton the choice of the next edge to be taken is uniquely determined by the automaton's current location, the input action, and the time the input event is offered.

The determinization procedure for ERA is given by [10], and is a conceptually simple extension of the subset construction used in the untimed case, only now the guards must be taken into account. We explain the technique in the following.

The initial location of the deterministic ERA is $L_0 = \{\bar{l}_0\}$. Assume now that the deterministic automaton occupies the locations L. Let E_a denote the set of edges starting in L, and labeled with action a. For every non-empty subset E'_a of E_a, the deterministic automaton contains an edge from L to the target locations of E'_a, labeled with action a and guard g. The guard g is constructed by conjuncting the guards in E'_a, and by conjuncting the result with the conjunction of the negated guards in $E_a - E'_a$. As a result, the guards of a edges from the same location L in the deterministic automaton are mutually exclusive, and the successor location is uniquely determined by the action, and the time it occurs.



(a) (b)

Figure 4.3. *Illustration of the determinization principle. Non-deterministic ERA (a), and equivalent deterministic ERA (b).*

Figure 4.3 shows the determinization principle. Observe how the a transitions from l_1 are *sorted* such that either both are enabled, or only one of them is.

Determinization of the coffee machine from Figure 4.2 yields the deterministic coffee machine illustrated in Figure 4.4.

Because the same clock is reset along every a edge, a state of the determinized automaton can be represented by a pair consisting of a set of locations and a single clock valuation: $\langle L, \bar{u}\rangle$. In an unrestricted timed automata, it would have to be represented by a set of configurations: $\{\langle \bar{l}_1, \bar{u}_1\rangle, \langle \bar{l}_2, \bar{u}_2\rangle, \ldots, \langle \bar{l}_n, \bar{u}_n\rangle\}$.

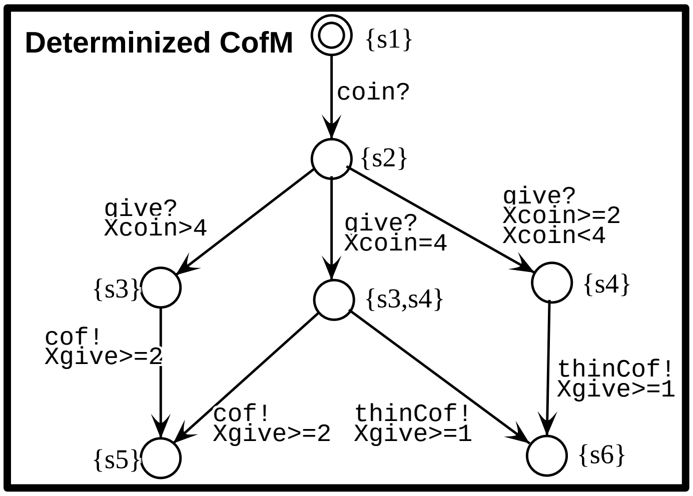


Figure 4.4. *Determinized coffee vending machine.*

4.2 Test Generation from Event Recording Automata

In this section we introduce our techniques for generating tests from ERA. The techniques will be presented top down. An overall algorithm specifies the main steps involved in generating a test case, and then follows a description of each step in further detail. The first main step is to compute the stable edge set partitions for ERA. This step is described later in this section. The remaining steps of the algorithm, which involve application of the symbolic reachability techniques, are described in succeeding sections in this chapter.

4.2.1 Overall Algorithm

Algorithm 4.5 presents the main steps of our generation procedure. According to our coverage strategy, it must be ensured that every reachable partition is tested by executing some prefix trace leading to the partition, followed by the required observations thereof.

Algorithm 4.5 *Overall Test Case Generation Algorithm*:
input: ERA specification $\mathcal{S}$.
output: A covering set of relevant timed Hennessy properties.

```
1. Compute $\mathcal{S}_p$ = Stable Edge Set Partition Graph($\mathcal{S}$).

2. Compute $\mathcal{S}_r$ = Reachability($\mathcal{S}_p$).

3. Label every $[L, z/p] \in \mathcal{S}_r$ with the sets $M$, $C$, $R$.

4. $Tested := \emptyset$

5. Traverse $\mathcal{S}_r$. For each $[L, z/p]$ in $\mathcal{S}_r$:

   if $\not\exists z'. \ [L, z'/p] \in Tested$ then
   $Tested := Tested \cup \{[L, z/p]\}$, and enumerate tests:

   (a) Choose $\langle \bar{l}, \bar{u} \rangle \in [L, z/p]$
   (b) Compute a concrete trace $\sigma$ from $\langle \bar{l}_0, \bar{0} \rangle$ to $\langle \bar{l}, \bar{u} \rangle$.
   (c) Make Test Cases:
       if $A \in M([L, p])$ then after $\sigma$ must $A$ is a relevant test.

       if $a \in C([L, p])$ then can $\sigma \cdot a$ is a relevant test.

       if $a \in R([L, p])$ then after $\sigma \cdot a$ must $\emptyset$ is a relevant test.
```

The result of step 1 is a graph that we refer to as the *partition graph*. Its nodes contain partitions, and its edges are labeled with an observable action. An edge represents the possibility of executing an action in a state in the source partition, waiting some amount of time, and thereby entering a state in the target partition. The partitioning algorithm implicitly determinizes the specification such that the partition graph is a deterministic representation of the state space. A partition will be represented by a pair $[L, p]$, where L is a set of location vectors in the determinized automaton, and p is boolean combination of the inequations describing the clock constraints that must hold for that partition, or equivalently, the set of clock valuations satisfying the constraints. A partition $[L, p]$ is thus the set of states $\{\langle \bar{l}, \bar{u} \rangle \mid \bar{l} \in L, \bar{u} \in p\}$.

Step 2 performs symbolic reachability analysis of the partition graph. This is done for two reasons. First, computing the reachable partitions and reachable parts thereof, gives the states that can be chosen for testing, and to which a trace exists. Also, to compute this trace some of the nodes in the partition graph may need to be traversed a number of times. Second, the reachability analysis gives a termination criterion; when no further states can be reached, test generation can be stopped because all reachable partitions in the specification have been covered. Steps 1 and 2 could be interleaved such that only the reachable parts of the partition graph are constructed. However, we consider this to be an implementation technique that should not be described in this abstract overall algorithm.

The result of step 2 is a *symbolic reachability graph* representing the symbolic execution of the specification and the resulting reachable state space. Nodes in this graph consists of symbolic states $[L, z/p]$ where L is a set of location vectors, and where z is a constraint characterizing a set of reachable clock vectors also in p. Specifically, z is a subset of the states contained in partition p. The edges are labeled with the observable action linking two symbolic states.

The nodes in the reachability graph are labeled with three pieces of information in step 3: The minimized set of must sets M holding in that symbolic state, the possible actions C in the state not contained in M, and the actions R that must be refused.

Step 4 initializes an empty set that contains the symbolic states from which tests have be generated so far.

Step 5 contains the generation process itself. Note that the same partition may be traversed many times during forward reachability analysis. The coverage criterion only requires that tests are generated from one point of the partition. Algorithm 4.5 therefore only generates test for the first symbolic state that reaches a given partition, and uses the set *Tested* to ignore subsequent passes over the same partition. This ensures that all the may, must, and refusal properties are only generated once per partition, and thus reduces the number of produced test cases. Other strategies such as testing all reached symbolic states, or only testing certain designated locations deemed critical by the user, can easily be implemented.

If a particular point in the symbolic state is of interest, such as an extreme value, this must be computed (step 5a). When a point has been chosen, a concrete trace leading to it from the initial state is computed (step 5b). This involves back propagation of the constraints characterizing the test point (or partition, if no particular point is required) back along the symbolic path used to reach the partition, and then choosing the specific delays in the timed trace.

Finally, in step 5c, a test case can be generated for each of the may, must, and refusal properties holding in that symbolic state, and can finally be output as a test automaton in whatever output format is desired.

It should be noted that the above algorithm generates individual timed Hennessy properties. In general, it is desirable to compose several of these properties into fewer more complex test cases in the form of trees, as indicated in Section 2.2.5. To facilitate test composition, the traversal and construction of test cases in step 5 should be done differently. A procedure will be proposed in Section 4.5. Furthermore, it should be noted that steps 1 and 2 can be interleaved. Because not all partitions may be reachable, interleaving its construction with reachability analysis could result in a smaller graph and less memory use during its construction.

4.2.2 State Partitioning

According to our coverage criterion, there should at least be one test point for each partition as defined by the same edges being enabled. The first step is therefore to compute the constraints that characterize each such partition. Because we need to take care of potential non-determinism, the partition constraints are computed from a set of location vectors rather than a single vector. For such set of locations the constraints that precisely characterizes the enabledness of a given subset (and only these) of outgoing edges can be computed. This is done formally in Definition 4.6 (1), where the constraints are computed by conjuncting the guards that must be enabled and by conjuncting the negated guards of edges that must be disabled.

Definition 4.6 *State partitioning $P(L)$*:
Let L be a set of location vectors, $E(L)$ the set of edges starting in a location vector in L, E a set of edges, and $\Gamma(E) = \{g \mid \bar{l} \xrightarrow{g,a} \bar{l}' \in E\}$. Recall from Definition 3.3 that $G(X)$ denotes the set of possible guards that can be generated using conjunctions of bounds on differences of clocks in X.

1. $P(L) = \{P_E \mid E \in 2^{E(L)}\}$, where $P_E = \displaystyle\bigwedge_{g\in\Gamma(E)} g \quad\wedge \bigwedge_{g\in\Gamma(E(L)-E)} \neg g$

   P_E, expressed as a disjunction of conjunctions of clock inequations only, such that $\bigvee_i \bigwedge_j \gamma_{ij} \iff P_E$, is said to be on *disjunctive normal form.* Recall that a guard is a conjunction of bounds on clock differences, γ. Therefore, P_E on disjunctive normal form can be written as $\bigvee_i g_i$. Let $\mathrm{DNF}(P_E) = \{g \mid g \in G(X)\}$ such that $\bigvee_{g\in\mathrm{DNF}(P_E)} \iff P_E$.

2. $P_{\mathrm{dnf}}(L) = \displaystyle\bigcup_{P_E\in P(L)} \mathrm{DNF}(P_E)$

□

The resulting partitions suffer from two serious drawbacks both caused by the use of negation of guards that are themselves conjunctions of simple constraints as defined in Definition 3.3. First, we would like the partitions to form contiguous convex polyhedra in the $|X|$-dimensional space such that it is possible to delay $\bar{u}$ inside p without encountering a change in the set of enabled edges. That is, for any $\bar{u} \in p$ there is a delay d such that for all $d' < d$ it holds that $\bar{u} + d \in p$, and for all $d' \geq d$ it holds that $\bar{u} + d' \notin p$. This can be ensured by allowing only convex polyhedra, which in turn can be obtained using conjunction only. Second, if the partitions form convex polyhedra, the existing efficient techniques for forward and backward reachability analysis can be reused.

We therefore use the slightly finer, but convex partitions, defined in Definition 4.6 (2) where each partition is described using only conjunction. This is easily accomplished by rewriting the original constraints P_E to their disjunctive normal form, and by treating each disjunct as its own partition.

We can view the state space of the specification as a graph of partitions. Specifically, a partition can be represented by the pair $[L, p]$, where L is a set of locations, and p is the constraints characterizing its set of enabled edges. A node in this graph contains a partition. An edge between two nodes are labeled with an observable action, and represents the possibility of executing an action in a state in the source node, waiting some amount of time, and thereby entering a state in the target node. The partition graph is defined formally in Definition 4.7.

Definition 4.7 *Partition Graph*:
The nodes and edges are defined inductively as:

1. The set $\{[L_0, p] \mid L_0 = \{\bar{l}_0\} \text{ and } p \in P_{\mathrm{dnf}}(L_0)\}$ are nodes.
2. if $[L, p]$ is a node, so is $[L_a, p_a]$, and $[L, p] \xrightarrow{a} [L_a, p_a]$ is an edge, where $L_a = \{\bar{l}' \mid \exists \bar{l} \in L.\ \bar{l} \xrightarrow{g,a} \bar{l}' \ \wedge\ (g \wedge p \neq \emptyset)\}$, and $p_a \in P_{\mathrm{dnf}}(L_a)$.

□

The graph is constructed by starting from an existing node $[L, p]$ (initially the partitions of the initial location), and then for each enabled action a, by computing the set of locations L_a that can be entered by executing the a action from the partition. Then the partitions p_a of location L_a can be computed according to Definition 4.6 (2). Every $[L_a, p_a]$ is then an a successor of $[L, p]$. It should be noted that only partitions whose constraints have solutions need to be represented. Further, not all may be reachable from the initial state. The partition graph for the coffee machine is depicted in Figure 4.8.

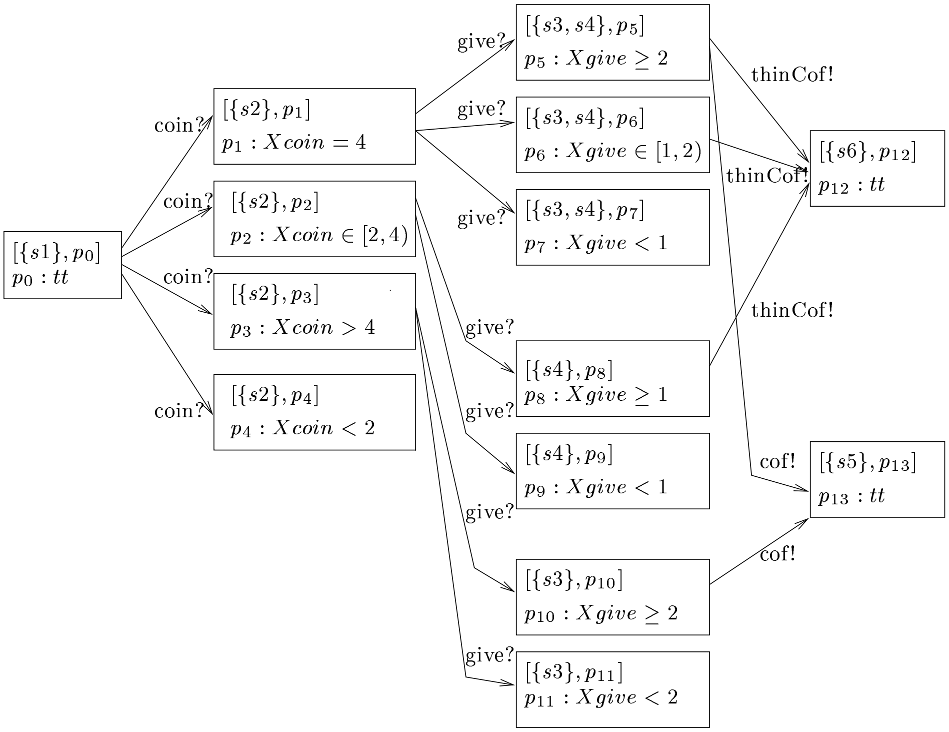


Figure 4.8. *Partition graph for the coffee machine.*

Proposition 4.9 states some further properties about our partitioning. First, by construction it holds that the edges enabled in the partition $[L, p]$ are the same for all concrete states in that partition. Therefore all concrete states of a partition satisfy exactly the same 'single action' tests (Hennessy test without a preceding trace). This property confirms that the partition graph represents different deadlock situations, which was part of the original motivation for choosing it. Second, the partition graph gives a *deterministic* representation of the state space.

Proposition 4.9 *Partition properties.*:

1. Single Action Tests. $\forall\langle L, \bar{u}\rangle, \langle L, \bar{v}\rangle \in [L, p]$.
   (a) $\langle L, \bar{u}\rangle \models \textbf{after } \epsilon \textbf{ must } A$ *iff* $\langle L, \bar{v}\rangle \models \textbf{after } \epsilon \textbf{ must } A$
   (b) $\langle L, \bar{u}\rangle \models \textbf{after } a \textbf{ must } \emptyset$ *iff* $\langle L, \bar{v}\rangle \models \textbf{after } a \textbf{ must } \emptyset$
   (c) $\langle L, \bar{u}\rangle \models \textbf{can } a$ *iff* $\langle L, \bar{v}\rangle \models \textbf{can } a$
2. Determinism: $\forall\langle L, \bar{u}\rangle \in [L, p]$.
   $\langle L, \bar{u}\rangle \xrightarrow{a} \langle L', \bar{u}'\rangle$ and $\langle L, \bar{u}\rangle \xrightarrow{a} \langle L'', \bar{u}''\rangle$ *implies* $\langle L', \bar{u}'\rangle = \langle L'', \bar{u}''\rangle$

Argument:

1. follows from the absence of τ actions and non-urgent actions, and from the construction of p which ensures that every state has exactly the same enabled and disabled transitions. Only the enabled transitions, and not the clock values, affect which single action tests are satisfied.
2. follows because 1) from a given set of locations L every subset of edges with the same action a (and only these edges) are formed, 2) the target location L_a is constructed to contain all locations that can be entered by executing an a action from any source location in L, and 3) the same clock x_a is reset on all a edges. Because the partitioning forms the subsets of *all edges*, it is finer than the one resulting from the determinization algorithm of ERA, which only requires subsets of all edges labeled with the same action [10]. □

Each partition $[L, p]$ can now be decorated with the action sets M, C, R defined in Definition 4.10 as needed by Algorithm 4.5.

Definition 4.10 *Decorated Partitions*:
Define $\text{Must}([L,p]) = \{A \mid \exists\langle L,\bar{u}\rangle \in [L,p].\ \langle L,\bar{u}\rangle \models \textbf{after}\ \epsilon\ \textbf{must}\ A\}$
$\text{Sort}([L,p]) = \{a \mid \exists\langle L,\bar{u}\rangle \in [L,p].\ \langle L,\bar{u}\rangle \xrightarrow{a}\}$

1. $M([L,p]) = min_{\subseteq}\text{Must}[L,p]$.
2. $C([L,p]) = \text{Sort}([L,p]) - \bigcup_{A \in M([L,p])} A$.
3. $R([L,p]) = \mathcal{O} - \text{Sort}([L,p])$.

□

The reader should observe the (intended) similarity between the partition graph and the untimed success graph defined in Section 2.3. In the timed case, we are further challenged by the problem of computing 1) the reachable partitions, 2) the reachable parts of these partitions (no all clock valuations of a partition are reachable from the initial state), and 3) the specific timed traces needed to generate test cases. These problems are addressed in the following section.

4.3 Symbolic Techniques

The reachable parts of the partitions must be computed before concrete test cases can be generated. Given the reachable parts, the desired test points can be chosen, and timed traces leading to the chosen test points can be synthesized. Also, a symbolic representation of the state space is needed for the termination of Algorithm 4.5. The algorithm stops when all reachable partitions have been reached and tested. The required computations consist of solving linear inequalities on clocks. The computations can be done efficiently using the so-called zone technique developed for model checking of timed automata [38, 18, 120, 119, 13, 64]. Specifically, we shall employ similar techniques as those developed for the UppAal tool [119, 13, 64].

4.3.1 Zones

A zone is a conjunction of linear inequalities on clock variables. The solution set of a zone is the set of all clock valuations that satisfies its constraints. A zone will for example be used to describe the solutions to a guard.

Definition 4.11 *Zones*:
Let $X = \{x_1 \dots x_n\}$ be a set of clocks. A zone z over clocks in X is a constraint system consisting of conjunctions of clock constraints of the following forms:

> $\{x_i - x_j \prec c_{ij} \mid i, j \leq n\} \cup \{a_i \prec x_i\} \cup \{x_i \prec b_i\}$, where $\prec \in \{<, \leq\}$, c_{ij}, a_i, b_i are integers including ∞, and $x_i, x_j \in X$.

□

Intuitively, a_i is a lower bound on clock i, b_i an upper bound, and c_{ij} a maximum difference of two clocks i, j. The needed symbolic computations rely on the tool box of operations on zones defined in Definition 4.12. We postpone their implementation until Section 4.3.2.

Definition 4.12 *Operations on zones*:

1. $z^{\uparrow} =_{\text{def}} \{\bar{u} + d \mid \exists d \in \mathbb{R}_{\geq 0}, \bar{u} \in z\}$. $z^{\uparrow}$ contains the *future* of z, i.e., the clock valuations that can be reached from z by delaying.

2. $z^{\downarrow} =_{\text{def}} \{\bar{u} \mid \exists d \in \mathbb{R}_{\geq 0}, \bar{u} + d \in z\}$. $z^{\downarrow}$ contains the *past* of z, i.e., the clock valuations that can reach z by delaying.

3. $z_{x_i:=0} =_{\text{def}} \{\bar{u}[x_i \mapsto 0] \mid \bar{u} \in z\}$. $z_{x_i:=0}$ is obtained by *resetting* the clock x_i in all clock valuations in z.

4. $\mathit{border}(z, x_i) =_{\text{def}} \{\bar{u} \in z \mid \bar{u}(x_i) = 0\}$. The *border* of a zone z on clock x_i is the subset of z where x_i equals zero.

5. $\mathit{free}(z, x_i) =_{\text{def}} \{\bar{u}[x_i \mapsto d] \mid \bar{u} \in z, d \in \mathbb{R}_{\geq 0}\}$. *Free* unconstrains the upper and lower bounds on the clock x_i, that is, x_i is allowed to assume any value.

6. $z_1 \wedge z_2 =_{\text{def}} z_1 \cap z_2$ gives the *intersection* of two zones, i.e., the clock valuation where the constraints of both zones are satisfied.

7. $z_1 \subseteq z_2$ is a predicate that determines whether one zone is fully contained in another, i.e., z_1 is implied by z_2.

8. $z = \emptyset$ is a predicate that determines whether the zone is empty. A zone is empty when its inequalities are unsatisfiable.

□

Perceived graphically, a zone forms a convex polyhedra in the n-dimensional space. The zone operations are depicted graphically in Figure 4.13.

Zones are closed under all the above operations. Thus, applying any of these operators on a zone results in a set of clock valuations describable by a zone.

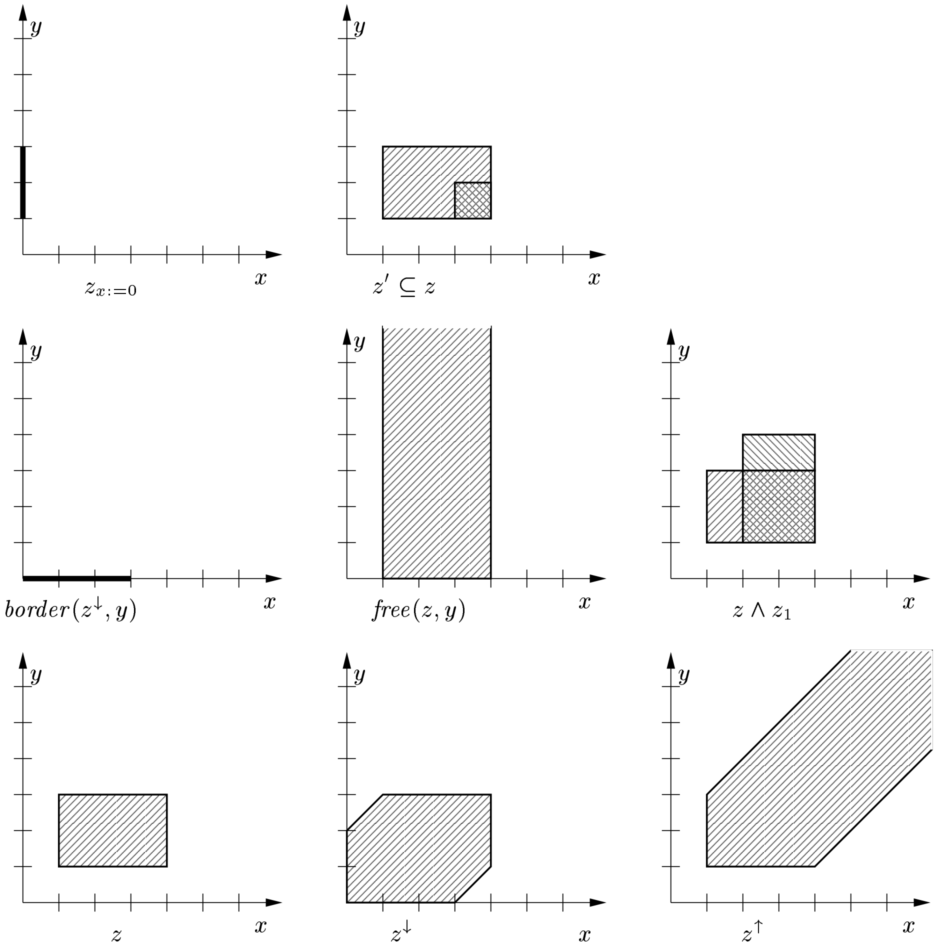


Figure 4.13. *Operations on zones.*

4.3.2 Difference Bound Matrixes

Zones can be repressented efficiently by the *difference bound matrix* (DBM) data structure. DBMs were first applied to represent clock differences by Dill in [38]. A DBM represents clock difference constraints of the form $x_i - x_j \leq c_{ij}$ by a $(n+1) \times (n+1)$ matrix such that c_{ij} equals matrix element (i, j), and where n is the number of clocks.

To represent constraints of the form $x_i \leq c$, DBMs use a special zero clock $\mathbf{0}$ that has the constant value 0. Thus, $x_i \leq c_i$ is represented as $x_i - \mathbf{0} \leq c_i$. Note that lower bound constraints of the form $x_i - x_j \geq c_{ij}$ are rewritten as $x_j - x_i \leq -c_{ij}$ to fit into a DBM. Similarly, $x_i \geq c_i$ is rewritten as $\mathbf{0} - x_i \leq -c_i$.

Also each matrix element encodes whether the comparator is strict or not, i.e, $<$ vs. $\leq$. Figure 4.14(a) shows an example of a DBM.

In general, several DBMs can represent the same constraint system. Closer inspection of z from Figure 4.14(a) reveals an implicit constraint on the upper bound of x_1 caused by the upper bound on the clock differences between x_1 and x_2, and the bound on x_2. Thereby, $x_1 \leq 7$. A DBM where all implicit constraints have been resolved such that for all $x_i - x_j \leq c_{ij}$, c_{ij} is the tightest bound possible such that the DBM represents the same solutions, is *canonical.* Two DBMs represents the same solution set iff the DBMs in canonical form are identical. The canonical DBM for z is shown in Figure 4.14(b).

The canonical representation of a DBM can be computed by an all pairs shortest path algorithm. This is the most expensive DBM operation; it has a cubic complexity in the number of clocks. The algorithms for checking the inclusion and emptyness of zones assume that the DBMs are represented in their canonical forms. Implementation of other zone operators are described in [14, 120].

$$\begin{array}{c|ccc}
 & \mathbf{0} & x_1 & x_2 \\
\hline
\mathbf{0} & \times & -5 & 0 \\
x_1 & \infty & \times & 4 \\
x_2 & 3 & \infty & \times
\end{array}
\qquad
\begin{array}{c|ccc}
 & \mathbf{0} & x_1 & x_2 \\
\hline
\mathbf{0} & \times & -5 & -1 \\
x_1 & 7 & \times & 4 \\
x_2 & 3 & -2 & \times
\end{array}$$

(a) (b)

Figure 4.14. *DBM representation of the constraint $z = x_1 - x_2 \leq 4 \ \wedge \ x_2 \leq 3 \ \wedge \ x_1 \geq 5$ (a), and the DBM on canonical form (b).*

4.3.3 Forward Reachability

We next describe how to compute the reachable subsets of the partitions. We introduce the notion of a *symbolic reachability graph* which represents the computations that a partition of the partitioned ERA can perform.

A *symbolic state*, written $[L, z/p]$, represents a reachable part of partition $[L, p]$. Let z be a zone of reachable clock valuations, and let $z \subseteq p$. The notation z/p denotes that z is a reachable fraction of p. Similarly, a *symbolic transition* does not represent a single transition but rather a whole set of possible transitions. We shall write $[L, z/p] \overset{a}{\Rightarrow} [L', z'/p']$ if z' contains all clock valuations in the partition p' reachable from the states $\langle L, \bar{u} \rangle$, $\bar{u} \in z$ by performing an a action followed by some delay. This relation is defined formally in Definition 4.15.

Definition 4.15 *Symbolic Transitions*:

$$[L, z/p] \overset{a}{\Rightarrow} [L', z'/p'] \text{ iff } [L, p] \overset{a}{\rightarrow} [L', p'] \text{ and}$$
$$z' = \{\bar{u}' \mid \exists \bar{u} \in z.\ \exists d \in \mathbb{R}_{\geq 0}.\ \langle L, \bar{u}\rangle \xrightarrow{a \cdot \varepsilon(d)} \langle L', \bar{u}'\rangle \ \wedge\ \bar{u}' \in p'\},\ \text{and } z' \neq \emptyset$$

□

The target zone of a symbolic transition can easily be computed using the equation: $z' = (z_{x_a:=0})^{\uparrow} \wedge p'$, and our defined operators on zones. First, $z_{x_a:=0}$ yields the clock valuations of z after the a action is executed, and consequently after a's event clock is reset. Second, the application of the future operator yields all states reachable after the a action by letting time pass, and finally the subset of p' reachable from z is computed by conjuncting with the p' constraint. This procedure is illustrated graphically in Figur 4.16.

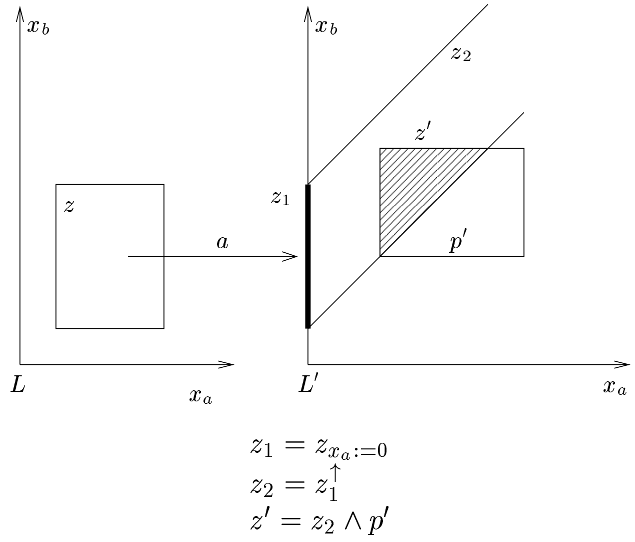


Figure 4.16. *Illustration of* $[L, z/p] \overset{a}{\Rightarrow} [L', z'/p']$.

Forward reachability analysis starts in the initial state, and iteratively computes the symbolic states that can be reached in one step from an existing symbolic state. This traversal can either be done depth first or breadth first. When a new symbolic state is contained in one previously reached, it can be concluded that no further states can be reached from it, and therefore, that it needs no further exploration. This procedure is repeated while new symbolic states are found.

Forward reachability analysis follows the direction of edges in the specification. The opposite, backward reachability analysis, starts from some desirable or undesirable symbolic state by following the edges in the opposite direction, and stopping when the initial state is found.

An algorithm inspired by Petterson, Daniels, and Yi [119, 83] for forward reachability analysis of the partition graph is formulated in Algorithm 4.17.

