Time-constrained Reactive Automata

A Novel Development Methodology for Embedded Real-time Systems

A thesis presented

by

Azer Bestavros

to

The Division of Applied Sciences

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

in the subject of

Computer Science

Harvard University

Cambridge, Massachusetts

August 1991

"Hear, my child, the instruction of thy father,
and forsake not the teaching of thy mother;
For they are a fair garland upon thy head,
and adorning pendants for thy neck."

Proverbs 1:8–9

To Mom and Dad.

Time-constrained Reactive Automata

A Novel Development Methodology for

Embedded Real-time Systems

Abstract

Embedded computing systems are characterized by the rigidity of their performance and reliability requirements, which are dictated by the critical nature of their missions and the demanding and often hostile environments with which they interact. Considering the vital role that such systems are playing and will continue to play in our world, it has become imperative that a rigorous and systematic treatment that recognizes their unique requirements be adopted. In this thesis we propose such a treatment based on the Time-constrained Reactive Automata (TRA) model – a novel formalism suitable for the specification, validation, verification, and implementation of embedded systems.

Previous studies in modeling real-time computing have focussed on adding the notion of time to formal modeling techniques of traditional systems without regard to physical realities of the modeled systems. The TRA model is a physically sound formalism. Among its salient features is a fundamental notion of space and time that restricts the expressiveness of the model in a way that allows the specification of only those systems that are potentially physically realizable. The TRA model is compositional and supports time, control, and computation non-determinism without violating the principles of causality and spontaneity.

Using the TRA model, an embedded system is viewed as a set of asynchronously interacting automata (TRAs), each representing an autonomous system entity. TRAs are input enabled; they interact by signaling events on their output channels and by reacting to events signaled on their input channels. The behavior of a TRA is governed by time-constrained causal relationships between computation-triggering events. The TRA model is compositional and allows time, control, and computation non-determinism. The TRA model allows the representation of both the external environment and the programmed system along with the available computational resources in a unique framework making it possible to prove safety and liveness properties and study transient and steady state performances of embedded real-time control systems. In particular, using the TRA formalism there is no conceptual distinction between a system and a property; both are specified as formal objects. This reduces the verification process to that of establishing correspondences – preservation and implementation – between such objects.

CLEOPATRA is a specification language based entirely on the TRA formalism. It features a C-like imperative syntax for the description of computation, which makes it easier to incorporate in real applications already using C; it is object-based, thus advocating modularity, reusability, and off-the-shelf hierarchical programming of embedded systems. *CLEOPATRA* is semantically sound. In particular, its objects can be transformed, mechanically and unambiguously, into formal TRA objects for verification purposes. We have developed a compiler that allows specifications written in *CLEOPATRA* to be executed in simulated time, thus providing a valuable tool for validation purposes.

We have used the TRA development methodology in the design, simulation, and analysis of various systems – specifically asynchronous digital circuits, sensori-motor activity management for autonomous systems, and intelligent controllers. Our experience has confirmed the suitability of this novel methodology for the specification, verification, and validation of embedded and time-critical applications. Its usefulness in the implementation of such systems, although not tackled in this thesis, is eminent.

Contents

Abstract **v**

List of Figures **x**

Acknowledgments **xii**

1 Introduction **1**

1.1 Embedded Systems 2

1.1.1 Aspects and Constraints 2

1.1.2 Development Requirements 4

1.2 Thesis Outline 5

2 A Survey of Related Research **6**

2.1 Formal Models and Verification Techniques 7

2.1.1 State-based Techniques 8

2.1.2 Process-Algebra-based Techniques 9

2.1.3 Logic-based Techniques 10

2.1.4 Petri-net-based Techniques 11

2.2 Specification Techniques 12

2.2.1 Requirement Specification Languages 12

2.2.2 Programming Languages 14

2.3 Development Support 17

2.3.1 Operating System Kernels 18

2.3.2 Scheduling Algorithms 19

2.4 Other Issues 20

3 The Time-constrained Reactive Automata Model **21**
3.1 Novelties 22
3.2 The TRA Model 23
3.2.1 Basic Definitions 24
3.2.2 The TRA Object 27
3.2.3 Sources of Non-determinism 31
3.3 Space and Time aspects of TRAs 32
3.3.1 The Conflict Relationship 32
3.3.2 Proper TRAs 33
3.3.3 TRA Control Components 33
3.4 The TRA Operational Semantics 36
3.4.1 TRA Intentions, Status, and Status Succession 36
3.4.2 TRA Executions, Schedules, and Behaviors 39
3.4.3 TRA Implementation 41
3.4.4 TRA Equivalence 42
3.5 Operations on TRAs 43
3.5.1 Hiding 43
3.5.2 Renaming 44
3.5.3 Composition 45

4 TRA-based Specification **50**
4.1 *CLEOPATRA*: A Specification Language 51
4.1.1 Classes and Objects 51
4.1.2 Time-constrained Event-driven Transaction 55
4.1.3 An Example 58
4.2 Relationship between *CLEOPATRA* and the TRA model 60
4.2.1 Soundness 60
4.2.2 Completeness 64

5 TRA-based Verification **66**
5.1 Modular Decomposition 67
5.2 Functional Decomposition 72
5.3 Hierarchical Decomposition 73

6 TRA-based Validation **79**

6.1 *CLEOPATRA*: A Simulation Language 80

6.1.1 Data Types 80

6.1.2 The `main` TRA-class 81

6.1.3 Object Instantiation 81

6.1.4 System-defined TRA-classes 83

6.1.5 Compatibility with C 85

6.1.6 Compilation and Execution 85

6.2 Simulation of Reactive Behaviors in *CLEOPATRA* 88

6.2.1 Servo Control Systems (Basic Behaviors) 88

6.2.2 Selective Control Systems (Subsuming Behaviors) 90

6.2.3 Teleo-selective Control Systems (Competing Behaviors) 92

6.2.4 Intelligent Control Systems (Intelligent Behaviors) 93

7 TRA-based Implementation **97**

7.1 *CLEOPATRA*: A Programming Language 98

7.2 TRA-based Development of Robotics Applications 99

8 Conclusion **103**

8.1 Summary 104

8.2 Directions for Future Research 105

Bibliography **107**

Index **115**

List of Figures

3.1 TRA objects and channels. 25
3.2 Signals, events, and actions. 25
3.3 Basic components of a TRA object. 27
3.4 A TRA time-constrained causal relationship. 29
3.5 State diagram of up/down counter. 30
3.6 TRA-specification of up/down counter. 30
3.7 An improper TRA specification of a counter. 34
3.8 A proper TRA specification of a counter. 35
3.9 The notion of a TRA status. 37

4.1 Partial Syntax of a TRA specification in $\mathcal{CLEOPATRA}$ 52
4.2 Specification of the class of integrators that use the trapezoidal rule. 53
4.3 $\mathcal{CLEOPATRA}$ specification of a ramp generator. 54
4.4 Time-constrained Event-driven Transaction (TET). 55
4.5 $\mathcal{CLEOPATRA}$ specification of the factorial computation. 57
4.6 $\mathcal{CLEOPATRA}$ specification of a finite length FIFO delay element. 58
4.7 $\mathcal{CLEOPATRA}$ specification of a 2-input synchronizer. 62
4.8 Formal TRA specification of a 2-input synchronizer. 63

5.1 $\mathcal{CLEOPATRA}$ specification of the installations $\mathcal{X}$ and $\mathcal{Y}$. 68
5.2 $\mathcal{CLEOPATRA}$ specification of the property $\mathcal{P}$. 71
5.3 TRA-specification of the property $\mathcal{P}$. 71
5.4 Ψ-mapping between specification and implementation. 75
5.5 TRA-specification of the installation $\mathcal{X}$. 78
5.6 TRA-specification of the composition $\mathcal{X} \times \mathcal{C}$. 78

6.1 The `main` TRA-class. 82
6.2 A stand-alone process control system. 83
6.3 Simulated behavior of an underdamped process control system. 84
6.4 The `fmonitor` system-defined TRA-class. 84
6.5 Compilation and simulation of *CLEOPATRA* specifications. 86
6.6 A typical *CLEOPATRA* compilation and execution session. 87
6.7 CMOS `nand` gate and a switching circuit approximation. 89
6.8 *CLEOPATRA* specification of the `nand` gate. 89
6.9 *CLEOPATRA* specification of the subsumption TRA. 91
6.10 Basic and emergent behaviors of Buggy in a typical simulation. 94
6.11 Interaction between behavioral planning and real-time control 95

7.1 Components of a *CLEOPATRA*-based implementation environment 99
7.2 Set-up for a sensori-motor activity coordination experiment 102

Acknowledgments

My first exposure to the difficulties involved in the development of embedded systems came in the summer of 1988 while working in the Robotics Laboratory of Harvard University on the implementation of an interface that allows the programmable control of an industrial robot arm from the Unix™-based environment of a Sun™ workstation. The interface worked and my summer project was over, but my quest for a scientific methodology for the development of embedded computing systems was just starting. Almost three summers later, I am hereby presenting my findings.

This work would have been impossible if it were not for the advice, attention, and encouragement of many people; I wish to thank them all. I also owe much to Harvard University for the intellectual wealth and the cultural diversity of its community. I feel privileged to belong to this unique institution.

In the first place, I am greatly indebted to my advisor Professor Thomas E. Cheatham, Jr. for his constant encouragement and continued support during the course of this work. I am grateful to him for he has taught me, among many other things, the art of being a researcher. I am really fortunate for having been under his tutelage. Also, I would like to express my gratitude to the other members of my thesis committee, Professor Ugo Gagliardi and Professor James Clark, who helped me in my research from start to finish with valuable counseling and helpful suggestions.

I am most thankful to the faculty, students, and staff of the Computer Science Department at Harvard University for giving me the opportunity to work in such a stimulating, and yet personal, environment. In particular, I want to thank Dan Stefanescu, Michael Kilian, and all members of the Languages and Systems group for their feedback and discussion of the many lengthy presentations of my work. My thanks are also due to members of Harvard's Robotics Laboratory, especially to Nicola Ferrier, George Thomas, and John Page, for their technical assistance and for putting up with my growling experiments. A number of people have generously helped me in preparing and presenting my work; I am grateful to all of them. I am particularly appreciative of the constructive comments of Yves Deville and César Galindo-Legaria on the final manuscripts of this thesis.

Finally, I wish to acknowledge the tuition and guidance of Professors Michael Rabin, Roger Brockett, Nancy Lynch, Harry Lewis, Meichun Hsu, and many other scholars at Harvard University, Massachusetts Institute of Technology, and Alexandria University.

This work was supported by DARPA N00039-88-C-0163

Chapter 1

Introduction

The use of computer systems to monitor and control real-time processes in industrial, medical, scientific, environmental, military, and other applications that are vital to our lives, continues to mushroom. The critical nature of these processes coupled with their inherent complexities, demand that a rigorous and systematic methodology be employed in their specification and implementation so as to guarantee a predictably safe operation. This thesis proposes such a treatment.

1.1 Embedded Systems

A computing system is *embedded* if it is explicitly viewed as being a component of a larger system whose primary purpose is to monitor and control an environment. The leaping advances in computing technologies that the last few decades have witnessed has resulted in an explosion in the extent and variety of such systems. This trend is likely to continue in the future.

1.1.1 Aspects and Constraints

Embedded systems are usually associated with critical applications, in which human lives or expensive machinery are at stake. Their missions are often long-lived and non-interruptible, making maintenance or reconfiguration difficult. Examples include command and control systems, nuclear reactors, industrial process-control plants, robotics, space shuttle and aircraft avionics, collision avoidance systems, automotive control, switching circuits and telephony systems, data-acquisition systems, and real-time databases, just to name a few.

Viewed simply, an embedded system has two parts: an *external interface* and a *programmed system*. The external interface consists of a number of devices such as sensors and actuators that interact with the environment. The programmed system collects information from the sensors and responds by producing actions to drive the actuators. The sustained demands of the environments in which such systems operate pose relatively rigid and urgent requirements on their performance. These requirements are usually stated as constraints on the real-time behavior of the programmed system. Wirth [Wirt77] singled out this processing-time dependency as the one aspect that differentiates embedded systems from other sequential and parallel systems. This led to a body of research on *real-time computing*, which, in many instances, was considered in isolation from other equally if not more important aspects of embedded systems. In particular, the critical nature of the missions associated with embedded systems poses stringent reliability requirements on their design. Furthermore, should these systems fail to meet their specified reliability or performance requirements, they should do so safely [Leve91].

In addition to the aforementioned performance and reliability requirements, the development of embedded systems is often governed by a number of other constraints. In particular, tasks in an embedded application often compete for limited resources, like processors and actuators, thus giving rise to *resource constraints*. They might have to concurrently execute in order to achieve a desirable effect, thus imposing *concurrency constraints*. They might be suspended, or aborted in favor of a higher priority task, thus abiding by *precedence constraints*. They might have to communicate and synchronize to insure the satisfaction of *consistency constraints*. They might have to execute on specific sites or use specific resources to achieve *fault-tolerance constraints*. Finally, and perhaps most importantly, it is often the case that the application itself might dictate *logistic constraints* pertaining to physical aspects such as placement and packaging, or mechanical and inertial properties such as stability, steady state errors, and communication delays.

The range of disciplines employed in developing the various components of an embedded application makes it extremely difficult to adopt an accurate and integrated view of the system in its entirety. This further complicates the process of specifying and verifying system-wide requirements. For example, in a simple sensori-motor robotic application [Clar91], algorithms from various disciplines like low-level imaging, active vision, tactile sensing, path planning, compliant motion control, and non-linear dynamics may be utilized [Fu87]. Not only are these disciplines very different with respect to their abstractions and programming styles, but they also differ greatly in their computational requirements, which range from single-board dedicated processors to massively parallel general-purpose computers.

Current embedded systems are expensive to build and their properties are usually verified with ad hoc techniques, or with expensive and extensive simulations [Stan88a]. Minor specification or implementation changes result in new rounds of testing and fixing. The often incomplete and evolving specifications of these systems further exasperates this problem. Schneider [Schn88] portrays the situation aptly by saying that “Unlike other engineering disciplines, our methods are not founded on science. Real-time systems are built one way or another because that was the way the ‘last one’ was built. And, since the ‘last one’ worked, we hope that the next one will”. This brute force approach is not likely to scale-up with future systems. A rigorous and systematic treatment of embedded systems that recognizes their unique requirements is imperative if we are to meet the needs and challenges of the future.

1.1.2 Development Requirements

Predictability – the ability to foretell that an implementation will not violate a set of specification requirements – is a crucial, highly desirable property of embedded time-critical systems. Therefore, the success of any embedded system development methodology is largely judged based on the degree with which such a methodology enhances and promotes the predictability of the developed system.

Validation is the process of determining whether customers' desires have been correctly specified. The complexity of embedded systems renders the specification of their desired functionalities and constraints very difficult. A complete and correct set of requirements is seldom known a priori. *Prototyping* an implementation is often the approach used for validation purposes. Such an approach, although useful for simpler and massively produced systems, becomes impractical for complex, one-of-a-kind systems. *Simulations* are likely to be used instead. When a potential implementation is prototyped or simulated, both the specification and the realization are tested, which makes the isolation of customer and implementor responsibilities difficult and sometimes impossible. To solve this problem, the validation process has to be completely independent from implementation decisions. This is only possible if the system specifications are executable, and therefore can be used directly to generate demonstrable behaviors. *Executable specifications* have the added advantage that they help debug the customer's requirements early in the development cycle, before any investment in implementation takes place.

Verification is the process of certifying that certain desired properties[1] are preserved in a given set of system specifications. *Formal verification* entails proving analytically that the desired properties follow from the given specifications. This requires that both the specifications and the properties to be certified be formally expressed. Due to the grandeur

[1]Safety (nothing bad will happen) or liveness (something good will happen) are examples of such properties.

and complexity of embedded systems, accurate mathematical representation is not always feasible. *Empirical verification*, relying on extensive testing of simulations and prototypes, has to be used instead.

As we hinted before, current approaches to the specification of an embedded system are notorious for their inaccuracy and incompleteness. This leads to frequent changes in the specification late in the development cycle. To be able to accommodate such changes gracefully, *modular* development methodologies, which support both functional and hierarchical decomposition, should be adopted to promote *reusability* and *adaptability*.

1.2 Thesis Outline

In chapter 2, we identify the various areas of research in embedded and real-time systems that have been addressed in the past few years and that need to be addressed in the future. In this respect, we single out the major research efforts in modeling and verification formalisms, specification and programming languages, and system development support.

In chapter 3, we present the backbone of our development methodology, namely the Time-constrained Reactive Automata (TRA) formalism. Following a brief overview of the guiding principles that motivated our choices, we formally present the basic components of the TRA model and its operational semantics. The remainder of the thesis is devoted to the various aspects of the TRA-based development of embedded real-time systems, namely specification, verification, validation, and implementation.