Algorithm 4.17 *Forward reachability computation*:

```
passed := $\emptyset$
waiting := $\{[L_0, z_0/p] \mid z_0 = \bar{0}^{\uparrow} \wedge p, p \in P_{\mathrm{dnf}}(L_0), L_0 = \{\bar{l}_0\}\ \}$
**while**(waiting $\neq \emptyset$)
    waiting := waiting $- \{[L, z/p]\}$
    **if** $\nexists [L, z'/p] \in$ passed. $z \subseteq z'$
        passed := passed $\cup \{[L, z/p]\}$
        **whenever** $[L, z/p] \overset{a}{\Rightarrow} [L', z'/p']$
            waiting := waiting $\cup \{[L', z'/p']\}$
```

The algorithm maintains two sets of symbolic states: The passed list contains the symbolic states already explored. The waiting list contains the states that still are to be explored. If the waiting list is accessed using a stack discipline, the state space is traversed depth first; if it is accessed using a queue discipline, the state space is traversed breadth first.

There are two technical remarks to be made about the algorithm. Because of the structure of the partition graph, the algorithm is started, not in the initial state $[L_o, \bar{0}]$, but in the symbolic states that the initial state can reach by only delaying. The second comment regards termination. As we discuss in Section 4.4 subset inclusion as used by the algorithm is not always sufficient to ensure termination. The passed symbolic states need to contain certain extrapolated states. We also discuss possible strategies when the specification becomes too large for a complete reachability analysis.

When the algorithm terminates, the passed list contains the reachable state space. A partition $[L, p]$ is reachable if there exists some $[L, z/p]$ in the passed list. The reachable states of a partition equals the union of all such z's.

When the algorithm traverses the state space breadth first, it also computes the shortest (not necessarily the fastest) traces leading to each partition: When the partition $[L, p]$ is encountered for the first time by a symbolic state $[L_n, z_n/p]$, the symbolic transitions taken by the algorithm from the initial location constitute a *shortest trace* to the partition. Using the shortest trace in a test case will help reducing the size of the test suite.

4.3.4 Back Propagation of Constraints

Given a symbolic trace, the next step is to compute a sample concrete trace leading from the initial state to the target partition, or to the desired subset thereof. To do this, we compute a symbolic state that contains the subset of states that *all* lead to the final state. To illustrate the need for strengthening, consider the specification in Figure 4.18 in which the terminal state $s3$ can only be reached if the b action is taken before 7 time units after the first a action.

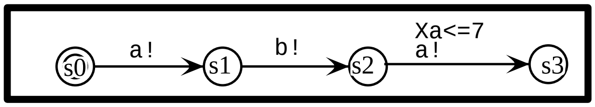


Figure 4.18. *Strengthening zones.*

From the definition of symbolic transitions it follows that if $[L, z/p] \overset{a}{\Rightarrow} [L', z'/p']$ then some of the states in $[L, z/p]$ can reach z', but not necessarily all of them. We introduce a *strengthened* symbolic state, written $[L, y/z/p]$, where y now is the subset of z which all will reach the desired states. We call y a *strengthened* zone of z. Definition 4.19 formally states the required relation between two strengthened symbolic states in a trace.

Definition 4.19 *Strengthened zones*:

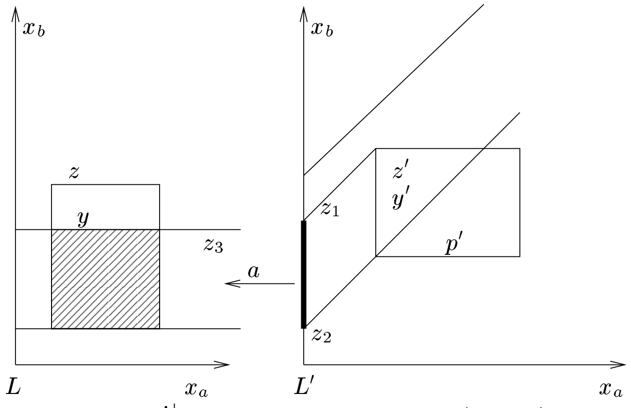
$$[L, y/z/p] \overset{a}{\Rightarrow} [L', y'/z'/p'] \textit{ iff } [L, z/p] \overset{a}{\Rightarrow} [L', z'/p'] \text{ and}$$
$$y = \{\bar{u} \in z \mid \exists d \in \mathbb{R}_{\geq 0}.\langle L, \bar{u}\rangle \xrightarrow{a\cdot\varepsilon(d)} \langle L', \bar{u}'\rangle \wedge \bar{u}' \in y'\}$$

□

A strengthened zone y can be computed essentially using the same techniques that would be used in backward reachability analysis:
$y = \mathit{free}(x_a, \mathit{border}(x_a, y'^{\downarrow})) \wedge z$

The steps for strengthening zone z to reach $y' = z'$ in Figure 4.16 are illustrated in Figure 4.20. First, application of the past operator to y' gives the set of clock valuations that can reach y' by delaying. Second, the border operation extracts the subset thereof where the clock x_a is zero, that is, the values just after the a action which implicitly assigns zero to x_a. Third, freeing x_a in this set gives the clock valuations just before the clock assignment that can reach y' by performing an a action and then delaying some. Finally, y can be computed as the intersection of the freed zone and z.

To reach the terminal zone or a subset y thereof, back propagation must be performed starting from the terminal zone and back towards the initial state.



$$z_1 = y'^{\downarrow} \qquad z_3 = \mathit{free}(z_2, x_a)$$
$$z_2 = \mathit{border}(z_1, x_a) \quad y = z \wedge z_3$$

Figure 4.20. *Backward propagation of constraints.*

4.3.5 Timed Trace Computation

The final step in the computation of a single test case is to compute a specific trace with concrete delays from a symbolic trace. This can be done by either going forwards or backwards in the strengthened trace. We elaborate on the forward approach because this will also be used in the test composition algorithm outlined in Section 4.5.

When a specific clock valuation in one zone has been chosen, the possible delays from that to the next zone in the trace can be computed. The specific delay can then be chosen according to a selection strategy. Given a fixed delay, the reached clock valuation in the successor zone can be computed. The procedure is started in the initial state $\langle L_0, \bar{0}\rangle$, and proceeds until the final state of the trace is reached.

Let $\bar{u}$ be chosen in y, and suppose that $[L, y/z/p] \overset{a}{\Rightarrow} [L', y'/z'/p']$. The set of clock valuations in y' that are reachable from the state $\langle L, \bar{u}\rangle$ can be computed as a forward reachability step: $y'' = (\bar{u}_{x_a:=0})^{\uparrow} \wedge y'$.

Now observe that y'' is a line segment, and that fixing a delay implies fixing a point on this line segment. The possible delays D are directly available as the upper and lower bound on clock x_a in the canonical DBM of y'' because x_a was reset when a was taken. Assume that the delay $d_1 \in D$ is chosen. Then the selected point $\bar{u}'$ in y'' can be computed as: $y'' \wedge x_a = d_1$.

The specific delay can be chosen freely in the range D. We introduce the selection function $\mathcal{D}(D)$, which chooses a delay according to a reasonable strategy.

There are three immediate strategies for $\mathcal{D}$:

1. Choose the smallest delay $d \in D$. This checks the *promptness* of the implementation by executing the succeeding action at the earliest time possible in the current trajectory.

2. Choose the delay (possibly stochastically) to be in the middle of D. This checks the *persistence* of the implementation, i.e., that the succeeding action can be executed in the interior of the line segment.

3. Choose the delay to be the largest delay in D. This tests the *patience* of the system, i.e., that the succeeding action is also executable at the latest required time.

Of the above strategies, it seems most important to check the promptness of the system as this checks for missed deadline errors, which are common in real-time systems. But also the patience may be important, because this may detect errors where a timer times out prematurely.

4.3.6 Extreme Value Selection

The experiences from testing of sequential programs [12] suggest that many bugs are found near the extreme values of the input domains, and therefore that testing should focus around these. This may well also be the case for real-time systems. We here suggest that extreme real-time inputs can be interpreted as extreme clock valuations in the reached symbolic states. Algorithm 4.5 is able to compute a trace leading to any reached (subset) of clock valuations, including extreme values.

In our setup we can define two types of extreme values, local and global extremes. The global extremes are defined by the reached extreme values of the partitions, whereas local extremes are defined by a single reached zone (a subset of a partition). Because an partition may be passed several times by a symbolic state, these two notions do not coincide.

One could define precisely what constitutes an extreme value in several ways, but we do not pursue this here. Figure 4.21 shows some possible candidates for the extreme values of a zone. The information required to compute such extreme values is directly available from its canonical DBM representation. There is no similar direct method of computing the global extremes with the DBM representation. One must compute all reached local extremes and select those that constitute global extremes.

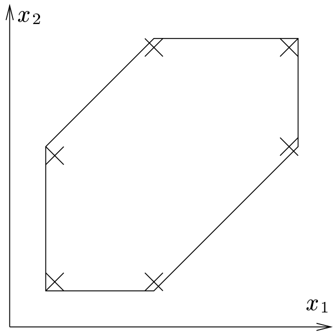


Figure 4.21. *Candidates for local extreme values.*

4.3.7 Symbolic Execution of Unrestricted Timed Automata

This subsection is devoted to a problem unrelated to analysis of ERA, but related to the problem of implementing the direct test generation lgorithm (Algorithm 3.11), outlined in Section 3.3 for unrestricted timed automata. That algorithm assumes two operations; one to compute the reachable states after a delay, and one to compute the reachable states after an action. It was stated that this potentially infinite set of states could be computed the using the symbolic reachability techniques just presented. In this subsection, we substantiate this claim by formulating Algorithm 4.22, which computes these sets.

One restriction to timed automata will be convenient: Only true guards will be permitted on urgent hidden actions. This restriction ensures that an urgent τ_u action is possible for all clock valuations in a location vector, or not possible at all in that location vector. Without this restriction we would need an (unpleasant) set-difference operator on zones to subtract the subset of states that cannot be reached because time may not pass when an urgent τ_u is possible.

The following notation is needed. Let the present state of the network be represented by a *set of symbolic states* $Z = \{[\bar{l}_1, z_1] \ldots [\bar{l}_n, z_n]\}$. Let x_t be an auxiliary timer not used in the automata, which will be reset after every observable action, thus recording the amount of time passed since then. Divide Z into two subsets Z_{nu} from which no symbolic state can perform an urgent τ_u action, and Z_u from which all symbolic states can perform an urgent τ_u action (they may also be able to perform a non-urgent τ). Let $r(z)$, $\mathit{free}(r, z)$, and $\mathit{border}(r, z)$ be the assignment, free, and border operations defined on zones generalized to sets of assignments. Define $\mathit{pre}(r, z) =_{\mathrm{def}} \mathit{free}(r, \mathit{border}(r, z^{\downarrow}))$.

Finally, let $\xrightarrow{g,\tau,r}_{\dagger}$ be a shorthand for $\xrightarrow{g,\tau,r}$ when the τ action arises from an explicit τ labeled edge, or for $\xrightarrow{g_1 \wedge g_2, \tau, r_1 \cup r_2}$ when the τ action arises from a synchronization between some $l_i \xrightarrow{g_1,a,r_1} l'_i$ and $l_j \xrightarrow{g_2,\bar{a},r_2} l'_j$.

Algorithm 4.22 *Symbolic execution of timed automata*:

```
$\mathbf{after}(Z,d) =_{\mathrm{def}}$        $\mathbf{after}_{nu}\ (Z_{nu},d) \bigcup \mathbf{after}_{u}\ (Z_u,d)$

$\mathbf{after}_{nu}(Z_{nu},d) =_{\mathrm{def}}$  $\{[\bar{l},z'] \mid [\bar{l},z] \in Z_{nu}, z' = (z^{\uparrow} \wedge I(\bar{l}) \wedge x_t = d), z' \neq \emptyset\}$
                          $\bigcup \mathbf{after}(Z',d)$, where
                        $Z' = \{[\bar{l'},z''] \mid \exists[\bar{l},z] \in Z_{nu}.\ \bar{l} \xrightarrow{g,\tau,r}_{\dagger} \bar{l'},$
                        $z'' = r(z^{\uparrow} \wedge I(\bar{l}) \wedge x_t \leq d \wedge g \wedge pre(r, I(\bar{l'}))), z'' \neq \emptyset\}$

$\mathbf{after}_{u}(Z_u,d) =_{\mathrm{def}}$    $\{[\bar{l'},z'] \mid [\bar{l},z] \in Z_u, \bar{l'} = \bar{l}\ \wedge\ z' = (z \wedge x_t = d), z' \neq \emptyset\}$
                          $\bigcup \mathbf{after}(Z',d)$, where
                        $Z' = \{[\bar{l'},z''] \mid \exists[\bar{l},z] \in Z_u.\ \bar{l} \xrightarrow{g,\tau,r}_{\dagger} \bar{l'},$
                        $z'' = r(z \wedge g \wedge x_t \leq d \wedge pre(r, I(\bar{l'}))), z'' \neq \emptyset\}$

$\mathbf{after}(Z,a) =_{\mathrm{def}}$        $\{[\bar{l'},z'] \mid \exists[\bar{l},z] \in Z.\ \exists\bar{l} \xrightarrow{g,a,r} \bar{l'},$
                        $z' = (r'(z \wedge g \wedge pre(r, I(\bar{l'})))),\ z' \neq \emptyset\}$,
                        $r' = r \cup \{x_t := 0\}$
```

Intuitively, we need to accumulate an amount of d time units across $\xrightarrow{\tau}^{*}$ and $\xrightarrow{\varepsilon(e)}$ transitions. The function $\mathbf{after}(Z,d)$ recursively follows τ actions, and collects all states that can be reached after d time units.

The function $\mathbf{after}_{nu}(Z_{nu},d)$ collects the symbolic states that can be reached by letting time pass with less than d time units, or reached after executing a non-urgent action. Similarly, $\mathbf{after}_{u}(Z_{nu},d)$ collects the symbolic states that can be reached after having executed an (urgent or non-urgent) internal action without letting time pass. Letting time pass is not permitted when an internal action is enabled.

The function $\mathbf{after}(Z,a)$ computes all symbolic states that can be reached from a symbolic state in Z by executing an a action.

Finally, we remark that computing the satisfaction of the Z **after** $\varepsilon(d)$ **must** A predicate is also rather involved. We say that a state $\langle\bar{l},\bar{u}\rangle$ is *pseudo stable* if it allows time to pass, i.e., if $\langle\bar{l},\bar{u}\rangle \xrightarrow{\varepsilon(d)}$ for some d. It is not pseudo stable if it has an enabled urgent τ action, or if it is positioned at the boundary of an invariant condition. All such pseudo stable symbolic states in $Z' = Z$ **after** $\varepsilon(d)$ must be able to perform one of the (urgent) actions in A. Both extracting the pseudo stable parts of symbolic states, and checking if a symbolic state is covered by some $a \in A$ involves negation of guards (or alternatively set difference operation) and conjuncting these to the symbolic states. As we have seen, this results in non-convex sets, or requires a representation as several convex zones.

4.4 Termination

This section further comments on the termination of the test case generation algorithm, and in particular on the forward reachability algorithm presented in Algorithm 4.17, because this implicitly terminates both the overall test case generation algorithm and the test composition algorithm. The first comment is of a technical nature concerning termination of the forward reachability analysis which is not guaranteed in the presented version of Algorithm 4.17. The second comment is about pragmatic approaches to handling large specifications. One problem is that memory and time limitations may prohibit complete symbolically analysis. Another is that the resulting test suite may be too large to be executed within the given time resources.

4.4.1 Termination of Forward Reachability

The forward reachability algorithm in Algorithm 4.17 stops further exploration when it finds a previously visited symbolic state containing the current one.

It is possible to construct ERA models, like timed automata in general, that exhibits a diverging behavior of its clock values. An example is shown in Figure 4.23. The value of clock x_a in location s_1 increases by one each time a b action is executed. Therefore, the reachability algorithm will never find an existing state in the passed list that contains the new value of x_a, and consequently never terminate.

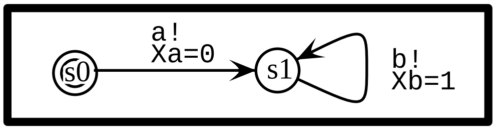


Figure 4.23. *Diverging specification.*

When the symbolic reachability techniques are used for model checking, this problem is solved by *extrapolating* clock values from the reached states such that termination will be ensured. We apply this technique, but also comment on its implications on testing.

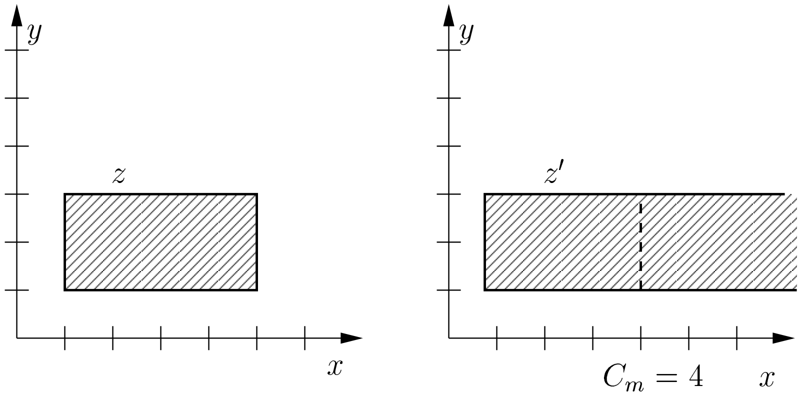
The essential observation is that the precise value of a clock is irrelevant when it exceeds a certain maximum value C_m. Larger values will not enable or disable further guards. This observation is also applied to achieve a finite representation of a discrete time semantics in Section 3.1.3, and in the finite region graph representation defined in Section 3.4.2. In ERA, it suffices to choose C_m larger than the largest constant used in its guards.

Intuitively, the *extrapolation* of a zone is the zone where all upper bounds larger that C_m has been removed, and where all lower bounds has been replaced by C_m, see Definition 4.24 and Figure 4.25. For further information see [83, 35].

Definition 4.24 *Extrapolation [83]*:
Let $z = \bigwedge_{ij} x_i - x_j \prec c_{ij}$ denote the constraints of a DBM in its canonical form, and let $z' = \bigwedge_{ij} x_i - x_j \prec c'_{ij}$ denote the extrapolated zone, where

$$c'_{ij} = \begin{cases} \infty, & \text{if } c_{ij} > C_m \\ -C_m, & \text{if } c_{ij} < -C_m \\ c_{ij} & \text{otherwize} \end{cases}$$

□



(a) (b)

Figure 4.25. *A zone z (a), and extrapolated zone z' (b).*

However, for testing it is important to observe that the extrapolated zone contains information that is not necessarily reachable. This implies that not all clock valuations in an extrapolated zone are valid for test selection. To avoid this problem, we propose the simple solution of storing the passed list in the reachability analysis as extrapolated zones, but storing the constructed reachability graph using only zones with exact information. Only the reachability graph is traversed during test generation.

This solution ensures that all reachable partitions will be reached by a portion of clock valuations, and guarantees termination (unless a discrete variable is unbounded), but at a cost of storing each symbolic state more than once (twice if the entire reachability graph is stored at the same time).

Moreover, because the extrapolated information helps terminating loops at which clock values diverge, the stored exact information will not contain all extreme values of the specification that are in fact reachable by unfolding such loops. This means that certain reachable states are unavailable for test selection. It is unknown what practical consequences this limitation has on error detection, but we expect them not to be severe, because the partition is tested by other values. Further, the test generation algorithm can be modified to unfold loops a number of times, and thereby reach additional extreme values.

4.4.2 Pragmatic Termination Criteria

We shall here propose a few pragmatic strategies for handling specifications whose reachability or partition graphs are too large to be completely computed, stored, or tested.

Limit trace length: Construction of the graph is terminated when its depth exceeds the maximum desired value.

Random Exploration: At each node, only a randomly chosen set of its successor nodes will be constructed. The construction is continued until the desired number of nodes is constructed, or a space or time limit is reached.

Bit-State Hashing: With bit-state hashing, a node is deemed previously reached if it hashes to the same hash table entry as another node. The nodes of the graph are stored in a hash table of length N. A hash function computes a hash key h_i for each node which is then stored in the table at entry h_i **mod** N. Without bit-state hashing, conflicts could be resolved using open hashing such that conflicting entries are stored in a chained list. With bit-state hashing however, only the first encountered node will be stored. When a second node is encountered, it is disregarded, and no further exploration is done from it. If the contents of the node is no longer needed, the hash entry can be implemented using only one bit per node, which thus gives a very compact representation. N would be chosen close to the available memory, or to the maximum desired state space.

The bit-state hashing technique was popularized by the SPIN model checker by Holzman [54]. He also gives a detailed account of its properties, and also outlines other partial search methods. It is believed to result in a better (under) approximation of the state space than random exploration, which has a tendency of confining itself to small parts of the state space. In our partition graph, hash keys can be computed from a combination of control location vector and integer variable values. In the reachability graph, it can be based on an identification number of the concave or convex paritіtions.

4.5 Composing Tests

The overall test generation algorithm given in Algorithm 4.5 has the disadvantage of generating a test case per must, can, and refusal property per symbolic state to be tested. Because the execution of the generated test suite can be a bottleneck, it is important to reduce its size, while maintaining coverage. The size of a test suite can be measured by the number of tests cases it contains, and their combined total length.

In addition, composing tests reduces the number of inconclusive verdicts experienced during test execution, and hence reduces the number of re-executions of the same test case.

It is possible to reduce the size of the test suite generated by Algorithm 4.5 by not repeating tests that have been generated as part of the trace leading to a symbolic state. Also, a test case should in case of non-determinism be constructed as a tree as indicated in Figure 2.22, such that test execution can be continued from all outcomes of a non-deterministic choice, provided the chosen branch is uncovered by a previous test. We propose an algorithm that constructs tree structured tests and attempts to reduce the test suite size.

4.5.1 Composition Algorithm

Because the exact algorithm is somewhat involved we shall here only outline the principles of how it works. The input to the algorithm is a spanning *tree* of the reachability graph, specifically the tree obtained during forward reachability analysis. In this so-called *reachability tree*, the leafs are either symbolic states from where no actions are possible, or symbolic states at which the search was terminated because the state was contained in one previously reached, or because one of the alternative termination strategies outlined in Section 4.4 was activated.

Algorithm 4.26 *Composition of timed tests*:
input: reachability tree.
output: A covering set of timed tests.

```
construct first test tree
while (more test trees) {
    while (more splits) {
        back propagate constraints
        generate timed trace tree
        output test case
        make new split
        }
    mutate test tree
}
```

The algorithm is outlined in Algorithm 4.26. It operates on a structure called a *test tree*, which is a subtree of the reachability tree, and which also forms the structure of the generated test. Each node is labeled with one of the must, can, or refusal properties for that node, i.e., a set of actions $A \in M$, $A = C$, or $A = R$. For each action a in A, there is an a labeled edge leading to a node which will contain the sub test to be executed after a. If the node represents a refusal ($A = R$), there are no such sub tests, and consequently no outgoing edges. The verdict of the test in a given node depends on the chosen property.

This procedure is in spirit identical to the construction of a test case in the untimed case in Algorithm 2.27, except it is now executed in the reachability tree. Given such a test tree, the algorithm propagates constraints back from the leafs, instantiates the tree by selecting a specific "trace tree" leading to the leafs, and outputs it in a suitable format. The algorithm then mutates the test tree such that a new test can be generated. This procedure is continued until all required properties are contained in some test case, and such that the reachability graph is covered as required.

The back propagation and test tree mutation steps require some further comments contained in the following Sections 4.5.2 and 4.5.3.

4.5.2 Back Propagation in the Test Tree

Because we shall need to compute a symbolic trace tree that will lead to all leaf nodes, we first back propagate constraints from the leafs such that a strengthened zones in the tree represents the states that will lead to the leafs of its sub trees. This can be done using the technique given in Section 4.3.4. The trace tree can then be computed using the timed trace generation algorithm given in Section 4.3.5.

Assume that a symbolic state has two transitions, a and b, as shown in Figure 4.27: $[L, y'/z/p] \overset{a}{\Rightarrow} [L_a, y_a/z_a/p_a]$ and $[L, y''/z/p] \overset{b}{\Rightarrow} [L_b, y_b/z_b/p_b]$.

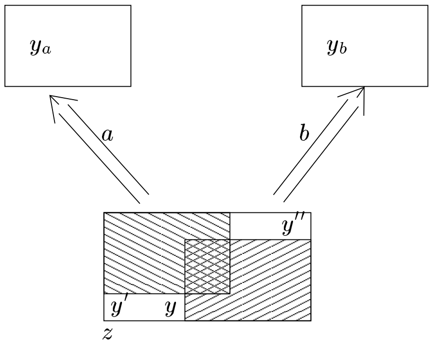


Figure 4.27. *Entire zone z does not reach both zone y_a and y_b, but zone $y = y' \wedge y''$ does. both*

y' contains the clock valuations that will reach the leafs of subtree a, and y'' contains the clock valuations that will reach the leafs of subtree b. The clock valuation y that will reach the leafs of both subtrees is hence the intersection of y' and y'': $y = y' \wedge y''$.

However, back propagation in a tree poses a problem because, in a given symbolic state, there may not be a common set of clock valuation that are able to reach the leafs of all its sub trees, i.e., y may be empty. When this happens, the single tests joined by the tree cannot be composed.

We propose to handle this problem by splitting the test tree by temporarily detaching the subtrees that caused a split. This is illustrated in Figure 4.28(a). After the test has been generated, the previously detached subtrees are reattached, but the subtrees now completed are detached, see Figure 4.28(b). In principle, the reattached nodes could cause a new split; splitting of a test tree can therefore happen a number of times.

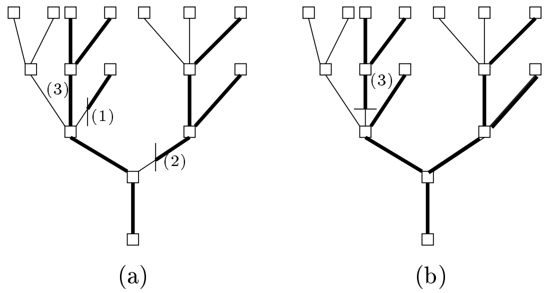


Figure 4.28. *Splitting a test tree. A reachability tree with a chosen test tree with two subtrees (1) and (2) cut off (a), and the test tree after reattachment (3) (b).*

4.5.3 Mutation of Test Trees

To compose as much behavior as possible, the test tree is constructed to be maximal at any given time: either its leafs are leafs in the reachability tree, or they are parents to *completed* nodes. A node is completed when either

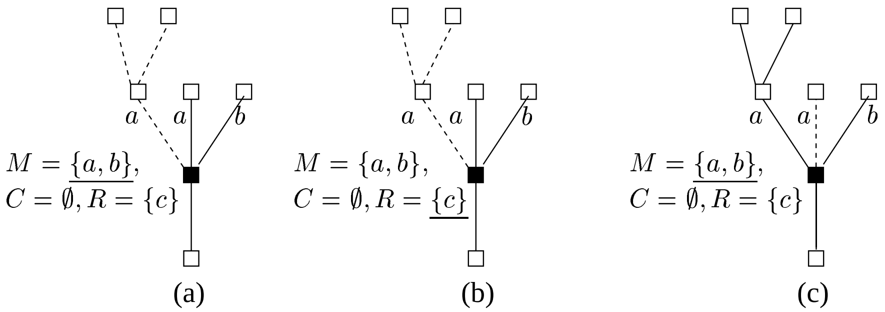
- all the nodes of its subtree are completed, and if it has been decided to generate tests for the node, then all the node's M, C, R propeties are contained in some generated test, or if
- all the nodes of its subtree are completed, and it is decided *not* to generate tests from this node.

When a test has been generated for the current test tree, the idea is to mutate all of its leaf nodes simultaneously whereby a new test tree is obtained, with as many modifications of the original as possible.

There are two options for mutating a node, which are depicted in Figure 4.29:

- replace a completed a labeled subtree with a new unexplored a labeled sub tree of the reachability tree. This involves selecting and constructing a new sub test tree, or
- mutate its property by choosing a new set of M, C or R actions. To ensure maximality, this possibly involves adding a new unexplored subtree, because a new action was enabled, and possibly removing a sub tree because an action was disabled.

The implementation of these rules must ensure that all sub trees of the reachability tree are included in a test tree at some point in time, and that all its



a
a
b
$M = \{a, b\}$,
$C = \emptyset, R = \{c\}$
(a)
a
a
b
$M = \{a, b\}$,
$C = \emptyset, R = \{c\}$
(b)
a
a
b
$M = \{a, b\}$,
$C = \emptyset, R = \{c\}$
(c)

Figure 4.29. *Mutating a test tree. A reachability tree (dashed lines) with superimposed test tree (solid lines) (a), mutation by changing property at the black node (b), and mutation by changing subtree (c).*

nodes are completed after termination. It should also be noted that this algorithm, like the overall Algorithm 4.5, skips symbolic states for which tests has already been generated. We propose to generate tests from a symbolic state the first time its partition is encountered during the reachability analysis, using either breadth first or depth first traversals as desired.

4.6 Implementation

We have implemented our approach and algorithms in a prototype tool called RTCAT. The purpose of the prototype is to evaluate our approach with respect to implementability, and the number and type of generated tests.

4.6.1 Facilities

RTCAT inputs an ERA specification in AutoGraph format [92]. A specification may consist of several ERA operating in parallel, and communicating via shared clocks and integer variables, but no internal synchronization is allowed as stated in Section 4.1. RTCAT performs partitioning, symbolic reachability analysis, and outputs a covering test suite to a file in DOT format [62]. Other features include:

Termination: By default, the entire partition and reachability graphs are constructed. Reachability terminates using subset inclusion with extrapolation. Construction of both graphs can also be terminated by specifying a trace depth, using bit-state hashing, or both.

Construction Order: Both breadth first and depth first construction of the partition and reachability graphs are implemented. Tests are generated for a node the first time its (convex) partition is passed during reachability graph construction.

Test Structure: Tests can be constructed either as individual timed Hennessy tests (Algorithm 4.5) or as test trees using the principles outlined in Section 4.5. To reduce the number of inconclusive verdicts at the intermediate steps in an individually generated test, a sub test with the strongest property containing the outgoing action is generated. Suppose that the property to be generated is **after** $a_1 \cdot a_2 \cdot a_3 \cdot \cdots \cdot a_n$ **must** A, and that it holds that **after** $a_1 \cdot a_2$ **must**$\{a_3, b\}$, then the generated test will compose both properties. Thus, the verdict after $a_1 \cdot a_2$ becomes **fail** rather than **pass** or **inconc**. No sub test after b will be generated in an individually generated test.

Trace Generation: Timed traces can be generated using prompt, interior, or patience selection as described in Section 4.3.5.

The tool contains a builtin test case checker to assist tool development and debugging. When enabled, it executes the generated tests against the specification. The specification must pass the tests generated from it. Little effort has been put into optimizing the tool, neither with respect to time nor memory consumption. No extreme value selection is currently implemented.

RTCAT is implemented in C++ and occupies about 22K lines of code. The implementation is based on code from a simulator for timed automata (part of an old version of the UppAal toolkit [65]). Its AutoGraph file format parser was reused with some minor modifications to accommodate the ERA syntax. Also the DBM implementation from the simulator was reused with some added operations for extrapolation and clock scaling.

4.6.2 Tool Options

The tool is invoked by the command `RTCAT options atg-filename` on the command line interface. The implemented options are summarized in Table 4.30.

4.6.3 Implementation Remarks

In this section, we make a few final but important implementation level remarks of the prototype.

option	meaning
-o *filename*	Write test cases to the file named *filename*.
-I	Write partition graph to file named `.initial.`*filename*.
-F dot$n$	Write test cases with extra state information, $0 \leq n \leq 4$.
-T	Generate individual test properties, not composed trees.
-C	Do not include refusals in generated tests.
-BFr	Construct reachability graph breadth first. Default is depth first.
-BFd	Construct partition graph breadth first. Default is depth first.
-d n	Limit trace depth to n.
-b n	Employ bit-state hashing in reachability graph construction using n hash table entries.
-n n	Employ bit-state hashing in partition graph construction using n hash table entries.
-Sd	use delay selection d: P=prompt, I=interior, L=persistent.
-V	Enable test case validator.

Table 4.30. *Tool options.*

Tool Architecture

The prototype operates in four distinct phases, i.e., the preceding must be completed before a new is started:

1. parsing and initialization
2. partition graph construction
3. reachability graph construction
4. test tree mutation and test output

This architecture was expected to be the easiest to construct and maintain during prototype implementation. An alternative architecture would be 'lazy' evaluation where the earlier phases are called on demand by the test tree generator. The lazy approach would have the advantage of only constructing the needed parts of the potentially large partition and reachability graphs. It would also make test cases available for execution immediately, and would be able to generate some test cases for larger systems that cannot be fully stored or analyzed.

Clock Scaling

A technical problem occurs in the computation of concrete traces, and in the selection of test points. Because clocks are drawn from a dense time domain, a selected clock valuation would therefore require a representation as a rational number. It is convenient to use the existing implemented zone operations and DBM data structure also for this purpose. However, this implementation operates on integers, and a direct reuse thus requires that the selected valuations are integral. Our implemented selection algorithms therefore prefer to choose the integral clock valuation nearest to the desired rational valuation.

However, not all zones contain integral valuations. One solution would be to use a floating point implementation of zones. But because floating points, as represented in computers, cannot be stored and manipulated entirely accurately, we have chosen a different approach. When needed, we instead change the time scale by multiplying the involved zones by a constant factor, and choose an integral valuation in the scaled zone. This scaling may be needed several times, and can be done for as long as no clock value exceeds the maximum allowed integer. We thereby accommodate many rationals.

Clock Reduction

A further issue is the number of clocks implicitly defined by ERA: One clock per action. This would for must specifications result in inefficient storage and manipulation of the zones, and could become a significant limiting factor when generating tests from untrivial specifications.

Although there is a declared clock for each action, only few of them are usually used in guards, and few are consequently relevant for the symbolic analysis. The prototype represents only such active clocks in the DBMs. The current trace generation algorithm measures delays by the clock of the action last executed. The remaining inactive clocks therefore map to the same auxiliary clock. It may be possible to reduce the number of clocks further by employing more sophisticated clock reduction algorithms proposed for model checkers, cf. [36].

4.6.4 Example of Generated Test Cases

RTCAT generates the 16 test cases shown in Figures 4.31 and 4.32 for the coffee vending machine specified in Figure 4.2. The tool options were set to generate composed tests, and to use interior selection.

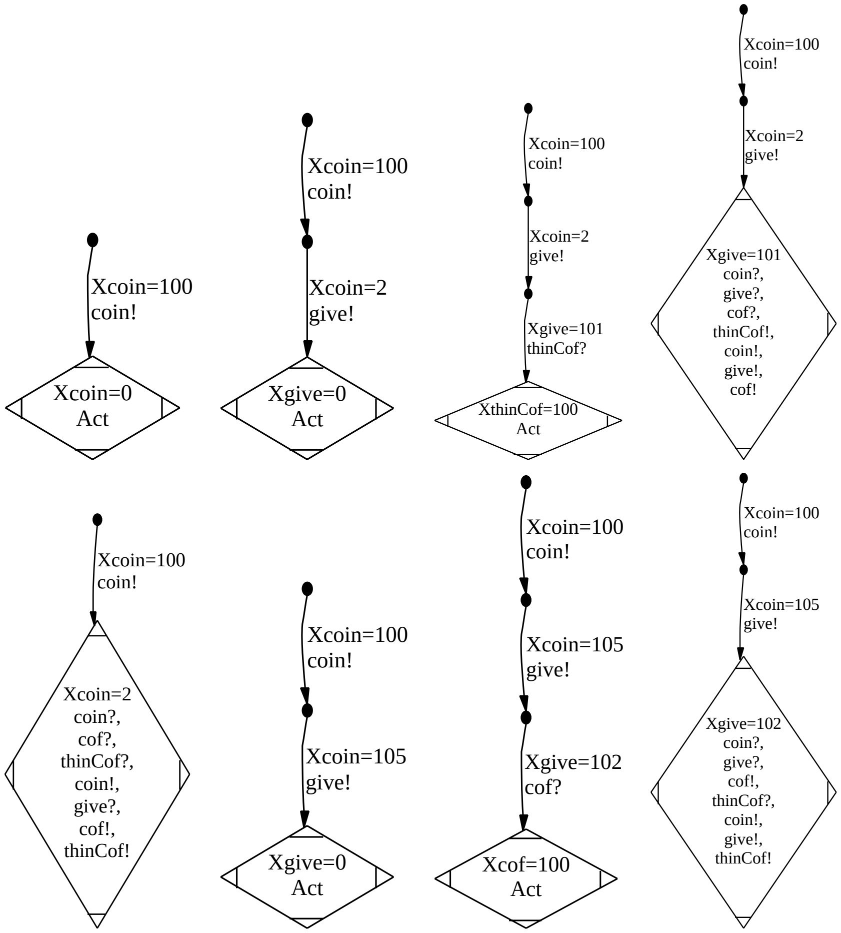


Figure 4.31. *Test cases for the coffee machine in Figure 4.2. Filled states are* **fail** *states, and unfilled states are* **pass** *states. Diamonds indicate a list of actions to be refused at the time indicated at the top of the list. Act is an acronym for all actions.*

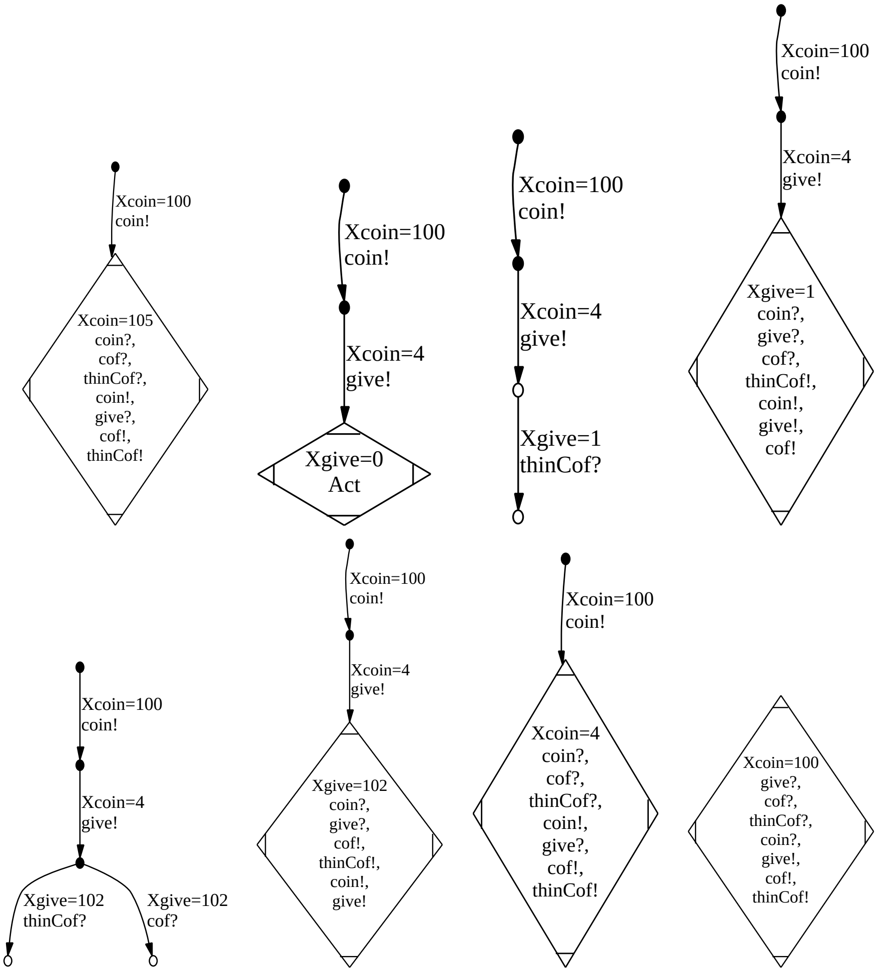


Figure 4.32. *Test cases for the coffee machine (continued).*

4.7 Summary

This chapter contains a detailed description of our symbolic test generation techniques. We apply our techniques on a restricted but determinizable class of timed automata, ERA. This involves computing the stable edge set partitions of the specification and performing symbolic reachability analysis on the partition graph. Dense timed systems must be analyzed and represented symbolically. For this purpose we use zones which represent infinite convex sets of clock valuations, and which can be manipulated efficiently. From the reachable parts of the partition graph, test cases can be generated relatively easily by using the symbolic techniques to compute a specific timed trace to any given target state chosen to be tested. The RTCAT tool implements our technique. Its features include breadth- and depth-first construction orders, various partial search methods, test composition, and prompt, persistent, and patient selection of time instances.

In the next chapter we evaluate the usefulness and feasibility of the techniques by applying them to a series of example specifications.

Chapter 5

Evaluation

In this chapter we evaluate our proposed techniques to determine their applicability and limitations, and give our preliminary experiences using RTCAT to generate tests. The starting point of the evaluation will be a collection of ERA specification examples contained in Section 5.1.

In Section 5.2 we report on the performance of our tool when applied to the sample specifications. The primary aspect to evaluate is the number of generated tests. This will indicate whether our selection strategy produces a reasonable number of tests, and what its limits are. The secondary aspect is the space and time used to generate them. We examine several configurations of the tool to determine the effect of traversal orders and test composition.

We use "testing setup" as a common term for the ingredients in an automatic testing method. It includes a specification language, an implementation relation, a test case language, and an overall test selection strategy. We evaluate these aspects in Section 5.3.

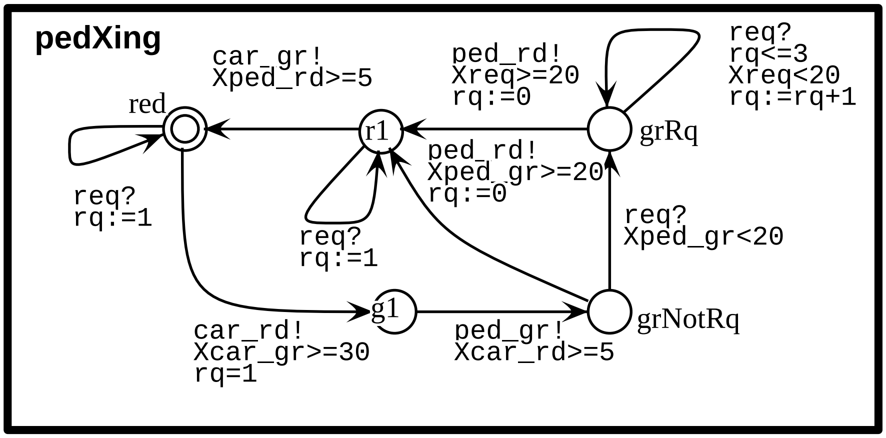
We outline in Section 5.4 our experiences of using and implementing symbolic methods for test generation, and finally, Section 5.5 summarizes the chapter.

5.1 Event Recording Automata Specifications

This section contains a couple of small toy like examples, and one larger specification: an attempt at specifying the Philips Audio Protocol.

5.1.1 Pedestrian Crossing

The ERA shown in Figure 5.1 is a specification of a controller for a pedestrian crossing at a busy road. The pedestrians face a traffic light that signals when they are permitted to walk. The cars face a similar signal. The signal is normally red for pedestrians and green for cars. For simplicity we have omitted a yellow light. When a pedestrian wish to cross the road, he makes a request on a button placed on a pole by the curb. The controller uses the five observable actions summarized in Table 5.2.



```
Config
    observable req, car_gr, car_rd, ped_rd, ped_gr;
    int rq;
    system pedXing;
```

Figure 5.1. *ERA for a pedestrian crossing light controller.*

Action	Description
`req`	a pedestrian has made a request
`car_gr`	change light for cars to green
`car_rd`	change light for cars to red
`ped_gr`	change light for pedestrians to green
`ped_rd`	change light for pedestrians to red

Table 5.2. *Pedestrian crossing actions.*

The controller must implement the following informal time requirements:

- The light is green for cars for at least 30 seconds to permit sufficient cars to cross the intersection, even if pedestrians have requested passage.

- When the light switches there must be a period of 5 second red-time in both directions: 1) After the car signal has switched to red, 5 seconds must elapse before pedestrians are given a green light. This delay allows cars to clear the intersection before pedestrians may start walking. Similarly, 2) after switching the pedestrian signal to red, the controller must allow 5 seconds before switching the car signal to green. This delay allows pedestrians to clear the crossing.

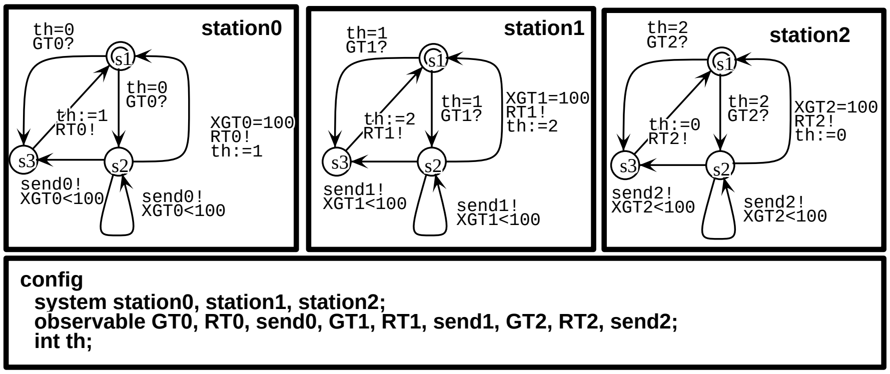
- The car signal must turn back to green after 20 seconds, if no new pedestrians request passage.

- If new pedestrians arrive and request passage while their signal is green, the green time can be extended with an additional 20 seconds. It can be extended at most three times in succession.

The specification assumes that the light signals react to the commands from the controller as soon as they are given. Also note that we have made no explicit assumptions about the frequency of requests, but they must be ignored (refused) by the controller when no request action is enabled.

5.1.2 Token Passing Protocol

The model presented in Figure 5.3 is a specification of a simple token ring network. A station is permitted to send only when it holds a token; it may hold the token for at most 100 time units per turn. A station receives the token from the network via the `GT`i action, releases it using the `RT`i action, and sends messages using the `send`i action, where i is the ID of the station.



```
config
  system station0, station1, station2;
  observable GT0, RT0, send0, GT1, RT1, send1, GT2, RT2, send2;
  int th;
```

Figure 5.3. *Token passing system with three stations.*

A station need not hold the token for the entire time slot, but may release it immediately (has nothing to send), or before the time slot expires (has no further pending messages). We have modeled this decision as a non-deterministic choice. The ring behavior of the network is achieved by a shared integer variable th that indicates the current token holder. The specification assumes that the transmission medium is able to accept messages at any time.

5.1.3 Philips Audio Protocol

The Philips Audio Protocol is a dedicated protocol for exchanging control information between audio/visual consumer electronic units. Consequently, the protocol must be simple and cheap to implement. The data is Manchester encoded, and transmitted on a shared bus implemented as a single wire. There are two interesting aspects of this protocol. One is that a certain tolerance is permitted on the timing of events to compensate for drift of hardware clocks and CPU contention. Philips permits a ±5% tolerance on all the timing, while still being able to decode the transmitted signal correctly. The second aspect is that the collisions of messages on the bus must be detected. The protocol was first studied by Bosscher et al. in [17]. It was here proven formally that the signals can be correctly decoded if tolerances are less than $\frac{1}{17}$. The protocol has since been studied numerous times in the context of model checking.

The goal of generating tests for the protocol is to compute a test suite that can be used to determine if a given audio component implements the Manchester encoding and collision detection correctly, and within the allowed tolerances.

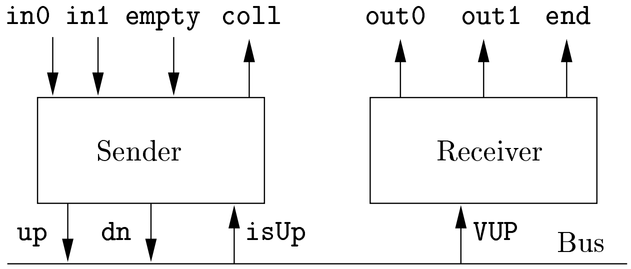


Figure 5.4. *Overview of the Philips audio protocol.*

As shown in Figure 5.4, a station is equipped with a module for encoding and transmitting data on the bus, and a module for receiving and decoding the data. The sender obtains the bit stream to be transmitted via three actions: in0, in1, and empty, respectively representing a zero-bit, a one-bit, and an end of message delimiter. The sender Manchester encodes these bits, and uses the actions up and dn to drive the bus voltage high and low respectively.

The bus works as a logical or, so whenever a station drives the bus high, the bus will be high even if other stations previously has set it low. A sender can detect collisions by checking that the bus is indeed low when the sender is sending a low. The `isUp` action is used for this purpose. If a collision is detected, the upper protocol layer is informed via the `coll` action.

The receiver informs the upper layer of the decoded bits via the `out1`, `out0`, and `end` actions. Philips uses rising edge triggering to decode the electrical signal. A rising edge is indicated to the receiver by the `VUP` action. To decode the signal using only rising edge triggering as required by Philips, messages must start with a logical one, and be odd in length.

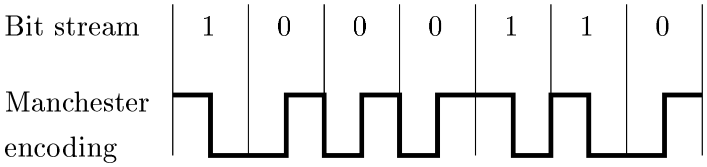


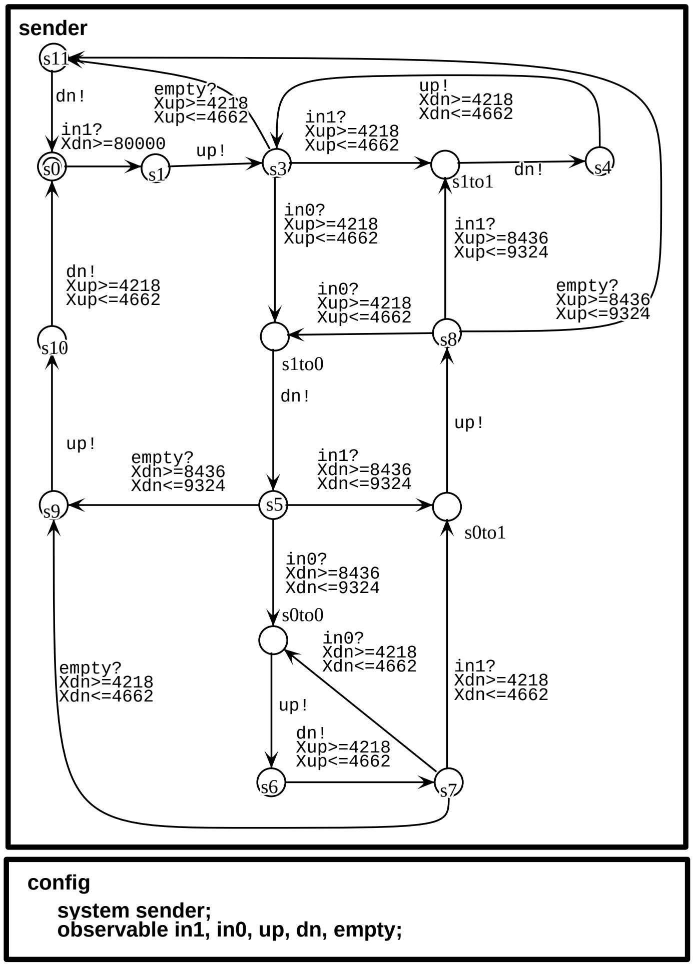
Figure 5.5. *Manchester encoding of the bit stream 1000110.*

Using Manchester encoding, the time axis is divided into equal sized *bit slots*. In every bit slot one bit can be sent. A bit slot is further halved into two intervals. A logical zero is represented by a low voltage on the wire during the first interval of a bit slot, a rising edge at half the bit slot, and high voltage during the last interval. A logical one is represented by a high during the first interval, followed by falling edge, and a low through the last half [109]. The Manchester encoding is illustrated in Figure 5.5.

A bit slot in the Philips protocol is $888\mu s$ long. In the modeling we use quarters of bit slots, denoted q, equaling $222\mu s$. The basic constants used in the model, and the derived tolerance levels are summarized in Table 5.6.

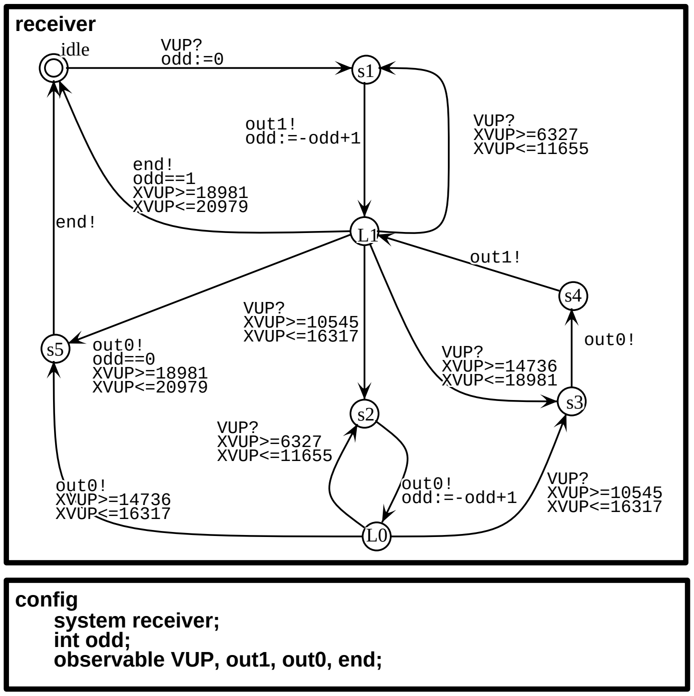
We present two versions of the sender, one with only the basic Manchester encoding, and one also including collision detection. The basic version is shown in Figure 5.7. The basic operating principle is that the sender inputs a new bit while encoding the current bit, i.e., it has read a bit ahead. The important states are labeled `SXtoY`, where X represents the bit currently being generated, and Y the bit to be generated next. Observe that whenever X and Y differ, the sender waits twice the normal duration before changing the status of the wire.

The receiver in Figure 5.8 is triggered by rising edges. The important states are `L0` and `L1`. The receiver is in `L0` when the last received bit was a zero, and in `L1` when the last bit was a one. According to [13] the model is a direct translation of the decoding algorithm described in the Philips documentation.



```
system sender;
observable in1, in0, up, dn, empty;
```

Figure 5.7. *The sender ERA without collision detection.*



```
config
    system receiver;
    int odd;
    observable VUP, out1, out0, end;
```

Figure 5.8. *The receiver ERA.*

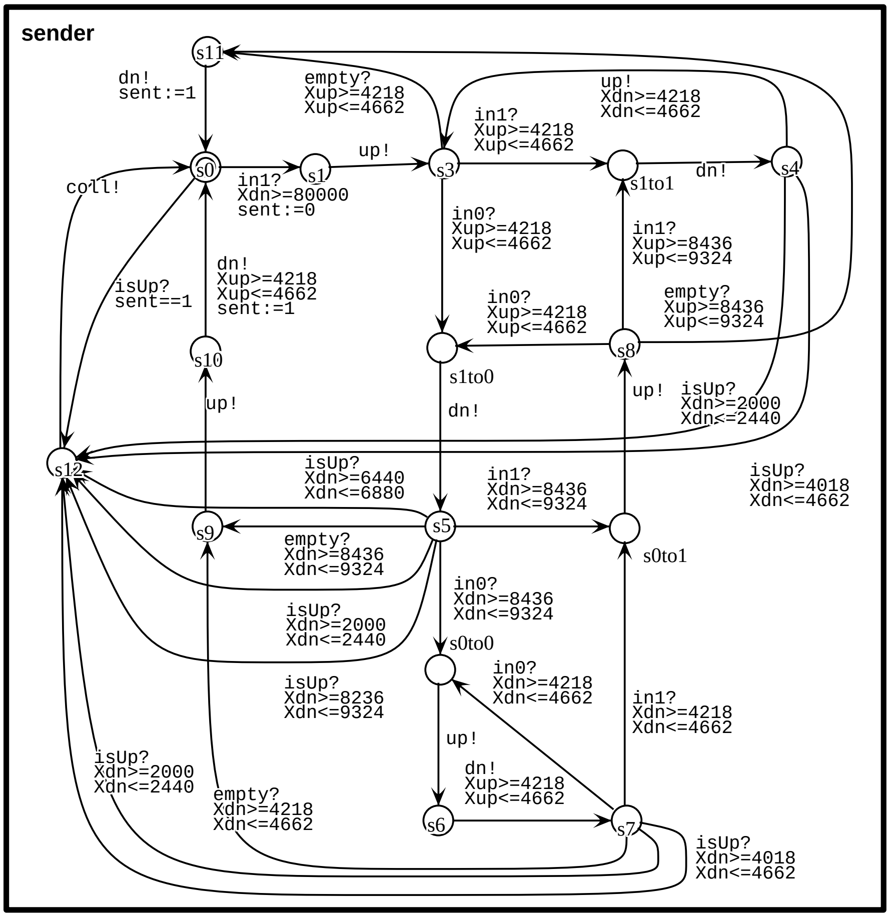
Symbol	Value	Meaning
q	2220	one quarter of a bit slot (220μs)
d	200	Detection 'just' before `up` (20μs)
g	220	'Around' 25% and 75% of the bit-slot (22μs)
w	80000	Station Silence (8ms)
t	0.05	Tolerance (5%)
A1min	2000	q-g
A1max	2440	q+g
A2min	6440	3q-g
A2max	6880	3q+g
Q2	4440	2q
Q2minD	4018	2q(1-t)-d
Q2min	4218	2q(1-t)
Q2max	4662	2q(1+t)
Q3min	6327	3q(1-t)
Q3max	6993	3q(1+t)
Q4minD	8236	4q(1-t)-d
Q4min	8436	4q(1-t)
Q4max	9324	4q(t+t)
Q5min	10545	5q(1-t)
Q5max	11655	5q(1+t)
Q7min	14763	7q(1-t)
Q7max	16317	7q(1+t)
Q9min	18981	9q(1-t)
Q9max	20979	9q(1+t)

Table 5.6. *Constants used in the ERA specification of the Philips audio protocol.*

As stated, a sender can detect a collision by checking if the wire is high when it is itself sending a low. According to Phillips the bus only needs to be sampled 'around' three specific time points, namely after a quarter of a bit slot after starting a low signal, again after three quarters (if still transmitting a low as in the one-to-zero transition), and 'just' before setting the bus high. The sender augmented with collision detection is depicted in Figure 5.9.

We note the following about our modeling:

- To function correctly the sender assumes that the medium is ready to perform the `up` and `dn` actions as soon as they are enabled by the sender.
- In our modeling of collision detection, the sender is required to be able to synchronize with the `isUp` action at all instances in the $\pm g$ interval.



```
config
  system sender;
  int sent;
  observable in1, in0, up, dn, empty, isUp, coll;
```

Figure 5.9. *The sender ERA with collision detection.*

This is probably not what the Philips engineerers have in mind. Rather, they intend to sample the bus at some point in this interval. However, this cannot be readily modeled in the current ERA language.

- The modeling presented here is inspired by the UppAal model described in [13]. The receiver could almost be directly translated from the timed automata model, but the sender had to be reformulated, see Section 5.3.1 fur further discussions on this.

- The tolerances are modeled by permitting the upper protocol layer to deliver the next bit to be transmitted at some point in the "window of opportunity". The sender is therefore required to accept bits at any time within the tolerance interval. One way the sender can implement this communication is to perform an Ada-like rendezvous call to the upper protocol layer (bit stream generator), but the called object may defer the acceptance of the call within the tolerances.

  Our model was derived from another timed automata model designed to verify that the protocol decodes a signal correctly despite timing tolerances and collisions. The interfaces to the components (sender/receiver) in an actual implementation may therefore not correspond to the interface used in this model. In consequence, the generated test cases may not be executable against that implementation.

5.2 Number of Generated Tests

In this section we shall evaluate the number of tests generated by RTCAT from the specifications presented in Section 5.1. An important goal is to bring down the size of the generated test suite.

We shall first evaluate what impact test composition and construction order of the reachability graph might have on the number of generated tests. We shall do this on basis of the pedestrian crossing and the Philips sender and receiver specifications. We shall also evaluate what impact larger specifications could have on the number of generated tests, and on the time and space consumed by RTCAT. This will be evaluated using upscaled versions of the token passing protocol.

5.2.1 Composition and Construction Order

Table 5.10 shows a series of measured parameters for three specifications: The pedestrian crossing (Figure 5.1), the Philips receiver module (Figure 5.8), and the Philips sender module with collision detection (Figure 5.9). The parameters measured are the number of reached (convex) partitions, the amount of time and memory RTCAT used to generate and output the test cases to a file, the number of reached symbolic states, the number of generated tests, and the total length of these tests.

The length of a test case is computed as the number of edges occuring in the test tree, counting refusal nodes as a single edge only. Other definitions are possible, such as the depth of the tree, but because the test trees should be covered when executed, we consider the chosen metric to be more reasonable. Observe that it is sometimes necessary to minimize the actual time it takes to execute the test suite. For example, when the same test suite is to be executed against several product items, the time spent on testing each product instance should be minimized. This is not our main concern here. The fastest test suite is not necessarily obtained using the shortest test suite.