In chapter 4, we introduce *CLEOPATRA*, a TRA-based specification language. We establish the soundness of *CLEOPATRA* and characterize its expressiveness in relation with the TRA formalism. In chapter 5, we present three formal verification techniques for the TRA model based on modular, functional, and hierarchical decomposition of systems. In chapter 6, we introduce those ingredients of *CLEOPATRA* that allow it to be executable and, thus, suitable for validation purposes via simulation. In chapter 7, we discuss the potentials of *CLEOPATRA* to serve as a programming language for implementation purposes. We conclude in chapter 8 with a summary of contributions and future research directions.

Chapter 2

A Survey of Related Research

In the past few years, various aspects of embedded and real-time systems have been studied, namely formal models, specification techniques, verification methodologies, development tools, and operating systems. The absence of a unifying formal framework that addresses the aforementioned issues severely limits the usefulness of these studies.

Wirth classified computation into three categories: sequential, parallel, and processing-time dependent [Wirt77]. The difficulty of specification, implementation, and verification of systems increases as parallelism and processing-time dependencies, which are characteristics of embedded systems, are incorporated. In this chapter, we identify the various aspects of embedded and real-time systems that have been addressed in the past few years. In particular, we single out the major research efforts in the development of formal models, specification techniques, verification methodologies, development tools, and operating systems.

2.1 Formal Models and Verification Techniques

Time has always been an observable but uncontrollable phenomenon, and unless it becomes possible to travel through it, we will always have to abide by its laws in dealing with "real" problems. Previous studies in modeling real-time systems have focussed on adding the notion of time to formal modeling techniques of traditional systems, namely: logic-based, process-algebra-based, Petri-net-based, and state-based. This view of adding the *time dimension* to all what traditional computing systems research has deemed "good" is yet to be justified.

Verification entails establishing that a solution is correct by showing that it satisfies a set of desired properties. *Formal verification* techniques prove the correctness of a solution by using the rules of a proof system developed for an underlying formal model. *Empirical verification* techniques establish the correctness of a solution using simulation, prototyping, and testing. Despite their elegance, formal verification techniques are not practical for real-world embedded applications. In particular, their soundness depends on how accurate and realistic the adopted abstractions are. In most of the cases, their usefulness is limited to proving properties of specifications rather than implementations.

Properties of embedded systems are usually classified as being either *safety* properties or *timeliness* properties. Safety properties deal with the requirement that "nothing bad will happen", whereas timeliness properties deal with the requirement that "something will happen in due time". It is the timeliness of embedded systems that qualifies them for being real-time systems. Timeliness corresponds to *liveness* – the requirement that "something will eventually happen" – in non real-time systems. Besides liveness, timeliness properties subsume other properties, like fairness and finite progress, often considered in traditional systems. Formal verification of timeliness properties requires proving that specific timing constraints are met. This involves determining the time of completion of actions, which may depend on the pattern and timing of the external environment stimuli, and the availability and capacity of the computing resources.

2.1.1 State-based Techniques

Early attempts at expressing the requirements of real-time systems shared a common view of these systems as Finite State Machines in which a response at any instance is completely determined by the system's present state and its future stimuli [Alfo77, Zave82]. Dasarathy [Dasa85] added *timer alarms* to Finite State Machines to allow for the modeling of real-time telephony systems. A timer alarm is an artificial stimulus that is generated if a specified timing deadline is missed; it acts as an interrupt signaling the occurence of an exception. Lewis [Lewi89] extended finite state graphs with uncertain timing constraints that are expressed as lower and upper delay bounds. This model is used to interpret formulae of branching-time logic, and is the basis for the verification algorithms of timing properties presented in [Lewi90]. Alur, Courcoubetis, and Dill [Alur90] proposed the use of *Timed Büchi Automata* to model the behavior of finite-state asynchronous real-time systems. Timed Büchi Automata are Büchi automata [Bü60] augmented with a mechanism to express constant bounds on the timing delays between system events. They suggested associating each automaton with a finite set of *clocks*, which can be tested or set instantaneously with automaton transitions.

In standard state-based specification techniques, a system is allowed to refuse unspecified inputs. Such specifications, therefore, impose restrictions on what the environment can and/or cannot do. While appropriate for protocol/interface specification, such a methodology seems unrealistic for an embedded systems, where no assumptions can be made about the behavior of the external environment. To avoid this undesirable property, Lynch [Lync88b] proposed the Input-Output Automata (IOA) model in which *inputs* actions are distinguished from *local* actions in that they are always enabled; a transition is defined for any input action and for every state of the automaton. The IOA model was used to develop proof techniques for the study of discrete event systems [Lync88a, Lync89a]. In [Best88a, Best90b] we proposed the Input-Output Timed Automata (IOTA) as an extension to Lynch's IOA model.[1] The IOTA model allows the specification of lower and upper bounds on the delay between the enabling of a locally-controlled action and its firing. Specification and simulation languages, proof techniques, and lower/upper bounds for a number of problems using this and other timed extensions of the IOA model were reported in [Best90a, Lync89c, Lync89b].

State proliferation is a property often attributed to state-based specification and verification techniques. As a remedy, Harel [Hare87] proposed a purely graphic formalism, called *Statecharts*, to reduce the number of states by introducing the multiple active-state notion. Later, Jahanian and Mok [Jaha88] introduced *Modecharts* as a compact and structured way of representing real-time systems. Although similar in some ways to Harel's Statecharts, Modecharts are specifically tailored to representing time-critical systems. The semantics of Modecharts is given in the Real-Time Logic of [Jaha86].

2.1.2 Process-Algebra-based Techniques

Several attempts have been made to extend traditional process-algebra techniques [Henn88] to represent time. The work of Reed and Roscoe [Reed88], extending Hoare's Communicating Sequential Processes (CSP) model [Hoar85], and the work of Baeten and Bergstra [Baet91c], extending Bergstra and Klop's Algebra of Communicating Processes (ACP) [Berg84] are two such examples.

In [Gerb89b], Gerber, Lee, and Zwarico suggest using the *Timed Acceptances Model* to capture the temporal constraints of concurrent programs. Their model, which they use to prove correctness properties of real-time programs, consists of a CSP-based language, a partially ordered semantic model, and an axiom system. Similar efforts have been reported in [Gosw88] using a simpler semantic domain. In an effort to bridge the gap between computational models and implementation environments, Lee, Gerber, and Davidson proposed

[1]Similar extensions to Lynch's IOA model were reported independently by Tuttle, Modugno, and Merritt in [Tutt88].

the *Communicating Shared Resources* (CSR) model [Gerb89a, Gerb90]. The CSR model is synchronous. It allows processes to be assigned to resources and execute thereon in an interleaved fashion according to their priorities. In order to allow for formal verification capabilities, they developed a Calculus for CSR (CCSR). CSR specifications can be translated into the CCSR formalism for verification using syntactic manipulations [Lee91].

A particularly interesting work is that of Baeten and Bergstra [Baet91b], in which their real time process algebra (ACP_ρ) [Baet91c] is extended into a real space-time process algebra ($\mathrm{ACP}_{\delta\rho}$), where processes are described using both space and time coordinates. This work is a first step toward tackling some of the concerns addressed in this thesis, namely *physical correctness*. Two versions of $\mathrm{ACP}_{\delta\rho}$ are developed, namely *classical* and *relativistic*. In [Baet91a], the classical version is used to study asynchronous communication in such a way that the motion of processes can be taken into account.

2.1.3 Logic-based Techniques

Temporal logics are appropriate for the description of the temporal properties of systems. In [Pnue77], Pnueli advocated the use of temporal logic formalisms for the behavioral specification of concurrent systems. He described a *time hierarchy of specifications* relating the occurence of time in formulae of a system to the expressive power of that system. Many interesting properties – like safety, liveness, and precedence – can be proved using such formalisms [Bern81, Mann82]. For example, Bochmann [Boch82] used temporal logic to specify and verify properties of an arbiter. Along the same lines, Clarke et al. [Mish83, Clar83] proposed the use of temporal logic in the automatic verification of asynchronous circuits. Moszkowski [Mosz85] defined a temporal logic to reason about hardware at the circuit level. Jahanian and Mok [Jaha86] proposed a first-order Real-Time Logic (RTL) to aid in the safety analysis of timing properties of real-time systems. Their model does not have modal operators to deal with time; instead, time is captured by a function that time-stamps events. Time constraints are expressed as first order assertions on these functions. Later, they used their logic in conjunction with Modechart specifications [Jaha88]. Allen [Alle81, Alle83, Alle84] proposed an interval temporal logic that is based upon time intervals rather than time points; it axiomatizes and uses seven relationships, with inverses, that can hold between two time intervals. This approach enables reasoning about non-instantaneous actions, for example in hardware specification [Wils89, Wils90] and in plan generation [Alle86].

A critical deficiency of temporal logics is their inability to express causal relationships between the various events in a system. Recently, Borriello and Amon [Borr90] addressed that problem by proposing a model for the executable specification of timing behavior that is based on a restricted version of the full first-order predicate calculus and which utilizes event ancestry[2] for the representation of complex timing relationships.

2.1.4 Petri-net-based Techniques

Petri-nets are attractive candidates for the specification of real-time systems. In particular, they offer an expressive technique for the representation of data dependencies and causality. There have been several proposals for extending the standard Petri-net model to include time. Ramchandani [Ramc74] proposed associating computational delays with transitions. Merlin and Farber [Merl74, Merl76] suggested the use of minimum and maximum bounds on uncertain transitional delays. Associating delays with Petri-net's transitions violates the instantaneous firing feature of the basic Petri-net model. This was remedied in the work of Sifakis [Sifa77] and, later, in the work of Coolahan and Roussopoulos [Cool83], by associating computational delays with places rather than transitions. Razouk [Razo83] proposed the use of both enabling and firing times; tokens are absorbed from input places after the enabling time has elapsed and do not reappear in the output places until after the firing time has elapsed. Ghezzi et al. [Ghez89] proposed a model where tokens are time-stamped environments; time constraints are associated with transitions and are modeled as predicates on the input tokens.

Timed Petri-net models have been used in studying various aspects of real-time systems – requirement specification [Cool83], performance evaluation [Holl87], and safety analysis [Leve87], to name a few.

[2]a weak notion of causality

2.2 Specification Techniques

The usual approach for specifying computing systems behavior is to enumerate the actions that a system participates in [Henn88, Lync89a]. Time is only perceived through the partial or total ordering of these actions. Such an ordering may be determined, not by the time in which actions were taken, but according to other considerations such as consistency [Eswa76] and serializability [Papa79, Yann84]. This artificial reordering of actions is only possible in applications where assumptions about the outside world can be made and enforced.[3] For embedded systems, such assumptions cannot be accommodated; time, as viewed by the environment in which a program executes, is a significant factor. Thus, the main challenge in the specification of real-time systems is how to incorporate the notion of time – how to extend programming notations to allow programmers to specify computations that are both *dependent* and *constrained* by time.

Complexity is another consideration in the specification of embedded systems. To manage a large and complex system, it is a good practice to hierarchically decompose it so that details be hidden from the higher levels of abstraction and exposed at the lower ones. This methodology allows implementors to reason about and establish the correctness of subsystems at each level independently. To deduce properties of the whole system from properties of its parts and the way these parts are combined, we must characterize a way to *compose* the real-time properties of parts to synthesize them for the whole. This might be subtle because these parts interact in ways that depend on resource and time constraints.

2.2.1 Requirement Specification Languages

Requirement specifications act as a contract between the customer and implementor of a system. There are two approaches for requirement specifications. In the conventional approach, systems are treated as black boxes; only the required characteristics of their external behavior are described – usually partially and informally. In the operational approach [Zave82], requirements are specified by formulating a system – using implementation-independent

[3]For instance, by undoing actions and aborting transactions in a database system

structures – that would generate the desired behaviors. In adopting one of these two approaches, one has to take into account several considerations, namely validation, verification, automation, maintenance, and management [Zave84]. We argue that, for embedded systems, the operational approach is appropriate. Its executable nature facilitates the validation process; its formality promises greater verification potentials; its modularity makes the evolution of customers' requirements easier to manage and encourages automation by advocating reusability and step-wise refinement.

The PAISLey language and environment [Zave84, Zave86, Zave88] were crafted with the operational requirements specification approach in mind. A system is described by a set of asynchronous processes, where each process has a state and goes through a sequence of discrete state changes (*process steps*). The computations occurring during these steps are specified using a functional notation to represent mappings. An upper bound, lower bound, or distribution of possible computational delays can be attached to any mapping. Only non-recursive time-constrained mappings are allowed. Special mappings called *exchange functions* are used to support interprocess interactions.[4] An exchange function is evaluated – even if its value is not needed – to produce global side-effects of synchronization and communication. The PAISLey environment provides tools for the execution of potentially incomplete specifications and testing for inconsistencies. Although designed for embedded applications, PAISLey fails in many respects. Time is added as an afterthought, time-constraints cannot be state-dependent, the notion of causality is non-existent, communication is blocking, broadcasting is not allowed, the use of exchange functions defeats the referential transparency property, and the functional flavor of PAISLey is not appropriate for the object-oriented/procedural nature of embedded systems.

ENCOMPASS is an environment aimed to support the incremental construction of Ada programs using executable specifications and formal techniques [Terw87a, Terw88]. It provides support for various aspects of software development including, specification, prototyping, testing, formal verification, documentation, configuration control and project management. In ENCOMPASS, software can be specified using PLEASE [Terw87c, Terw87b] an Ada-based executable specification language which can be automatically translated into

[4]Exchange functions are very similar to the CSP input and output primitives [Hoar85].

Prolog. In ENCOMPASS, software components are first specified using a combination of conventional programming languages and predicate logic. These abstract components are then incrementally refined into components in an implementation language. Each refinement is verified before another one is applied which guarantees that the final components satisfy the original specification. PLEASE allows a procedure or function to be specified with pre- and post-conditions written using Horn clauses. PLEASE specifications may be used in proofs of correctness. They may also be transformed into prototypes which use Prolog to execute pre- and post-conditions. ENCOMPASS and PLEASE were designed with the "software engineering" problem in mind. They do not support any notion of time, distribution or parallelism.

RT-ASLAN is a state-based formal language for specifying real-time systems at different levels of abstractions with the motivation of verifying them [Auer86]. A real-time system is viewed as a set of processes communicating via an *interface process*. Process' transitions can be either periodic or non-periodic. Time is maintained by a process that increments a time variable after each *tick* transition. Assertions written in first-order predicate logic can be attached to RT-ASLAN specifications. Assertions denote either *invariants* or *constraints*. Verification is done using a state-based inductive approach with the tick transition as the inductive step. In addition to its non-realistic communication and time management assumptions, a number of other simplifying assumptions are made in RT-ASLAN.[5]

2.2.2 Programming Languages

Until recently, most of the time-critical parts of an embedded application were (and are still being) implemented in low-level assembly or machine languages. This is primarily caused by a common misconception that "real-time computing is equivalent to fast computing" [Stan88a]. The objective of real-time computing is to meet the specified timing requirements. A faster computer makes it easier to meet these requirements, but does not guarantee it. The most important property of real-time programming languages is, thus, predictability.

[5]For example, each process is assumed to run on a dedicated processor, thus systems that can be specified using RT-ASLAN are neither process nor resource restricted.

With few exceptions, most of the real-time programming languages developed for embedded applications failed to meet that single property. For example, Ada is designed for embedded time-critical applications and is intended to support static priority scheduling of tasks. However, the definition of Ada tasking allows a high-priority task to wait for a low-priority task for an unpredictable duration. Ada and Modula-2 are examples of early attempts at developing general-purpose real-time programming languages. Logically correct programs are written using mechanisms such as coroutines, processes, priorities, interrupts, and exception handling to control the execution behavior. Knowledge of the runtime environment is required to tailor the program to meet timing specifications, which makes the program sensitive to hardware characteristics and system configuration.

Other factors – besides efficient coding and hardware characteristics – determine predictability. Implementation languages should be expressive enough to prescribe complex timing constraints. Current real-time programming languages provide little (if any) support for expressing time constraints. This state of affairs is very well pronounced in Berry et al.'s heavily quoted statement [Berr83]: "...paradoxically, one can verify that the current so-called 'real-time programming languages' do not provide any explicit means of correctly expressing time constraints. A fortiori, they provide no insurance that the constraints would be respected when executing the program." For example, in a language like Ada where only lower bounds on time delays can be expressed, there is no way upper bounds can be asserted.

Esterel [Berr85] represents the first attempt at permitting direct expression of timing requirements in programs. Programmers are allowed to specify deadlines for procedural invocations, leaving for the runtime system the responsibility to ensure their satisfaction. In addition to the timing requirements, programmers are allowed to specify exception handlers to be invoked if the specified requirements cannot be honored at run-time. The lack of compile-time analysis in Esterel, however, means that predictability, in the strong sense of completing without exception, is lost.