	PedX	Philips (R)	Philips (S)
Reached Partitions	19	60	47
Time (s)	5	2	2
Memory (MB)	5	5	5
Breadth First			
Symbolic States	77	71	97
C-Number of Tests	30	97	68
C-Total Length	151	527	393
I-Number of Tests	39	118	85
I-Total Length	188	614	467
Depth First			
Symbolic States	247	85	98
C-Number of Tests	26	86	67
C-Total Length	595	1619	487
I-Number of Tests	39	118	85
I-Total Length	676	2103	587

Table 5.10. *Experimental results from generating tests from sample specifications, and with different tool configurations. C=composed test, I=individually generated test.*

There is a total of four possible combinations of construction order and test composition, all included in Table 5.10. The number of reached partitions is the same in all four combinations, as it should be. The tabulated time and memory usage figures are the maximum values observed for the four configurations. The most time consuming configurations were the ones involving depth first construction. From an applicability point of view, these differences are insignificant. All test suites were generated in a matter of few seconds and used about 5Mb of memory.

More importantly, observe that the test suite size in all combinations is quite manageable, and constitute test suites that could easily be executed in practise. One may even worry whether they contain too few tests. There is thus a large margin allowing for more test points per partition, or longer tests.

Next compare the effect of construction order. The construction order may influence the number and length of tests because tests will only be generated for the first symbolic state reaching a partition during reachability analysis.

Our results show that depth first construction generates slightly fewer tests than breadth first, but also considerably longer test suites. Thus, when the goal is to minimize the time needed to execute the test suites, breadth first construction is clearly preferable (assuming that the average delay between events is the same). When the goal is to check the behavior of the implementation after longer sequences of events, but still ensuring coverage, depth first generation is preferable.

Next compare the effect on composing tests versus generating them individually. Composing tests consistently results in both fewer, and also shorter test suites. However, the obtained reductions are only in the range of 10-25% that, although significant, is less than hoped. A possible explanation for this is that these specifications contain only little non-determinism (in fact, only the Philips receiver is). Therefore, the test composition algorithm is only able to mutate the test case at its single leaf. The reduction therefore only avoids sub-traces of already generated tests.

5.2.2 Scalability

To examine how RTCAT behaves when the size of the specifications increase, we have benchmarked its behavior against the token passing protocol illustrated in Figure 5.3 with an increasing number of stations.

The results contained in Table 5.11 indicates a problem, not with the number of partitions or the number of generated tests, but with how reachability analysis is performed, and with the number of symbolic states needed to terminate. Observe that the number of symbolic states used to represent the state space increases dramatically as the number of stations increase. At the same time neither the number of partitions nor the number of tests increases at the same rate. In this example, the bottleneck is our reachability analysis, and not the number of generated tests.

Stations	1	3	5	7	8	$5'$	$10'$
Symbolic States	14	191	3584	15427	52976	62	125
Reached Partitions	6	18	30	42	48	30	60
Number of Tests	11	31	51	71	81	51	101
Total Length	22	126	310	574	736	310	1120
Time (s)	1	2	35	541	4050	1	3
Memory (MB)	5	5	11	39	136	5	5

Table 5.11. *Experimental results of scalability experiment. The tool is configured to compose tests, and to use breadth first construction. The columns labeled $5'$ and $10'$ refer to a modified token passing system using only one clock.*

Primarily caused by high memory consumption, but also by lack of speed, the tool is consequently incapable of generating a guaranteed covering test suite for systems with more than eight stations using breadth first reachability analysis.

There is a number of explanations for this behavior:

1. The reachability analysis is done on the level of (convex) partitions. This means that when performing forward reachability, the symbolic successors for each such partition is computed. In contrast, typical model checkers uses coarser states, and typically only adds one symbolic state whenever a new control vector is visited. Because our partitioning is finer, our symbolic states are smaller, and more symbolic states are therefore needed.

2. Not only the number of stations increase in this example, but because every station has its own clock to measure the token holding time, so does the number of clocks. For an eight station system there are nine active clocks (one for each `GT`i action, and one shared by all other actions). We believe that the number of clocks and the relatively small symbolic states makes it difficult for the tool to find a symbolic state in the passed list fully containing the current one, and in consequence, the tool is forced to unfold each state a number of times before concluding that no further partitions are reachable. If we reduce the number of active clocks in the specification by using the same `GT` action in all stations, the required number of states drops dramatically, although there are still many compared to the number of partitions. The clock reduced versions with respectively five and ten stations are labeled $5'$ and $10'$ in Table 5.11.

3. The extreme amount of memory is used to store more than the passed list. Both the passed list and an explicit test graph is build. A node in the test graph contains, in addition to a symbolic state, various book keeping information such as lists of the (single step) must, can, and refusal properties holding in that state, and a DBM for back propagated constraints, etc. In fact, every symbolic state is stored twice, once in the passed list, and once in the test graph. The rationale for this design is that 1) the symbolic states in the passed list may contain extrapolated information which cannot be used in test time point selection, and 2) our design is made such that only the part of the test *tree* needed for each test case (or closely related test cases) currently being generated need to be stored. However, because our prototype, for the ease of making modifications to the code, operates in distinct phases, this option is not utilized, and we experience the penalty of constructing the full test graph.

4. The long execution times are primarily caused by the problems outlined in points 1 and 2. Because a large number of symbolic states are instantiated per partition, and because members of the same (concave) partition hash to the same entry in the hash table implementing the passed list, there is consequently many collisions. This again means that much time is spent on checking inclusion between a new symbolic state, and all the states in the same entry.

5. Depth first construction is more memory efficient than breadth first construction in this case; it about halves the number of symbolic states, and is therefore faster and uses less memory. However, as previously noted, the test suites become much longer; up to ten times as long in this case.

Whereas the number of tests generally equals the number of reached partitions multiplied by the total number of must, may, and refusal properties in each partition, it is more difficult to predict how the number of partitions increase with specification size, for example measured by its the number of edges or parallel components. In this example, the number of partitions apparently grows linearly with the number of stations. This well behavedness should only be expected with similar specifications where only one of the parallel components is active at a time, and where the guards are simple. In general, an exponential growth should be expected.

5.3 Testing Setup

The main ingredients in our testing "set up" is the specification language, the test language, and the test selection strategy. These will be evaluated in turn. We also discuss an hitherto untouched aspect of test generation, namely environment modeling.

5.3.1 Event Recording Automata

The following sums up our experiences using the ERA formalism as a specification language. We originally decided to use ERA because it constitute a simple and a clean subset of timed automata, and not the least, a determinizable subset. However, these benefits come at a cost. First, the restrictions could reduce the convenience of timed automata; convenience meaning that the required behavior can be expressed with a reasonable effort and elegance. Second, the restrictions could imply a loss of expressiveness, i.e., it becomes impossible to express requirements that are expressible in the unrestricted language.

On the theoretical expressiveness, it was shown by Fix et al. in [10] that ERA are a strict subclass of timed automata. This means that there are timed languages that can be accepted by a timed automaton that cannot be accepted by an ERA. Not even all deterministic timed automata can be expressed as a trace equivalent ERA. Thus, ERA are less expressive than timed automata. On the other hand the same paper shows that the timed transition system model by Henzinger et al. [51] can be transformed into a (trace) equivalent ERA model, and thereby obtaining the result that language inclusion is decidable for timed transition systems. This indicates that ERA is at least as expressive as another well established specification language. The transitions in a timed transition system are labeled with an action, and an upper and lower time bound. A transition is enabled when the time elapsed after the source location of the transition was entered is between the lower and upper bound.

We have shown through a series of examples how ERA can be used to specify interesting and practically relevant properties. Recall that the main restriction in ERA is that clocks are tied to observable actions, and that the corresponding event clock is reset automatically when the action occurs. When the specification can be expressed as a single automaton, this restriction is in our experience only a minor inconvenience. We have been able to work around this restriction by adding extra states or extra action names:

- Frequently we wish to measure the time elapsed after a location is entered. However, the location might be entered by different observable actions. In timed automata one would simply reset the same clock along all entry actions. In ERA this is impossible. Instead, the single location must be replaced by several locations (one for each entry action) such that the proceeding behavior can refer to the correct event clock. Alternatively, integer variables can encode which of a set of clocks to use in guards.

- Sometimes we wish to measure the time elapsed since the first occurrence of an action, but not wanting the event clock to be reset on succeeding occurrences. A simple work around is to use several names, and thus more clocks, for the same real action.

- When an ERA model is being developed, it is frequently convenient to use τ actions. Because ERA do not allow internal actions, these must be eliminated later. Our experience suggest that this is often possible by syntactically rewriting the specification.

- ERA models tend to use more clocks than timed automata models. At the syntactic level, there is a clock for each observable action. Because the number of clocks affects the complexity of the symbolic analysis, the execution time and space consumption of the generator may increase to an impractical level. However, all clocks will rarely be used in guards, and therefore their values are irrelevant to the analysis, and can consequently be eliminated. Even when these obviously inactive clocks are eliminated, an ERA model typically uses more clocks compared to a manual timed automata specification. We expect that the clock reduction algorithms that have been developed in model checkers can be used to reduce this potential problem to a manageable level.

We have also identified a set of more serious limitations, which makes us hesitate in accepting ERA as the ideal specification language:

No Internal Actions: We have seen that τ actions can be used in the construction of a model if they can be eliminated afterwards. The lack of τ actions implies that an ERA cannot autonomously change state. It can only advance from one state to another by synchronizing with the environment. This means that an ERA cannot express a situation where an action is required to recur indefinitely without also requiring synchronization with the environment in the interim. An example is the square wave generator timed automaton shown in Figure 5.12. The state of the wave (high or low) is indicated by the automaton by its preparedness

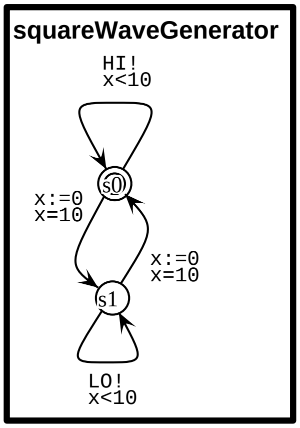


Figure 5.12. *Timed automaton specification of a square wave generator.*

to synchronize on a `high` respectively `low` action. This can only be expressed as an ERA if the number of squares that should be generated between two synchronizations are bounded by a finite number.

We expect there to be practical cases where such autonomous behavior is required.

No Internal Communication: The basic ERA language consists of a single automaton without τ actions. This means that a network of concurrent and communicating automata cannot easily be expressed in the basic ERA language: When synchronization is possible on several actions, one pair is chosen non-deterministically; this implies that it also becomes non-deterministic which event clock is reset. The environment is then no longer in control of which clock is reset, which is otherwise a profound property of ERA.

This limitation has a profound impact on the development methodology envisioned in Section 1.2.4. The idea was that tests should be generated from a quite concrete model (a set of communicating concurrent components) of the system under test. If ERA are to be used, test generation should start from a more abstract level where the required observable behavior is specified, but where the system has not been decomposed into components. Sometimes it is possible to manually perform the required abstraction by systematically removing the internal actions and synchronizations among components.

Alternatively, or supplementary, each component could be tested separately. It is also possible to use ERA to specify only certain aspects or *properties* of a system. If such a property model contains enough behavioral information, the generated sequences will still constitute a relevant test for the system.

No Timing Uncertainty: In our interpretation of ERA, it is not possible to specify that an action must become enabled at some time point in an interval, but that the implementation is free to choose which.

A workaround that some times is applicable is to use a non-deterministic choice between an edge that enables the action early, and between one that enables it later. This will at least produce the correct verdicts of the generated test cases. However, if many time requirements are of this form, the large number of required extra states and edges makes the workaround impractical. This will also result in a large number of inconclusive verdicts at test execution time, because the event did not occur at the expected time instant.

In our work we have assumed synchronous and fully symmetrical communication, but a distinction between inputs and outputs further clarifies the issue. Often, the environment is in control of when to supply the inputs to the system, whereas the system decides when to produce outputs. We find our communication model most reasonable for input actions because the system is expected to accept them at any time when the action is enabled. The nature of outputs is normally different. The readiness of the system dictates when outputs are delivered, and it does not generally appear reasonable to require this after a single specific delay, but rather before a certain amount of time has expired. Thus, whether an output is delivered or not, and when it is delivered is not controllable by the tester.

We believe that there are a significant number of applications in which the present strict timing mode is appropriate, e.g, in the area of strictly timed embedded controllers. On the other hand, for example in communication protocols, one can rarely require a message to be delivered after a specific delay, but only within certain bounds. In general, timing uncertainty is needed.

The simplicity of the ERA language, and its ease of analysis, made it an excellent vehicle for exploring our ideas. However, we also conclude that the basic ERA lacks some convenience and expressiveness.

There are two possible strategies for generalizing the specification language. The first strategy is to extend the ERA model with the needed facilities, while still trying to preserve its ease of analysis with respect to the particular analysis our test generator needs to perform. One could imagine that a slightly extended ERA language could be used as an assembler language into which a more convenient user visible specification language could be compiled. The second approach is to generalize our symbolic algorithms to handle a larger subset of timed automata. Further work is needed to determine what approach will be the most promising.

5.3.2 Test Language

Our test language consisting of timed may and must properties were obtained as a straightforward extension of the untimed versions. These tests dictates the specific time points at which communication should be attempted.

One may argue that it is difficult to test these properties in practice because it is impossible to ensure that an action is enabled in precisely the required moment. However, it is our belief that a successful synchronization in a small interval around the desired time instant is sufficient in most practical situations. The absolute size of the interval obviously depends on the magnitude of the delays in the trace, of the required precision, and also of the precision of the physical timers in the test system.

Further, the test language can check that the implementation refuses an action that is refused by the specification at a single time point. An important improvement of the test language will be to check refusal of actions over durations of time rather than only at specific time points. Fortunately, this improvement seems straightforward to implement with our current specification language and symbolic analysis techniques.

The test language is most effective when the specification requires "strict" timing of the implementation, as indeed is the case in our interpretation of ERA. If actions with timing uncertainty (e.g., delivery of an action at some point in an interval) dominated the specification, our testing language would be ineffective: Many test executions would result in inconclusive verdicts because the implementation should only possibly engage in the synchronization at the specified time point. Instead, it would be more effective if the tester continuously offered its action during the entire interval in which the implementation was expected to accept the action. This extension would benefit from a change from offline generation to online generation. See Section 6.1.2 for a detailed discussion of online and offline testing. In an online generator, the specific time point at which the synchronization took place could be fed to the generator that could then narrow the expected interval of the next action. Online testing will require that the test system is sufficiently fast to interpret the specification symbolically in real-time. Alternatively, symbolic test cases could be generated, meaning that the clock constraints characterizing the symbolic path leading to the partition to be tested should be preserved in the test case, and be interpreted and instantiated at test execution time.

As we shall elaborate on in Section 6.2.1, our test language does not fully characterize the testing preorder. It would be ideal to generate tests based on such a characterization, but to our knowledge such a theory does not exist for timed automata. Also, candidate theories should be carefully evaluated to balance the increased testing power with the potential increase in the number of test cases, and the potentially increased complexity of generating them.

In conclusion, timed must tests is adequate for the current setup. It has constituted a challenging case for our test case generation method. However, it should be extended to include continuous refusal and enabling of actions.

5.3.3 Environment modeling

Throughout the thesis we have implicitly assumed that a specification is self contained, and that the implementation should operate correctly in a universal and unrestricted environment. However, in some cases the operational environment of the implementation must also be taken into consideration. The environment consists of the components with which the implementation under test communicates. This can either be components in the computer system, or components in the external physical environment.

The dependency on the environment occurs at least in two situations:

- Sometimes the specification is developed under the assumptions of a specific environment. This issue arised numerous times in the specifications given in Section 5.1.
- It may not be necessary to establish that a component is correct in any environment, but frequently it suffices to establish correctness in a specific environment only. First, the component may not work (and is not required to) at all in other environments, and establishing correctness wrt. to these are meaningless. Second, it may be possible to reduce the size of the test suite when a specific environment is considered. This means that only the test cases that are relevant with respect to this environment should be generated.

The environment assumptions can be modeled explicitly by a collection of timed automata, and these can be fed to the test generator along with the specification. Formally, the test generator should generate test from a synchronous product of the specification and environment model. Our current interpretation of ERA offers only urgent actions, and is therefore not generally appropriate for environment modeling; the environment usually has a substantial freedom in deciding when to perform which actions. Timing uncertainty is desired for this purpose.

Manually modifying the ERA specification to implicitly include the environment assumptions is rarely possible because a test generator would need to distinguish between events that are forbidden and events that are not relevant. Otherwise, irrelevant or unsound tests could result. Thus, for both methodological and technical reasons, our approach should be extended to facilitate explicit environment modeling.

5.3.4 Test Selection

Our approach to test selection is primarily guided by the chosen partitioning. We have focused on one possible partitioning strategy, stable edge sets, which we believe, and have argued in favor of, would constitute a good compromise between generating a reasonable number of tests, and still capturing sufficient timing aspects to be able to detect significant timing faults.

So far we have no practical experiences using our approach to test real systems, and have consequently little factual knowledge about its fault detection potential in practise. An alternative way of evaluating its fault detection potential could be to state a *fault model* describing the implementation faults that could be detected, and then state the assumptions on the implementation necessary to detect all such faults. Some central observations towards such a characterization include:

- The implementation behaves *uniformly* in each partition, i.e., if it behaves correctly in one point, it does so for all. Extreme values can be used to support this assumption.
- The implementation does not introduce further partitions, or at least, the number of partitions in the implementation is explicitly bounded.
- The symbolic transitions that take place between partitions are implemented correctly. This can be ensured by applying checking sequences, see Chapter 6, and assuming uniformity.

The development and formalization of such a fault model and its assumptions would be academically very fruitful, and provide valuable insight into the underlying properties of the chosen strategy. On the other hand, its practical value is less obvious because implementations rarely are timed automata that are simple mutations (change clock constants or add/remove edges, locations and timers) of the specification automata. Instead they are typically implemented using a programming language, operating system, and hardware. A more valuable approach in practice would be to characterize the timing faults that actually occur in real implementations, trace these back to the specification, and develop the selection strategy based there on. This would require a large amount of practical work and empirical studies.

We find the low number of generated tests very encouraging.

Our proposed partitioning strategy has two potential disadvantages:

- If the specification is very large, the number of generated test cases may exceed what can be executed. There are two strategies to deal with this. The first is to limit the size of the partition and reachability graphs used in the test generation process, for example by using the partial techniques proposed in Section 4.4.2. The second strategy is to choose a coarser partitioning, for example by only insisting on testing every action or every edge. It is an open question under which circumstances which strategy is preferable. Currently, only options from the first category are supported by RTCAT.

  In the other extreme, when specifications are fairly small, there is a choice between executing longer traces by unfolding loops, or generating tests from a finer partitioning. Only the first choice is straightforward to include in the current tool.

  We would prefer a technique where the user could tune the partitioning to his particular needs, or perhaps even let the tool make an optimal choice, given a certain amount of testing resources.

- A theoretical computational limitation lies in the partitioning itself. From a given set of control locations, the constraints characterizing the enabledness of each subset of edges must be computed. This is exponential in the number of edges, and can potentially become a limiting factor in practice. Two factors contribute to the number of partitions. One is the degree of non-determinism: The more non-deterministic, the larger the set of location vectors in the determinized automaton, and the larger the number of edges from the union of these control vectors. The second factor is the complexity of the guards. If one is very unfortunate, all subsets of edges could have a solution. Currently, the RTCAT implementation forms all subset of edges from a node in the partition graph, and checks if the resulting constraints have solutions. Our experience indicates a practical limit of this approach of less than 20 edges with non-trivially true guards.

Also, because we have decided to treat the disjuncts of the disjunctive normal form representation of a partition as separate convex "smaller" partitions, the number of conjunctions in guards also contribute with a blow up.

It is easy to give unrealistic specifications that approaches these limits, but to what extend the practical applicability will be limited is a rather different issue, that we believe best can be resolved through further specification of real-life systems, and application of our techniques.

We are cautiously optimistic about the size of specifications that can be handled using our technique on the following grounds. First, specifications seem to be only modestly non-deterministic, and guards are rarely very complex. Second, ERA models only need to specify the externally observable behavior. Such specifications rarely consist of a large number of internal components, and consequently rarely has a large number of simultaneously enabled edges. Further, as briefly discussed in Section 5.3.1, it might sometimes be possible to decompose a specification into smaller parts modeling different aspects of the system behavior, or into properties, that can be tested separately. Thus, there seem to be a large number of potential applications that could benefit from our technique.

5.4 Implementation

The following sums up our experiences implementing test generation using symbolic reachability techniques.

5.4.1 Symbolic Reachability Techniques for Testing

We have shown how symbolic techniques invented for model checking can be used to generate test cases. The two main ingredients in our solution are 1) a partitioning of the state space characterized by enabled edges, and 2) the use of symbolic reachability analysis to find the reachable parts of the partitions, and to compose test properties into test trees.

The strength of this approach is that it systematically explores the state space, and generates tests for the partitions yet uncovered. The resulting test suite is well defined, and guarantees full coverage.

A potential weakness is the means available to handle very large specifications where attempts to generate the partition graph and perform exhaustive reachability analysis would be futile.

Our use of the symbolic techniques requires several phases: Partitioning, reachability analysis, back propagation, and trace generation, and in each phase many small details must be considered. Some of the encountered principal problems have been the representation of concave solution sets, and the “chopping” up of the state space into a large number of symbolic states, especially when many clocks are involved. The techniques are therefore non-trivial to employ efficiently, and require a substantial implementation effort.

5.4.2 Prototype Tool Implementation

The current implementation can and should be optimized both with respect to speed and memory usage of reachability analysis, and the amount of space used to store book keeping information for test generation.

Specifically, our experiences suggest that RTCAT could be improved significantly by storing “coarser” information in the passed list, and by performing reachability analysis also on a coarser level. However, changing the reachability analysis is non-trivial to implement because the test tree required for test case composition would then have to be reconstructed afterwards from the coarser symbolic states. This would effectively mean that parts of the reachability analysis would be computed twice, once in forward reachability analysis, and once to construct the partitions and their individual successors. Avoiding this extra amount of computation was our original motivation for doing reachability on partition level. However, given the potential space savings and the current termination problems, this decision should be reconsidered.

It is also clear that only the part of the test graph used for the current test (tree) should be stored to save memory. This implies that the test tree construction algorithm, test composition algorithm, and test output should operate alternatingly, rather than in the present distinct phases. This idea can be extended to a general tool architecture where all lower level modules are being invoked on a need basis (lazy evaluation) as dictated by the upper modules.

The current DBM implementation is reused from an old simulation/visualization tool. Significantly improved implementations both with respect to space and time are now used in newer versions of the UppAal tool, cf., [64]. Application of these newer techniques will enable exhaustive test generation from even larger specifications.

5.4.3 Verification Techniques and Testing

We stated in Section 1.2.4 that verification and testing were complementary techniques solving different problems. Yet we have successfully applied model checking techniques for the generation of tests. This raises the question of what their differences and similarities are seen from a tool development point of view. We shall here try to convey our experiences gained through the present work.

The main objective of verification is to automatically *prove* by exact means that a formalized model satisfies certain properties, or that certain refinement or simulation relations exist between such formal objects. Recurring themes in this line of research are the expressiveness of the used languages, their decidability properties, and the size of the models that can be analyzed. In contrast, testing does not in practice aim at establishing a proof, but aims at obtaining best possible confidence in the correct operation of a physical system under the given circumstances and resources. Applicability of testing is not necessarily limited by the size of the system to be tested, or by the availability of specifications that can be exactly or fully analyzed. This gives the test tool developer approximation options (and challenges) not available to verification tool developers. Testing techniques as means of approximating verification have been investigated by Clarke and Lee in [29, 28, 30], see Section 6.2.3.

Despite of these differences there is also a lot of common ground between these approaches. In particular, specification languages, their formal semantics, and implementation relations seem to be in common when using automated testing. Furthermore, progress in symbolic execution techniques, and data structures and algorithms for representing and analyzing the specification at the semantic level will benefit both camps.

The differences seem to dissipate when test tools attempt to guarantee a certain coverage, as in our case, or to construct optimal test cases by either minimizing test suites, or by reducing the amount of inconclusive verdicts.

It should also be noted that other verification techniques such as exploiting structural symmetries in the specification and partial order reduction techniques have interpretations as test selection strategies [73, 20]. We argue that the analysis techniques developed for verification in many cases also can be used to improve testing.

5.5 Summary

The evaluatation of our technique was done both quantitatively and qualitatively by examining a number of example specifications. One is an attempt to model and generate test for a realistic real-life example, the Philips audio protocol.

The simplicity of the ERA language, and its ease of analysis, made it an excellent vehicle for exploring our ideas. We find that ERA are sufficiently expressive, but are sometimes inconvenient to use. More seriously, our current interpretation lacks a way of specifying timing uncertainty. The problem with timing uncertainty also appears in our testing language which allows instantiated time traces only. Because the occurrences of outputs from the implementation are uncontrollable, the result is too many inconclusive verdicts during test execution. Our technique should also be extended to allow modeling of environment assumptions.

The number of computed partitions, and consequently, the number of generated test cases, is very reasonable, and can easily be executed in practice. Although potentially a bottleneck, also the space and time used to generate the tests is very encouraging, and suggests that even larger specifications can be handled. The current implementation can be optimized in many ways, most notably by employing a lazy evaluation architecture where only the needed parts of the partition and reachability graphs are computed on a need basis.

Chapter 6

Related Work

In the following we review some of the work that have appeared on methods and tools for automated testing, and relate this to our work. The discussion is structured into an untimed part appearing in Section 6.1, and a timed part appearing in Section 6.2. We outline the novelties of our approach in Section 6.3.

We shall introduce an alternative method of testing based on checking sequences for finite state machines. This has recently been applied also to real-time systems. Other topics include the choice of implementation relation, the approach to test generation, be it online or offline, and the strategy used to select tests.

6.1 Untimed Testing

This section discusses related work that does not deal explicitly with real-time, but which is nevertheless important. We first take a deeper look at both the theoretical and practical issues that arise from testing of concurrent systems. Then two extreme approaches to automatic testing, offline and online test generation, are identified. The description of test generation using checking sequences follows. Finally, we discuss test selection. Our test selection method is rooted in well known sequential testing techniques.

6.1.1 Test Observations

In the presented untimed testing theory we have assumed that the outcome of executing a test could be found by reading the verdict label of the state in which the execution of the tester and implementation *deadlocks*. However, this assumption is not without practical complications, as we shall discuss in the following. The comments also apply to our real-time preorder.

The first problem is how to conclude that the execution of the tester and implementation has entered a deadlocked configuration, i.e., that no further synchronizations are ever possible. It is of course impossible to wait an infinite amount of time to determine whether a deadlock has occurred, or whether the implementation is just too slow to respond. In practise, extremely long response times are unacceptable, and consequently expiration of a carefully set timer can be used to declare deadlock. Another consequence of our assumption is that deadlocks and internal divergence (live lock) in the implementation are indistinguishable. However, such information about divergence could be of great value when diagnosing why a test failed.

The second problem is related to the certainty with which observations can be made in face of non-determinism. The essential problem is that satisfaction of a must property is quantified over all computations of the implementation, whereas a single test execution only reveals one.

When the execution of a must test deadlocks in a fail state, it can safely be concluded that the system is erroneous. However, from the observation of a successful computation, it cannot be concluded that the implementation always passes that test. Similarly, when a may test is successful, the implementation certainly had the desired trace, but when it is inconclusive, neither presence nor absence of the trace can be concluded.

Even if the same test were executed multiple times, it is unlikely that the implementation has followed all possible paths or interleavings. Hence, there is no means by which a black box tester can guarantee satisfaction of a must test. It is frequently assumed in practice that a finite number of re-executions of the same test will pass through all computations of the implementation (called the *complete-testing* assumption in [108]). An alternative to the complete testing assumption is to equip the tester with capabilities for monitoring which internal computations have been taken, or possibly even controlling which are taken, c.f., Brinch-Hansen [48] and Taylor et al. [110].

Deadlocks are not the only possible observations; indeed an abundance have been proposed. For an overview and classification we refer to Glabbeek [113]. Different assumptions induce different implementation relations that differ in how discriminating they are. For example, we could assume that the tester, in addition to observing deadlocks, also could recover from the deadlock and *continue* testing by enabling an alternative set of actions. Adopting observation of such communication failures leads to an implementation relation called the *failure trace* preorder. It turns out that such observations become central when time is taken into consideration, see Section 6.2.1.

Another fascinating example, albeit quite exotic, is observations corresponding to making copies[1] of the implementation in its current state. Each copy can then be subjected to different experiments. This technique enables testing of the very discriminating the 2/3-bisimulation (ready-simulation) [67] preorder.

Nearly all of these theoretically based preorders are based on symmetric and synchronous communication between tester and implementation. However, in practice communicating systems often distinguish between inputs and outputs: *Inputs* signify data given to the system, and *outputs* data being produced by the system. Inputs cannot be refused, but may of course be ignored, i.e, system entities are assumed to be input enabled. Tretmans proposes in [112] a preorder **ioconf** that relates processes when the outputs of the implementation after a trace are included in the outputs of the specification after the same trace. Tretmans also defines a stronger version **ioco** that additionally requires that the implementation only refuses to deliver outputs when the specification also refuses to deliver outputs.

6.1.2 Approaches to Test Generation

We distinguish between two extreme types of test generators: Online and offline generators. An *offline* generator does all work offline before test execution begins. It interprets the specification, constructs the success graph, traverses it to construct test cases, and performs test selection. The entire test suite is thus constructed a priori, and is typically stored in a set of files from where it can be recalled when a new product is to be tested.

An *online* generator constructs a test case as it is being executed. During test execution the test driver simulates the specification. The test driver constructs a set of actions that the implementation is expected to be able to synchronize on in its current state. This set of actions is typically chosen randomly from the success sets from the specification's current state. If synchronization were successful, the synchronization action is fed to the simulator that computes the states that the specification can reach after performing the action. A new success set is then computed and the procedure is continued. If the synchronization attempt was unsuccessful the test driver stops execution and reports the appropriate verdict. The test driver can be restarted to perform a new test for as long as time and other resources permit. Thus, only the parts of the state space and success graph actually executed needs to be constructed. Online testers are thus special cases of *environment simulators* where the interactions are derived from a well defined testing theory.

[1] Provided that we have yet to master the sophisticated technology required to realize Star Trek Replicators [63] we may need to settle with snapshots or core dump approximations.

An example of an offline generator is the TGV (Test Generation with Verification technology) toolkit, developed at the University of Rennes, France by Jéron et al. [57, 19, 41]. Input to TGV consists of an SDL or LOTOS specification and a collection of test purposes. A *test purpose* expresses a particular property that the implementation must satisfy. The tool then constructs one *test graph* for each test purpose such that failure to pass the test implies failure to satisfy the test purpose. A test purpose is modeled as an automaton with a subset of states labeled with 'pass' or 'reject' verdicts. The test graph is a controllable subset of the graph consisting of traces leading to pass verdicts. A test case is controllable if it never has a choice of outputs, or a choice of producing an output or accepting an input. TGV is based on an asynchronous testing theory that distinguishes between inputs and outputs.

The test graph is constructed in several steps. First the specification is τ reduced and determinized. It is then composed with the test purpose via a synchronous product construction. The result is the behavior that is relevant for the test purpose. The 'reject' verdict is used to limit the size of the product graph by not constructing behavior that has been deemed irrelevant by the test engineer. The graph is then traversed in two phases to resolve controllability conflicts. It should be noted that TGV performs τ reduction and determinization *on-the-fly*, i.e., only the used state space is constructed and processed.

The test purpose plays an essential role in this approach since it determines which tests to be are generated. TGV thus uses test purpose based test selection. Because the test purposes are written by test engineerers, this is a manual strategy.

The techniques of TGV has been integrated into the commercial testing tool OBJECTGEODE from Verilog [59].

TROJKA by de Vries and Tretmans [37] is a tool for online testing. It accepts specifications written in the Promela protocol specification language [53]. A TROJKA configuration consists of three logical components: A specification interpreter, a test driver, and the implementation under test. The current interpreter is a modified version of the SPIN model checker [53]. The interpreter keeps track of the states reachable after the trace executed so far, and computes the actions possible in the next step. The test driver is the "middle man" which supplies the inputs produced by the interpreter to the implementation, and which invokes the interpreter with the resulting output.

In our view a good online tester systematically constructs and applies all possible tests, and avoids re-executing a test that has already been passed. Thereby it may obtain better coverage. From the description in [37] TROJKA does not appear to address the issue of obtaining or measuring coverage: It does not save information about the test cases already executed, and thus risks re-executing the same test. It is also unclear whether the tool chooses between the actions produced by the interpreter randomly or in order.

The VVT-RT (Verification, Validation, and Test for Reactive Real-Time Systems) testing tool by Peleska et al. [81, 79] is also based on an online approach, but addresses the issues of systematic test generation and re-execution of test cases. A CSP specification is compiled to a deterministic graph labeled with refusal information similar to our notion of a success graph by the CSP refinement checking tool FDR [93, 94]. This graph is then interpreted at run time to generate test cases and evaluate their outcome.

In addition, a *test monitor* is used to determine the achieved coverage. It is concerned with two types of coverage. The first type is what parts of the refusal graph have been covered, i.e., which requirements remain to be tested. The second type detects what internal paths or components have been activated during a test. Because the implementations may be non-deterministic, this may vary from one execution to another. It may be important for the test engineerer to know precisely which components or paths have been activated, for example which of a set of redundant components was active in a fault tolerant system. For this reason, the testing monitor can also be equipped with probes into the internals of the implementation under test that enables the monitor to track which internal paths have been executed. The necessary instrumentation of the implementation must usually be done manually.

A final remark is that VVT-RT proposes *hardware in-the-loop* testing. This means that a separate test computer is used instead of the implementations operational environment, and that the test computer is connected to the external physical interface of the implementation. This tool also has the capability of generating real-time tests, see Section 6.2.3.

Our approach is based on an offline approach where we systematically analyze the specification and cover this with tests. However, as noted in Section 5.3.2, timing uncertainty requires symbolic test cases or using an online approach. We shall therefore consider how to interpret event recording automata dynamically in future work.

6.1.3 Checking Experiments

A substantial amount of research was carried out in the period from the 1950's to the early 1970's on theories and algorithms for testing of sequential hardware circuits. This research resulted in efficient test generation algorithms that even guarantees full fault coverage under a specific set of assumptions. We refer to Lee and Yannakakis [68] for a recent survey of these results. The techniques are now being resurrected, generalized, and tried out in the context of conformance testing of communication protocols and reactive real-time systems. We shall therefore outline the key points of these techniques.

The theories were originally developed for Mealy-machines which are finite state machines where transitions are labeled with an input/output pair $s \xrightarrow{i/o} s'$ such that the machine upon receiving input action i produces output action o. Two *states are equivalent* iff for every input sequence both states produce the same output sequence. Two *machines are equivalent* iff for every state in one automata there exists an equivalent state in the other, and vice versa. This implies trace equivalence. A *checking sequence* is a sequence of input actions that is able to distinguish inequivalent implementation machines from a known specification machine under the following assumptions [68]:

1. The specification and implementation are both *deterministic* Mealy machines.
2. The machines has the same set of input and output actions.
3. The specification is minimized (it has no equivalent states).
4. The specification is strongly connected (for every pair of states there is an input sequence that transfers the machine from one state to the other).
5. The specification is completely specified (for every state there is an outgoing transition for every input action).
6. The specification has n states, and the implementation has at most m states, $m \geq n$.
7. The particular W-testing method discussed below also requires a reliable reset operation.

The basic idea in a checking experiment is to ensure that every transition of the specification is correctly implemented by the implementation. A checking experiment follows the following generic algorithm:

1. For every specification transition $s \xrightarrow{i/o} s'$, apply an input sequence that transfers (a correct) implementation to state s.
2. Apply input i, and verify that the output equals o.
3. Verify that the destination state is (equivalent to) state s'.

There is a host of techniques for verifying that the implementation is in the expected state (algorithm step 3). One, the so called W-method proposed by Chow in [27], uses a *characterizing set* for state verification. A characterizing set W for the specification machine is a set of input sequences that can distinguish the behaviors of all states, that is, for every pair of states s, s' $(s \neq s')$ there is an input sequence in W such that the output sequence produced by s differs from the output sequence produced by s'. This method thus requires that the same state is checked using all input sequences in W. This implies re-application of the transfer sequence in step 1, hence the need for a reliable reset action.

Let P be a set of input sequences that visits every transition of the specification machine (i.e., that constitutes a transition cover). P is the set of actions needed in steps 1–2 in the above algorithm. If $n = m$ the sequences $P \circ W$ constitutes an exhaustive test suite ($X \circ Y =_{\text{def}} \{x \cdot y \mid x \in X, y \in Y\}$). These sequences can be joined via a reset-action to form a checking sequence. When $m > n$ it must be checked that the extra $m - n$ states has an equivalent state in the specification. Because the extra states could be attached anywhere to the minimal implementation machine, additional input sequences of exponential length are required to ensure that all extra states are visited by the transition cover, i.e., the sequences $P \circ Act_I^{m-n} \circ W$ constitute an exhaustive test suite (Act_I^{m-n} is the set of all input sequences of length less than or equal to $m - n$).

The total length of a checking sequence constructed using the W-method when $m = n$ is consequently at most $n^3 k$, where k is the number of input actions, and can be constructed in polynomial time. When $m > n$ the length grows to $n^2 m k^{m-n+1}$, and thus becomes exponential in the number of extra states [27].

Test generation tools for FSMs using state characterization techniques exist. An example is TAG (Test Automatic Generation) developed by Tan et al. at the University of Montreál [107]. The tool uses harmonized state identification sets instead of Chow's characterization set. This produces fewer test cases.

The methodology was developed for Mealy machines, but have in [108] been re-formulated for LTSs, and are thus applicable in our testing setup in the case of deterministic systems. Some work exists on generalizing the theory to non-deterministic systems [71, 108], but it does not appear as well developed as the deterministic theory.

The employed implementation relation is trace equivalence. Ours is based on Hennessy tests which, contrary to traces, also have the capability of detecting deadlocks. We believe that this is essential for testing concurrent and distributed systems. Checking experiments appear ideal for relatively small deterministic systems, but where full fault coverage is critical.

6.1.4 Domain Based Selection

Most introductory books on software engineering or testing, e.g., Pressman [85] and Beizer [12], describe a technique for functional black box testing involving partitioning of the input values into equivalence classes (called domains in [12]) in which the implementation is expected to behave similarly. It is usually recommended to test each equivalence class once in its interior and a number of times on its borders or extreme values.

The rationale for this approach can be explained by considering the faults that can occur. A *computation fault* is wrong processing of all inputs in a domain, thus potentially causing wrong outputs. Hence interior selection. A *domain fault* is an incorrectly implemented domain, and thus inputs are classified wrongly. This is expected to occur most frequently on the border or at extreme values of the domain. Hence extreme value selection.

There is no formal definition of what precisely constitutes an equivalence class or what "same behavior" is. When the source code is available, inputs that follow the same path can be considered equivalent [116, 31]. Another common approach is to collect all predicates occurring in the formal or informal specification describing the pre- and post-conditions of its operations, rewrite these to a disjunctive normal form in order to obtain nice domains, and find their dependencies. Each disjunct is then treated as an sub-domain which is tested separately [12].

These ideas have been applied to formal specifications with the aim of ensuring coverage and automatizing testing, especially test selection and test outcome evaluation. Hierons [52], and Hörcher and Peleska [55] aim at automatizing testing against Z specifications.

Raymond et al. propose in [87] a technique for testing deterministic discrete time reactive systems against specifications given in the synchronous data flow language Lustre. Their work does not deal with testing of real-time constraints, but whether the implementation computes permissible output *values*, given an history of input *values*. Their work shares with ours the use of efficient data structures commonly used in model checkers for computing relevant inputs. They use binary decision diagrams to solve boolean equations, and use convex polyhedra to represent solutions to numerical constraints. Their test inputs are then randomly chosen from these solution sets. They neither propose an explicit notion of coverage nor measure the resulting coverage.

In our approach we apply these selection principles to clock valuations, thus regarding clocks as parameters, although oddly behaving ones. We use the actions possible in a partition and its deadlocks properties as "outputs" to verify that the implementation responds correctly. We also propose extreme value selection to check that the time constraints are implemented correctly. Guards could be implemented erroneously by initializing timers with wrong values, or timers could be reused unsafely. A premature timeout could be caught by an "upper" or late extreme value because an action that should have been enabled (resp. disabled) have been disabled (resp. enabled). A missed deadline could be detected by a "lower" or early extreme because an action that should have been enabled (resp. disabled) is still disabled (resp. enabled). The notions of computation faults and domain faults therefore also make perfect sense in the time domain.

6.2 Timed Testing

The introduction of real-time influences all aspects of automated testing. We first discuss potential revisions of the theoretical foundation and the implementation relation in Section 6.2.1, and thereafter turn to, in our view, mostly theoretical testing methods based on checking sequences. Obtaining a manageable set of tests is a key issue in real-time testing. Some promising approaches are discussed in Section 6.2.3. We make some remarks in Section 6.2.4 about other potentially interesting algorithms for the analysis of real-time systems.

6.2.1 Observations and Timed Preorders

Our timed testing preorder was derived by including time in the traces of the untimed may and must properties. It is important to note that the satisfaction of the resulting implementation relation $\sqsubseteq_{\mathbf{tte}}$ does not imply that no arbitrary test automaton can distinguish the implementation from the specification. An ideal preorder would satisfy the relations stated in Definition 6.1.

Definition 6.1 *Test Preorder*:
Let $\mathcal{L}_{tta}$ be the class of test automata, i.e., timed automata whose locations has been labeled with verdicts **pass** or **fail**. Let $\mathcal{S}, \mathcal{I}$ be timed automata.

1. $\mathcal{S} \sqsubseteq_{\mathbf{must}} \mathcal{I}$ *iff* $\forall \mathcal{T} \in \mathcal{L}_{tta}.\ \mathcal{S}$ **must** $\mathcal{T}$ *implies* $\mathcal{I}$ **must** $\mathcal{T}$
2. $\mathcal{S} \sqsubseteq_{\mathbf{may}} \mathcal{I}$ *iff* $\forall \mathcal{T} \in \mathcal{L}_{tta}.\ \mathcal{S}$ **may** $\mathcal{T}$ *implies* $\mathcal{I}$ **may** $\mathcal{T}$
3. $\mathcal{S} \sqsubseteq_{\mathbf{te}} \mathcal{I}$ *iff* $\mathcal{S} \sqsubseteq_{\mathbf{must}} \mathcal{I} \land \mathcal{S} \sqsubseteq_{\mathbf{may}} \mathcal{I}$

□

That is, $\mathcal{L}_{\mathbf{tmust}}$ does not fully characterize such a preorder. One reason is that the observations one would naturally make in a timed model change because the progression of time can be used to observe refusal of actions.

The notion of *refusal testing* in the untimed setting was first explored by Phillips in [84]. Contrary to our Hennessy based tests which deadlock when the implementation refuses to engage in one of the offered actions, refusal testing assumes that the tester can observe such refusals and continue with an alternative set of actions. In [84] this is explained as a button pressing experiment where the implementation is equiped with a button for each action as well as a green light. In a basic experiment a set of actions are continuously pressed until one is accepted, or until the green light goes off. The green light is constructed to be on while the implementation has internal processing to do, and to be off when the implementation has reached a stable state without being able to synchronize with the offered actions.

In a timed setting this observation of refusals becomes more natural since one can offer a set of actions and wait some amount of time. If the timer goes off a time bounded refusal has been observed.

This idea of refusals seems to underly the testing theory developed by Hennessy and Regan in [50] for the discrete Timed Process Language (TPL). They define the notions of may and must testing, and give an alternative characterizations based on barbs, and based on passing tests in an associated test language (F-tests). Further, they also give a proof system for TPL. This work is thus a timed dual to the classical untimed work of De Nicola and Hennessy [75].

To apply the notion of refusals to dense timed automata we conjecture the need for test automata structured like the one illustrated in Figure 6.2. The automaton continuously offers a set of actions for c time units, and uses an internal action to time out.

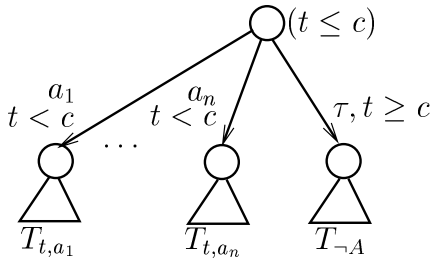


Figure 6.2. *Test automaton for densely timed automata? t is a clock used by the test automata, c is a real-valued constant, T_{t,a_i} is the sub test after executing a_i at time t, and $T_{\neg A}$ is the sub test after refusing $A = \{a_1 \dots a_n\}$ for c time units.*

The *timed failures* model of timed CSP [102] could also serve as basis for a continuous time testing theory. A timed failure is a timed trace and a set of refusal tokens describing which actions are continuously refused in which time intervals along that trace. The semantics of timed CSP is defined using timed failures.

An alternative testing theory is developed by Cleaveland and Zwarico in [33]. In this theory an internal computation step (τ action) is defined to take one time unit. Systems are related by a *faster-than* relation. This work thus suggests a link between real-time conformance testing and performance testing.

We conclude that the theoretical ground for real-time testing is not completely covered. We are looking forward to a well-developed and practical theory for timed automata.

We finally remark that Mok's Real-Time Logic (RTL) [56] like ERA expresses time constraints on the occurrences of events. Central to RTL is the occurrence relation $R(e, i, t)$, which states that the ith occurrence of event e happens at time t. Time constraints are expressed by a first order predicate logic on time- and occurrence variables. The event recording automata model is similar in the sense of stating timing constraints on event occurrences. However, in the basic definition, event recording automata only permit reference to the last occurrence of an event, and permits a limited set of guards only.

6.2.2 Checking Experiments for Real-time Systems

It would be natural to assume that exhaustive testing of densely timed systems would be impossible because of the infinite state spaces. However, it was shown by Springintveld et al. in [105] that a *finite* set of finite length tests suffices. Like the untimed case, exhaustiveness is only ensured under a set of assumptions about the implementation. Before stating the exact result and its assumptions, some definitions are necessary.

Recall regions from Definition 3.14. Define the *grid automata* $\mathcal{G}(\mathcal{A}, \delta)$ as the sub automata of timed automaton $\mathcal{A}$ that only contains clock valuations that are multiples of δ, where $0 < \delta < 1$. Let S be the states in $\mathcal{A}$, and X the set of clocks. Thus, a state $\langle \bar{l}, \bar{u} \rangle \in S$ is also a state in the grid automaton iff $\forall x \in X.\ \exists k \in \mathbb{N}.\ \bar{u}(x) = k\delta$. $\mathcal{G}(\mathcal{A}, \delta)$ represents a discrete version of $\mathcal{A}$ with discretization step δ.

The algorithm developed in [105] generates test cases for a flavor of timed automata called Bounded Time Domain Input/Output Automata (TIOA). The TIOA model distinguish between input and output actions; inputs are controlled by the environment and outputs by the automaton itself. A TIOA is input enabled which means that it is able to receive inputs at every time instant. The time domain of a clock is a bounded interval of real numbers united with the infinite element ∞. Intuitively, the value of a clock is defined to be ∞ when it exceeds the upper bound of the interval. Beyond this bound the exact value of the clock is irrelevant—it suffices to know that it is large.

[105] proves that bisimilarity of two TIOA can be decided by checking bisimilarity of their (finite state) grid automata, provided that the step size δ is chosen sufficiently small, i.e., $\mathcal{A}_1 \simeq \mathcal{A}_2 \quad \textit{iff} \quad \mathcal{G}(\mathcal{A}_1, \delta) \simeq \mathcal{G}(\mathcal{A}_2, \delta)$.

The basic idea is now to use Chow's algorithm to derive a checking sequence from such a sufficiently fine grained specification grid automata. Note that in the deterministic case trace equivalence coincides with bisimilarity. A sufficiently small step size is 2^{-n} where n is chosen to be greater than or equal to the number of *regions* in the product automata of the specification and implementation TIOA.

Because the number of regions in realistic specifications is very large, it should be clear that the step size becomes infinitesimal, and consequently that the algorithm, while theoretically exhaustive, is highly impractical.

The assumptions of the algorithm can now be stated as:

1. The specification is a known controllable deterministic TIOA. The implementation can be modeled by some controllable deterministic TIOA.
2. The number of regions in the implementation does not exceed n', and the step size is chosen sufficiently small as defined above.
3. The number of states in the grid automaton for the implementation does not exceed m.

A more recent result also using checking sequences of grid automata is presented by En-Nouaary et al. in [39]. The algorithm also uses a deterministic TIOA-model, but here the step size is chosen much larger than in [105]. It has been shown that when the step size is chosen to be $1/(|X|+2)$ [66], all reachable regions has a representative state in the grid automaton. The checking sequence derived from the grid automata can thus be viewed as a checking sequence for the region graph of the specification. A fault model based on wrongly implemented regions is also presented.

The resulting test suite is exhaustive wrt. trace equivalence if *uniformity* can be assumed about the implementation. Their uniformity assumption states that if the implementation behaves correctly on some points in a clock region, it also behaves correctly for the remaining points. Although not explicit from the paper, it also seems necessary to assume that the implementation uses the same number of clocks as the specification, and has a no more than m states in its grid automaton.

The authors of [39] give an example of an on-off switch specification. The timed automata has two locations, two edges, one clock, and uses a maximum clock constant of 1. For this example, their algorithm generates 30 test cases. Thus, although the step size is more reasonable than [105], we still believe that it will be too small for most practical applications.

A final effort using checking sequences is reported by Cardell-Oliver and Glover in [26]. Their specification language, termed timed transition systems, is different from timed automata in that no explicit clocks exist. Instead actions are guarded by an upper and lower bound. One of the enabled actions must be taken from a state before any of them disables. Their testing methodology assumes a discrete time interpretation of deterministic, finite state, deadlock and live lock free specifications and implementations. As usual an upper bound on the number of states in the implementation must be assumed. Their approach is implemented in a tool which is applied to a series of small cases. Their result indicates that the approach is feasible, at least for small systems, but problems arise if the implementation has more states than the specification.

Recently Cardell-Oliver [24, 25] has outlined how to generate tests in the form of timed traces from continuously timed automata. Testing is based on generating a checking sequence from a digitized approximation of the original automaton. However, it is unclear from the presentation what properties this approximation has. The step size is chosen much larger than that required to visit every region, and possibly only such that every edge can be visited. Further, it is unclear what kind of communication interface is assumed to exist between the tester and the implementation: She seem to assume that the tester can observe the values of the clocks and state variables in the implementation.

Our symbolic method maintains exact information of the state space of the specification, and only assumes communication with the implementation via synchronizing on actions.

6.2.3 Real-Time Testing

The application of black-box domain testing to real-time systems is also proposed by Clarke and Lee in [29, 28, 30]. Although their primary goal of using testing as a means of approximating verification to reduce the state explosion problem is different from ours, their generated tests could potentially be applied to physical systems as well. Their tests are not applied to a physical system, but to an Algebra of Communicating Shared Resources (ACSR) model thereof.

Time requirements are specified as directed acyclic graphs called *constraint graphs*. Nodes in a constraint graph correspond to actions, and edges express a time constraint between the source and target action. An edge is labeled with two pieces of information; an interval describing the permissible delays between the two actions, and a set of actions that may not occur during this interval. Tests can be automatically generated from such constraint graphs.

The authors define the *domain* of an action to be the permissible delays preceding the action. They further define different coverage criteria for these domains, such as observation of all actions, and/or observation of all extreme values in the domains. These criteria are then organized in a subsumption hierarchy.

Their domains are "nice" linear intervals that are directly available in the constraint graph. Also, since their constraint graphs must be acyclic this only permit specification of finite behaviors. Our specifications are given as event recording automata without these restrictions. Our stable edge set partitioning were obtained, not only by looking at single actions, but sets thereof, i.e., we do not assume independence of these. Moreover, since we operate with constraints over many clocks, our partitions are no longer just intervals, but of a dimension corresponding to the number of clocks. We further subdivided these into convex polyhedra, and applied symbolic reachability analysis to find the reachable parts thereof. Thus, we are faced with a more difficult analysis problem, and the constraint graph can to some extend be viewed as the outcome of this analysis.

Braberman et al. [20] describe an approach where a structured analysis/structured design real-time model is represented as a timed Petri net. Analysis methods for timed Petri nets based on constraint solving can be used to generate a symbolic *timed reachability tree* up to a predefined time bound. From this, specific timed test sequences can be chosen. This work shares with ours the generation of tests from a symbolic representation of the state space. The paper also proposes other selection criteria, mostly based on the type and order of the events in the trace. However, they seem to be concerned with generating traces only, and not on deadlock properties as we are. The paper describes no specific data structures or algorithms for constraint solving, and states no results regarding their efficiency. Their approach does not appear to be implemented.

The VVT-RT (now known as RT-Tester) tool developed in cooperation between Bremen University and Verified Systems GmbH. [81, 79, 78], briefly discussed in Section 6.1.2, also facilitate real-time testing. The specification language is untimed CSP extended with a set of special actions $\texttt{set}_i$ and $\texttt{elapse}_i$ for setting and waiting for timers provided by the runtime system. CSP specification processes can synchronize on these actions, and use them to signal error if an action is not received when required, or for delaying inputs, etc. There are two types of timers. Fixed timers time out after a specified amount of time. Random timers time out at a random instant in a specified time interval.

It should be noted that the specification accepted by RT-TESTER is not a model of the desired target system behavior, but rather is a *test specification* consisting of CSP expressions representing the *joined* behavior of a collection of use cases, each describing a communication scenario, that the test engineerer wish to have tested. Thus much of the burden of generating and selecting test cases lies with the test engineerer. In the reviewed literature there is no description of how a test specification can be derived systematically from a model of the environment and desired target system behavior.

The employed coverage criterion aims at executing every edge in the refusal graph of the test specification at least once, including timer events. There is no coverage criterion for the time domain except that all time outs will be covered. Their approach has detected implementation faults in industrial applications [80, 98, 23, 82], but whether a more systematic and detailed treatment of time could reveal further faults is an open issue.

The approaches reviewed so far are based on so called behavioral specification languages. Another school is logic specifications. Mandrioli et al. [72] proposes a technique for tool *assisted* generation of tests from TRIO discrete time temporal logic specifications. The user assists in selecting the tests to be generated by guiding the decomposition of the specification into subformulae. The tool then generates a history (execution trace) satisfying the chosen subformula. With appropriate input/output labeling this trace can be used as a test case. The authors propose to measure coverage in terms of the number of axioms and predicates that have been tested.

6.2.4 Algorithms

Yannakakis and Lee [117] describe an alternative algorithm for computing a minimized reachable symbolic transition system from a deterministic timed automaton. The symbolic states resulting from their algorithm are *stable* in the sense that all its members have the same symbolic a successor states for all actions a, including the immediate time successor action. Our symbolic states do not have this property which implies that we must strengthen the symbolic states through a back propagation step prior to trace generation. Their algorithm is of potential interest to us because avoiding back propagation will be a big advantage if our techniques are to be applied in an online testing approach where the test case is generated while being executed. It will enable us to systematically visit all equivalence states without use of back propagation. It is unclear whether we will benefit from the acclaimed efficiency of the algorithm because the initial partitioning that must be provided (although the same as ours where the same edges must be enabled) is required to be convex.

Clock Difference Diagrams [11] is a binary decision diagram inspired data structure that permits representation of non-convex unions of convex sets. This data structure will allow a much more compact representation of the passed list than presently done. It will possibly also enable reachability analysis to be made based on non-convex partitions, rather than on their present convex subsets.

6.3 Novelties of Our Approach

Our work focuses on fully automatic generation of tests for timed automata using an offline approach. Compared to the related work outlined in this chapter, our work distinguishes itself by treating time thoroughly and systematically, yet in a way we claim have practical relevance.

We have defined a partitioning of the timed automata specification which is much coarser than the previous approaches based on regions. It is our view that the region based techniques in most cases are too fine grained, and neither scale well, nor provide good guidance in the test selection process.

We have given algorithms that systematically explore the partition graph and cover this with tests. To our knowledge, the employed zone and DBM based algorithms and data structures for symbolic execution and reachability analysis of the specification have not previously been applied to testing.

A further novelty is that we permit both *non-deterministic* timed specifications and implementations. Most other related work on timed testing limits attention to only deterministic systems, whereas non-determinism is permitted in most untimed approaches. Our work thus levels out this discrepancy. To handle non-determinism, we made a slight generalization to Hennessy's testing theory and adopted a specification language, event recording automata, that enabled us to perform the necessary analysis. Interestingly, the ease of analyzing event recording automata does not seem to have been exploited elsewhere.

6.4 Summary

We have identified two main approaches to test generation. In the preorder based approach, an implementation relation defines the correcness of implementations. A tool interprets the specification with respect to this preorder and generates test cases. Checking sequence based test generation checks that the states of the specification are equivalent to those of the implementation, i.e., that the implementation has no output- or transfer-faults. Both techniques have also been applied in the timed setting.

Test case generators can be online or offline. Online generators execute test cases as they are being generated. Offline generators output completed test suites before test execution. Further, test selection can be manual or fully automatic.

Our approach is preorder based, generates tests offline, and selects tests fully automatically. Our work focuses on testing real-time constraints. The novelties include automatic test selection from a coarse grained state partitioning, handling non-deterministic timed specifications, and the application of symbolic verification techniques to test generation.

Chapter 7

Conclusions and Future Work

This theses is concerned with the development of correct distributed real-time systems, and has made two main contributions to this field.

Our first main contribution is in the area of testing where we have developed a new technique for automatic generation of real-time conformance tests against formal specifications. We implemented our techniques in the RTCAT tool, and performed an evaluation thereof. The second main contribution is a formally defined specification and programming language that facilitate reuse of real-time software components. The conclusions of this line of research is contained in Section A.7. The remainder of this chapter presents conclusions concerning our work on testing, the lessons learned, and its potential implications.

It has been our goal to develop an automatic testing technique for densely timed systems specified using timed automata. Further, tests should be generated from a sound theoretical basis with a well defined implementation relation. It has also been our goal to select tests systematically such that the generated test suite would give a well defined coverage. To approach these goals we employed three existing techniques:

Hennessy's Testing Theory: This theory is a widely accepted testing theory for concurrent systems. In addition to checking the traces of the implementation, it also checks that the implementation has no unspecified deadlocks.

Classical Black-box Test Selection: A common approach to selecting tests for black box sequential procedures is to use partition or domain testing, where inputs are partitioned into sub domains in which the procedure, according to the specification, is expected treat identically. We regard the clocks of a timed specification as (oddly behaving) input parameters.

Symbolic Reachability Analysis: In the recent years efficient constraint solving techniques for model checking of timed automata have been developed. These techniques provide the necessary means for computing the reachable partitions, for covering the specification systematically, and for computing the timed test sequences.

In Chapter 2 we present Hennessy's testing theory and associated test language formally. We define the class of relevant Hennessy tests, and we show how to generate such tests from specifications given as communicating extended finite state machines provided with a labeled transition system semantics. To further study the properties of the involved algorithms, we implement an untimed test generation tool, TestGen, that constructs the success graph data structure containing the information necessary for test generation. Once constructed, the success graph eases systematic generation of Hennessy tests. We also give some examples of specifications and the resulting success graphs.

In Chapter 3 we propose to use Hennessy's tests lifted to include timed traces as test language, knowing that this does not fully constitute an alternative characterization of a real-time testing preorder. We formally define a quite unrestricted model of timed automata, and outline an algorithm based on symbolic execution of timed automata for generating timed Hennessy tests. However, this algorithm does not systematically generate tests suites from a well defined coverage criterion.

There are numerous ways of defining input partitions, and the precise definition affects the nature and the number of tests that will be generated. We therefore organize these in a common framework, and define a number of possible instances thereof. One of these, stable edge set partitioning, is chosen for further investigation in a prototype implementation. Stable edge set partitioning is chosen partly on intuitive grounds, and partly because the members of the partitions are equivalent with respect to certain deadlock and action properties (Hennessy tests without a preceding trace).

Construction of a timed version of the success graph by applying our partitioning and symbolic reachability ideas to the unrestricted model of timed automata turned out to be very challenging, for both technical and principal reasons. The principle problems are caused by the indeterminizability of timed automata. Obtaining a finite deterministic success graph, as is desired for generating timed Hennessy tests, may therefore be impossible. The technical problems are related to computing the desired partitions and their reachable parts. One problem is that the adopted symbolic techniques cannot represent concave unions of sets of states; another is maintaining a common "time base" when different clocks were reset along non-deterministic choices.

For these reasons we decide to develop our techniques for Alur's restricted but determinizable timed automata model, called event recording automata, which restricts the resets of clocks. Using this model, most technical problems vanish, and it therefore turns out to be an excellent vehicle for developing our techniques.

Chapter 4 contains the main results of our work on real-time testing. We show how to generate tests for event recording automata in an automatic and systematic way. Our technique involves three main steps. The first step constructs a partition graph based on the stable edge set partitioning. The second step is symbolic reachability analysis of that graph. The final step is the generation of the tests themselves. For this we give two algorithms; one basic algorithm that also supports extreme value selection, and one that aims at reducing the size of the test suite by composing test cases. Our techniques are implemented in the RTCAT tool.

In Chapter 5 we evaluate our techniques with respect to usability and expressiveness of the event recording automata model, the number of generated tests, and implementability.

Through a series of examples we demonstrate how event recording automata can specify untrivial and practically relevant timing behavior. While theoretically less expressive than timed automata, it has proven sufficiently expressive for our examples, but sometimes causing minor inconveniences. More importantly, however, our present interpretation lacks means of expressing timing uncertainty and modeling of environment assumptions.

Generating tests from these specifications using RTCAT results in fairly small covering test suites. The smallest covering test suite is obtained by configuring RTCAT to compose tests, and to perform breadth first test generation. Our preliminary experience with respect to the number of tests that will be generated is therefore very encouraging, and we believe that our technique is feasible for even larger specifications. However, guaranteeing coverage of large specifications could become problematic because both test execution and reachability analysis may become bottlenecks. We have yet to determine how large realistic and practical specifications our techniques can handle. Alternatively, it is sometimes possible to decompose the specification into more manageable properties that can be tested separately.