One way to insure predictability is to restrict expressiveness. This is the approach taken by Real-Time Euclid [Klig86]. The language definition forces every use of its constructs to be both time- and space-bounded, thus avoiding many of the dynamic aspects found in languages designed for programming traditional systems – recursion for example.

Time bounds and time-out exception handlers have to be specified for unbounded loops, wait statements, and device requests. Real-Time Euclid programs can, thus, be analyzed for guaranteeing schedulability given a specific hardware organization.

Another approach to guarantee that stringent timing constraints are always met is to sacrifice accuracy for predictability. This approach forms the basis for the body of research on *imprecise computations* [Lin87, Liu91]. The imprecise computation techniques prevents timing faults and achieves graceful degradation by producing approximate results of acceptable quality whenever exact results cannot be produced in time. The Flex language [Lin88, Lin91] is intended for systems where the this methodology is applicable. Three main techniques are used to specify time-constrained computations in Flex. The *milestone method* is appropriate for monotone time-critical tasks. A task is monotone if the quality of its intermediate result does not decrease as it executes longer. Flex provides constructs for the specification of intermediate result variables and error indicators. Should the task terminate prematurely due to a hard time constraint, the latest recorded intermediate results and error indicators are readily available. Another technique for trading off quality for time is the *sieve method* in which, if needs be, computation steps can be skipped to save time. In applications where the milestone and sieve methods are not applicable, the multiple version method is used. Using this approach, programmers specify two (or more) versions for each time-critical task. At run-time, the appropriate version is chosen based on the available time to produce a result. The imprecise computations approach warrants more scheduling flexibility in order to meet deadlines. In [Liu87, Chun90, Shih91] various algorithms for scheduling imprecise computations are presented.

The real-time programming languages we discussed thus far are all imperative. In [Faus86], Faustini and Lewis show how to extend Lucid, an equational dataflow language, for real-time purposes. In Lucid, programmers think in terms of streams and filters. A filter is used to construct one *output stream* out of a number of *input streams*, each with known properties. A Lucid program is, thus, a set of equations modeling a dataflow graph. Time is incorporated in a Lucid program by associating a stream of *time windows* with each stream of data values. Attaching time windows to input and output streams can be viewed as imposing timing constraints on their generation. Another similar language is LUSTRE [Caps87], a synchronous data-flow language aimed for programming real-time systems. LUSTRE is primarily designed for mathematically describable systems.[6] A program is a system of time-dependent equations representing *invariant assertions* that hold at each point in time. In LUSTRE there is no notion of execution, control or sequentiality. Only discrete systems are considered. Thus, time is projected onto the set of naturals and variables are infinite sequences of values. The equational semantics of Lucid and LUSTRE is much too simple to be practically usable in complex systems. In particular, some tasks cannot be fully described using systems of equations. A solution suggested in [Caps87] is to allow a LUSTRE program to call external functions written in a host language (namely C). To preserve the equational semantics, these external functions have to compute in zero time and have to produce no side-effects. Both of these assumptions are unacceptable for embedded real-time applications.

Computations in embedded systems are likely to be not only *time-constrained* but also *time-dependent*. ARCTIC [Rubi87] is an example for a language for describing the behavior of time-dependent concurrent activities. The fundamental idea in ARCTIC is that variables, and behaviors in general, can be described using functions of time. In ARCTIC, timing is explicitly indicated and is not a consequence of sequential execution. ARCTIC provides a set of tools to describe continuous as well as discrete signals. It has been used in the production of computer music and other digital audio sounds.

2.3 Development Support

The increasing complexity of embedded real-time systems dictates that powerful tools be available to aid in their design, implementation, and support. In particular, programming environments must provide powerful tools for testing, debugging, and simulating the operation of real-time programs. Also, they must facilitate the reuse, adaptation and tailoring of real-time software modules. In the past, these activities were done mostly in an ad hoc manner.

[6]For example, automatic control and signal processing applications.

2.3.1 Operating System Kernels

Operating systems play a key role in managing system resources so that programmers can focus on the application specific problems rather than the underlying system issues. Typically, real-time operating systems will have to allocate resources, keep track of deadlines and raise exceptions in case they are not met. In embedded and real-time systems, however, the operating system and the application are tightly intertwined and it is not clear how they can be decoupled. This represents a dilemma and, thus, a challenge; how to provide high level abstractions for programmers and yet meet performance requirements which are fundamentally dependent on the implementation and the environment. Abstractions, like processes, fairness, and finite progress, although useful in connection with conventional operating systems, are not necessarily adequate for time-critical applications [Schn88]. Current operating systems offer no solutions for the aforementioned dilemma; they are inadequate and must evolve to cope with the demands of real-time programmers.

Most of the existing real-time kernels are simply stripped down, optimized versions of conventional timesharing operating systems. VRTX [Read86], VxWorks [Wind89], and Lynx [Baue90] are classic examples. They promote a hardware independent architecture that is independent from the file system and the I/O system. Their prominent features include fast context switching, efficient interrupt handling, fast data-acquisition, real-time clock support, user-defined watchdog timers and interrupts, and priority scheduling. In an effort to provide a basis for evaluating such operating systems, POSIX.4 and an extension thereof have been proposed as IEEE standards [IEEE90]. Compliance with the POSIX standards is expected to rapidly become mandatory for commercial systems [Gall91].

REX [Bake86] promotes a different kernel structure. It introduces the notion of an *executive* – a software layer that runs on top of an operating system and which is responsible for scheduling and storage management. An executive acts as an interface between applications and the lower level operating system functions such as interrupt handling. CHAOS [Schw87] and AT&T's NRTX [Cox88] are similar kernels in that they offer an object-based view of embedded systems. Such a view promises significant improvement in modularity, reconfigurability, and maintainability. Both systems are aimed at robotics applications.

CHAOS evolved from an earlier kernel called GEM. It provides programmers with a view of the system as a set of interacting objects. NRTX is a real-time executive derived from UNIX™. Along with C^{++}, it offers a programming environment for the development of software for embedded systems. The Spring kernel [Stan87, Stan89] is built around the relatively new principle of *segmentation*, in which resources are divided into units to be manipulated by the various parts of the kernel – the scheduler, for example – in such a manner as to provide predictability with respect to timing constraints [Stan88b].

An additional responsibility for real-time operating systems is the management of information about the real-world and/or any active real-time tasks. This information should be viewed as a shared resource that multiple processes (including the operating system itself) might want to access (read or update) concurrently. This access, however, has to be regulated to insure some level of consistency and recency. In real-time systems, a significant portion of the data acquired from external interfaces is highly perishable in the sense that it has value to the mission only if used quickly. To satisfy timing requirements, the degree of concurrency must be increased through some kind of interaction between concurrency control protocols and real-time scheduling algorithms. It is not clear whether the classical theory of concurrency control (the serializability theory) [Papa79, Yann84] is appropriate for embedded systems. We believe it is not because of the limitation in concurrency allowed by serializable concurrent executions.

2.3.2 Scheduling Algorithms

The scheduling problem is that of allocating the available limited resources in a way that guarantees the satisfaction of the specified timing constraints. This can be done either by the programmer or by the run-time system. These two choices represent the extremes of a continuum. Scheduling for real-time systems is very different from scheduling problems considered in other areas where the goal is to find an optimal *static* scheduling policy that would minimize the response time for a given set of tasks. In real-time systems, the major goal is to schedule as many jobs as possible, subject to meeting deadlines. This does not necessarily mean minimizing response times. In addition, real-time systems are highly dynamic, thus requiring adaptive scheduling algorithms.

The interaction between verification and scheduling in real-time systems is subtle. This is basically due to the fact that scheduling affects the timing properties of programs and it is these properties that should be verified. On the one hand, one might think of the verification process as one in which, given the problem specification and the available resources, it is required to show the *existence* of at least one schedule that satisfies the specifications. In this case, the job of the scheduler is to find such a schedule. On the other hand, we might assume that the scheduling policy is known and thus verifying the correctness of a program entails showing that the *composition* of the program, scheduling policy and available resources meets the given specifications. This latter approach is both appealing and realistic. It is appealing because it allows the exposure of the available resources to the verification process, thus making it possible to provide clues about the minimum required resources and to compare different design alternatives. It is realistic because in almost all real-time application, the scheduling policy is usually predetermined.[7] The challenge in adopting this approach, though, is the need to represent programs, schedulers and available resources in a unified framework.

2.4 Other Issues

Embedded and real-time computing is a wide open research area for intellectually challenging computer science problems. There are a number of aspects and research areas of real-time systems that we have not considered in our review as they are not directly related to our work. These include programming environments, databases, artificial intelligence, general and special purpose architectures, communication protocols, fault-tolerance, testing, and safety analysis. An overview of these research areas and others can be found in [Stan88b, Stan92, Burn90, Tilb91a, Tilb91b].

[7]Usually based on some priority scheme.

Chapter 3

The Time-constrained Reactive Automata Model

Using the TRA model, an embedded system is viewed as a set of asynchronously interacting automata (TRAs), each representing an autonomous system entity. TRAs are input enabled; they interact asynchronously by signaling events on their output channels and by responding to events signaled on their input channels. The behavior of a TRA is governed by time-constrained causal relationships between computation-triggering events. The TRA model is compositional and allows time, control, and computation nondeterminism. Among its salient features, the TRA *model allows the specification of only those systems that are potentially physically realizable. In that respect, it abides by the causality and spontaneity principles.*

The TRA model [Best91b] has evolved from our work in [Best90b] extending Lynch's IOA model [Lync88b, Lync88a] to suit embedded and time-constrained computation.

3.1 Novelties

Previous studies in modeling real-time computing have focussed on adding the notion of time without regard to physical properties of the modeled systems. This makes it possible to specify systems that do not abide by principles like causality and spontaneity. Our work remedies such situations by dealing not only with the notion of time, but also with the notion of space. Events occur at uniquely identifiable points in time as well as in space. Events occurring at the same time and place are undistinguishable. The payoff for the dual treatment of space and time is manifold. For example, requirement specifications become more expressive since they can constrain the time as well as the space coordinates of system events. Also, mappings between various levels of abstractions for compilation and verification purposes become more robust as the formalism becomes more structured.

The TRA model differs from others in that it does not allow the specification of systems that are not *reactive*. A system is reactive if it cannot block the occurence of events not under its control. This property is crucial for accurate and realistic modeling of embedded and real-time systems. A sufficient condition for reactivity is the *input enabling* property proposed in [Lync88b]. The TRA model is input enabled. It distinguishes clearly between environment-controlled actions, which cannot be restricted or constrained, and locally-controlled actions, which can be scheduled and disabled.

Among state-based models, the TRA formalism is unique in that it admits the causal nature of physical processes. The *causality* of the TRA model follows the standard definition of causality for non-deterministic systems. A system is *causal* if given two inputs that are identical up to any given point in time, there exist outputs (for the respective inputs) that are also identical up to the same point in time. The TRA model enforces causality by requiring that any locally-controlled actions be produced only as a *result* of an earlier *cause*. In our work, a clear distinction is made between causality and dependency. An event occurs as a result of exactly one earlier event but may depend on many others as reflected in the state of the system. This spares our formalism from dealing with clairvoyant and capricious behaviors [Stua91].

Spontaneity is a notion closely related to causality.[1] A system is *spontaneous* if its output actions at any given point in time t cannot depend on actions occuring at or after time t. In particular, if an output occurs simultaneously with (say) an input transition, the same output could have been produced without the simultaneous input transition [Sree90]. Simultaneity is, thus, a mere coincidence; the output event could have occured spontaneously even if the input transition was delayed. The TRA model enforces spontaneity by requiring that simultaneously occuring events be independent; time has to *necessarily* advance to observe dependencies.

The TRA model distinguishes between two notions of time: *real* and *perceived*. Real time cannot be measured by any single process in a given system; it is only observable by the environment. Perceived time, on the other hand, can be specified using uncertain time delays. The TRA model, therefore, does not provide for (or allow the specification of) any *global* or *perfect clocks*. As a consequence, the only measure of time available for system processes has to be relative to *imperfect*, *local clocks*. This distinction between real time and perceived time is important when dealing with embedded applications where time properties are stated with respect to real time, but have to be preserved relying on perceived time.

3.2 The TRA Model

An embedded system is viewed as a set of asynchronously interacting Mealy [Hopc79] automata (TRAs), each representing an autonomous system entity. TRAs are input enabled; they communicate by signaling events on their output channels and by reacting to events signaled on their input channels. The behavior of a TRA is governed by time-constrained causal relationships between computation-triggering events. The TRA model is compositional and allows time, control, and computation non-determinism. In this section, we formally define the TRA model.

[1]Actually both spontaneity and causality are directly related to the past and future light cones of an event in space-time [Hawk88].

3.2.1 Basic Definitions

We adopt a continuous model of time similar to that used in [Alur90, Lewi90], where time is considered a measurable, continuous, infinitely divisible quantity. We represent any point in time by a nonnegative real $t \in \Re$. Time intervals are defined by specifying their end-points which are drawn from the set of nonnegative rationals $\mathcal{Q} \subset \Re$. A time interval is viewed as a traditional set over nonnegative real numbers. It can be an empty set, in which case it is denoted by ε, it can be a singleton set, in which case it is denoted by the $[t,t]$, $t \in \mathcal{Q}$, or else it can be an infinite (dense) set, in which case it is expressed in one of the following formats $[t_l, t_u]$, $(t_l, t_u]$, $[t_l, t_u)$, or (t_l, t_u), denoting the closed, right-closed, left-closed, and open time intervals, respectively, where $t_l, t_u \in \mathcal{Q}$ and $t_l < t_u$. We use $|[t_l, t_u]|$ to denote a dense time interval in any one of these formats. The set of all dense time intervals is denoted by $\mathcal{D}$. Throughout this thesis, we use the symbol δ to denote a time interval and the symbol Δ to denote a set of such time intervals.

Let $\delta_i = |[t_a, t_b]|$ and $\delta_j = |[t_c, t_d]|$ be two dense intervals. We say that δ_i contains δ_j if $\delta_j \subseteq \delta_i$. The union, intersection, and difference of δ_i and δ_j are denoted by $\delta_i \cup \delta_j$, $\delta_i \cap \delta_j$, and $\delta_i - \delta_j$, respectively.[2] We define the sum of δ_i and δ_j to be the new time interval $\delta_k = \delta_i + \delta_j$ whose end-points are obtained by adding the corresponding end-points of δ_i and δ_j, namely $\delta_k = |[t_a + t_c, t_b + t_d]|$. As a special case, we define the time interval δ_k obtained by shifting the time interval δ_i by τ to be $\delta_k = \delta_i + \tau = \delta_i + [\tau, \tau] = |[t_a + \tau, t_b + \tau]|$.[2]

A real-time system is viewed as a set of interacting automata called TRAs (Time-constrained Reactive Automata). TRAs communicate with each other through *channels* (see Figure 3.1). A channel is an abstraction for an *ideal* unidirectional communication. The information that a channel carries is called a *signal*, which consists of a sequence of *events*. An event, denoted by $\langle \pi : t \rangle$, underscores the occurence of an *action* π at a specific point in time t.

To illustrate the notions of actions, events, and signals (see Figure 3.2), consider the channel `MOVE` of some TRA. Let `North`, `South`, `East`, and `West` be the possible values that can be signaled on `MOVE`. `MOVE(East)` is, therefore, a possible action of the TRA. The

[2]The format of the resulting set (closed, semi-closed, or open) depends on the formats of δ_i and δ_j.

instantiation of `MOVE(East)` at time t_1 denotes the occurence of an event $\langle$`MOVE(East)`$: t_1\rangle$. Furthermore, the sequence of events $\langle$`MOVE(East)`$: t_1\rangle\langle$`MOVE(North)`$: t_2\rangle\langle$`MOVE(South)`$: t_3\rangle$ ...*etc.* constitutes a signal. Events occuring on different channels can be simultaneous. We use $\langle \pi_1, \pi_2, \ldots, \pi_m : t\rangle$ to denote the occurence of the set of simultaneous events $\langle \pi_1 : t\rangle$, $\langle \pi_2 : t\rangle$, ..., $\langle \pi_m : t\rangle$.