We also conclude that our Hennessy based test language should be extended with continuous enabling and refusal of actions. It should be further examined whether Hennessy's recent testing theory [50] for real-time systems could serve as practical basis for this extension, see Chapter 6.

Our implementation effort shows that our techniques are indeed implementable and operational, but it also indicates that neither partitioning, nor the reachability techniques are straightforward to apply efficiently. We find that future implementations could be improved by employing a lazy evaluation architecture, and perform the reachability analysis using coarser symbolic states.

In our review of related work, we found only a moderate amount dealing explicitly and systematically with generation of real time tests. Compared to these our approach is novel. Specifically, compared to present attempts at generating tests for timed automata based on a representation using regions or digitized clocks, our approach seem advantageous. We believe that our techniques deserve further attention, and that future work can improve on its applicability and efficiency.

Future Work

There are many aspects of our work that require further work. Listed below, in what we believe should be an approximate order of significance, is a number of topics for future work.

Applications: So far we have only generated tests from a few more or less realistic hypothetical specifications. An important next step is to apply our techniques to test of real-life systems. This entails both specification, test generation, and test execution.

Such case studies will contribute with essential information about the applicability of our technique, including the outstanding question of the error detection abilities of our selection scheme, and for which kinds of applications this approach is useful.

Environment Assumptions: A facility should be added to our techniques to permit modeling of the environment of the specification, and to take this into account in the test generation process. We expect that it will be unproblematic to develop an algorithm that factors in the environment behavior in the reachability analysis, although some amount of implementation work is required.

Timing Uncertainty: The perhaps most inhibiting aspect of our interpretation of the ERA specification language is its inability to specify uncertain timing behavior. We see no evidence that this should be a profound limitation of ERA. It was rather introduced by our application of timed Hennessy tests, and by our desire to keep matters reasonably simple. We envision two types of actions, urgent and non-urgent actions, as included in our general timed automata model.

Because the need for timing uncertainty most frequently arises in conjunction with actions that one would logically regard as being outputs, a distinction between input (controlled by the environment) and output actions (controlled by the system) should be considered.

Fortunately, most of our techniques appear to be applicable also in this setting. The partitioning and reachability analysis techniques are largely unaffected. Further, it will be easy to define a timed version of the input/output conformance relations reported in Chapter 6 in the same way we did with Hennessy's untimed tests.

However, it will be more difficult to change the test notation language, and the algorithms producing the tests: To be effective, it should allow actions to be continuously enabled and refused over a time interval. Also, since the time instant where a synchronization took place is only available during test execution, we think that better or stronger tests could be generated using online testing. This does not rule out the use of our symbolic techniques per se, as the basic operations are sufficiently efficient to be executed in real-time (obviously depending on the time scale of the system being tested). Alternatively, the generated test cases should be stored in symbolic form and be instantiated at execution time. Significant implementation effort will be needed to change test language, and rewrite the implementation for online testing.

Implementation Improvements: Two important improvements of the implementation should be made. Reachability analysis should be applied differently to avoid chopping up the state space into many small symbolic states. The memory usage should be reduced by only storing the parts needed for generation of the current test case. Finally, with a lazy evaluation tool architecture, it would be easier to accommodate large specifications. Implementing these improvements requires a large redesign and implementation effort.

Testing Reusable Specifications: We have proposed a model where a reusable specification consists of separately specified untimed objects and time constraints. One approach to derive tests from such specifications, is to compose the components and their constraints, and to analyze their combined behavior using similar techniques as we have proposed for timed automata. Thus, the separation itself seems to cause only minor challenges. It is unclear whether, or to what extend, tests can be generated from the separate specifications, thus reusing parts of the test generation effort.

However, the particular Real-Time Synchronizers and Actor languages adds two difficult challenges. First, actors communicate asynchronously via messages. This requires a distinction between inputs and outputs, and observing that messages may arrive in an order different from their transmission order. Both aspects affect testing theory and algorithms since these are not directly applicable to asynchronous systems. The second challenge is to partition these specifications and to analyze them symbolically. The symbolic methods we propose for timed automata does not appear to be directly applicable to Real-Time Synchronizers, although similar techniques may be feasible.

Discrete Variables and Value Passing: Currently, value passing is not supported in our specification languages. Discrete variables are handled only as an extension of the control location. A logical next step would be to extend our partitioning to also incorporate discrete variables and parameters; after all, our approach to real-time test selection was inspired by the way discrete parameters are traditionally handled in sequential testing techniques. We expect that these can be incorporated into an automata based specification language, and that symbolic constraint solving techniques can be applied. However, the specific algorithms remain to be worked out.

Hybrid Systems: Hybrid systems evolve by discrete transitions and continuous behavior described by differential equations. It would be interesting to find approximations to their continuous behavior that can be analyzed and tested using similar techniques to the ones developed here for real-time systems. Again, results developed for model checking may be a source of inspiration. Alternatively, it could be investigated how simulation techniques can be incorporated to deal with differential equations that cannot be solved feasibly by exact means.

Probabilistic and Performance Testing: The problem of testing real-time conformity interfaces to a related problem, that of performance testing. In performance testing absolute correct timing is not required, but only requirements like average response time, or whether a certain fraction of responses is earlier than a given limit, should be checked. The outcome of executing such performance tests could be evaluated against probabilistic timed automata specifications.

Although it may be possible to test hybrid and probabilistic systems using similar ideas to the ones presented here, this will require a completely new research effort.

The goal of the work presented here was to show how real-time test cases could be derived systematically with a guaranteed coverage using the symbolic analysis techniques developed for model checking. Given our positive results in this respect, we plan in the near future to further apply our techniques and to examine how to deal more effectively with a number of the identified practical issues. Specifically, we intend to add environment modeling, timing uncertainty, and distinguish between inputs and outputs. We believe that our basic techniques are still applicable in this setup although some re-implementation effort is needed towards online testing.

Appendix A

Towards Reusable Real-Time Objects

Brian Nielsen

Aalborg University
Dept. of Computer Science
Fredrik Bajersvej 7E
DK-9220 Aalborg, Denmark

Email: bnielsen@cs.auc.dk

Gul Agha

University of Illinois at Urbana-Champaign
Dept. of Computer Science
1304 W. Springfield Av.
Urbana, Illinois 61801, U.S.A

Email: agha@cs.uiuc.edu

Abstract‡

Large and complex real-time systems can benefit significantly from a component-based development approach where new systems are constructed by composing reusable, documented and previously tested concurrent objects. However, reusing objects which execute under real-time constraints is problematic because application specific time and synchronization constraints are often embedded in the internals of these objects. The tight coupling of functionality and real-time constraints makes objects interdependent, and as a result difficult to reuse in another system.

We propose a model which facilitates separate and modular specification of real-time constraints, and show how separation of real-time constraints and functional behavior is possible. We present

‡This chapter is previously published in [77].

our ideas using the *Actor model* to represent untimed objects, and the *Real-time Synchronizers* language to express real-time and synchronization constraints. We discuss specific mechanisms by which *Real-time Synchronizers* can govern the interaction and execution of untimed objects.

We treat our model formally, and succinctly define what effect real-time constraints have on a set of concurrent objects. We briefly discuss how a middleware scheduling and event-dispatching service can use the synchronizers to execute the system.

A.1 Motivation

Real-time systems remain among the most challenging systems to build, and often projects are late and products faulty. Developers are faced with ever more stringent requirements for building larger, more complex systems at a faster pace and without proportional resources. However, because current tools and techniques to deal with complexity do not scale linearly with size of programs, development problems worsen. We believe that real-time systems can benefit significantly from a development approach based on components where new systems are constructed by composing reusable, documented and previously tested components. Unfortunately, current software development methods and tools do not properly support such construction.

Because real-time systems are safety critical and often unattended, they must operate under strict end-to-end time constraints and be dependable. Dependability requirements entail both correctness and tolerance to faults. Real-time systems can be loosely defined as systems where timely response is equally important as correct response. Real-time systems typically monitor and regulate physical equipment. Some well-known examples include: manufacturing plant automatization, where the production steps must be supervised and coordinated; chemical processes which are automatically monitored and regulated through sensors and actuators; safety systems aboard trains and cars; financial applications where stock rates must be guaranteed up-to-date and where transactions must be completed within specific time bounds.

Historically, real-time systems were built using low level programming languages and executed on dedicated hardware and specialized operating systems: efficiency, high resource utilization, and integration with hardware were the primary concerns, software modularity and reuse were only secondary. In the light of more stringent development requirements, we believe the emphasis should now be on building modular and reusable components, which can be used in many applications.

Middleware services (i.e. general purpose services located between platforms and applications [16]) can then be used to help integrate the components.

However, reusing real-time components is often problematic because application specific time and synchronization constraints are embedded in the internals of these components. This kind of tight coupling makes components interdependent, and consequently unlikely to be reusable in other systems. Properly supported component-based software development will allow components to be developed individually and later be composed with other individually developed or existing components, making it possible to reuse components in different applications. Thus component-based software has emerged as an active area of research. Our work makes a contribution to this area.

A.2 Separation and Reuse

In what follows we use collections of concurrent objects to represent components in a distributed real-time system. Typically these objects model real-world entities or act as proxies for them. The objects execute concurrently and communicate by exchanging messages containing computation results or information about their local states. Objects may be larger entities than data structures such as lists or trees, they need not be heavyweight processes.

Designing reusable objects is difficult and requires skilled engineers. Building reusable concurrent real-time objects is even more difficult, and necessitates particular restrictions:

1. Objects should not schedule themselves by setting their priorities or by specifying deadlines and delays on method invocations, e.g., use expressions such as `object.method(args) deadline 10`, or contain any other type of scheduling related information.

2. Objects should not manipulate timers for programming delays or timeouts. Timer manipulation includes requesting, cancelling, and handling timer signals.

3. Objects should not have hardwired synchronization constraints. In a concurrent system, certain restrictions on order of events must be enfored in order to ensure safety and liveness. This concerns both the order of invocations of a single object, and the interaction between invocations on a set of objects.

Priorities, real-time constraints, timer values, and synchronization constraints are all properties that are likely to differ between applications, and therefore objects that embed such behavior cannot be readily reused. In addition most of these properties are *global* properties, not properties belonging internally to a single object. For example, a priority level only makes sense when compared to the priorities of other objects. Similarly, an object is usually part of a sequence of objects chained by method calls which together must obey an end-to-end deadline. A deadline on a method invocation only represents a single object's time budget along the call chain. New applications using the object will usually have different end-to-end deadlines and different call sequences. Therefore the objects would have to be modified, and consequently re-tested, to accommodate a new time budget.

Parameterizing objects with timing and scheduling information would solve these problems only to a very limited extent. This is partly because it is difficult to know which attributes should be parameterized, and partly because concurrency constraints among objects are difficult to capture through parameters. We argue that it is better to handle the composition by a *composition software agent*, and use design methods and programming languages/environments that explicitly provide notations and abstractions for this decoupling.

Another source of reuse is the constraints themselves. We expect that many instantances of the same constraints will recur in different applications. It would therefore be advantageous to reuse them. However, a more important reason for reuse is that real-time and synchronization constraints can be extremely tricky to specify correctly. Constraints that work as desired should be reused rather than be replaced by new similar ones. An effective and modular language should enable the programmer to factor out common constraint instances as *constraint patterns* and support their composition.

We propose a model in which both real-time and synchronization constraints can be specified in an integrated manner, enabling a fairly general set of constraints to be specified. For example, a time constraint could specify that a controller object must receive sensor data from a sensor object every 20 milliseconds. A synchronization constraint temporarily disables some actions until others have taken place, for example, to prevent a producer from inserting in a full buffer. We refer to combined real-time and synchronization constraints as interaction constraints. Both types constrain dynamic interactions *among* objects.

Our interaction constraints are conceptually installed "above" ordinary objects, and they actively enforce the developers' constraints, see Figure A.1. The enforcing agent is the scheduler (or schedulers) which bases its decisions on the supplied constraints.

Interaction constraints are expressed in terms of enabling conditions on communication events occurring on the interface of objects. These events consti-

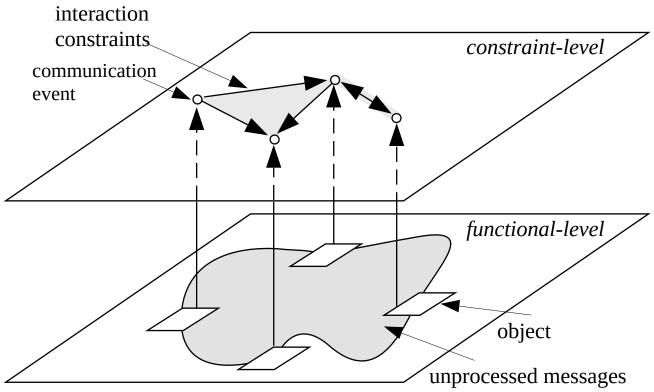


Figure A.1. *Separation of constraints and objects.*

tute the observable behavior of a system. What goes on inside an object is encapsulated, and cannot be constrained. Specifically, a collection of *synchronizer entities* constrain by delaying or accelerating message invocations. Each synchronizer implements a constraint pattern. We use the object-oriented *Actor* model to describe objects.

Section A.3 introduces and exemplifies our model. Since we are interested in providing a clean and sound model, it is accompanied by a description of its semantics. Our goal is to succinctly define constraints and their effects on the objects they constrain. Section A.4 provides the formal definitions. Finally, in Section A.5 we discuss implementation.

A.3 Specification of Interaction Constraints

We use the object-oriented Actor model [1, 2, 5] to describe distributed computing entities (hardware or software). An actor encapsulates a state, provides a set of public methods, and potentially invokes public methods in other objects by means of message passing. Unlike many object-oriented languages, message passing is *non-blocking* and *buffered.* This means that when an actor sends a message, it continues its computation without waiting for, or getting a reply from, the receiver. Further, messages sent but not yet processed by the receiver are conceptually buffered in a mailbox at the receiver. The receiver receives and processes messages one at a time. In addition, actors execute *concurrently* with other actors. An actor system is illustrated in Figure A.2.

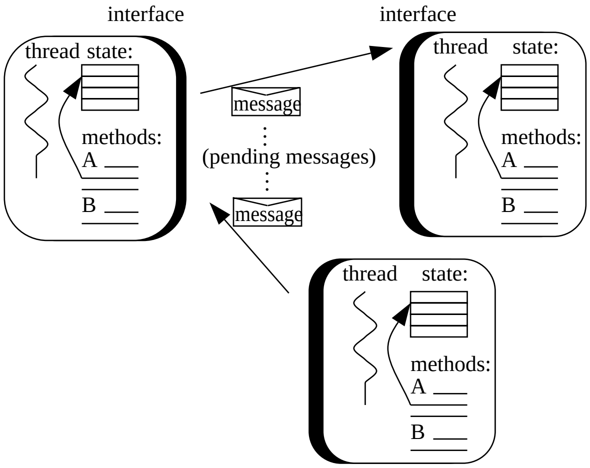


Figure A.2. *Illustration of an actor system.*

```
actor pressureSensor ( ) {
   real value;
   method read(actorRef customer) {
      send customer.reading(value);
   }
}
actor steamValve ( ) { . . . } //  unspecified
actor controller (actorRef sensor,valve) {
   method loop( ) {
      send self.loop( );
      send sensor.read(self);
   }
   method reading(real pressure) {
      newValvePos=computeValvePos(pressure);
      send valve.move(newValvePos);
   }
}
```

Figure A.3. *Steam boiler.*

Each actor is identified by a unique name, called its *mail address*. A mail address can be bound to state variables of type *actor reference*. To send a message an actor executes the **send** $a.m(pv)$ primitive, where a is an actor reference variable containing the mail address of the target actor (possibly the actor itself), m is the method to be invoked, and pv is the value(s) passed. It is possible to communicate mail-addresses through messages thus allowing dynamic configuration of the communication topology.

The example actor program in Figure A.3 describes part of a simple boiler control system consisting of a pressure sensor, a controller, and a valve actuator. These entities are modelled as actors. The goal is to maintain a pre-specified pressure level in the boiler. The controller is the heart of the system. It repeatedly executes a method which sends a request to the pressure sensor for the current boiler pressure. The iteration is implemented by having the controller send itself a loop message which causes requests to be sent to the pressure sensor. The parameter of this message specifies which actor the result must be sent to, in this case the controller itself. Upon request, the pressure sensor sends a reply containing its current pressure reading back to the initiator of the request (i.e., the controller). Based on that value, the controller computes an updated steam valve position, and sends a message to invoke the move method on the valve.

The RT-Synchronizers^{-} language that we define in this paper to express constraints is purposely distilled: it does not include syntactic sugar for convenient description of common constraint patterns. This allows us to focus on the central ideas, and makes it easier to define a complete semantics. A *synchronizer* is an object that enforces user specified constraints on messages sent by actors. Such constraints express real-time or ordering constraints on pairs of message invocations. The messages of interest are captured by means of *patterns* that represent predicates over message contents and synchronizer state. The structure of an RT-Synchronizers^{-} declaration is given in Figure A.4. It consists of 4 parts: A set of instantiation parameters, declarations of local variables, a set of constraints, and a set of triggers.

A constraint has one of the following forms:

$p_1 \Rightarrow p_2 \prec y$**:** Here p_1 and p_2 are message patterns and y is a variable or constant with positive real value. Let $a_1(cv_1)$ and $a_2(cv_2)$ be message invocations matching p_1 and p_2 respectively. This constraint then states that after an $a_1(cv_1)$ has occurred, an $a_2(cv_2)$ must follow before y time units elapse. We say that event $a_1(cv_1)$ *fires* the constraint, and causes a *demand* for $a_2(cv_2)$.

$p_1 \Rightarrow p_2 \succ y$**:** After $a_1(cv_1)$ occurs, at least y time units must pass before $a_2(cv_2)$ is permitted.

synchronizer $(a_1, \ldots, a_n)\{$
$StateDeclaration$

$p_{11} \Rightarrow p_{21} \sim y_1$
$\vdots$
$p_{1n} \Rightarrow p_{2n} \sim y_n$

$p_1 : \overline{x} := \overline{exp}$
$\vdots$
$p_k : \overline{x} := \overline{exp}$
$\}$

Figure A.4. *Structure of RT-Synchronizers*$^-$. $\sim \in \{\succ, \prec\}$.

In both cases there are no constraints on $a_2(cv_2)$ until after $a_1(cv_1)$ fires. A pattern has the form $x_1(x_2)$ **when** b, where b is a boolean predicate (guard) over the message parameter x_2 and the state of the synchronizer. x_1 is a state variable containing an actor address. Intuitively, a message satisfies a pattern if it is targeted at x_1 and the boolean predicate evaluates to true. If a message satisfies a pattern, the invocation is affected by a constraint which must then be satisfied before the invocation can take place. When a constraint forbids the invocation of a message, it is buffered until a later time when the constraint enables it. A disabled message may become enabled when a delay has expired, or when the synchronizer changes state through a trigger operation.

A trigger command specifies how the synchronizer's state variables change when a message invocation satisfies a given pattern. Specifically, assignment of the trigger $p : \overline{x} := \overline{exp}$ is executed when a message satisfying p is invoked. Synchronizers can thus adapt to the system's current state.

To promote modularity of interaction constraints, the constraints can be specified as a *collection* of synchronizer objects executing concurrently.

A.3.1 Example 1: Steam Boiler Constraints

The synchronizer in Figure A.5 describes the real-time constraints for the simple boiler control system from Figure A.3. The controller should read the pressure periodically (every 20 time units plus or minus some tolerance). The controller must receive sensor data from the pressure sensor within 10 time units measured from the start of the period, and it must update the steam valve position no later than 5 time units after receiving sensor data. A message sequence chart illustrating the communication among the boiler objects and the associated timing constraints is shown in Figure A.6.

This example shows how real-time constraints can be expressed and imposed separately from the functionality. It also shows how periodic constraints can be expressed by combining deadlines and delays. To make the language easier to use, common constraint patterns such as those enforcing periodicity can be specified as macros.

```
actor pressureSensor ( ) { . . . };
actor steamValve ( ) { . . . };
actor controller (actorRef sensor,valve) { . . . };

synchronizer boilerConstraints (actorRef: controller,valve) {
   // periodic loop:
   controller.loop ⇒ controller.loop ≺ 20+ϵ
   controller.loop ⇒ controller.loop ≻ 20-ϵ
   //deadline on reading:
   controller.loop ⇒ controller.reading ≺ 10
   //deadline on move:
   controller.reading ⇒ valve.move ≺ 5
}
```

Figure A.5. *Steam boiler constraints.*

A.3.2 Example 2: New Boiler

In a new boiler application, the pressure sensor must be polled approximately every 100 time units for pressure readings, and the pressure valve must be moved accordingly no later than 20 time units after the appropriate reading. However, in situations where the pressure in the boiler is high, the system must operate with a higher frequency. The pressure sensor must then be polled every 50 time units. Two threshold values, `NormToHighThr` and `HighToNormThr`, define which pressure values cause mode change.

The functional part is *reused* from Example 1, i.e., the actors and their respective behaviors are unmodified, but they are now composed by the "newBoiler-Constraints" synchronizer given in Figure A.7. The synchronizer maintains a

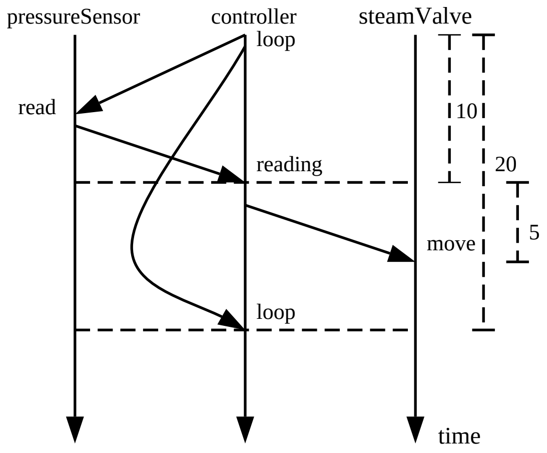


Figure A.6. *Message sequence chart with annotated timing constraints for the steam boiler example.*

```
synchronizer newBoilerConstraints (actorRef: sensor, controller, valve){
   enum Mode {Normal,High} mode = Normal;
   //normal pressure periodic:
   sensor.read when mode==Normal ⇒ sensor.read ≺ 100+ϵ
   sensor.read when mode==Normal ⇒ sensor.read ≻ 100-ϵ
   //high pressure periodic:
   sensor.read when mode==High ⇒ sensor.read ≺ 50+ϵ
   sensor.read when mode==High ⇒ sensor.read ≻ 50-ϵ
   //deadline on move:
   sensor.read ⇒ valve.move ≺ 20
   //Trigger mode change
   controller.reading(pressure) when pressure ≥ NormToHighThr: mode=High;
   controller.reading(pressure) when pressure ≤ HighToNormThr: mode=Normal;
}
```

Figure A.7. *New steam boiler.*

mode state variable which tracks whether the system operates in high or normal pressure mode. The example also illustrates RT-Synchronizers^{-}'s ability to capture the dynamic changes that are common to many real-time systems through the use of a state variable, and to change time constraints accordingly.

A.3.3 Example 3: Time Bounded Buffer

This and the next example show two common real-time constraint patterns: a real-time producer-consumer relation, and rate control. These examples also show how RT-Synchronizers^{-} can express synchronization (event ordering) constraints.

```
actor q {
  method put(Item item) {// store item };
  method get(actorRef customer) { send customer.processItem(item);}
}
actor consumer( ) {                              actor producer( ) {
  method consume( ) {                              method produce ( ) {
    send q.get(self);                                send q.put(item);
    send self.consume( );                            send self.produce( );
  }                                                }
  method processItem(Item item) { ... }          }
}
synchronizer bbConstraints (actorRef: producer, consumer, q) {
  int n=0; // no of elements in queue
  q.put ⇒ q.get ≺ 20; // time bound on get
  disable consumer.consume when n ≤ 0; // buf empty?
  disable producer.produce when n ≥ maxBufSz; // buf full?
  producer.produce: n++;
  consumer.consume: n−−;
}
```

Figure A.8. *Bounded buffer with time constraints.*

Figure A.8 shows a time bounded buffer where each element must be removed 20 time units after it has been inserted. In addition, the usual restrictions of not putting on a full buffer and not getting from an empty buffer are enforced. Note that the code uses a shorthand, **disable** p, to temporarily prevent messages matching the pattern p from being invoked. **disable** p can be written as

$e_0 \Rightarrow p \succ \infty$, where e_0 is a pattern assumed to be fired at system startup time. This synchronizer could be used, for example, in a multimedia system where the queue is an actor capable of decompressing a compressed video stream: the actor has a fixed storage capacity for frames. Until a compressed frame is decompressed and consumed, it occupies a buffer slot. The actor accepts messages containing a compressed frame and messages removing an uncompressed frame. The frames may only stay in the actor for a bounded amount of time. The buffer space must then be freed up for processing of new, fresh frames.

A.3.4 Example 4: Rate Control

The example shown in Figure A.9 illustrates how rate control can be described. At most 20 move operations can safely be performed on an actuator in any time window of 30 time units.

```
actor actuator { method move( ) { ... }}
actor eventGenerator {
  method timeout( ) { send self.timeOut( ); }
}
synchronizer rateControll (actorRef: actuator, eventGen) {
  int credit=20; //  max no of events in window
  // timeout 30 tu's after move:
  actuator.move ⇒ eventGen.timeOut ≺ 30;
  actuator.move ⇒ eventGen.timeOut ≻ 30;
  // event permitted?
  disable actuator.move when credit ≤ 0;
  // timeOut must be after move!
  disable eventGen.timeOut when credit ≥ 20;
  actuator.move: credit−−;
  eventGen.timeOut: credit++;
}
```

Figure A.9. *Rate control.*

We use an *event generator actor* to produce message invocations so that the synchronizer changes state at certain time-points. An event generator actor does not add any functionality per se, but is necessary for the proper functioning of the synchronizer. This programming technique obviates the need for a special internal event concept in RT-Synchronizers$^-$.

A.4 Formal Definition

In this section we provide a formal definition of our model. The formal model defines the permissible behavior of a constrained actor program, which is crucial for determining which executions on a physical machine will be considered correct.

The separation of functionality and constraints is maintained in the formal definition, and this enables the semantics for Actors and RT-Synchronizers$^-$ to be given as independent transition systems. The meaning of a program composed of actors and synchronizers is then given by putting the two transition systems in "parallel". Figure A.10 gives an overview of the transition systems to be defined. A numerator denominator-pair should be read as $\frac{\text{Premise}}{\text{Conclusion}}$, where the premise is the condition that must hold in order for the conclusion to hold. The κ transitions define semantics for Actors, the γ transitions for individual constraints, and the σ transitions for synchronizer objects. Finally, $\kappa\sigma$ transitions define the behavior of a constrained actor-system.

$$\frac{\frac{}{\text{Actors: } \longrightarrow_{\kappa}} \qquad \frac{\text{Single constraints: } \longrightarrow_{\gamma}}{\text{Synchronizers: } \longrightarrow_{\sigma}}}{\text{Constrained System: } \longrightarrow_{\kappa\sigma}}$$

Figure A.10. *Dependencies of the transition systems to be defined.*

A.4.1 Semantics of Actors

We first define a transition system κ for an actor language. This defines how the state of an actor system changes when a primitive operation is performed, thus giving an abstract interpretation. The actor semantics presented here is inspired by the work of Agha et. al. [5] where a well-developed theory of actors can be found. However, note that we present actor semantics in imperative style rather than the applicative style used in previous work. Our semantic model abstracts away the notion of methods. Instead, each actor has a single behavior—a sequence of statements—which it applies to every incoming message.

When an actor has completed processing a message it executes the **ready** command to indicate its readyness to accept a new message. As an aside, readers familiar with the classic Actor literature will note that the original **become** primitive has been replaced with **ready**. When an actor executed a **become** it created a new anonymous actor to carry out the rest of its computation, and prepared itself to receive a new message. Thus, in the classic model, actors were multi-threaded, and tended to be extremely fine-grained. In recent literature [3], the simpler **ready** has replaced **become**, with essentially no loss of expressiveness. In addition we have, due to brevity, omitted the semantic definition of dynamic actor creation.

The state of an actor system is represented by a configuration which can be thought of as an instantaneous snapshot of the system state made by a conceptual observer. It is modeled as a pair $\langle\ \alpha\ \mid \mu\ \rangle$ where α represents *actor states*, and μ is the set of *pending messages*. The α mapping maintains the state of all actors in the system. An actor state holds the execution state of an actor: the values of its state variables and the commands that remain to be executed. An actor state is written $[E \vdash b]_a$ where a is the actor's address, E is an environment (mapping from identifiers to their values) tracking the values of the state variables, and b is the remainder of the actor's behavior. In each computation step the actor reduces the behavior until it reaches a **ready**(x) statement. This juncture signifies that the actor a is waiting for an incoming message whose contents should be bound to x. When a message arrives, the actor continues its execution. A message is a pair $\langle a \Leftarrow cv\rangle$ consisting of a destination actor address a, and a value to be communicated cv. In general cv encodes information about which method to invoke along with the values of the method's parameters.

$\langle \mathbf{fun} : a\rangle$

$$\frac{E \vdash b \ \longrightarrow_{\lambda}\ E' \vdash b'}{\langle\ \alpha\,,\ [E \vdash b]_a\ \mid \mu\ \rangle \longrightarrow_{\kappa} \langle\ \alpha\,,\ [E' \vdash b']_a\ \mid \mu\ \rangle}$$

$\langle \mathbf{snd} : a, \langle a' \Leftarrow cv\rangle\rangle$

$$\langle\ \alpha\,,\ [E \vdash \mathbf{send}(a', cv); b]_a\ \mid \mu\ \rangle \longrightarrow_{\kappa} \langle\ \alpha\,,\ [E \vdash b]_a\ \mid \mu\,,\ \langle a' \Leftarrow cv\rangle\ \rangle$$

$\langle \mathbf{rcv} : a, \langle a \Leftarrow cv\rangle\rangle$

$$\langle\ \alpha\,,\ [E \vdash \mathbf{ready}(x); b]_a\ \mid \mu\,,\ \langle a \Leftarrow cv\rangle\ \rangle \longrightarrow_{\kappa} \langle\ \alpha\,,\ [E[x \mapsto cv] \vdash b]_a\ \mid \mu\ \rangle$$

Figure A.11. *Configuration transitions* $\longrightarrow_{\kappa}$.

The semantics of actors is given in Figure A.11. The **fun** transition defines the effect on system state when an actor performs an internal computation step, i.e. a reduction of an expression. The transition system $\longrightarrow_\lambda$ defines the semantics of the sequential language used to express actor behaviors. Since we do not rely on a specific language, we have omitted its definition.

The interpretation of **send** is given by the **snd**-rule. The newly sent message is added to μ. Message reception is described by the **rcv** transition. When an actor executes a $\mathbf{ready}(x)$ command, it becomes ready to accept a new message in an environment with the updated state variables left by the previous processing. Also, the actual value carried by the message is bound to the formal argument x. Finally, the message is removed from μ. It is exactly these receive transitions that will be constrained by RT-Synchronizers$^-$. Other transitions are only affected indirectly.