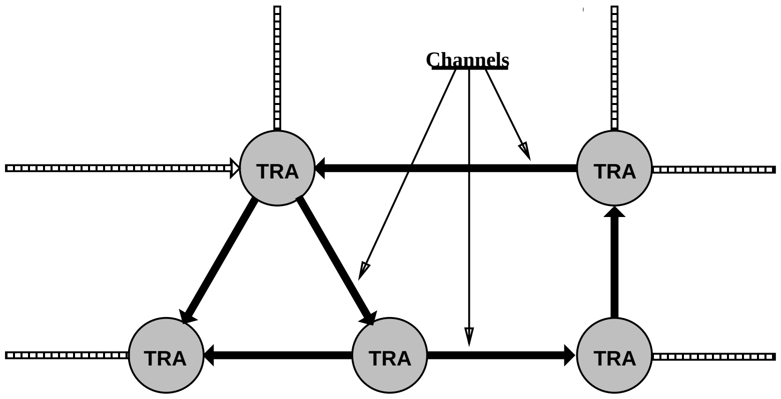
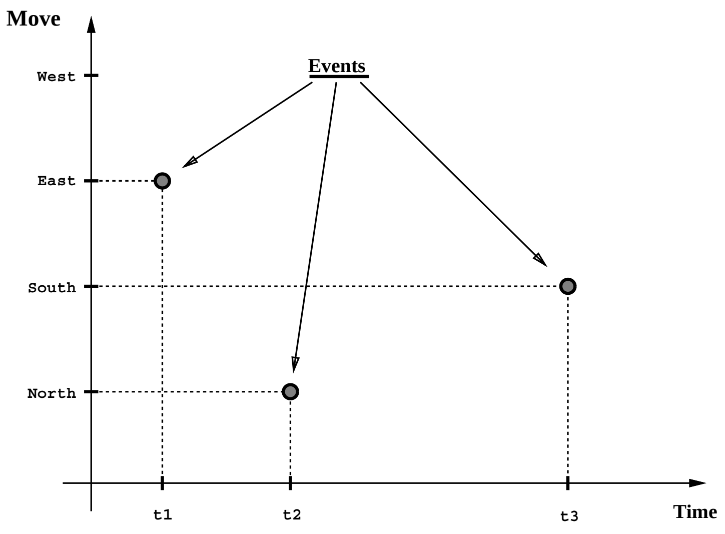


Figure 3.1: TRA objects and channels.



Signal: ... <Move(East),t1> <Move(North),t2> <Move(South),t3> ...

Figure 3.2: Signals, events, and actions.

To be identifiable, events on a given channel must be signaled at different points in time. Simultaneity can only be observed between events fired on different channels. Therefore, a signal $\langle a_0 : t_0\rangle\langle a_1 : t_1\rangle \ldots \langle a_k : t_k\rangle \ldots$, is totally ordered $(t_k < t_{k+1}, k \geq 0)$. In an earlier model [Best90b], we enforced this requirement by associating a minimum *switching time* with every channel; two events signaled on the same channel have to be separated by an amount of time equal to at least the minimum switching time. Imposing a finite positive switching time means that channels have *finite* capacities and thus cannot carry infinitely many events at the same time. Associating a lower bound on the switching time of input channels, however, seemed to violate the input-enabled principle. This is why, in the TRA model, we only require that a *positive* (rather than minimum) switching time exist.

At any point in time, a TRA is in a given *state*. The set of all such possible states defines the TRA's *state space*. The state of a TRA is visible and can only be changed by local *computations*. Computations (and thus state transitions) are triggered by actions and might be required to meet specific *timing constraints*.

In the following presentation we use capital letters (*e.g.* Θ, Γ, Δ) to denote sets, and small letters (*e.g.* θ, γ, δ) to denote members of these sets. For example, we use Θ to denote a set of states, and θ to denote an element of Θ. Subscripted capital letters are used to denote subsets. For example, Θ_i denotes a subset of the set Θ. We use superscripts to identify sets belonging to a given TRA object. For example, $\Theta^{\mathcal{A}}$ denotes the set of states associated with the TRA $\mathcal{A}$. Superscripts are dropped whenever the association is understood from the context. The dimensionality of a cross-product $\Theta = \Xi_1 \times \Xi_2 \times \ldots \times \Xi_n$ is n. Furthermore, if $\theta \in \Theta = (\xi_1, \xi_2, \ldots, \xi_n)$, $\xi_i \in \Xi_i$, where $1 \leq i \leq n$, then the r^{th} component of θ (namely ξ_r) is denoted by $\theta[r]$. A sequence s is an ordered string of symbols taken from an alphabet A. The set of all the prefixes of a sequence s is denoted by $\mathit{pref}(s)$ and the set of all the prefixes of any sequence in a set S is denoted by $\mathit{pref}(S)$. The catenation of two sequences s_1 and s_2 is denoted by $s_1 s_2$. An empty sequence is denoted by $\perp$. The cardinality of a set Θ is denoted by $|\Theta|$; its power set is denoted by 2^{Θ}. The sets of natural, integer, rational, and real numbers are denoted by $\mathcal{N}$, $\mathcal{Z}$, $\mathcal{Q}$, and $\Re$, respectively. The set ϕ denotes an empty set. We generalize a function $f : A \rightarrow B$ over subsets of its domain A by defining $f(A') = \bigcup_i f(a_i)$, where $a_i \in A' \subseteq A$.

3.2.2 The TRA Object

In this section we formally define the various components of a TRA object and informally introduce its operational behavior. A formal treatment of the operational semantic of a TRA object will be given in section 3.4.

Definition 1 *A* TRA *object*[3] *is a sextuple* $(\Sigma, \sigma_0, \Pi, \Theta, \Lambda, \Upsilon)$, *where*

- Σ, *the* TRA *signature, is the set of all the channels of the* TRA. *It is partitioned into three disjoint sets of input, output, and internal channels. We denote these by* Σ_{in}, Σ_{out}, *and* Σ_{int}, *respectively. The set consisting of both input and output channels is the set of external channels* (Σ_{ext}). *These are the only channels visible from outside the* TRA. *The set consisting of both output and internal channels is the set of local channels* (Σ_{loc}). *These are the locally controlled channels of the* TRA.

- $\sigma_0 \in \Sigma_{\text{in}}$ *is the start channel.*

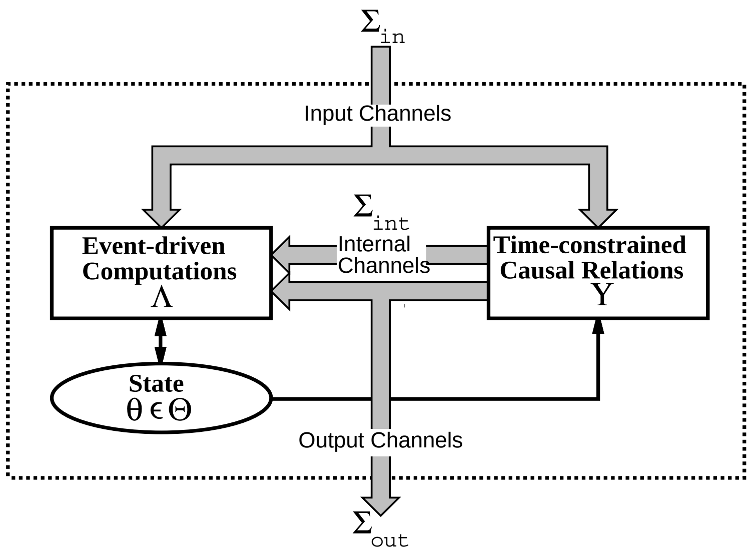


Figure 3.3: Basic components of a TRA object.

[3]see Figure 3.3 for an illustration.

- Π, *the signaling range function, maps each channel in* Σ *to a possibly infinite set of values that can be signaled as actions on that channel. Action sets of different channels are disjoint. The set of all the actions of a* TRA *is given by* $\Pi(\Sigma)$. *The set of input, output, internal, external, and local actions are similarly given by* $\Pi(\Sigma_{\text{in}})$, $\Pi(\Sigma_{\text{out}})$, $\Pi(\Sigma_{\text{int}})$, $\Pi(\Sigma_{\text{ext}})$, *and* $\Pi(\Sigma_{\text{loc}})$, *respectively.*

- Θ *is a possibly infinite set of states of the* TRA. *The set* Θ *can be expressed as the cross product of a finite number of subspaces* $\Theta = \Phi_1 \times \Phi_2 \times \ldots \times \Phi_p$, *where* $p \geq 1$ *denotes the dimensionality of* Θ.

- $\Lambda \subseteq \Theta \times \Pi(\Sigma) \times \Theta$ *is a set of possible computational steps of the* TRA. TRA*s are input enabled which means that for every* $\pi \in \Pi(\Sigma_{\text{in}})$, *and for every* $\theta \in \Theta$, *there exists at least one step* $(\theta, \pi, \theta') \in \Lambda$, *for some* $\theta' \in \Theta$. *Thus,* Λ *defines a total multifunction* $\Lambda : \theta \times \Pi(\Sigma_i n) \rightarrow \theta$.

- $\Upsilon \subseteq \Sigma \times \Sigma_{\text{loc}} \times \mathcal{D} \times 2^{\Theta}$ *is a set of time-constrained causal relationships (or simply time constraints) of the* TRA. *A time constraint* $v_i \in \Upsilon$ *is a quadruple* $(\sigma_i, \sigma_i', \delta_i, \Theta_i)$ *whose interpretation is that: if an action is signaled at time* $t \in \Re$ *on the channel* σ_i, *then a corresponding action must be fired on the channel* σ_i' *at time* t', *where* $t' - t \in \delta_i$, *provided that the* TRA *does not enter any of the states in* Θ_i *for the open interval* (t, t').[4] *The channel* $\sigma_i \in \Sigma$ *is called the trigger of the time constraint, whereas* $\sigma_i' \in \Sigma_{\text{loc}}$ *is called the constrained channel.* $\Theta_i \subseteq \Theta$ *defines the set of states that disable the time constraint; once triggered a time constraint becomes and remains active until satisfied or disabled. A time constraint is satisfied by the firing of an action on the channel* σ_i *within the imposed time bounds; it is disabled if the* TRA *enters in one of the disabling states in* Θ_i *before it is satisfied.*[5] *The interval* δ_i *specifies upper and lower bounds on the delay between the triggering and satisfaction (or disabling) of the time constraint* v_i.

[4]Notice that this condition does not necessitate the existence of a computational step $(\theta, \pi', \theta') \in \Lambda$ for each $\theta \in \Theta - \Theta_i$, where $\pi' \in \Pi(\sigma')$ and $\theta' \in \Theta$, since the specification of the TRA might avoid being in θ when σ' is scheduled to fire.

[5]see Figure 3.4 for an illustration.

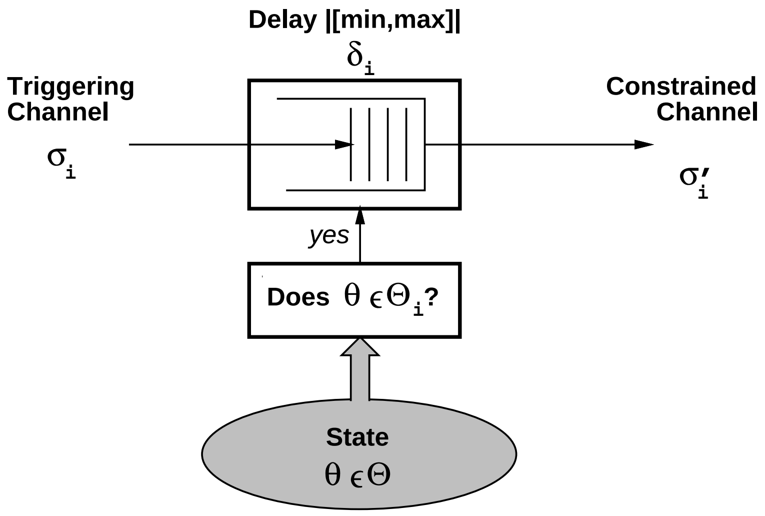


Figure 3.4: A TRA time-constrained causal relationship.

As an example of a TRA specification, consider the the up/down counter whose state diagram is shown in Figure 3.5. The counter accepts commands issued on the input channel `cmd` to count up or down and signals the value of the current count (state) on the output channel `cnt`. The counter starts its operation once an action is fired on the `init` channel. The value of the `init` signal determines the starting state of the counter. The counter is constrained to produce a count every at least 1.9 and at most 2.1 units of time, once it starts execution. Figure 3.6 shows the TRA-specification of such a counter.

The first three components of a TRA sextuple can be largely viewed as defining an interface between the TRA object and its environment. In particular, to be able to use the counter of Figure 3.6, it suffices to know its external signature $\Sigma_{\text{in}} = \{\texttt{init}, \texttt{cmd}\}, \Sigma_{\text{out}} = \{\texttt{cnt}\}$, the identity of the start channel $\sigma_0 = \texttt{init}$, along with the signaling range of all the channels in Σ_{ext}. The last three components of a TRA sextuple are responsible for its behavior. The state space defines the spatial structure of the computation. For the counter of Figure 3.6, this structure is unidimensionally spanned by the single state variable θ. The set of computational steps defines the effect of events on the state of the TRA. The set of time-constrained causalities defines the rules governing the *scheduling* of the TRA's local events. For the counter of Figure 3.6, there are two such rules.

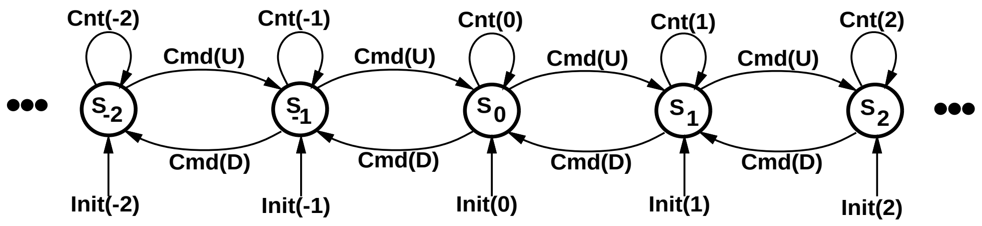


Figure 3.5: State diagram of up/down counter.

- ◇ Σ, the signature of the counter, consists of the union of the following classes: $\Sigma_{\text{in}} = \{\texttt{cmd}, \texttt{init}\}$, $\Sigma_{\text{out}} = \{\texttt{cnt}\}$, and $\Sigma_{\text{int}} = \phi$.
- ◇ $\texttt{init} \in \Sigma_{\text{in}}$ is the start channel.
- ◇ Π, the signaling range function, is defined as follows: $\Pi(\texttt{init}) = \mathcal{Z}$, $\Pi(\texttt{cmd}) = \{\texttt{UP}, \texttt{DOWN}\}$, and $\Pi(\texttt{cnt}) = \mathcal{Z}$.
- ◇ Θ, the set of states of the counter is given by: $\{\theta_i : i \in \mathcal{Z}\}$.
- ◇ Λ, the set of computational steps of the counter is given by:

$$\begin{aligned}\Lambda = {} & (\textstyle\bigcup_{i,j\in\mathcal{Z}}\{(\theta_i, \texttt{init}(j), \theta_j)\}) \cup \\ & (\textstyle\bigcup_{i\in\mathcal{Z}}\{(\theta_i, \texttt{cmd(UP)}, \theta_{i+1})\}) \cup \\ & (\textstyle\bigcup_{i\in\mathcal{Z}}\{(\theta_i, \texttt{cmd(DOWN)}, \theta_{i-1})\}) \cup \\ & (\textstyle\bigcup_{i\in\mathcal{Z}}\{(\theta_i, \texttt{cnt}(i), \theta_i)\}).\end{aligned}$$

- ◇ Υ, the set of time constraints is given by: $\Upsilon = \{(\texttt{init}, \texttt{cnt}, [1.9, 2.1], \phi), (\texttt{cnt}, \texttt{cnt}, [1.9, 2.1], \phi)\}$.

Figure 3.6: TRA-specification of up/down counter.

3.2.3 Sources of Non-determinism

The behavior of a TRA is generally non-deterministic. In particular, three sources of non-determinism can be singled out.

1. **Control non-determinism**: At any given point in time there might be a number of choices concerning the action to be fired. Each one of these choices results in a different computational step, and thus in a different execution.
2. **Timing non-determinism**: TRA timing constraints specify lower and upper bounds on the delay between causes and effects, thus leaving the TRA with a potentially infinite number of choices concerning the exact delay to be exhibited. Each one of these choices results in a different event, and thus in a different execution.
3. **Computation non-determinism**: The computation associated with specific actions might be non-deterministic. In this case, firing the same action from the same state might result in different next states, and thus in different executions.

Considered separately, each one of the above forms of non-determinism is benign. A combination thereof, however, deserves a closer attention. In particular, an unrestricted combination of control non-determinism and timing non-determinism might prove to be undesirable; it might allow behaviors that violate the spontaneity principle.

To illustrate this point, consider a TRA, A, for which two possible steps are: $(\theta_i, \pi_1, \theta_j)$ and $(\theta_i, \pi_2, \theta_k)$, where $\theta_j \neq \theta_k$. Furthermore, assume that $\mathcal{A}$ entered state θ_i at time t and that both π_1 and π_2 are scheduled. Now, if the timing constraints for π_1 and π_2 are specified such that both actions can fire on different channels at some later time t', then "what will be the next state of A? Will it be θ_j or θ_k or neither?" [6] The issue here is not whether the next state should be θ_j or θ_k. Rather, the issue is whether or not such a situation should have been allowed in the first place. In the next section, we impose some constraints on TRAs so as to avoid such malignant situations.

[6] The argument given here is made assuming that both π_1 and π_2 are locally-controlled actions. The same argument, however, can be made if either π_1 or π_2, or both are input actions.

3.3 Space and Time aspects of TRAs

The interplay between control non-determinism and timing non-determinism is interesting because it is related to the notions of space and time. Control non-determinism refers to uncertainties about the identity of the channel that will be fired; it refers to a *spatial uncertainty*. As such, and to abide by the spontaneity principle, it must reduce the range of possible *timing uncertainty*.

3.3.1 The Conflict Relationship

Two computational steps *conflict* if both of them introduce changes to at least one of the subspaces of the TRA's state space. This is formally defined below.

Definition 2 *Two computational steps* $(\theta_i, \pi_i, \theta'_i), (\theta_j, \pi_j, \theta'_j) \in \Lambda$ *conflict if and only if for some dimension* k *of* Θ, $\theta_i[k] \neq \theta'_i[k]$ *and* $\theta_j[k] \neq \theta'_j[k]$, *where* $1 \leq k \leq n$.

It is important to realize that the conflict relationship depends not only on a TRA's computational behavior, but also on the structure of its state space. In particular, two TRAs with isomorphic computational steps could have very different conflict relationships depending on their state space characterizations.

The notion of conflicting computational steps can be easily extended to actions and channels. This is formally defined below.

Definition 3 *Two actions* π_i *and* π_j *conflict if there exist at least two conflicting computational steps* $(\theta_i, \pi_i, \theta'_i), (\theta_j, \pi_j, \theta'_j) \in \Lambda$. *Two channels* σ_i *and* σ_j *conflict if at least one action from* $\Pi(\sigma_i)$ *and one action from* $\Pi(\sigma_j)$ *conflict.*

From the above definitions, it is obvious that for a given TRA object, all conflicting computational steps, actions, and channels can be properly identified.

3.3.2 Proper TRAs

The input channels of a given TRA are not under its control; they can fire at any time.[7] To preserve the non-blocking (input-enabled) nature of the TRA model, it is, therefore, necessary to insure that input actions on different channels do not conflict.

Definition 4 *A* TRA $\mathcal{A}$ *is improper if at least two of its input channels conflict, otherwise it is proper.*

The counter shown in Figure 3.7 is an example of an improper TRA specification,[8] in which the input channels `up` and `down` conflict over the state variable θ. A proper specification of that counter is shown in Figure 3.8, in which the conflict between `up` and `down` has been removed by shifting the responsibility for changing the state variable θ to two internal (local) channels `go_up` and `go_down`. Notice that the two counters do not specify the same behavior. In particular, the proper counter of Figure 3.8 introduces a non-zero delay between the occurence of an input action on the `up` or `down` channels and the respective change in the value of the state variable θ.

For the remainder of this thesis, it will be assumed that any TRA is *proper* unless otherwise stated.

3.3.3 TRA Control Components

The conflict relationship depicts computational dependencies that emerge due to sharing information about state. For two local actions to conflict, their respective channels must be under the control of a single *component* of the TRA. The transitive closure of the conflict relationship, therefore, defines a partition on the locally-controlled channels of a given TRA.

Definition 5 *Two local channels σ_i and σ_j belongs to the same component (class) if they conflict.*

[7] In particular, two input actions signaled on different channels can occur simultaneously.

[8] Unlike the counter of Figure 3.6, the counters specified in Figures 3.7 & 3.8 have separate channels for accepting requests to count `up` or `down`.

- ⋄ $\Sigma_{\text{in}} = \{\texttt{up}, \texttt{down}, \texttt{init}\}$, $\Sigma_{\text{out}} = \{\texttt{cnt}\}$, and $\Sigma_{\text{int}} = \phi$.
- ⋄ $\sigma_0 = \texttt{init} \in \Sigma_{\text{in}}$.
- ⋄ $\Pi(\texttt{init}) = \mathcal{Z}$, $\Pi(\texttt{up}) = \{\}$, $\Pi(\texttt{down}) = \{\}$, and $\Pi(\texttt{cnt}) = \mathcal{Z}$.
- ⋄ $\Theta = \{\theta_i : i \in \mathcal{Z}\}$.
- ⋄ $\Lambda = (\bigcup_{i,j\in\mathcal{Z}}\{(\theta_i, \texttt{init}(j), \theta_j)\}) \cup (\bigcup_{i\in\mathcal{Z}}\{(\theta_i, \texttt{cnt}(i), \theta_i)\}) \cup$
  $(\bigcup_{i\in\mathcal{Z}}\{(\theta_i, \texttt{up}(), \theta_{i+1})\}) \cup (\bigcup_{i\in\mathcal{Z}}\{(\theta_i, \texttt{down}(), \theta_{i-1})\})$.
- ⋄ $\Upsilon = \{(\texttt{init}, \texttt{cnt}, [1.9, 2.1], \phi), (\texttt{cnt}, \texttt{cnt}, [1.9, 2.1], \phi)\}$.

Figure 3.7: An improper TRA specification of a counter.

The partition of the TRA's locally-controlled channels into classes captures some of the structure of the system the automaton is modeling or the set of requirements it is specifying. In particular, each class of channels is intended to represent the set of channels locally-controlled by *some* system component. This partitioning retains the basic control structure of the system's primitive components.

To illustrate the notion of system components, consider the local channels `go_up`, `go_down`, and `cnt` of the counter specified in Figure 3.8. Obviously, `go_up` and `go_down` conflict since they modify the single state variable θ. Hence, they must be under the control of a single TRA component; they cannot fire simultaneously. The channel `cnt`, on the other hand, conflicts with neither `go_up` nor `go_down`. Thus, it belongs to a different TRA component, and as such, might fire simultaneously with either one of them.

The notion of system components we are presenting here is novel and entirely different from that used in untimed models to express fairness [Lync88b] by requiring that, in an infinite execution, each of the system's components gets infinitely many chances to perform its locally-controlled actions. In timed systems, the major concern is *safe* and not necessarily *fair* executions [Schn88]. Even if required, fairness can be enforced by treating it as a safety property; liveness properties can be handled in infinite execution by requiring time to grow unboundedly.[9] This led to the abandoning of the idea of partitioning a system into components in our earlier model proposed in [Best90b]. Lynch and Vaandrager [Lync91] followed suit in their recent modification of the model proposed in [Tutt88].

- ◇ $\Sigma_{\text{in}} = \{\texttt{up}, \texttt{down}, \texttt{init}\}$, $\Sigma_{\text{out}} = \{\texttt{cnt}\}$, and $\Sigma_{\text{int}} = \{\texttt{go\_up}, \texttt{go\_down}\}$.
- ◇ $\sigma_0 = \texttt{init} \in \Sigma_{\text{in}}$.
- ◇ $\Pi(\texttt{init}) = \mathcal{Z}$, $\Pi(\texttt{up}) = \{\}$, $\Pi(\texttt{down}) = \{\}$, $\Pi(\texttt{go\_up}) = \{\}$, $\Pi(\texttt{go\_down}) = \{\}$, and $\Pi(\texttt{cnt}) = \mathcal{Z}$.
- ◇ $\Theta = \{\theta_i : i \in \mathcal{Z}\}$.
- ◇ $\Lambda = (\bigcup_{i,j\in\mathcal{Z}}\{(\theta_i, \texttt{init}(j), \theta_j)\}) \cup (\bigcup_{i\in\mathcal{Z}}\{(\theta_i, \texttt{cnt}(i), \theta_i)\}) \cup$
  $(\bigcup_{i\in\mathcal{Z}}\{(\theta_i, \texttt{up}(), \theta_i)\}) \cup (\bigcup_{i\in\mathcal{Z}}\{(\theta_i, \texttt{down}(), \theta_i)\}) \cup$
  $(\bigcup_{i\in\mathcal{Z}}\{(\theta_i, \texttt{go\_up}(), \theta_{i+1})\}) \cup (\bigcup_{i\in\mathcal{Z}}\{(\theta_i, \texttt{go\_down}(), \theta_{i-1})\})$.
- ◇ $\Upsilon = \{(\texttt{up}, \texttt{go\_up}, [0.1, 0.2], \phi), (\texttt{down}, \texttt{go\_down}, [0.1, 0.2], \phi),$
  $(\texttt{init}, \texttt{cnt}, [1.9, 2.1], \phi), (\texttt{cnt}, \texttt{cnt}, [1.9, 2.1], \phi)\}$.

Figure 3.8: A proper TRA specification of a counter.

In the TRA model we use system components to represent what can be termed as *spatial locality*. Different actions can be signaled at the same "time" only if they are not signaled from the same "place"; they can be produced at the same "place" only if they do not occur at the same "time". This intuition is inspired from physical systems, where events are characterized and distinguishable by their time-space coordinates [Hawk88].

[9]Such executions were called *admissible* in [Lync91].

3.4 The TRA Operational Semantics

In this section, we describe the rules governing the reaction of a TRA object to the events occuring on its input channels. Once these rules are specified, the possible behaviors of a given TRA can be determined.

3.4.1 TRA Intentions, Status, and Status Succession

In standard automata theory, there is no distinction between choosing a transition and firing it; they constitute a unique, instantaneous, and atomic activity. In the TRA model a distinction is made whereby choosing (scheduling) a transition and executing (committing) that transition are separate activities. They are *distinct* in that they are separated in time. In fact, a scheduled transition does not have to be committed; it can be abandoned due to unforseeable conditions. The distinction between the two activities is also pronounced in the way the TRA model differentiates between input and local events. Input events are uncontrollable; they are not scheduled. Local events are.

The *state* of a TRA at an arbitrary point in time is not sufficient to construct its *future behavior*. To explain why this is true consider the example shown in Figure 3.9, where a TRA is known to be in some state s at time t_1. Assume that, due to a triggering event at some earlier time t_0, an action is scheduled to fire at some point in a future interval given by $|[t_0 + t_{\text{lo}}, t_0 + t_{\text{hi}}]|$. Knowing only the state of the TRA at time t_1 is not sufficient to predict future behaviors. In addition to the state, the intervals of time where scheduled transitions might fire have to be recorded. We encapsulate this knowledge in our notion of *intentions*.

Consider the time constraint $v_i = (\sigma_i, \sigma_i', \delta_i, \Theta_i) \in \Upsilon$. v_i identifies a time-constrained causal relationship between the events signaled on σ_i and those signaled on σ_i'. In particular, the occurence of a triggering event on σ_i results in an intention to perform an action on σ_i' within the time frame imposed by δ_i. The commitment or abandonment of such an intention in due time is conditional on the states assumed by the TRA from when the intention is posted until when it is committed or abandoned. At any given point in time, a TRA might have several outstanding intentions. In particular, the occurence of a single

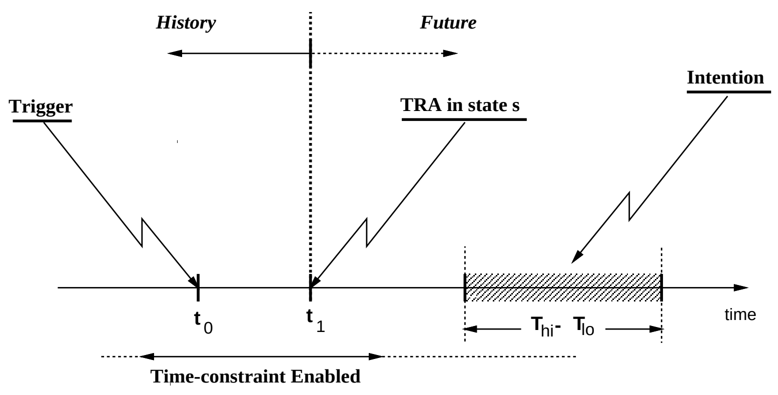


Figure 3.9: The notion of a TRA status.

event might generate a number of intentions, each dictated by a different time constraint. Different outstanding intentions are not necessarily imposed by different time constraints. In particular, the repeated occurence of a triggering event might generate a number of outstanding intentions, all of which are posted by the same time constraint.

For a given TRA, we define the *intention vector* $I = \vec{\Delta}$ to be a vector of r sets of intentions, where $r = |\Upsilon|$. Each entry in I is associated with one of the TRA's time constraints. In particular, if $v_i = (\sigma_i, \sigma'_i, \delta_i, \Theta_i) \in \Upsilon$ is one of the TRA's time constraints, then $I[v_i] = \{\delta_{i1}, \delta_{i2}, \ldots, \delta_{ik}, \ldots \delta_{im}\}$ denotes a set of m time intervals during which actions on the channel σ'_i have been scheduled to fire as a result of earlier triggers on σ_i. Each one of the intervals in Δ_i can be thought of as an independent *activation* of the time constraint v_i. An empty intention set, $I[v_i] = \phi$, indicates the absence of any activations of v_i. The empty intention vector, I_ϕ, consists of r such empty intention sets. At any point in time, the intention vector of a TRA can be thought of as an extension of the TRA's state. This is encapsulated in our notion of a TRA status.

Definition 6 *The status of a* TRA $(\Sigma, \sigma_0, \Pi, \Theta, \Lambda, \Upsilon)$ *at any point in time* $t \in \Re$ *is the tuple* (θ, I)*, where* $\theta \in \Theta$ *and* I *are the* TRA*'s state and intention vector at time* t*, respectively.*

At any point in time, a TRA can be in exactly one status. A TRA changes its status in response to the occurence of any number of events at a given point in time.

Definition 7 *Assume that the status (θ, I) of a* TRA *was entered at time t. Furthermore, assume that at a later time $t' > t$, a set of simultaneous actions $\pi_1 \in \Pi(\sigma_1), \pi_2 \in \Pi(\sigma_2), \ldots, \pi_m \in \Pi(\sigma_m)$ were fired, where $\sigma_j \in \Sigma, 0 \leq j \leq m$. As a result, the* TRA *will assume a new status (θ', I'), where $I' = (I \cup I'_{\text{enabled}}) - (I'_{\text{fired}} \cup I'_{\text{disabled}})$.*

The status (θ', I') is called a valid successor *of the status (θ, I) due to the occurence of the set of simultaneous events $\langle \pi_1, \pi_2, \ldots, \pi_m : t' \rangle$, if and only if the following conditions hold:*

1. *Spontaneity:*
   The channels $\sigma_1, \sigma_2, \ldots, \sigma_m$ do not conflict; they belong to different TRA *components.*
2. *Legality:*
   There exists some sequence of transitions $(\theta, \pi_1, \theta_1), (\theta, \pi_2, \theta_2), \ldots (\theta, \pi_m, \theta_m) \in \Lambda$, such that $\theta_m = \theta'$.
3. *Safety:*
   For every intention $\delta_{ik} \in I[\upsilon_i]$, $t'' \in \delta_{ik}$ for some $t'' > t'$, $t'' \in \Re$, where $\upsilon_i \in \Upsilon$.
4. *Causality:*
   For all $\sigma_i \in \Sigma_{\text{loc}}$, the following conditions hold
   a. *If $\sigma_i \neq \sigma_j$ for all $1 \leq j \leq m$ then for every $\upsilon_k = (\sigma_k, \sigma'_k, \delta_k, \Theta_k) \in \Upsilon$ for which $\sigma'_k = \sigma_i$, $I'_{\text{fired}}[\upsilon_k] = \phi$.*
   b. *Otherwise, let $\Upsilon_i \subseteq \Upsilon$ be the set of time constraint with σ_i as the constrained channel, then there must exist exactly one time constraint $\upsilon_r \in \Upsilon_i$ such that:*
      - ◇ *$I'_{\text{fired}}[\upsilon_r] = \{\delta_{rk}\}$, where $\delta_{rk} \in I[\upsilon_r]$ and $t' \in \delta_{rk}$, and*
      - ◇ *$I'_{\text{fired}}[\upsilon_k] = \phi$, where $\upsilon_k \in \Upsilon_i$ and $\upsilon_k \neq \upsilon_r$.*
5. *Consistency:*
   For every time constraint $\upsilon_k = (\sigma_k, \sigma'_k, \delta_k, \Theta_k) \in \Upsilon$, the following conditions hold
   a. *If $\theta' \in \Theta_k$, then*
      - ◇ *$I'_{\text{disabled}}[\upsilon_k] = I[\upsilon_k]$ and*
      - ◇ *$I'_{\text{enabled}}[\upsilon_k] = \phi$.*
   b. *Otherwise*
      - ◇ *$I'_{\text{disabled}}[\upsilon_k] = \phi$, and*
      - ◇ *If $\sigma_k = \sigma_j$ for some $1 \leq j \leq m$, then $I'_{\text{enabled}}[\upsilon_k] = \{(t' + \delta_i)\}$, else $I'_{\text{enabled}}[\upsilon_k] = \phi$.*

In the above definition, the *spontaneity* condition allows the occurence of simultaneous events only if they do not conflict. As we have mentioned before, time has to elapse for dependencies to be manifested. The *legality* condition ensures that the change in the state of the TRA from θ to θ' is the result of defined computational steps. Notice that the spontaneity condition ensures that the transition from θ to θ' is independent of the ordering of the computational steps. The *safety* condition guarantees that no active time constraint expires. In other words, outstanding intentions are either committed or abandoned in due time. The *causality* condition necessitates that local events be causal; they are signaled only if intended due to an earlier trigger. In particular, the causality condition guarantees that there is exactly one committed intention per local event. In other words, every local event satisfies exactly one intention. The *consistency* condition requires that the intentions in I continue to exist in I' unless otherwise dictated by the occurence of the set of simultaneous events $\langle \pi_1 : t' \rangle \langle \pi_2 : t' \rangle \ldots \langle \pi_m : t' \rangle$.