From this semantics one can make no assertions about the execution time of an actor program; how, then, can we meet real-time requirements? To make this point clear, we temporarily introduce time into the Actor semantics.

Time can be added to transition systems by introducing a special set of *delay actions* written as $\varepsilon(d)$ where d is a finite positive real-valued number representing the passage of d time units. The idea is that system execution can be observed by alternatingly observing a set of instantaneous transitions and observing a delay. In [76] this idea was termed the *two-phase functioning principle*: system state evolves alternatingly by performing a sequence of instantaneous actions and by letting time pass.

By adding the rule: $\langle\ \alpha\ \mid \mu\ \rangle \xrightarrow{\varepsilon(d)}_\kappa\ \langle\ \alpha\ \mid \mu\ \rangle$, we extend the $\longrightarrow_\kappa$ transition relation with the ability to let time pass. The rule states that any actor configuration is always able to delay transitions for some (finite) amount of time. The consequence is that one cannot tell how long a time an actor program takes to finish; indeed the interval between any pair of actions is indeterminate. This is a reasonable model for untimed concurrent programs, where no assumptions on the relative order or timing of events should be made. However, a language with this semantics is unsuitable for real-time system: from the code one can only make assertions about eventuality properties, not about bounded timing. A real-time programming language should make assertions about time bounds possible, and its semantics should define when and by how much can time advance.

A.4.2 RT-Synchronizers$^-$ Semantics

We start by defining semantics for single constraints ($\longrightarrow_\gamma$ transition system), and thereafter proceed to a synchronizer object ($\longrightarrow_\sigma$ transition system); the latter is essentially a state plus a collection of constraints and triggers. The state variables of a synchronizer will be represented by an environment V mapping identifiers to their values. Constraints and patterns are evaluated in this environment.

Recall that a constraint has the form $p_1 \Rightarrow p_2 \sim y$. Whenever an invocation matches p_1 the constraint fires thereby creating a new demand instance for an invocation matching p_2. Such a *demand* will semantically be represented by the triple $p_2 \sim d$, where d is a real number denoting the deadline or release time of p_2, depending on $\sim$. d is initialized with the value of state variable y, $V(y)$, when fired.

$\langle \mathbf{Sat}_{\prec} : a(cv) \rangle$

$$\frac{c_s = \begin{cases} \emptyset & \mathit{if}\, a(cv) \models p_2 \\ p_2 \prec d' & \mathit{otherwise} \end{cases} \qquad c_f = \begin{cases} p_2 \prec V(y) & \mathit{if}\, a(cv) \models p_1) \\ \emptyset & \mathit{otherwise} \end{cases}}{\langle\!\langle\ \xi_{\prec} \mid \chi \uplus p_2 \prec d'\ \rangle\!\rangle \xrightarrow{a(cv)}_\gamma \langle\!\langle\ \xi_{\prec} \mid \chi \uplus c_f \uplus c_s\ \rangle\!\rangle}$$

$\langle \mathbf{Sat}_{\succ} : a(cv) \rangle$

$$\frac{c_s = \begin{cases} \emptyset & \mathit{if}\, a(cv) \models p_2 \wedge d' \leq 0 \\ p_2 \prec d' & \mathit{otherwise} \end{cases} \qquad c_f = \begin{cases} p_2 \succ V(y) & \mathit{if}\, a(cv) \models p_1) \\ \emptyset & \mathit{otherwise} \end{cases}}{\langle\!\langle\ \xi_{\succ} \mid \chi \uplus p_2 \succ d'\ \rangle\!\rangle \xrightarrow{a(cv)}_\gamma \langle\!\langle\ \xi_{\succ} \mid \chi \uplus c_f \uplus c_s\ \rangle\!\rangle}$$

$\langle \mathbf{Sat}_{\emptyset} : a(cv) \rangle$

$$\frac{c_f = \begin{cases} p_2 \sim V(y) & \mathit{if}\, a(cv) \models p_1) \\ \emptyset & \mathit{otherwise} \end{cases}}{\langle\!\langle\ \xi_{\sim} \mid \emptyset\ \rangle\!\rangle \xrightarrow{a(cv)}_\gamma \langle\!\langle\ \xi_{\sim} \mid \emptyset \uplus c_f\ \rangle\!\rangle}$$

$\langle \mathbf{Delay}_{\sim} : e \rangle$

$$\frac{\forall p_2 \prec d_i \in (\chi \ominus e).d_i \geq 0}{\langle\!\langle\ \xi_{\sim} \mid \chi\ \rangle\!\rangle \xrightarrow{\varepsilon(e)}_\gamma \langle\!\langle\ \xi_{\sim} \mid \chi \ominus e\ \rangle\!\rangle}$$

$$a(cv) \models x_1(x_2)\mathbf{when}\ b =_{def} a = V(x_1) \wedge b(V[x_2 \mapsto cv])$$

Figure A.12. *Semantics for single constraints* $\longrightarrow_\gamma$, $\sim \in \{\succ, \prec\}$.

Since a constraint can fire many times successively, a constraint may induce many outstanding demand instances. The state of a single constraint is therefore represented as a *constraint configuration* $\langle\!| \xi_\sim \mid \chi |\!\rangle$ where $\xi_\sim$ stands for the (static description of a) constraint of the form $p_1 \Rightarrow p_2 \sim y$, and χ is a multi-set of demands instantiated from the static description $\xi_\sim$. The semantic rules are shown in Figure A.12.

The function c_s determines whether the pattern of a demand instance is satisfied, and if so, removes it from the demand instance set. If the pattern is not satisfied, the demand is maintained. Similarly, the function c_f determines whether or not the constraint fires and therefore whether or not to add a new demand instance. Thus the **Sat**-rules ensure that whenever a constraint fires, a demand (c_f) is added to χ. Also, whenever a demand (c_s) is satisfied, it is removed from χ. Due to the possibility of a single message matching both p_1 and p_2 the **Sat**-rules are prepared to both satisfy and fire a demand. The demand instance to be removed is chosen non-deterministically, giving the implementation maximal freedom to choose the demand it finds the most appropriate, e.g., the one with the tightest deadline.

Passage of time is controlled by the **Delay**-rule such that the elapsed amount of time (e) is subtracted from d_i in each demand $p_i \sim d_i$. This is written $\chi \ominus e$. Thus for $p \succ d$, d is the amount of time that *must* pass before p is enabled. In particular, p will be enabled when d is less than 0. This requirement is enforced by the c_s function of the $\langle \mathbf{Sat}_\succ : a(cv)\rangle$ rule. For $p \prec d$, d is the amount of time that *may* pass before p will be disabled. p would be disabled if d is less than 0. However the $\langle \mathbf{Delay}_\prec : e\rangle$ rule prevents time from progressing that much. In effect, the delay rule ensures that deadline constraints are *always* satisfied in the semantics. This corresponds to the declarative meaning one would expect from a constraint: something that must be enforced. Without this strict definition, our constraints would degenerate to mere assertions and not convey their intended meaning. Note that an actual language implementation may not always be able to give this guarantee — either statically or dynamically — for two reasons. First, because physical resources may not exist to realize them, and second, because finding feasible schedules for general constraints is computationally very complex.

Conflicting constraints that have no solutions should be detected as part of the compiler's static program check. Ren has shown how RT-Synchronizers$^-$ constraints can be mapped to linear inequality systems for which polynomial time algorithms exist for detecting solvability [90, 88].

The following transition sequence illustrates application of the transition rules for a constraint:

$$\begin{array}{ll}
\langle\!\langle\, p_1 \Rightarrow p_2 \prec 7 \mid \emptyset \,\rangle\!\rangle & \xrightarrow{a_1(cv)}_{\gamma} \\
\langle\!\langle\, p_1 \Rightarrow p_2 \prec 7 \mid p_2 \prec 7 \,\rangle\!\rangle & \xrightarrow{\varepsilon(3)}_{\gamma} \\
\langle\!\langle\, p_1 \Rightarrow p_2 \prec 7 \mid p_2 \prec 4 \,\rangle\!\rangle & \xrightarrow{a_1(cv)}_{\gamma} \\
\langle\!\langle\, p_1 \Rightarrow p_2 \prec 7 \mid p_2 \prec 4, p_2 \prec 7 \,\rangle\!\rangle & \xrightarrow{\varepsilon(4)}_{\gamma} \\
\langle\!\langle\, p_1 \Rightarrow p_2 \prec 7 \mid p_2 \prec 0, p_2 \prec 3 \,\rangle\!\rangle & \xrightarrow{a_2(cv)}_{\gamma} \\
\langle\!\langle\, p_1 \rightarrow p_2 \prec 7 \mid p_2 \prec 3 \,\rangle\!\rangle & \xrightarrow{a_2(cv)}_{\gamma} \\
\langle\!\langle\, p_1 \rightarrow p_2 \prec 7 \mid \emptyset \,\rangle\!\rangle &
\end{array}$$

Given that the behavior of each individual constraint is well defined, it is easy to define the behavior of a collection of constraints as found within a synchronizer. Essentially the individual constraints are conjoined, i.e., we require that all constraints agree on a given invocation. Similarly, they must all agree on letting time pass.

A synchronizer is represented by a *synchronizer configuration* $[\bar{\gamma}|V]$ where $\bar{\gamma}$ is a *set* of constraint configurations (ranged over by γ). As previously stated V represents the state variables of a synchronizer and is a mapping from identifiers to their values. The necessary definition is shown in Figure A.13. A synchronizer can engage in message reception $a(cv)$ or delay $\varepsilon(e)$ only when it is permitted by every constraint.

We have omitted the rather simple definition of the effect of triggers: V' is V simultaneously updated with the specified assignments in the matched triggers.

$$\langle \textbf{Action} : \ell \rangle$$

$$\frac{\forall i \in [1..n].\gamma_i \xrightarrow{\ell}_{\gamma} \gamma_i'}{[\gamma_1, \ldots, \gamma_n | V] \xrightarrow{\ell}_{\sigma} [\gamma_1', \ldots, \gamma_n' | V']}, \ell \in \{a(cv), \varepsilon(e)\}$$

Figure A.13. *Semantics for a synchronizer* $\longrightarrow_{\sigma}$.

A.4.3 Combining Actors and RT-Synchronizers$^-$

The preceding sections defined Actor and RT-Synchronizers$^-$ languages independently. The effect of constraining an actor program can now be defined here as a special form of parallel composition (denoted by $||$) that preserves the meaning of constraints. We call a collection of synchronizers an *interaction constraint system configuration* which is written $(\sigma_1, \ldots, \sigma_n)$ where σ ranges over synchronizer configurations. The composition $\parallel$ of an actor configuration and an interaction constraint system configuration is defined in Figure A.14.

Unaffected Actions

$$\frac{\langle\ \alpha\ |\mu\ \rangle \xrightarrow{\ell}_{\kappa} \langle\ \alpha'\ |\mu'\ \rangle \qquad \ell \in \{\langle \mathbf{fun} : a\rangle, \langle \mathbf{snd} : a, m\rangle, \langle \mathbf{ready} : a\rangle\}}{\langle\ \alpha\ |\mu\ \rangle \parallel (\sigma_1,\ldots,\sigma_n) \xrightarrow{\ell}_{\kappa\sigma} \langle\ \alpha'\ |\mu'\ \rangle \parallel (\sigma_1,\ldots,\sigma_n)}$$

Receive

$$\frac{\langle\ \alpha\ |\mu\ \rangle \xrightarrow{\ell}_{\kappa} \langle\ \alpha'\ |\mu'\ \rangle \qquad \bigwedge_{i\in[1..n]} \sigma_i \xrightarrow{a(cv)}_{\sigma} \sigma'_i \qquad \ell = \langle \mathbf{rcv} : a, \langle a \Leftarrow cv\rangle\rangle}{\langle\ \alpha\ |\mu\ \rangle \parallel (\sigma_1,\ldots,\sigma_n) \overset{\ell}{\to}_{\kappa\sigma} \langle\ \alpha'\ |\mu'\ \rangle \parallel (\sigma'_1,\ldots,\sigma'_n)}$$

Delay

$$\frac{\bigwedge_{i\in[1..n]} \sigma_i \xrightarrow{\varepsilon(d)}_{\sigma} \sigma'_i}{\langle\ \alpha\ |\mu\ \rangle \parallel (\sigma_1,\ldots,\sigma_n) \xrightarrow{\varepsilon(d)}_{\kappa\sigma} \langle\ \alpha\ |\mu\ \rangle \parallel (\sigma'_1,\ldots,\sigma'_n)}$$

Figure A.14. *Combined behavior* $\longrightarrow_{\kappa\sigma}$.

Transitions unaffected by interaction constraints altogether are message sends and local computations. These only have effect on the actor configuration. Message invocations $\langle \mathbf{rcv} : a, m\rangle$ are the interesting events affected by constraints. Note that the same invocation may be constrained by several synchronizers, and *all* must certify the invocation, i.e., synchronizers, like constraints, are composed conjunctively. The idea is that adding more synchronizers should further restrict the behavior of objects. A consequence of this idea is that the synchronizers also must agree on letting time pass.

The combined semantics define all correct transition sequences $(\longrightarrow^{*}_{\kappa\sigma})$. A transition sequence corresponds to one possible *schedule* of the implemented system (consisting of actors, constraints, operating system, runtime system, and hardware resources), and thus a primary task of the language implementation is to schedule events in the system such that the resulting schedule can be found in the program's semantics. Thus, a program consisting of actors and RT-Synchronizers^{-} can be viewed as a specification for the set of possible systems.

Observe that not all transition sequences defined by $\longrightarrow^*_{\kappa\sigma}$ are realizable on a physical machine. The problem is related to the progress of time and our intuition about causal ordering. Suppose event e_1 is a method invocation resulting in the sending of a message which eventually causes a method invocation, event e_2, then we surely would expect that time has progressed between these events. That is, in terms of a fictitious global clock C, it should hold that $C(e_1) < C(e_2)$. However, in our semantics, time is not *required* to pass between causally related events, but only permitted to.

There are two related problems, time locks and cluster points. A time lock occurs when no time progress is possible, i.e., the delay transition is eternally disabled. In our model this occurs as consequence of an unsatisfiable deadline constraint. A cluster point is a bounded interval of time in which an infinite number of events occur. It is possible to write such a specification in RT-Synchronizers$^-$. However, it will not be implementable on a (finitely fast) computer! Since our goal is to define the permissible implementations, and since time locks and cluster points are only required when explicitly specified, we have taken no measure to prohibit such behavior. A compiler should, however, warn developers about such unsatisfiable constraints.

A.5 Middleware Scheduling of RT-Synchronizers$^-$

The examples in Figures A.5–A.9 illustrated how our language can be used as a specification or modeling language that defines the structure and permissible behavior of a computer system consisting of hardware and system software executing an application.

An attractive approach to implementing a language that supports separation of objects and time constraints is to use a *middleware scheduling/event dispatching* service. Such a service is depicted in Figure A.15. An application consists of two parts, objects and time constraints. A set of potentially reusable objects are composed by middleware services for communication and scheduling. Communication typically includes request-reply communication, point-to-point real-time communication, and group communication. The scheduler(s) are responsible for event dispatching and resource (typically processor) allocation, based on information that is specified by the application separately from the objects. Thus, objects are being controlled by the middleware, rather than controlling themselves or each other.

Specifically, given a set of synchronizers as input, this service should, preferably without further programmer involvement, schedule message invocations in accordance with the specified real-time and synchronization constraints.

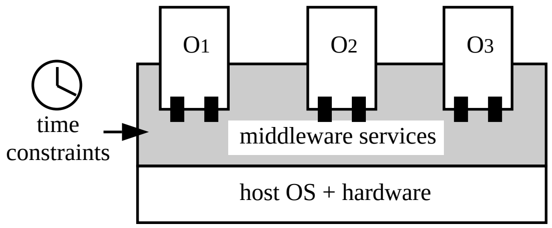


Figure A.15. *Middleware integrates pre-built objects.*

The remainder of this section is devoted to uncovering what work such a service must do to execute the specification directly.

Implementing our full model is not an easy task, but the difficulty is mostly related to the generality of the constraints that can be expressed, rather than due to the separation of functionality and time constraints. We have identified three main tasks a compiler and scheduling service should address:

Scheduling: One challenge is to find a scheduling strategy that satisfies the deadline constraints when the RT-Synchronizers$^-$ program is executed on a physical machine with limited resources. In addition, hard and firm real-time systems require an *a priori* guarantee (or at least a solid argument) that timing constraints will be satisfied on the chosen platform during runtime.

Constraint propagation: In RT-Synchronizers$^-$ the programmer need only specify end-to-end timing relations, not intermediate constraints on all events along the call chain. Assume that actor a receives a message $m1$; a then responds with a message $m2$ to actor b which in turn sends a message $m3$ to actor c. Let a_{m1}, b_{m2} and c_{m3} denote the reception events of these messages. Then a typical interaction constraint would be $a_{m1} \Rightarrow c_{m3} \prec 10$. This scenario is depicted in Figure A.16. Consequently, there is an implicit constraint on event b_{m2} which is to happen (well) before c_{m3}. Ideally, the compiler/runtime system should be able to perform constraint propagation along the call chain, and derive the intermediate deadlines.

Synchronizer distribution: If the synchronizer entities are maintained as runtime objects, how should their state be distributed? Here there is a classic compromise between a centralized solution where consistent

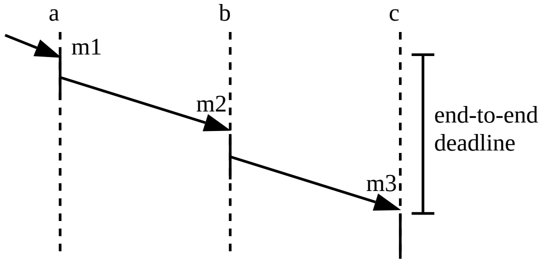


Figure A.16. *End-to-end deadlines require computation of intermediate deadlines along the call chain.*

updates are easy versus a distributed solution that potentially reduces bottlenecks and increases fault tolerance, but by increasing the cost of maintaining consistency.

Our implementation idea seems practical for soft real-time systems only: we provide no procedure, whether automatic or manual, for establishing the guarantees of satisfaction of time constraints as required by hard real-time systems, and for the unrestricted type of real-time and synchronization constraints that we permit in our language. Additionally, a full *verification* of the implemented system is rarely practical. To make schedulability analysis practical, one often restricts the types of constraints to *periodic* constraints. Similar restrictions can be made to RT-Synchronizers^{-}. With simple dependencies between periodic tasks generalized rate-monotonic analysis can be utilized [104].

Constraint directed scheduling is an implementation technique that dynamically uses the information of the fired constraints in the synchronizers to assign deadlines and release times to messages (see Figure A.17). Synchronizer objects are thus maintained at run time as data objects, whose state can be inspected by the scheduler.

Time-based scheduling such as Earliest-Deadline-First (EDF) can then be used to dispatch messages based on their deadlines. We propose to use EDF-scheduling because it is dynamic and optimal: if a feasible schedule exists EDF will produce one. Obviously, EDF does not in itself guarantee that a feasible schedule exists and constraint violations may therefore occur. An advantage of our strategy is that it does more than simply monitor the time constraints; it constructively applies information from the synchronizers to its scheduling decisions.

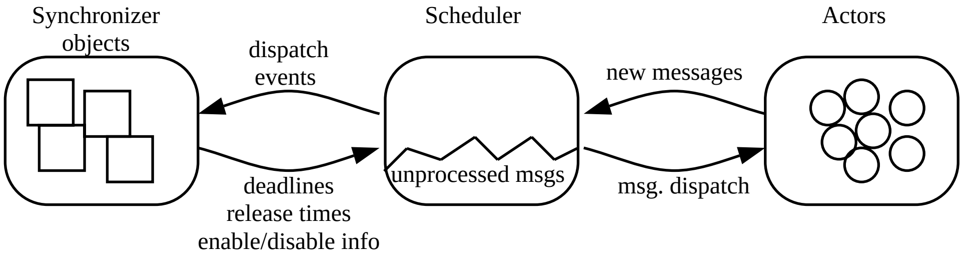


Figure A.17. *Implementation architecture with constraint directed scheduling.*

We propose to let the compiler compute a conservative version of the call graph annotated with worst case execution time and message propagation delays, and include a copy of it at runtime [88]. The runtime system then has the information necessary to propagate constraints automatically when this cannot be done statically by the compiler. Moreover, we expect that in many cases the compiler would be able to compile away synchronizers entirely. It can generate code (similar to remote-procedure-call stubs) which can be linked with the objects. This code implements the time constraints by manipulating timers, setting priorities and/or instructing the scheduler about method call deadlines, etc.

It is interesting to note that the operational semantics can assist in the implementation of a constraint directed scheduling system. An operational semantics can often be constructed such that it constitutes an abstract algorithm for the language implementation. However, because our semantics abstracts away any notion of resources and execution time, in our case, this algorithm can only be partial. In particular, it does not solve the constraint propagation problem mentioned earlier.

The following example demonstrates two potential benefits of the semantics. First, it shows how the semantics manipulates the synchronizer data structure by adding and removing constraints, and second it indicates how release times and deadlines for messages can be deduced. Recall the boiler example in Section A.3.1. We show how the runtime system may execute that specification. We maintain two important data structures, the set of fired demands, and the pool of unprocessed messages. We reuse the notation for demands from the semantics: $\langle\!|\ \xi_{\sim} \mid \chi\ |\!\rangle$ where $\xi_{\sim}$ stands for the static description of a constraint, and χ is the multi-set of instantiated demands. A message is written as $o.m[R,D]$ where o is the target object, m the method to be invoked, and R and D respectively the release time and deadline of the message. In the following, we measure time relative to a global clock $\mathbf{t}$, and not using individual timers as was convenient in the semantics. Each row in Figure A.18 shows the global time at which a given event (i.e., message invocation) occurs, the resulting synchronizer state, and the set of unprocessed messages (including those produced by the event).

t	Event	Synchronizer State	Message Pool
0	(initial)	$\langle\!\langle\ \text{c.loop} \Rightarrow \text{c.loop} \prec 20+\epsilon \mid \emptyset\ \rangle\!\rangle$ $\langle\!\langle\ \text{c.loop} \Rightarrow \text{c.loop} \succ 20-\epsilon \mid \emptyset\ \rangle\!\rangle$ $\langle\!\langle\ \text{c.loop} \Rightarrow \text{c.reading} \prec 10 \mid \emptyset\ \rangle\!\rangle$ $\langle\!\langle\ \text{c.reading} \Rightarrow \text{v.move} \prec 5 \mid \emptyset\ \rangle\!\rangle$	$\text{c.loop}[0,\infty]$
1	c.loop	$\langle\!\langle\ \text{c.loop} \Rightarrow \text{c.loop} \prec 20+\epsilon \mid \text{c.loop} \prec 1+20+\epsilon\ \rangle\!\rangle$ $\langle\!\langle\ \text{c.loop} \Rightarrow \text{c.loop} \succ 20-\epsilon \mid \text{c.loop} \succ 1+20-\epsilon\ \rangle\!\rangle$ $\langle\!\langle\ \text{c.loop} \Rightarrow \text{c.reading} \prec 10 \mid \text{c.reading} \prec 1+10\ \rangle\!\rangle$ $\langle\!\langle\ \text{c.reading} \Rightarrow \text{v.move} \prec 5 \mid \emptyset\ \rangle\!\rangle$	$\text{c.loop}[21-\epsilon,21+\epsilon]$ $\text{s.read}[0,6]^{\ddagger}$
4	s.read	$\langle\!\langle\ \text{c.loop} \Rightarrow \text{c.loop} \prec 20+\epsilon \mid \text{c.loop} \prec 1+20+\epsilon\ \rangle\!\rangle$ $\langle\!\langle\ \text{c.loop} \Rightarrow \text{c.loop} \succ 20-\epsilon \mid \text{c.loop} \succ 1+20-\epsilon\ \rangle\!\rangle$ $\langle\!\langle\ \text{c.loop} \Rightarrow \text{c.reading} \prec 10 \mid \text{c.reading} \prec 1+10\ \rangle\!\rangle$ $\langle\!\langle\ \text{c.reading} \Rightarrow \text{v.move} \prec 5 \mid \emptyset\ \rangle\!\rangle$	$\text{c.loop}[21-\epsilon,21+\epsilon]$ $\text{c.reading}[0,11]$
9	c.reading	$\langle\!\langle\ \text{c.loop} \Rightarrow \text{c.loop} \prec 20+\epsilon \mid \text{c.loop} \prec 1+20+\epsilon\ \rangle\!\rangle$ $\langle\!\langle\ \text{c.loop} \Rightarrow \text{c.loop} \succ 20-\epsilon \mid \text{c.loop} \succ 1+20-\epsilon\ \rangle\!\rangle$ $\langle\!\langle\ \text{c.loop} \Rightarrow \text{c.reading} \prec 10 \mid \emptyset\ \rangle\!\rangle$ $\langle\!\langle\ \text{c.reading} \Rightarrow \text{v.move} \prec 5 \mid \text{v.move} \prec 9+5\ \rangle\!\rangle$	$\text{c.loop}[21-\epsilon,21+\epsilon]$ $\text{c.move}[0,14]$
13	v.move	$\langle\!\langle\ \text{c.loop} \Rightarrow \text{c.loop} \prec 20+\epsilon \mid \text{c.loop} \prec 1+20+\epsilon\ \rangle\!\rangle$ $\langle\!\langle\ \text{c.loop} \Rightarrow \text{c.loop} \succ 20-\epsilon \mid \text{c.loop} \succ 1+20-\epsilon\ \rangle\!\rangle$ $\langle\!\langle\ \text{c.loop} \Rightarrow \text{c.reading} \prec 10 \mid \emptyset\ \rangle\!\rangle$ $\langle\!\langle\ \text{c.reading} \Rightarrow \text{v.move} \prec 5 \mid \emptyset\ \rangle\!\rangle$	$\text{c.loop}[21-\epsilon,21+\epsilon]$
21	c.loop	$\langle\!\langle\ \text{c.loop} \Rightarrow \text{c.loop} \prec 20+\epsilon \mid \text{c.loop} \prec 21+20+\epsilon\ \rangle\!\rangle$ $\langle\!\langle\ \text{c.loop} \Rightarrow \text{c.loop} \succ 20-\epsilon \mid \text{c.loop} \succ 21+20-\epsilon\ \rangle\!\rangle$ $\langle\!\langle\ \text{c.loop} \Rightarrow \text{c.reading} \prec 10 \mid \text{c.reading} \prec 21+10\ \rangle\!\rangle$ $\langle\!\langle\ \text{c.reading} \Rightarrow \text{v.move} \prec 5 \mid \emptyset\ \rangle\!\rangle$	$\text{c.loop}[41-\epsilon,41+\epsilon]$ $\text{s.read}[0,26]$

Figure A.18. *Sample execution of the boiler specification.*

At time 0, the system is shown in the initial state in which the message pool contains an initialization message (*controller.loop*) and in which no synchronizer demands have been fired. Suppose the scheduler invokes the *controller.loop* message at time 1. This invocation matches three constraints and consequently causes the synchronizer to issue three new demands. The two first constitute the periodic constraint on a future loop message and the last one determines the deadline on the sensor reading. During processing of the loop message the controller sends out two new messages, the loop message to itself, and a read request to the pressure sensor.

The new loop message matches two demands, and according to the semantics these are applied conjunctively. The runtime system can therefore deduce the release time and the deadline (an ϵ interval around time 21) for the loop message from the demands. Deducing a deadline for *sensor.read* constitutes a more difficult case (labeled with a ‡ symbol in Figure A.18). There is no immediate matching demand on which to base the deadline. But it can be noted that there is a demand for which no matching message exists in the message pool. It is therefore likely that invocation of the unmatched *sensor.read* message will cause sending of the demanded message (as it indeed turns out to be the case in this example). Therefore the *sensor.read* message should be assigned a deadline before the demanded deadline (at time 11). The specific choice of deadline is in general a heuristic function of slack time and method computation time. Here time 6 is chosen.

The approach of assigning unmatched messages deadlines based on the most urgent unmatched demand will generally constrain the system unnecessarily, but selecting precisely the right message to constrain is generally impossible without extra information about potential causal relations between messages. This information is exactly what needs to be generated by the compiler. Less ideally, the missing constraints could be resolved explicitly by the programmer by providing additional synchronizers. In a less expressive real-time programming languages where end-to-end constraints cannot be expressed, the programmer would always be forced to do this.

Resuming the example at time 4 where *sensor.read* is invoked, the sensor responds with a *controller.reading*. Since this message matches a demand, it inherits the deadline from that (time 11). The result of invoking the reading message (at time 9) is the firing of a new demand on the valve movement and the sending of a *valve.move* message. Again, the runtime system is able to deduce the deadline on the move message from the move demand. Finally, at time 21, the loop message is invoked. This satisfies the remaining two demands, but at the same fires two new demands, which starts the next period.

A.6 Related Work

Real-time CORBA (Common Object Request Broker Architecture) [47] is a highly visible research effort where practitioners are shifting towards component-based real-time systems. An object request broker can be viewed as middleware facilitating transparent client-server communication in a heterogeneous distributed system. It also contains other communication services to facilitate building distributed applications. However, according to [99], current ORBs are ill-suited for real-time systems for at least four reasons. They lack interfaces for specifying quality of service, quality of service enforcement, real-time programming facilities, and performance optimizations.

Current proposals for real-time CORBA [99, 34, 40, 58] use a quality of service metaphor for specifying real-time constraints. Typically, the interface definition language is extended with QoS-datatypes. In TAO ORB [99], these parameters, which are necessary for guaranteeing schedulability according to rate monotonic scheduling, include worst case execution time, period, and importance. In NRad/URI's proposal [34] for a dynamic CORBA, time constraints are specified in a structure containing importance, deadline and period, and the constraints specify time bounds on a client's method invocations on a server. The proposed runtime system uses this information to compute dynamic scheduling and queuing priorities. The Realize proposal [58] associates deadline, reliability, and importance attributes to application tasks, where a task is defined as a sequence of method invocations between an external input and the generation of an external result. That is, deadlines in Realize are true end-to-end deadlines.

We see a clear trend in specifying real-time requirements through interface definitions and letting middleware enforce them. Clients and servers are largely unaware of the imposed real-time requirements. However, we think that these approaches—although an improvement—are imperfect:

- The quality of service attributes seem to be derived from what current run-time systems can manage rather than forming a coherent set. We have opted for a clean language instead of a more or less arbitrary collection of attributes.

- The types of constraints that can be specified are restrictive, e.g., only periods or deadlines between request and reply events. In addition, the constraints are static; once assigned they cannot be modified to respond to dynamic changes in the system's state of affairs. We allow for a fairly general set of constraints to be specified.

- Synchronization constraints are not considered. In our proposal, synchronization constraints are specified using the same mechanism as time constraints.