We use the notation $(\theta, I) \overset{\langle \pi_1, \pi_2 \ldots \pi_m : t' \rangle}{\longmapsto} (\theta', I')$ to denote the *direct status succession* from (θ, I) to (θ', I') due to the firing of the set of simultaneous events $\langle \pi_1 : t' \rangle$, $\langle \pi_2 : t' \rangle$, ..., $\langle \pi_m : t' \rangle$. Furthermore, we use the notation $(\theta, I) \overset{\alpha}{\longmapsto} (\theta', I')$ to denote the *extended status succession* from (θ, I) to (θ', I') due to a number of direct status successions.

A TRA is said to have reached a *stable status* $(\hat{\theta}, \hat{I})$, if all entries of the intention vector are empty $(\hat{I} = I_\phi)$. A TRA remains in a stable status until excited by an input event. This follows directly from the causality requirement for a status succession.

3.4.2 TRA Executions, Schedules, and Behaviors

To start executing, a TRA $(\Sigma, \sigma_0, \Pi, \Theta, \Lambda, \Upsilon)$ is put in a stable status (θ_0, I_0), where $I_0 = I_\phi$ and $\theta_0 \in \Theta$. The status (θ_0, I_0) is called an *initial status*. The execution is initiated at time t_0 with the firing of an action π_0 on the start channel σ_0, where $\pi_0 \in \Pi(\sigma_0)$. The event $\langle \pi_0 : t_0 \rangle$ is called the *initiating event.*

An *execution fragment* of a TRA is a possibly infinite string of alternating statuses and events. A *partial execution* of a TRA is an execution fragment that starts with an initial status followed by an initiating event.

Definition 8 *A legal execution (or simply an execution) of a* TRA *is a partial execution e, which satisfies one of the following conditions:*[10]

a. *The execution e terminates in a stable status (finite execution), or*

b. *The execution e contains an infinite number of status successions (infinite execution).*

The following definitions extend those originally proposed in [Lync88b] and are similar to those reported in [Best90b].

A *schedule* α of an execution e is the sequence consisting of *all* the events appearing in e. In looking at schedules of TRAs, we will be often interested in events occuring on only a subset of the TRA channels. For this purpose, we define the projection operation "$|$".

Definition 9 *Let $\Sigma_p^{\mathcal{A}}$ be a subset of the signature $\Sigma^{\mathcal{A}}$ of some* TRA *$\mathcal{A}$. If γ is a sequence of events over some signature Σ, containing at least one initiating event for $\mathcal{A}$, then the projection $\gamma|\Sigma_p^{\mathcal{A}}$ consists of all the events signaled on or after $t_0^{\mathcal{A}}$ on any of the channels in $\Sigma_p^{\mathcal{A}}$, where $t_0^{\mathcal{A}}$ is the time of the first initiating event $\langle \pi_0^{\mathcal{A}} : t_0^{\mathcal{A}} \rangle$ for $\mathcal{A}$. If γ contains no initiating events for $\mathcal{A}$, then the projection $\gamma|\Sigma_p^{\mathcal{A}}$ is the empty sequence $\perp$.*

Since internal events are invisible from outside a TRA, we will often be interested only in external events. We define β to be a *behavior* of a TRA $\mathcal{A}$, if it consists of all the *external* events appearing in some schedule α of $\mathcal{A}$. In particular, the behavior β of a TRA $\mathcal{A}$ is obtained from a schedule α by projecting the latter on $\mathcal{A}$'s external signature. That is, $\beta = \alpha|\Sigma_{\text{ext}}^{\mathcal{A}}$.

We denote the set of all the possible legal executions of a TRA $\mathcal{A}$ by $execs(\mathcal{A})$, the set of all its possible legal schedules by $scheds(\mathcal{A})$, and the set of all its possible legal behaviors by $behs(\mathcal{A})$. Obviously, $behs(\mathcal{A})$ describes all the possible interactions that the TRA $\mathcal{A}$ might be engaged in, and, therefore, constitutes a complete specification of the system that $\mathcal{A}$ models.

[10]In most of the interesting cases we will be concerned with executions including only one initiating event.

3.4.3 TRA Implementation

A TRA $\mathcal{A}$ is said to *implement* another TRA $\mathcal{B}$ if $\mathcal{A}$ does not produce any behavior that $\mathcal{B}$ could not have produced [Lync88b]. In other words, all of $\mathcal{A}$'s behaviors (the implementation) are possible behaviors of $\mathcal{B}$ (the specification). The reverse, however, is not true. There might exist behaviors of $\mathcal{B}$ that cannot be generated by $\mathcal{A}$. The notion of a TRA implementing another will be used mainly in verification.

Definition 10 *A* TRA $\mathcal{A}$ *implements a* TRA $\mathcal{B}$ *with respect to the signature* σ_p*, if and only if* $(behs(\mathcal{A})|\Sigma_p) \subseteq (behs(\mathcal{B})|\Sigma_p)$*. The* TRA $\mathcal{A}$ *is called the implementation* TRA*; the* TRA $\mathcal{B}$ *is called the specification* TRA*.*

We differentiate between two types of implementations. *Weak implementations* are established based on behaviors projected on the *common* external signature of both the specification and implementation TRAs, whereas *strong implementations* are established based on behaviors projected on the external signature of the specification TRA alone.

Definition 11 *A* TRA $\mathcal{A}$ *weakly implements a* TRA $\mathcal{B}$*, if and only if* $(behs(\mathcal{A})|\Sigma^{\mathcal{A}}_{\text{ext}} \cap \Sigma^{\mathcal{B}}_{\text{ext}}) \subseteq (behs(\mathcal{B})|\Sigma^{\mathcal{A}}_{\text{ext}} \cap \Sigma^{\mathcal{B}}_{\text{ext}})$.

The above definition does not constrain the signature of the implementation TRA. This leads to a trivial weak implementation of any TRA, namely the *nil* TRA, which has an empty external signature. Weak implementations will be used later to verify *property preservation.*

Definition 12 *A* TRA $\mathcal{A}$ *strongly implements (or simply implements) another* TRA $\mathcal{B}$*, if and only if* $behs(\mathcal{A})|\Sigma^{\mathcal{B}}_{\text{ext}} \subseteq behs(\mathcal{B})$.

Lemma 1 *A necessary condition for a* TRA $\mathcal{A}$ *to (strongly) implement another* TRA $\mathcal{B}$ *is that* $\Sigma^{\mathcal{A}}_{\text{in}} = \Sigma^{\mathcal{B}}_{\text{in}}$*, and* $\Pi^{\mathcal{A}}(\sigma_i) = \Pi^{\mathcal{B}}(\sigma_i), \forall \sigma_i \in \Sigma^{\mathcal{A}}_{\text{in}}$.

Proof: Immediate from the definition and the input enabled property of TRAs.

■

The above conditions for a strong implementation do not constrain the output signature or signaling ranges of the implementation. This is so because, in general, it might be possible to avoid the use of one (or more) of the output channels (actions). That is, the implementing TRA might elect to always produce behaviors that do not include actions signaled on some output channels.[11] In most of the interesting cases, however, the implementing TRA will not be able to discard any output channels (actions) and thus will have external signature and signaling ranges identical to those of the implemented TRA. In the remainder of this thesis, we will assume that this is indeed the case.

3.4.4 TRA Equivalence

Two TRAs are equivalent if there is no way of identifying one from the other just by comparing their behaviors.

Lemma 2 *A* TRA $\mathcal{A}$ *is equivalent to another* TRA $\mathcal{B}$ *if and only if:* $\mathcal{A}$ *implements* $\mathcal{B}$*, and* $\mathcal{B}$ *implements* $\mathcal{A}$*.*

Proof: Both the *if* and the *only if* parts can be proved by contradiction.

- ◇ Let $\mathcal{A}$ be equivalent to $\mathcal{B}$ and assume that $\mathcal{A}$ does not implement $\mathcal{B}$.[12] It follows that there exists at least one behavior of $\mathcal{A}$ that is not a behavior of $\mathcal{B}$. Using this behavior, the TRAs $\mathcal{A}$ and $\mathcal{B}$ can be identified. Hence, they are not equivalent – a contradiction.
- ◇ Let $\mathcal{A}$ be an implementation of $\mathcal{B}$ and $\mathcal{B}$ be an implementation of $\mathcal{A}$ and assume that $\mathcal{A}$ and $\mathcal{B}$ are not equivalent. It follows that there exists a behavior of $\mathcal{A}$ (or $\mathcal{B}$) that is not a behavior of $\mathcal{B}$ (or $\mathcal{A}$). Hence $\mathcal{A}$ (or $\mathcal{B}$) does not implement $\mathcal{B}$ (or $\mathcal{A}$) – a contradiction.

■

[11] If such behaviors are allowed by the specification.

[12] The case in which $\mathcal{B}$ does not implement $\mathcal{A}$ is symmetric.

3.5 Operations on TRAs

A basic aspect of the TRA model is its capability to model a complex system by operating on simpler system components. In this section we examine three basic operations, namely hiding, renaming, and composition.

3.5.1 Hiding

The hiding operation is a binary operation that, given a TRA $\mathcal{A}$ and a subset of its output channels $\Sigma_h \subseteq \Sigma^{\mathcal{A}}_{\text{out}}$, produces a new TRA $\mathcal{A}' = \mathcal{A}\sharp\Sigma_h$ whose signature is identical to $\mathcal{A}$ except that channels in Σ_h are classified as internal rather than output (external). Obviously, any behavior of $\mathcal{A}'$ does not reflect any actions signaled on any of the channels in Σ_h.

Definition 13 *The* TRA *resulting from hiding the set of output channels* Σ_h *of a* TRA $\mathcal{A}$ *given by the sextuple* $(\Sigma, \sigma_0, \Pi, \Theta, \Lambda, \Upsilon)$, $\Sigma_h \subseteq \Sigma^{\mathcal{A}}_{\text{out}}$, *is given by* $\mathcal{A}' = (\Sigma', \sigma_0, \Pi, \Theta, \Lambda, \Upsilon)$, *where:*

- $\Sigma' = \Sigma^{\mathcal{A}'}_{\text{in}} \cup \Sigma^{\mathcal{A}'}_{\text{out}} \cup \Sigma^{\mathcal{A}'}_{\text{int}}$, *where:*
  - ◇ $\Sigma^{\mathcal{A}'}_{\text{in}} = \Sigma^{\mathcal{A}}_{\text{in}}$.
  - ◇ $\Sigma^{\mathcal{A}'}_{\text{out}} = \Sigma^{\mathcal{A}}_{\text{out}} - \Sigma_h$.
  - ◇ $\Sigma^{\mathcal{A}'}_{\text{int}} = \Sigma^{\mathcal{A}}_{\text{int}} \cup \Sigma_h$.

Lemma 3 *Let* $\mathcal{A}$ *be the* TRA $(\Sigma, \sigma_0, \Pi, \Theta, \Lambda, \Upsilon)$, *and let* $\Sigma_h \subseteq \Sigma^{\mathcal{A}}_{\text{out}}$. *The executions, schedules, and behaviors of* $\mathcal{A}\sharp\Sigma_h$ *are given by:*

1. $execs(\mathcal{A}\sharp\Sigma_h) = execs(\mathcal{A})$,
2. $scheds(\mathcal{A}\sharp\Sigma_h) = scheds(\mathcal{A})$,
3. $behs(\mathcal{A}\sharp\Sigma_h) = behs(\mathcal{A})|(\Sigma^{\mathcal{A}} - \Sigma_h)$.

Proof: The proof of the Lemma follows immediately from the definition of hiding and the definitions of executions, schedules, and behaviors.

■

Theorem 1 *If* TRA $\mathcal{A}$ *implements* TRA $\mathcal{B}$ *then* $\mathcal{A}\sharp\Sigma_h$ *implements* $\mathcal{B}\sharp\Sigma_h$*, where* $\Sigma_h \subseteq \Sigma^{\mathcal{A}}_{\text{out}}$.

Proof:

- ◇ Assume that $\beta \in behs(\mathcal{A}\sharp\Sigma_h)$, and let $e \in execs(\mathcal{A}\sharp\Sigma_h)$ be the execution exhibiting β. From Lemma 3, we get that $e \in execs(\mathcal{A})$. (1)
- ◇ Let β' be the behavior of $\mathcal{A}$ exhibited in e, namely $\beta = \beta'|(\Sigma^{\mathcal{A}} - \Sigma_h)$. Since $\mathcal{A}$ implements $\mathcal{B}$, we get that $behs(\mathcal{A}) \subseteq behs(\mathcal{B})$. (2)
- ◇ From (1) and (2) we get that $\beta' \in behs(\mathcal{B})$ and $\beta = \beta'|(\Sigma^{\mathcal{A}} - \Sigma_h) \in behs(\mathcal{B}\sharp\Sigma_h)$. Therefore, $behs(\mathcal{A}\sharp\Sigma_h) \subseteq behs(\mathcal{B}\sharp\Sigma_h)$, from which we conclude that $\mathcal{A}\sharp\Sigma_h$ implements $\mathcal{B}\sharp\Sigma_h$.

■

3.5.2 Renaming

The renaming operation is a binary operation that, given a TRA $\mathcal{A}$ and a one-to-one function onto $F : \Sigma \rightarrow \Sigma'$, produces a new TRA $\mathcal{A}' = \mathcal{A}/F$ which is identical to $\mathcal{A}$ except that its channels are renamed according to F. The function F is called a *renaming function.* Obviously, the TRAs $\mathcal{A}$ and $\mathcal{A}'$ are isomorphic.

Definition 14 *A* TRA $\mathcal{A}$ *is an isomorphic implementation (or equivalent) to another* TRA $\mathcal{B}$ *if there exists a renaming function* F *for* A *such that* $\mathcal{A}/F$ *implements (or is equivalent to) the* TRA $\mathcal{B}$*.*

An important property of the renaming operation is that it preserves the implementation and equivalence relationships. This is established in the following easy to prove theorem.

Theorem 2 *If* TRA $\mathcal{A}$ *implements (or is equivalent to)* TRA $\mathcal{B}$ *and* F *is a renaming function for* $\mathcal{A}$ *then the* TRA $\mathcal{A}/F$ *implements (or is equivalent to) the* TRA $\mathcal{B}/F$*.*[13]

[13]It can be shown that if $\mathcal{A}$ implements (or is equivalent to) $\mathcal{B}$, then a renaming function for $\mathcal{A}$ is also a renaming function for $\mathcal{B}$.

3.5.3 Composition

The composition of a countable collection of compatible TRAs, $\{\mathcal{A}_i : i \in \mathcal{I}\}$, is a new TRA $\mathcal{A} = \mathcal{A}_0 \times \mathcal{A}_1 \times \ldots \times \mathcal{A}_i \times \ldots = \Pi_{i \in \mathcal{I}} \mathcal{A}_i$. The execution of $\mathcal{A}$ involves the execution of all its components $\mathcal{A}_{i \in \mathcal{I}}$, each starting from an initial status and observing every external event signaled by either the environment (input) or by any TRA in the collection $\{\mathcal{A}_i : i \in \mathcal{I}\}$. The *compatibility* condition for composition insures that, for each channel in the composition, there is at most one writer, a finite number of readers, and that the signaling ranges of readers and writers are compatible. We formally define these notions below.

Definition 15 *Given a countable collection of* TRA*s, $\{\mathcal{A}_i : i \in \mathcal{I}\}$, and a channel σ, we define the fan-in of σ to be:*

$$\mathit{fin}(\sigma) = |\{i : i \in \mathcal{I}, \sigma \in \Sigma^{\mathcal{A}_i}_{\text{out}}\}|$$

Definition 16 *Given a countable collection of* TRA*s, $\{\mathcal{A}_i : i \in I\}$, and a channel σ, we define the fan-out of σ to be:*

$$\mathit{fout}(\sigma) = |\{i : i \in \mathcal{I}, \sigma \in \Sigma^{\mathcal{A}_i}_{\text{in}}\}|$$

For a given collection of TRAs, the fan-in of a channel σ represents the number of TRAs that use that channel as an output channel; these are the *writers* of σ. Similarly, the fan-out of σ represents the number of TRAs that use that channel as an input channel; these are the *readers* of σ.

Definition 17 *A countable collection of* TRA*s, $\{\mathcal{A}_i : i \in \mathcal{I}\}$, is said to be I/O compatible if and only if, for all $i, j \in \mathcal{I}, i \neq j$, the following is satisfied:*

1. *If $\sigma \in \Sigma^{\mathcal{A}_i}_{\text{out}}$ and $\sigma \in \Sigma^{\mathcal{A}_j}_{\text{in}}$ then $\Pi^{\mathcal{A}_i}(\sigma) \subseteq \Pi^{\mathcal{A}_j}(\sigma)$, and*
2. *$\Sigma^{\mathcal{A}_i}_{\text{int}} \cap \Sigma^{\mathcal{A}_j} = \phi$.*

The first condition for I/O compatibility ensures that actions fired by a TRA on one end of a channel are within the range of expectation of the TRA on the other end. This requirement is necessary to preserve the *input enabled* property of composed TRAs. The second condition guarantees that the actions of internal channels are unobservable by the environment.[14]

Definition 18 *A countable collection of I/O compatible* TRA*s, $\{\mathcal{A}_i : i \in \mathcal{I}\}$, is said to be strongly compatible (or simply compatible) if and only if:*

1. *Every channel $\sigma \in \bigcup_{i \in \mathcal{I}} \Sigma^{\mathcal{A}_i}$ satisfies the following conditions:*
   - a. *$fin(\sigma) \leq 1$, and*
   - b. *$fout(\sigma)$ is finite.*
2. *The start channels of the collection satisfy the following conditions:*
   - a. *$fin(\sigma_0^{\mathcal{A}_0}) = 0$, and*
   - b. *If $fin(\sigma_0^{\mathcal{A}_i}) = 0$ then $\sigma_0^{\mathcal{A}_i} = \sigma_0^{\mathcal{A}_0}$.*

In the above definition, the first condition guarantees that at most one writer and only a finite number of readers are specified for each channel. We impose these restrictions to insure realistic modeling. In particular, forcing two signals produced by different TRAs to be identical violates the assumption of control autonomy for TRAs. Also, signaling an action to an infinite number of readers requires an infinite amount of energy – a physical impossibility. The second condition requires that all TRAs with start channels that are not specified as outputs of other TRAs share the same start channel; at least one such TRA exists and, without loss of generality, we assume that $\mathcal{A}_0$ falls into that category.

The composition operation is only defined for collections of strongly compatible TRAs. The input signature of the composed TRA consists of those channels that are inputs to one or more of the component TRAs, and which are not outputs of any of the component TRAs. The output signature of the composed TRA consists of all the outputs of all the component TRAs. Similarly, the internal signature of the composed TRA consists of all the internal channels of all the component TRAs. The start channel of the composed TRA is the start channel of one or more of its component TRAs.[15] The signaling range function of the composed TRA is

[14]This was termed as the "privacy respect" condition in [Merr89].

[15]Without loss of generality, we assume that TRA to be $\mathcal{A}_0$.

defined so as to preserve its input-enabled property. In particular, the signaling range of an input channel consists of only those actions that can accepted by all readers of that channel. A computational step of the composed TRA is necessarily a step of one of its components. Similarly the time-constrained causal relationships of the composed TRA are exactly those of the component TRAs.

Definition 19 *The composition $\mathcal{A} = \prod_{i\in\mathcal{I}} \mathcal{A}_i$ of a strongly compatible collection of* TRA*s, $\{\mathcal{A}_i : i \in \mathcal{I}\}$, is the* TRA *defined as follows:*

- *The signature, $\Sigma^{\mathcal{A}} = \Sigma^{\mathcal{A}}_{\text{in}} \cup \Sigma^{\mathcal{A}}_{\text{out}} \cup \Sigma^{\mathcal{A}}_{\text{int}}$, where:*
  - ◇ $\Sigma^{\mathcal{A}}_{\text{in}} = (\bigcup_{i\in\mathcal{I}} \Sigma^{\mathcal{A}_i}_{\text{in}}) - (\bigcup_{i\in\mathcal{I}} \Sigma^{\mathcal{A}_i}_{\text{out}})$.
  - ◇ $\Sigma^{\mathcal{A}}_{\text{out}} = \bigcup_{i\in\mathcal{I}} \Sigma^{\mathcal{A}_i}_{\text{out}}$
  - ◇ $\Sigma^{\mathcal{A}}_{\text{int}} = \bigcup_{i\in\mathcal{I}} \Sigma^{\mathcal{A}_i}_{\text{int}}$
- *The start channel is $\sigma_0^{\mathcal{A}_0}$.*
- *The signaling range function, $\Pi^{\mathcal{A}}$, is defined as follows:*
  - ◇ *If $\sigma \in \Sigma^{\mathcal{A}}_{\text{in}}$ then $\Pi^{\mathcal{A}}(\sigma) = \bigcap_{\{i:\sigma\in\Sigma^{\mathcal{A}_i}_{\text{in}}\}} \Pi^{\mathcal{A}_i}(\sigma)$, where $i \in \mathcal{I}$*
  - ◇ *If $\sigma \in \Sigma^{\mathcal{A}_i}_{\text{loc}}$ then $\Pi^{\mathcal{A}}(\sigma) = \Pi^{\mathcal{A}_i}(\sigma)$, where $i \in \mathcal{I}$.*
- *The set of states is given by: $\Theta^{\mathcal{A}} = \Theta^{\mathcal{A}_0} \times \Theta^{\mathcal{A}_1} \times \ldots \times \Theta^{\mathcal{A}_i} \times \ldots,$ where $i \in \mathcal{I}$.*
- *The set of computational steps, $\Lambda^{\mathcal{A}}$, is defined as follows:*
  - ◇ $\Lambda^{\mathcal{A}} = \{(\vec{\sigma_1}, \pi, \vec{\sigma_2}) : \forall i \in \mathcal{I}, \quad \text{if } \pi \in \Pi^{\mathcal{A}_i}(\Sigma^{\mathcal{A}_i}) \text{ then } (\vec{\sigma_1}[i], \pi, \vec{\sigma_2}[i]) \in \Lambda^{\mathcal{A}_i}, \text{ else } \vec{\sigma_1}[i] = \vec{\sigma_2}[i]\}.$
- *The set of timing constraints, Υ, is defined as follows:*
  - ◇ $\Upsilon^{\mathcal{A}} = \{(\sigma_k, \sigma'_k, \delta_k, \Theta_k) : \exists i \in \mathcal{I}, (\sigma_k, \sigma'_k, \delta_k, \Theta) \in \Upsilon^{\mathcal{A}_i},$ where $\Theta_k = \Theta^{\mathcal{A}_0} \times \Theta^{\mathcal{A}_1} \times \ldots \times \Theta^{\mathcal{A}_{i-1}} \times \Theta \times \Theta^{\mathcal{A}_{i+1}} \times \ldots,$ where $i \in I\}$.

The following Lemma establishes the relationship between the behavior of the composed TRA and the behaviors of its constituent TRAs.

Lemma 4 *Let $\mathcal{A}$ be the composition of a collection of strongly compatible* TRA*s $\{\mathcal{A}_i : i \in \mathcal{I}\}$. If $\beta \in pref(behs(\mathcal{A}))$ then $\beta|\Sigma_{\text{ext}}^{\mathcal{A}_i} \in pref(behs(\mathcal{A}_i))$ for all $i \in \mathcal{I}$.*

Proof: The proof is by induction on the length l of β. Without loss of generality, we only consider a constituent TRA $\mathcal{A}_j$, for some $j \in \mathcal{I}$.

<u>Base</u> ($l = 0$)

◇ By definition, the null prefix is a prefix of any behavior in $behs(\mathcal{A}_j)$.

<u>Induction</u> ($l > 0$)

◇ Assume that the Lemma is valid for $(l-1)$-long prefixes of behaviors of $\mathcal{A}$. That is, for any $(l-1)$-long sequence of events $\beta \in pref(behs(\mathcal{A}))$, $\beta|\Sigma_{\text{ext}}^{\mathcal{A}_j} \in pref(behs(\mathcal{A}_j))$.

◇ Any l-long prefix of a behavior of $\mathcal{A}$ can be rewritten as $\beta\langle\pi : t\rangle$ for some $(l-1)$-long sequence of events β and for some event $\langle\pi : t\rangle$, where $t \in \Re$ and $\pi \in \Pi(\Sigma_{\text{ext}}^{\mathcal{A}})$. From the induction assumption we get that $\beta|\Sigma_{\text{ext}}^{\mathcal{A}_j} \in behs(\mathcal{A}_j)$ (1)

◇ Three possibilities exist for the event $\langle\pi : t\rangle$.

a. π <u>is not an action of</u> $\mathcal{A}_j$:
From (1), it follows that $\beta\langle\pi : t\rangle|\Sigma_{\text{ext}}^{\mathcal{A}_j} = \beta|\Sigma_{\text{ext}}^{\mathcal{A}_j} \in pref(behs(\mathcal{A}_j))$. (2.a)

b. π <u>is an input action of</u> $\mathcal{A}_j$:
From (1) and from the input-enabled property of TRAs, it follows that $\beta\langle\pi : t\rangle|\Sigma_{\text{ext}}^{\mathcal{A}_j}$ must be a prefix of some behavior of $\mathcal{A}_j$. (2.b)

c. π <u>is an output action of</u> $\mathcal{A}_j$:
The definition of strongly compatible TRAs guarantees that the fan-in of any local channel of a composition is exactly one. Since π is an output action of $\mathcal{A}_j$, it follows that the event $\langle\pi : t\rangle$ could not have been produced by any other TRA except for $\mathcal{A}_j$. This argument, along with (1), imply that $\beta\langle\pi : t\rangle|\Sigma_{\text{ext}}^{\mathcal{A}_j}$ must be a prefix of some behavior of $\mathcal{A}_j$. (2.c)

◇ Collectively, statements (2.a), (2.b), and (2.c) prove the induction step, thus, concluding the proof of the Lemma.

■

Lemma 5 *Let $\mathcal{A}$ be the composition of a collection of strongly compatible* TRA*s $\{\mathcal{A}_i : i \in \mathcal{I}\}$. If channels $\sigma, \sigma' \in \Sigma^{\mathcal{A}}$ conflict in $\mathcal{A}$ then they conflict in some* TRA *$\mathcal{A}_j$, where $j \in \mathcal{I}$.*

Proof: *(sketch)*
The proof can be constructed by assuming that σ and σ' conflict in $\mathcal{A}$ but not in any other TRA from the collection $\{\mathcal{A}_i : i \in \mathcal{I}\}$ and showing that such an assumption, along with Definition 2 and the construction of $\Lambda^{\mathcal{A}}$ using Definition 19, implies a contradiction. ■

Lemma 6 *The composition of a collection of strongly compatible proper* TRA*s is proper.*

Proof: Directly from Definition 4 and Lemma 5. ■

Lemma 7 *Let $\mathcal{A}$ be the composition of a collection of strongly compatible* TRA*s $\{\mathcal{A}_i : i \in \mathcal{I}\}$. Two channels $\sigma, \sigma' \in \Sigma^{\mathcal{A}}$ belong to the same component of $\mathcal{A}$ only if they belong to the same component of some* TRA *$\mathcal{A}_j$, where $j \in \mathcal{I}$.*

Proof: Directly from Definition 5 and Lemma 5. ■

The TRA composition operation is more general than those reported in [Lync88b, Tutt88, Best90b] in that it allows the specification of both *parallel* and *sequential* composition. In particular, the introduction of the *start channel* permits the execution of two TRAs to be concurrent if they share the same start channel, or to be serialized if the start channel of one (child) is an output of the other (parent). Through appropriate composition, our model is capable of representing all of the composition operations in [Lyon89, Lyon90].

Chapter 4

TRA-based Specification

$\mathcal{CLEOPATRA}$, a $\mathcal{C}$-based $\mathcal{L}$anguage for the $\mathcal{E}$vent-driven $\mathcal{O}$bject-oriented $\mathcal{P}$rototyping of $\mathcal{A}$synchronous $\mathcal{T}$ime-constrained $\mathcal{R}$eactive $\mathcal{A}$utomata, is a convenient language for the specification of embedded systems under the TRA *formalism. $\mathcal{CLEOPATRA}$ specifications are executable and can be transformed, mechanically and unambiguously, into formal* TRA *objects for verification purposes.*

In this chapter, we introduce *CLEOPATRA*[1] a TRA-based specification language. We present a subset of its syntax, which would allow us to specify embedded systems and properties to be verified thereof in a convenient way. We establish the soundness of *CLEOPATRA*'s semantics by showing that any *CLEOPATRA* specification maps unambiguously to a formal TRA object. We defer the presentation of the executable aspects[2] of *CLEOPATRA* to a later chapter.

4.1 *CLEOPATRA*: A Specification Language

In *CLEOPATRA*, systems are specified as interconnections of TRA objects. Each TRA object has a set of *state variables* and a set of *channels.* Time-constrained causal relationships between events occuring on the different channels, and the computations (state transitions) that they trigger, are specified using *Time-constrained Event-driven Transactions* (TETs). The behavior of a TRA object is described using TETs. TRA objects can be composed together to specify more complex TRAs.

4.1.1 Classes and Objects

A TRA object specification in *CLEOPATRA* consists of two components: a header and a body. An object's header specifies its name, the parameters needed for its instantiation, and its signature. An object's body specifies its behavior. In its simplest form, this entails the specification of the TRA's state space and its potentially time-constrained set of reactions to the different events visible to it. More complex behaviors include (among others) the specification of: internal channels, initialization code, and interconnection of local (composed) objects. Figure 4.1 shows a BNF-like description of a TRA in *CLEOPATRA*.

[1]*CLEOPATRA* is acronym for *C*-based *L*anguage for the *E*vent-driven *O*bject-oriented *P*rototyping of *A*synchronous *T*ime-constrained *R*eactive *A*utomata.

[2]We have developed a compiler that allows specifications written in *CLEOPATRA* to be compiled and executed in simulated time under a UNIX^{TM} environment, thus providing a valuable tool for validation purposes. Real-time behaviors can be obtained by executing the compiled *CLEOPATRA* specifications under a VxWorks^{TM} real-time kernel, thus making of *CLEOPATRA* a programming language suitable for implementation purposes.

```
<tra-object> := <tra-header> ‘{’ <tra-body> ‘}’
<tra-header> := ‘TRA-class’ <tra-name> {‘(’ <tra-params-spec> ‘)’} <signature>
<tra-params-spec> := {<type> <param-id> {‘;’ <tra-params-spec>}}
<signature> := {<ch-list-spec>} ‘->’ {<ch-list-spec>}
<ch-list-spec> := <ch-id> ( <type> ) {‘,’ <ch-list-spec>}
<type> := ‘int’ | ‘double’ | ‘bool’ | ...
<tra-body> := {<declarations>} {<init>} {<transactions>}
<declarations> := {<state>} {<internal>} {<included>}
<state> := ‘state:’ <state-var-def>
<state-var-def> := <type> <var-list-def> ‘;’ {<statevar-def>}
<var-list-def> := <var-id> {‘=’ <constant-exp>} {‘,’ <var-list-def>}
<internal> := ‘internal:’ <signature>
<included> := ‘included:’ <included-objects>
<included-objects> := <tra-instantiation> ‘;’ {<included-objects>}
<tra-instantiation> := <tra-name> {‘(’ <actual-param-list> ‘)’} <ext-binding>
<actual-param-list> := <constant-exp> {‘,’ <actual-param-list>}
<ext-binding> := {<ch-list>} ‘->’ {<ch-list>}
<ch-list> := <ch-id> {‘,’ <ch-list>}
<init> := <code>
<transactions> := {<xact> {<transactions>}}
<xact> := <xact-header> ‘:’ <xact-body>
<xact-header> := {<trigger-list>} ‘->’ <out-sig-spec>
<trigger-list> := <in-sig-spec> {‘,’ <trigger-list>}
<in-sig-spec> := <ch-id> ‘(’ {<var-id>} ‘)’
<out-sig-spec> := <ch-id> ‘(’ {<exp>} ‘)’
<xact-body> := <act> | ‘{’ <acts> ‘}’
<acts> := <act> {<acts>}
<act> := <computation> | {<condframe>} <fire-acts> | {<timeframe>} <fire-acts>
<computation> := ‘commit’ ‘{’ <code> ‘}’ | ‘do’ ‘{’ <code> ‘}’
<condframe> := ‘unless’ ‘(’<cond>‘)’ | ‘while’ ‘(’<cond>‘)’
<timeframe> := <closed-timeframe> | <open-timeframe>
<closed-timeframe> := ‘within’ ‘[’<constant-exp>‘~’<constant-exp>‘]’
<open-timeframe> := ‘before’ <constant-exp> | ‘after’ <constant-exp>
```

Figure 4.1: Partial Syntax of a TRA specification in *CLEOPATRA*

In *CLEOPATRA*, TRAs are defined in *classes*. For example, Figure 4.2 shows the *CLEOPATRA* specification of the class of integrators that use trapezoidal approximation.

```
TRA-class integrate(double TICK, TICK_ERROR)
     in(double) -> out(double)
{
 state:
  double x0 = 0, x1 = 0, y = 0;
 act:
  in(x1) -> :
    ;
  init(),out() -> out(y):
    within [TICK-TICK_ERROR~TICK+TICK_ERROR]
      commit { y = y+TICK*(x0+x1)/2; x0 = x1; }
}
```

Figure 4.2: Specification of the class of integrators that use the trapezoidal rule.