The concept of separating functional behavior and interaction policies for Actors was first proposed by Frølund and Agha in [43] and a detailed description, operational semantics and implementation can be found in [42]. That work only considered constraints on the order of operations. Our work is a continuation of this line of research where we have extended it to apply to real-time systems and provided a formal treatment of the extended model. However, to what extent real-time and synchronization constraints can always be cleanly separated from functionality remains an open issue, and one which we think can be best resolved through larger case studies.

Another approach which permits separate specification of real-time and synchronization constraints for an object-oriented language is the composition filter model [6, 15]. Real-time input and output filters declared in an extended interface enable the specification of time bounds on method executions. Among the differences between composition filters and RT-Synchronizers$^-$ is that RT-Synchronizers$^-$ takes a global view of a collection of objects whereas the composition filter model takes a single object view. No formal treatment of composition filters appears to be available in the literature.

The Real-time Object-Oriented Modeling method (ROOM) [103], which has many notions in common with the Actor model, has recently been extended with notions for specifying real-time properties [96]: message sequence charts with annotated timing information can now be used to express activation periods of methods or end-to-end deadlines on sequences of message invocations. With these two kinds of constraints and a few design guidelines, the authors show how scheduling theory can be applied to ROOM-models.

Our approach to defining the semantics is inspired by recent research in formal specification languages for real-time systems, and the use of timed transition systems is borrowed from these languages. These languages often take the form of extended automata (Timed automata [7], Timed Graphs [7, 76]), or process algebras such as Timed CSP [102]. A different approach is to include a model of the underlying execution resources. This approach is taken in [97] and [121]. The resulting semantics includes an abstract model of the execution environment (number of CPU's, scheduler, execution time of assignments etc.). The process algebra Communicating Shared Resources (CSR) has been designed with the explicit purpose of modeling resources [44, 45]. A process always runs on some, possibly shared, resource. A set of processes can be mapped to different sets of resources, hence describing different implementations. Thus, these approaches model relatively concrete systems, rather than being specifications for a set of possible systems, as was our goal.

A recent implementation result is [61] where certain aspects of RT-Synchronizers$^-$ are implemented in their DART framework where constraints are used to dynamically instruct the scheduler about delays and deadlines of messages. However the paper gives no systematic (automatic) translation of constraints to scheduling information. We expect that our semantics can help in filling up this gap.

A.7 Discussion

Developers of modern real-time systems are required to construct increasingly large and complex systems, preferably at no extra cost. To satisfy this requirement, it is essential that developers can build real-time systems from existing components, and that newly developed components can be reused in several applications. We argued that in order to facilitate reuse of real-time objects, the real-time and synchronization constraints governing the object's interaction should be specified separately from the objects themselves. However, current development methods do not adequately support such separation.

We formulated our ideas in the context of Actors, and an associated specification language, RT-Synchronizers$^-$. Combined, they enable separate and modular specification of real-time systems: computing objects are glued together by synchronizer entities that express real-time and synchronization constraints. However, we believe that these ideas are applicable beyond these specific languages.

Our model is explained both conceptually and formally. Through a series of examples we indicated how separate specification is possible. Our operational semantics defines exactly what constraints are and what their effect on a given set of objects should be.

Our work on semantic modeling has clarified our understanding of the behavior of our model, and provides a succinct and detailed definition of synchronizers and constrained actor programs. In particular, we have gained new insight in three areas, which made the effort worthwhile:

- We defined the semantics in a modular fashion by composing a transition system for the untimed object-model with a transition system which interprets the time constraints. This composition explicitly points out which, object transitions are affected and how: reception of messages and time-progress may only occur when permitted by the constraints. Other object transitions are only indirectly affected.

The modularity opens the possibility of plugging in a different constraint specification language, i.e., the $\longrightarrow_\sigma$ transition could be replaced with the semantics for the new language. The composition will work when affected transitions remain as above, and when the semantics of the new language can be given as a timed transition system. Thus, our *constraining* concept is captured by the composition.

- Our semantics helped uncover some of the semantic subtleties of our constraint language, such as what happens when patterns and constraints overlap. For example, the same message may both fire a new demand as well as satisfy an existing one. Moreover, we decided that overlapping constraints should be interpreted conjunctively, i.e., both must be satisfied. Finally, we decided that adding more synchronizers should further restrict the behavior of objects; i.e., synchronizers must be satisfied conjunctively.

  It should also be noted that the rules defining the semantics of individual constraints appear complicated. This should give food for thought when revising the language or the semantics.

- The last major benefit is that our semantics suggests an implementation strategy suitable for soft real-time systems. The synchronizer entities can be maintained at runtime and can be used to extract information about release times and deadlines of messages. The semantics gives an abstract interpretation of the synchronizer objects and specifies how demands should be added or removed.

Building real-time components and architectures for integrating them is an area of active research. We believe that with additional research, component-based development will allow more complex real-time systems to be developed on schedule. However, additional work is needed, both on the models used for separate specification and on the middleware services necessary to implement them.

Acknowledgments

This work was made possible in part by support from the US National Science Foundation under contracts NSF CCR-9523253 and NSF CCR-9619522; by support from the US Air Force Office of Scientific Research, under contract AF DC 5-36128. The authors would like to thank other members of the Open Systems Laboratory for their comments and critical insights into the work related in this paper. In particular, we would like to thank Shangping Ren for her contribution to the definition of RTSynchronizers, and to Nadeem Jamali for his comments on a draft of this paper. A part of this research was done while the first author was a visitor to the University of Illinois Open Systems Laboratory under a fellowship from The Danish Technical Research Foundation and the Danish Research Academy.

Bibliography

[1] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Los Alamitos, California, 1986. ISBN 0-262-01092-5.

[2] Gul Agha. Concurrent Object-Oriented Programming. *Communications of the ACM*, 33(9):125–141, September 1990.

[3] Gul Agha. Modeling Concurrent Systems: Actors, Nets, and the Problem of Abstraction and Composition. In *17th International Conference on Application and Theory of Petri Nets*, Osaka, Japan, June 1996.

[4] Gul Agha, Svend Frølund, Rejendra Panwar, and Daniel Sturman. A Linguistic Framework for Dynamic Composition of Dependability Protocols. In *Conference on Dependable Computing for Critical Applications, Sicily, IFIP 1992*, 1992.

[5] Gul Agha, Ian A. Mason, Scott F. Smith, and Carolyn L. Talcott. A Foundation for Actor Computation. *Journal of Functional Programming*, 7:1–72, 1997.

[6] Mehmet Akşit, Jan Bosch, William van der Sterren, and Lodewijk Bergmans. Real-Time Specification Inheritance Anomalies and Real-Time Filters. In *European Conference on Object Oriented Programming (ECOOP'94)*, pages 386–407, 1994.

[7] Rajeev Alur, Costas Courcoubetis, and David Dill. Model–Checking for Real–Time Systems. In *Fifth IEEE Symposium on Logic in Computer Science*, pages 414–425, 1990.

[8] Rajeev Alur and David Dill. The Theory of Timed Automata. In *Real Time: Theory in Practice, REX Workshop '91*, volume LNCS 600 of *Lecture Notes in Computer Science, Springer-Verlag*, pages 45–73, 1992.

[9] Rajeev Alur and David L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126(2):183–235, 25 April 1994.

[10] Rajeev Alur, Limor Fix, and Thomas A. Henzinger. Event-Clock Automata: A Determinizable Class of Timed Automata. In *6th Conference on Computer Aided Verification*, 1994. Also in LNCS 818.

[11] Gerd Behrmann, Kim G. Larsen, Justin Pearson, Carsten Weise, and Wang Yi. Efficient Timed Reachability Analysis Using Clock Difference Diagrams. In *Computer Aided Verification (CAV'99)*, volume LNCS 1633, pages 22–24. Springer Verlag, July 1999. Trento, Italy.

[12] Boris Beizer. *Software Testing Techniques.* International Thompson Computer Press, 1990. 2nd edition, ISBN 1850328803.

[13] Johan Bengtsson, W. O. David Griffioen, Kåre J. Kristoffersen, Kim G. Larsen, Fredrik Larsson, Paul Petterson, and Wang Yi. Verification of an Audio Protocol with Bus Collision using UppAal. In *9th Intl. Conference on Computer Aided Verification*, pages 244–256, 1996. LNCS 1102.

[14] Johan Bengtsson and Fredrik Larsson. UppAal - A Tool for Automatic Verification of Real-Time Systems. Technical Report DOCS 96/67, Department of Computer Science. Uppsala University, february 1996. ISSN 0283-0574.

[15] Lodewijk Bergmans and Mehmet Akşit. Composing Synchronization and Real-Time Constraints. In *The Object Oriented Real-Time Systems Workshop (OORTS)*, October 1995. San Antonio, TX, USA. In conjunction with 7th IEEE Symposium on Parallel and Distributed Computing Systems.

[16] Philip A. Bernstein. Middleware — A Model for Distributed System Services. *Communications of the ACM*, 39(2):86–98, February 1996.

[17] Doeko Bosscher, Indra Polak, and Frits Vaandrager. Verification of an Audio Protocol. TR CS-R9445, CWI, Amsterdam, The Netherlands, 1994. Also in LNCS 863, 1994.

[18] Ahmed Bouajjani, Stavros Tripakis, and Sergio Yovine. On-the-fly Symbolic Model-Checking for Real-Time Systems. In *1997 IEEE Real-Time Systems Symposium, RTSS'97*, San Fransisco, USA, December 1996. IEEE Computer Society Press.

[19] Marius Bozga, Jean-Claude Fernandez, Lucian Ghirvu, Claude Jard, Thierry Jéron, Alain Kerbrat, Pierre Morel, and Laurent Mounier. Verification and Test Generation for the SSCOP Protocol. *Science of Computer Programming*, 36(1):27–52, 2000.

[20] V. Braberman, M. Felder, and M. Marré. Testing Timing Behaviors of Real Time Software. In *Quality Week 1997. San Francisco, USA.*, pages 143–155, April-May 1997 1997.

[21] Ed Brinksma. A Theory for the Derivation of Tests. In *Protocol Specification Testing and Verification VIII (PSTV'88)*, pages 63–74, 1988.

[22] Ed Brinksma, Jan Tretmans, and Louis Verhaard. A Framework for Test Selection. In *Protocol Specification Testing and Verification (PSTV'91)*, pages 216–231, 1991.

[23] Bettina Buth, Michel Kouvaras, Jan Peleska, and Hui Shi. Deadlock Analysis for a Fault-Tolerant System. In Michael Johnson, editor, *Algebraic Methodology and Software Technology. AMAST'97, Sidney, Australia*, pages 60–75, December 1997. Springer LNCS 1349.

[24] Rachel Cardell-Oliver. Conformance Testing of Real-Time Systems with Timed Automata. In *Nordic Workshop on Programming Theory*, Oct 6-8 1999.

[25] Rachel Cardell-Oliver. Conformance Testing of Real-Time Systems with Timed Automata. *Formal Aspects of Computing*, (12):350–371, 2000.

[26] Rachel Cardell-Oliver and Tim Glover. A Practical and Complete Algorithm for Testing Real-Time Systems. In *5th international Symposium on Formal Techniques in Real Time and Fault Tolerant Systems (FTRTFT'98)*, pages 251–261, September 14–18 1998. Also in LNCS 1486.

[27] Tsun S. Chow. Testing Software Design Modeled by Finite-State Machines. *IEEE Transactions on Software Engineering*, 4(3):178–187, may 1978.

[28] Duncan Clarke. *Testing Real-Time Constraints.* PhD thesis, A dissertation in Computer and Information Science. University of Pennsylvania. Department of Computer and Information Science, December 1996.

[29] Duncan Clarke and Insup Lee. Testing Real-Time Constraints in a Process Algebraic Setting. In *17th International Conference on Software Engineering*, 1995.

[30] Duncan Clarke and Insup Lee. Automatic Test Generation for the Analysis of a Real-Time System: Case Study. In *3rd IEEE Real-Time Technology and Applications Symposium*, 1997.

[31] Lori A. Clarke, Johnette Hassel, and Debra J. Richardson. A Close Look at Domain Testing. *IEEE Transactions of Software Engineering*, 8(4):380–390, 1982.

[32] Rance Cleaveland and Matthew Hennessy. Testing Equivalence as a Bisimulation Equivalence. *Formal Aspects of Computing*, 5:1–20, 1993.

[33] Rance Cleaveland and Amy E. Zwarico. A Theory of Testing for Real-Time. In *Proceedings, Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 110–119, Amsterdam, The Netherlands, 15–18 July 1991. IEEE Computer Society Press.

[34] Gregory Cooper, Lisa Cingiser DiPippo, Levon Esibov, Roman Ginis, Russel Johnston, Peter Kortman, Peter Krupp, John Mauer, Michael Squadrito, Bhavani Thuraisingham, Steven Wohlever, and Victor Fay Wolfe. Real-Time CORBA Development at MITRE, NRaD, Tri-Pacific and URI. In *IEEE Workshop on Middleware for Distributed Real-time Systems and Services*, pages 69–74. IEEE, December 1997. San Francisco, CA, USA.

[35] Conrado Daws and Stavros Tripakis. Model Checking of Real-Time Reachability Properties using Abstractions. In *Algorithms for the Construction and Analysis of Systems (TACAS'98)*, pages 313–329, 1998. LNCS 1055.

[36] Conrado Daws and Sergio Yovine. Reducing the Number of Clock Variables of Timed Automata. In *1996 IEEE Real-Time Systems Symposium, RTSS'96*, Washington, DC, USA, december 1996. IEEE Computer Society Press.

[37] René G. de Vries and Jan Tretmans. On the fly Conformance Testing using Spin. In *4th International Spin Workshop*, 1998. In Conjunction with IFIP FORTE/PSTV98.

[38] David L. Dill. Timing Assumptions and Verification of Finite-State Concurrent Systems. In *International Workshop on Automatic Verification Methods for Finite State Systems*, pages 197–212, Grenoble, France, June 1989. LNCS 407.

[39] Abdeslam En-Nouaary, Rachida Dssouli, and Ferhat Khendek. Timed Test Cases Generation Based on State Characterization Technique. In *19th IEEE Real-Time Systems Symposium (RTSS'98)*, pages 220–229, December 2–4 1998.

[40] W. Feng, U. Syyid, and J. W.-S. Liu. Providing for an Open, Real-Time CORBA. In *IEEE Workshop on Middleware for Distributed Real-time Systems and Services*, pages 75–80. IEEE, December 1997. San Francisco, CA, USA.

[41] Jean-Claude Fernandez, Claude Jard, Thierry Jéron, and César Viho. An Experiment in Automatic Generation of Test Suites for Protocols with Verification Technology. *Science of Computer Programming*, 29:123–146, 1997.

[42] Svend Frølund. *Coordinating Distributed Objects: An Actor-Based Approach to Synchronization.* MIT Press, 1996.

[43] Svend Frølund and Gul Agha. A Language Framework for Multi-Object Coordination. In O. Nierstrasz, editor, *European Conference on Object Oriented Programming (ECOOP '93)*, LNCS 707, pages 346–360, Kaiserslautern, Germany, July 1993. Springer-Verlag.

[44] Richard Gerber and Insup Lee. Communicating Shared Resources: A Model for Distributed Real-Time Systems. In *Real-Time Systems Symposium (RTSS'89)*, pages 68–78, Santa Monica, CA, USA, 1989. IEEE.

[45] Richard Gerber and Insup Lee. A Layered Approach to Automating the Verification of Real-Time Systems. *IEEE Transactions on Software Engineering*, 18(9):768–784, September 1992.

[46] Jens Chr. Godskesen. Fault Models for Embedded Systems (extended abstract). In *CHARME'99*, September 1999. Springer-Verlag LNCS 1703.

[47] Object Management Group. Realtime CORBA - A White Paper - Issue 1.0. Technical Report ORBOS/96-09-01, Object Management Group, December 1996.

[48] Per Brinch Hansen. Reproducible Testing of Monitors. *Software—Practice and Experience*, 8:712–729, 1978.

[49] Matthew Hennessy. *Algebraic Theory of Processes.* MIT Press Series in the Foundations of Computing. MIT Press, Cambridge, Massachusetts, 1988. ISBN 0-202-08171-7.

[50] Matthew Hennessy and Tim Regan. A Process Algebra for Timed Systems. *Journal of Information and Computing*, 117:221–239, 1994.

[51] Thomas A. Henzinger, Zohar Manna, and Amir Pnueli. Timed Transition Systems. In *Real Time: Theory in Practice, REX Workshop*

'91, volume LNCS 600 of *Lecture Notes in Computer Science, Springer-Verlag*, pages 226–251, 1992.

[52] Robert M. Hierons. Testing from a Z Specification. *Software Testing, Verification and Reliability*, 7:19–33, 1997.

[53] Gerard J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, New Jersey, U.S.A, 1991. ISBN 0-13-539925-4.

[54] Gerard J. Holzmann. *Design and Validation of Computer Protocols*, chapter 11. Prentice Hall, New Jersey, U.S.A, 1991. ISBN 0-13-539925-4.

[55] Hans-Martin Hörcher and Jan Peleska. Using Formal Specifications to Support Software Testing. *Software Quality Journal*, 7:309–327, 1995.

[56] Farnam Jahanian, Aloysius K. Mok, and Douglas A. Stuart. Formal Specification of Real-Time Systems. Technical Report UTCS-TR-88-25, University of Texas, 1988.

[57] Thierry Jéron and Pierre Morel. Test Generation Derived from Model-Checking. In *International Conference on Computer Aided Verification (CAV'99)*, July 7–10 1999. Italy.

[58] V. Kalogeraki, P.M. Melliar-Smith, and L.E. Moser. Soft Real-Time Resource Management in CORBA Distributed Systems. In *IEEE Workshop on Middleware for Distributed Real-time Systems and Services*, pages 46–51. IEEE, December 1997. San Francisco, CA, USA.

[59] Alain Kerbrat, Thierry Jéron, and Roland Groz. Automated Test Generation from SDL Specifications. In *Ninth SDL Forum*, 21-25 June 1999. Montral, Qubec, Canada.

[60] Don Kiely. Are Components the Future of Software. *IEEE Computer*, 32(2):10–11, February 1998.

[61] Brian Kirk, Libero Nigro, and Francesco Pupo. Using Real Time Constraints for Modularisation. In *Joint Modular Language Conference*, March 1997. Linz.

[62] Eleftherios Koustsofios and Stepen C. North. Drawing Graphs with dot. Technical Report `http://www.research.att.com/sw/tools/graphviz/dotguide.ps.gz`, AT&T Bell Laboratories, Murray Hill, NJ, U.S.A.

[63] Lawrence M. Krauss. *The Physics of Star Trek.* HarperCollins Publishers, New Yourk, U.S.A, 1995. ISBN 0-06-097710-8.

[64] Kim G. Larsen, Fredrik Larsson, Paul Petterson, and Wang Yi. Efficient Verification of Real-Time Systems: Compact Data Structures and State-Space Reduction. In *18th IEEE Real-Time Systems Symposium*, pages 14–24, 1997.

[65] Kim G. Larsen, Paul Pettersson, and Wang Yi. UppAal in a Nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1):134–152, 1997.

[66] Kim G. Larsen and Wang Yi. Time Abstracted Bisimulation: Implicit Specifications and Decidability. In *Mathematical Foundations of Programming Semantics (MFPS 9)*, volume LNCS 802. Springer Verlag, April 1993. New Orleans, U.S.A.

[67] Kim Guldstrand Larsen and Arne Skou. Bisimulation Through Probabilistic Testing. *Information and Computation*, 94(1), 1991.

[68] David Lee and Mihalis Yannakakis. Principles and Methods of Testing Finite State Machines—A Survey. *Proceedings of the IEEE*, 84(8):1090–1123, august 1996.

[69] Nancy G. Leveson and Clark S. Turner. An investigation of the Therac-25 Accidents. *IEEE Computer*, 26(7):18–41, July 1993.

[70] Harry R. Lewis and Christos H. Papadimitriou. *Elements of the Theory of Computation*. Prentice Hall, New Jersey, U.S.A, 1981. ISBN 0-13-273426-5.

[71] Gang Lou, Gregor v. Bochmann, and Alexandre Petrenko. Test Selection Based on Communicating Nondeterministic Finite-State Machines Using a Generalized Wp-Method. *IEEE Transactions on Software Engineering*, 20(2):140–162, february 1994.

[72] Dino Mandrioli, Sandro Morasca, and Angelo Morzenti. Generating Test Cases for Real-Time Systems from Logic Specifications. *ACM Transactions on Computer Systems*, 13(4):365–398, 1995.

[73] F. Michel, P. Azéma, and K. Drira. Selective Generation of Symmetrical Test Cases. In *IFIP TC6 9th International Workshop on Testing of Communication Systems (IWTCS'96*, pages 191–206, September 1996. Darmstadt, Germany.

[74] Robin Milner. *Communication and Concurrency*. Prentice Hall International (UK), 1989. 0-13-114984-9.

[75] R. De Nicola and M.C.B Hennessy. Testing Equivalences for Processes. *Theoretical Computer Science*, 34:83–133, 1984.

[76] Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Compiling Real-Time Specifications into Extended Automata. *IEEE Transactions on Software Engineering*, 18(9):805–816, September 1992.

[77] Brian Nielsen and Gul Agha. Towards Re-usable Real-Time Objects. *The Annals of Software Engineering*, 7, 1999. Special Issue on Real-Time Software Engineering.

[78] Jan Peleska. Formal Methods and the Development of Dependable Systems. Habilitationsschrift 9612, Institut für Informatik und Praktische Mathematik, Christian-Albrechts Universität, Kiel, 1996.

[79] Jan Peleska, Peter Amthor, Sabine Dick, Oliver Meyer, Michael Siegel, and Cornelia Zahlten. Testing Reactive Real-Time Systems. In *Material for the School – 5th International School and Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT'98*, 1998. Lyngby, Denmark.

[80] Jan Peleska and Bettina Buth. Formal Methods for the International Space Station ISS. In E.-R. Olderog and B. Steffen, editors, *Correct System Design*, pages 363–389, 1999. Springer LNCS 1710.

[81] Jan Peleska and Michael Siegel. Test Automation of Safety-Critical Reactive Systems. *South African Computer Journal*, 19:53–77, 1997.

[82] Jan Peleska and Cornelia Zahlten. Test Automation for Avionic Systems and Space Technology. In *GI Working Group on Test, Analysis and Verification of Software*, 1999. Munich, Extended Abstract.

[83] Paul Petterson. *Modelling and Verification of Real-Time Systems using Timed Automata: Theory and Practice.* PhD thesis, Uppsala University Department of Computer Science, 1999.

[84] Iain Phillips. Refusal Testing. *Theoretical Computer Science*, 50:241–284, 1987.

[85] Roger S. Pressman. *Software Engineering—A practioner's Approach.* McGraw-Hill Series in Software Engineering and Technology. McGraw-HILL, Inc., New York, second edition, 1987. ISBN 0-07-100232-4.

[86] Jens Ramskov. Ubrugelige Måledata fra Ørsted. *Ingeniøren*, (23), 1999.

[87] Pascal Raymond, Xavier Nicollin, Nicolas Halbwacs, and Daniel Weber. Automatic Testing of Reactive Systems. In *19th IEEE Real-Time Systems Symposium (RTSS'98)*, December 2–4 1998.

[88] Shangping Ren. *An Actor-Based Framework for Real-Time Coordination.* PhD thesis, Department Computer Science. University of Illinois at Urbana-Champaign, 1997. PhD. Thesis.

[89] Shangping Ren and Gul Agha. RT-Synchronizer: Language Support for Real-Time Specifications in Distributed Systems. *ACM Sigplan Notices*, 30(11), November 1995. Also in *ACM Sigplan 1995 Workshop on Languages, Compilers, and Tools for Real-Time Systems.*

[90] Shangping Ren and Gul Agha. A Modular Approach for Programming Embedded System. In Frits Vaandrager and Grzegorz Rozenberg, editors, *Lectures on Embedded Systems. European Educational Forum Lectures on Embedded Systems*, LNCS 1494, pages 170–207, Veldhoven, The Netherlands, November 1996. Springer-Verlag.

[91] Shangping Ren, Gul Agha, and Masahiko Saito. A Modular Approach for Programming Distributed Real-Time Systems. *Journal of Parallel and Distributed Computing*, 36(1):4–42, 1996.

[92] Annie Ressouche, Robert de Simone, Amar Bouali, and Valérie Roy. The FCTOOLS User Manual. Technical Report `ftp://ftp-sop.inria.fr/meije/verif/fc2.userman.ps`, INRIA Sophia Antipolis.

[93] Andrew W. Roscoe. *Model-Checking CSP.* Prentice-Hall, May 1994. ISBN 0-13-294844-3.

[94] Andrew W. Roscoe, Poul H.B. Gardiner, Michael H. Goldsmith, Jason R. Hulance, David M. Jackson, and J. Bryan Scattergood. Hierarchical Compression for Model-Checking CSP *or* How to Check 10^{20} Dining Philosophers for Deadlock. In Uffe H. Engberg, Kim G. Larsen, and Arne Skou, editors, *Proceedings of the Workshop on Tools and Algorithms for The Construction and Analysis of Systems, TACAS* (Aarhus, Denmark, 19–20 May, 1995), number NS-95-2 in Notes Series, pages 187–200, Department of Computer Science, University of Aarhus, May 1995. BRICS.

[95] John Rushby. Formal Methods: Instruments of Justification or Tools of Discovery. In *Nordic Seminar on Dependable Computing Systems (NSDCS'94)*, Lyngby, Denmark, August 1994. Tutorial.

[96] M. Saksena, P. Freedman, and P. Rodziewicz. Guidelines for Automated Implementation of Executable Object Oriented Models for Real-Time Embedded Control Software. In *18th IEEE Real-Time Systems Symposium*, pages 240–251. IEEE, December 1997.

[97] Ichiro Satoh and Mario Tokoro. Semantics for a Real-Time Object-Oriented Programming Language. In *Int. Conf. on Computer Languages*, pages 159–170, Toulouse, France, 1994. IEEE.

[98] Holger Schlingloff, Oliver Meyer, and Thomas Hülsing. Correctness Analysis of an Embedded Controller. In *Data Systems in Aerospace (DASIA99). ESA SP-447, Lisbon, Portugal*, pages 317–325, 1999.

[99] Douglas C. Schmidt, Rajeev Bector, and David L. Levine. An ORB Endsystem Architecture for Statically Scheduled Real-time Applications. In *IEEE Workshop on Middleware for Distributed Real-time Systems and Services*, pages 52–60. IEEE, December 1997. San Francisco, CA, USA.

[100] Douglas C. Schmidt and Mohamed E. Fayad. Lessons Learned Building Reusable OO Frameworks for Distributed Software. *Communications of the ACM*, 40(10):85–87, October 1997.

[101] Douglas C. Schmidt and Mohamed E. Fayad. Object-Oriented Application Frameworks. *Communications of the ACM*, 40(10):32–38, October 1997.

[102] Steve Schneider. An Operational Semantics for Timed CSP. *Information and Computation*, 116:193–213, 1995.

[103] Bran Selic, Garth Gullekson, and Paul T. Ward. *Real-time Object-Oriented Modeling.* Wiley Professional Computing. John Wiley & Sons, Inc., New York, 1994. ISBN 0-471-59917-4.

[104] Lui Sha, Ragunathan Rajkumar, and Shirish S. Sathaye. Generalized Rate-Monotonic Scheduling Theory: A Framework for Developing Real-Time Systems. *Proceedings of the IEEE*, 82(1):68–82, January 1994.

[105] J. Springintveld, F. Vaandrager, and P.R. D'Argenio. Testing Timed Automata. TR CTIT 97-17, University of Twente, 1997. To appear in *Theoretical Computer Science.*

[106] William Stallings. *Operating Systems—Internals and Design Principles, 3rd ed.* Prentice Hall, New Jersey, U.S.A, 1998. ISBN 0-13-887407-7.

[107] Q.M. Tan, A. Petrenko, and G. v. Bochmann. A Test Generation Tool for Specifications in the Form of State Machines. Technical Report IRO 1016, Department d'IRO, Université de Montreál, february 1996.

[108] Q.M. Tan, A. Petrenko, and G. v. Bochmann. Checking Experiments with Labeled Transition Systems for Trace Equivalence. In *IFIP 10th International Workshop on Testing of Communication Systems (IWTCS'97*, 1997. Korea.

[109] Andrew S. Tanenbaum. *Computer Networks.* Prentice Hall, New Jersey, U.S.A, 3ed edition, 1996. ISBN 0-13-349945-6.

[110] Richard N. Taylor, David L. Levine, and Cheryl D. Kelly. Structural Testing of Concurrent Programs. *IEEE Transactions on Software Engineering*, 18(3):206–215, March 1992.

[111] Jan Tretmans. *A Formal Approach to Conformance Testing.* PhD thesis, University of Twente, Haag, The Nederlands, 1992. ISBN 90-9005643-2.

[112] Jan Tretmans. Test Generation with Inputs, Outputs, and Quiescence. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96)*, pages 127–146, 1996. LNCS 1055.

[113] R.J. van Glabbeek. The Linear Time — Branching Time Spectrum. In *International Conference on Concurrency Theory (CONCUR '90)*, pages 278–297, December 1990. Madras, India, Also in LNCS 458.

[114] Volker Diekert, Paul Gastin, Antoine Petit. Removing epsilon-Transitions in Timed Automata. In *14th Annual Symposium on Theoretical Aspects of Computer Science, STACS 1997*, pages 583–594, Lübeck, Germany, February 27 - March 1 1997. Lecture Notes in Computer Science, Vol. 1200, Springer, 1997, ISBN 3-540-62616-6.

[115] Joachim Wegener and Matthias Groctmann. Verifying Timing Constraints of Real-Time Systems by Means of Evolutionary Testing. *Real-Time Systems*, (15):275–298, 1998.

[116] Lee J. White and Edward I. Cohen. A Domain Strategy for Computer Program Testing. *IEEE Transactions of Software Engineering*, 6(3):247–257, 1980.

[117] Mihalis Yannakakis and David Lee. An Efficient Algorithm for Minimizing Real-Time Transition Systems. *Formal Methods in System Design*, 11:113–136, 1997. Extended Abstract in Proceedings of Compute Aided Verification CAV'93, LNCS 697.

[118] Wang Yi and Bengt Jonsson. Decidability of Timed Language-Inclusion for Networks of Real-Time Communicating Sequential Processes. In *14th Conference on Foundations of Software Technology and Theoretical Computer Science*, December 1994. Madras, India, Also in LNCS 880.

[119] Wang Yi, Paul Pettersson, and Mats Daniels. Automatic Verification of Real-Time Communicating Systems by Constraint Solving. In *7th Int. Conf. on Formal Description Techniques*, pages 223–238, 1994. North-Holland.

[120] Sergio Yovine. Model Checking Timed Automata. In Frits Vaandrager and Grzegorz Rozenberg, editors, *Lectures on Embedded Systems. European Educational Forum School on Embedded Systems*, LNCS 1494, Veldhoven, The Netherlands, November 1996. Springer-Verlag.

[121] P. Zhou and J. Hooman. A Proof Theory for Asynchronously Communicating Real-Time Systems. In *Real-Time Systems Symposium (RTSS'92)*, pages 177–186, Phoenix, AZ, USA, 1992. IEEE.