TRA classes are parametrized. For instance, the specification of `integrate` given in Figure 4.3 includes the parameters `TICK`, and `TICK_ERROR`, which have to be specified before *instantiating* an object from that class.

The header of a TRA class determines its external signature and signaling range function. For example, any TRA from the class `integrate` specified in Figure 4.2 has a signature consisting of an input channel `in` and an output channel `out`. Both `in` and `out` carry actions whose values are drawn from the set of reals. In *CLEOPATRA*, the start channel of any given TRA-class is called `init`. Start channels do not have to be explicitly included in the header of a TRA-class. For example, in the definition of the `integrate` TRA-class given in Figure 4.2, there is no mention of any `init` channels in the external signature specified in the header, yet, `init` is used later in the body of `integrate`.

The body of a TRA class determines the behavior of objects from that class. Such a behavior can be either *basic* or *composite*. The description of a basic behavior involves the specification of a state space in the `state:` section, the specification of an initialization of that space in the `init:` section, and the specification of a set of Time-constrained Event-driven Transactions in the `act:` section. The behavior of an object belonging to the TRA-class `integrate` shown in Figure 4.2 is an example of a basic behavior. Composite behaviors, on the other hand, are specified by composing previously defined, simpler TRA-classes together in the `include:` section. For example, in Figure 4.3, the class `ramp` is defined by composing the `integrate` and `constant`[3] classes together.

```
TRA-class constant(double VAL, TICK, TICK_ERROR)
  -> out(double)
{
 act:
  init(), out() -> out(VAL):
    within [TICK-TICK_ERROR~TICK+TICK_ERROR]
      ;
}


TRA-class ramp
  -> y(double)
{
 internal:
  x(double) -> ;

 include:
  constant -> x() ;
  integrate x() -> y() ;
}
```

Figure 4.3: *CLEOPATRA* specification of a ramp generator.

[3]The behavior of an object from the `constant` class is to signal the value `VAL` on its only output channel `out` every `TICK` $\pm$ `TICK_ERROR` units of time.

4.1.2 Time-constrained Event-driven Transaction

In CLEOPATRA, the time-constrained causal relationships between events occuring on the different channels of a TRA-class, and the computations (state transitions) that they trigger, are specified using *Time-constrained Event-driven Transactions* (TET) A TET describes the reaction of a TRA to a subset of events. Such a reaction might involve responding to triggers and/or firing action(s). Figure 4.4 explains the relation between the triggering and firing of actions using TETs.

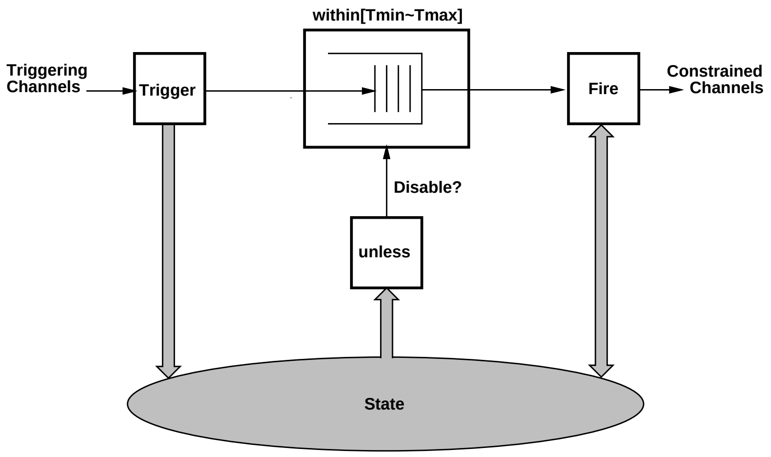


Figure 4.4: Time-constrained Event-driven Transaction (TET).

The description of a TET consists of two parts: a header and a body. The header of a TET specifies a set of triggering channels (trigger section) and a controlled channel (fire section). The trigger section specifies the effect of the triggering actions on the state of the TRA. In particular, it specifies at most one state variable (per triggering channel) where the value of a trigger on that channel is to be recorded. The fire section specifies the action value to be signaled on the controlled channel as a result of firing the TET. This value can be any expression on the state of the TRA. An absent expression means that a random value from the signaling range of the controlled channel is to be signaled. The body of a TET describes possible reactions to the TET triggers. Each reaction is associated with a disabling condition, a time constraint, and a state transformation schema. For example, consider the following TETs of the `integrate` class shown in Figure 4.2.

```
  in(x1) -> :
    ;
  init(),out() -> out(y):
    within [TICK-TICK_ERROR~TICK+TICK_ERROR]
      commit { y = y+TICK*(x0+x1)/2; x0 = x1; }
```

The first TET in the specification above is an example of a transaction with only a trigger section. Every time an action is signaled on the input channel `in`, its value is stored in the state variable `x1`, thus, resulting in a potential input transition. The second TET, on the other hand, is an example of a transaction with both a trigger section and a fire section. In particular, every time an action is signaled on one of the triggering channels (`init` or `out`) an output action is fired on `out` after a delay of `TICK` $\pm$ `TICK_ERROR` units of time elapses.

A TET with no triggering section is triggered every time an action is signaled on any channel of the TRA. In other words, its trigger set is considered to be the same as the TRA's signature. TETs of this sort are useful to specify iterative computations of *fixed points*. For example, consider the specification of the TRA-class `factorial` shown in Figure 4.5. The second TET of that class, defines a fixed point. In particular, once triggered by the occurence of a `request(x)` action, this TET will continue to be triggered and to fire (thus triggering itself again) until reaching a fixed point in which the value of the state variable `x` ceases to be positive. At this point, the value of the state variable `f` would have accumulated the value of the factorial function for the original request.

As we mentioned before, each reaction in the body of a TET is associated with three pieces of information: A disabling condition, a time constraint, and a state transformation schema.

The disabling condition (unless clause) is a boolean expression (predicate) on the state of the TRA.[4] In order to be committed, a reaction's disabling condition has to remain `false` from when the reaction is triggered until it commits. In other words, an intended reaction is aborted if at any point in time after its triggering (scheduling), the disabling condition becomes `true`. The absence of a disabling condition in a reaction implies that, once scheduled, it cannot be disabled.

[4]No side effects are permitted in the evaluation of this condition.

```
TRA-class fact
  request(int) -> result(int)
{
 state:
  int x = -1, f = 0;
 act:
  request(x) -> :
    commit { f = 1 ; }
  -> :
    unless(x<1)
      commit { f=f*x ; x-- ; }
  -> result(f):
    unless (x>0)
      ;
}
```

Figure 4.5: *CLEOPATRA* specification of the factorial computation.

The time constraint (within clause), determines a lower and upper bound for the real-time delay between scheduling a reaction and committing it. Only constant expressions are allowed to be used in the specification of time bounds. Open, closed, and semi-closed time intervals can be used provided they specify an interval of time from the set $\mathcal{D}$.[5] The absence of a time constraint from a TET specification implies that the causal relationship between the trigger and its effect is unconstrained in time. A lower bound of 0 and an upper bound of ∞ is assumed in such cases.

The state transformation schema (commit clause) specifies a *method* for computing the next state of the TRA once a reaction is committed. We adopt a C-like syntax for the specification of TET methods. Statements in a TET method are executed sequentially. The state transition caused by the execution of a TET method is assumed to be atomic and instantaneous. An absent commit clause implies that committing the reaction does not cause any state changes.

[5]Current *CLEOPATRA* processors accept only dense intervals of three forms: $(0, T_u)$, (T_l, ∞), or $[T_l, T_u]$, where $T_u > T_l \geq 0$. These are introduced using the `before`, `after`, and `within` clauses, respectively.

4.1.3 An Example

Figure 4.6 shows the specification of a finite length FIFO element in *CLEOPATRA*. The behavior of the FIFO is such that values fed into it are delayed for some amount of time before producing them as outputs.

```
TRA-class fifo(int N)
   in(float) -> out(float), overflow(), ack()
{
 state:
  float y[N];
  int i, j;
  bool f;

 act:
  init() -> ack():
   before DLY_MIN
    commit { i = 0; j = 0; f = FALSE; }
  in(y[i]) -> ack():
   before DLY_MIN
    commit { i = (i+1)%N ; if (i==j) f = TRUE ; }
  in() -> out(y[j]):
   unless (f)
    within [DLY_MIN~DLY_MAX]
     commit { j = (j+1)%N ; }
  in() -> overflow():
   unless (!f)
    within [DLY_MIN~DLY_MAX]
     ;
}
```

Figure 4.6: *CLEOPATRA* specification of a finite length FIFO delay element.

The header of the `fifo` TRA-class identifies the channel `in` as input, and the channels `out`, `ack` and `overflow` as outputs. Although not explicitly specified as such, the channel `init` (the start channel) is assumed to be an input channel. The signaling range for channels `in` and `out` is the set of floating point numbers, whereas the signaling range for channels `ack` and `overflow` consists of only one value.

The body of the `fifo` TRA-class contains two sections. In the `state:` section, the state space of a `fifo` object is described by four state variables: a vector `y[]` of `N` floating point values, two integer values `i` and `j`, and a boolean value `f`. In the `act:` section, the behavior of a `fifo` object is described by four TETs, each of which underscores a causal relationship between the events triggering its execution and those resulting from its execution.[6]

The first TET in the body of the FIFO establishes a causal relationship between events signaled on `init` and and those signaled on `ack`. In particular, firing an action on `init` (the trigger) *causes* the firing of an action on `ack` (the result) after a a delay of at most `DLY_MIN`. The second TET establishes a similar causal relationship between events signaled on `in` and `ack`. The third TET establishes a causal relationship between events signaled on `in` and `out`. In particular, firing an action action on `in` *causes* the firing of an action on `out` after a delay of at least `DLY_MIN` and at most `DLY_MAX` elapses, provided that the FIFO did not overflow as of the last initialization. The causal relationship that the fourth TET establishes can be explained similarly.

Each TET in a TRA-class specifies up to two possible state transitions. Consider, for example, the second TET in the FIFO specification given in Figure 4.6. In response to a trigger on `in`, the value of the triggering signal is stored in the state variable `y[i]`, thus resulting in a possible state change. Notice that this transition cannot be blocked or delayed; it is an *input transition*. The second state transition, an *output transition*, occurs with the firing of an action on `ack`, resulting in the adjustment of the values of the state variables `i` and `f`. Notice that the value of the action signaled on a local (output or internal) channel does not reflect the state change associated with it. For instance, in the fourth TET of Figure 4.6, the value signaled on the `out` channel, namely `y[j]`, does not reflect the changes introduced in the `commit` clause, namely advancing the pointer `j`.

It is important to realize that `fifo` objects will behave as expected only if inputs from the environment meet certain conditions. In particular, the value of the index `i` is not incremented as a result of an input on the channel `in` until at least `DLY_MIN` units of

[6] In other words, between input and output transitions.

time elapse following the signaling of that input. It follows that an erroneous behavior will result if two or more events are signaled on the channel `in` in a duration of time shorter than `DLY_MIN`. To avoid such a malignant behavior, the environment must wait for an acknowledgment `ack()`[7] or else, must wait for at least `DLY_MIN` before signaling a new input. In the next chapter, we show how such conditions can be formally verified.

4.2 Relationship between $\mathcal{CLEOPATRA}$ and the TRA model

The correspondence between $\mathcal{CLEOPATRA}$ and the TRA formalism is straightforward. Every $\mathcal{CLEOPATRA}$ specification corresponds to a formal TRA object, but not every TRA object is describable in $\mathcal{CLEOPATRA}$. In a sense, $\mathcal{CLEOPATRA}$ specifications are *sound* but *not complete* with respect to the TRA model.

4.2.1 Soundness

The following Lemma establishes a mapping from $\mathcal{CLEOPATRA}$ TRA-classes to their corresponding formal TRA objects. This amounts to a formal specification of $\mathcal{CLEOPATRA}$'s semantics.

Lemma 8 *Any instantiation of a $\mathcal{CLEOPATRA}$ TRA-class $\mathcal{C}$ specifies a formal TRA sextuple $\mathcal{A} = (\Sigma, \sigma_0, \Pi, \Theta, \Lambda, \Upsilon)$.*

Proof: (sketch)
We prove the Lemma by showing how to construct the sextuple $\mathcal{A} = (\Sigma, \sigma_0, \Pi, \Theta, \Lambda, \Upsilon)$ for any TRA class $\mathcal{C}$. The proof considers basic TRA-classes only. Composite TRA-classes, which include other classes in their specifications, can be constructed using the composition, hiding, and renaming operations on TRAs.

- ◇ The signature Σ of $\mathcal{A}$ can be deduced from the external signature of $\mathcal{C}$ specified in the header, and its internal signature specified in the body (if any). If not explicitly specified, the channel `init` is added to the input signature.

[7] An `ack()` event is signaled when the previous input has been processed.

- ◇ The start channel σ_0 is `init`.
- ◇ The signaling range $\Pi(\sigma)$ for any channel σ consists of all the values allowed by the declared type of σ in $\mathcal{C}$.
- ◇ The state space Θ of $\mathcal{A}$ is simply the cross product of all the state variables of $\mathcal{C}$.
- ◇ The set of computational steps in Λ is obtained as follows:
  - For every possible state $\theta \in \Theta$ and for every possible value u that can be signaled on the start channel σ_0, we define a computational step for $\mathcal{A}$. In particular, let θ_0 be the state resulting from executing the `init:` section[8] of $\mathcal{C}$ starting from θ under $\sigma_0(u)$, then the computational step should be on the form $(\theta, \sigma_0(u), \theta_0)$.
  - For every possible state $\theta \in \Theta$ and for every possible value u that can be signaled on any triggering channel[9] $\sigma \neq \sigma_0$ of any TET of $\mathcal{C}$, we define a computational step for $\mathcal{A}$. In particular, if V is the set of state variables specified to hold the value of the trigger σ, then the computational step should be on the form $(\theta, \sigma(u), \theta')$, where θ' is identical to θ except for the value of any $v \in V$ which should be made equal to u in θ'. If no state variables are specified, then $\theta' = \theta$.
  - For every reaction of a TET in $\mathcal{C}$ and for every state $\theta \in \Theta$, for which the reaction's disabling condition evaluates to *false*, we define a computational step for every possible value fired on the TET's controlled channel σ and for every possible next state. In particular, if $p(\theta)$ is the value to be signaled on the controlled channel and θ' is the state resulting from executing the reaction starting from θ, then the computational step should be on the form $(\theta, \sigma(p(\theta)), \theta')$. If no commit section is associated with the reaction then $\theta' = \theta$.
- ◇ The set of timing constraints in Υ is obtained as follows:
  - For every triggering channel[10] σ_i and every controlled channel[11] σ_i' of every TET in $\mathcal{C}$, a time constraint $(\sigma_i, \sigma_i', \delta_i, \Theta_i) \in \Upsilon$ is defined, where δ_i is the reaction's delay and Θ_i is the subset of Θ for which the reaction's disabling condition is true.[12]

■

[8]This should include as well the initializations of the state variables specified in the `state:` section of $\mathcal{C}$.

[9]The case where $\sigma = \sigma_0$ is considered an initialization and is dealt with separately (see previous step).

[10]TETs with no triggering channels are assumed to have every channel in Σ as a trigger.

[11]A dummy channel should be added to $\Sigma^{\mathcal{A}}_{\text{int}}$, for each TET with no controlled channels.

[12]For TETs with multiple reactions, a time constraint should be added for every reaction.

To illustrate the construction presented in the proof of Lemma 8, consider the specification of `sync2`, the TRA-class of 2-input synchronizers, shown in Figure 4.7. To instantiate an object of the TRA-class `sync2`, two parameters need to be specified, namely `TMIN` and `TMAX`. An `sync2` object has two input channels `in0` and `in1`, and one output channel `sync`. The absence of a *type* specification for all three channels implies that they have the default `unit` type. In addition, the start channel is (by default) named `init` and assumed to be of `unit` type. A channel whose type is `unit` is I/O compatible with channels of any other type.

```
TRA-class sync2(TMIN,TMAX)
  in0(),in1() -> sync()
{
 state:
  bool flag0, flag1 ;

 init:
  flag0 = FALSE;
  flag1 = FALSE ;

 act:
  in0() -> :
    do { flag0 = TRUE; }
  in1() -> :
    do { flag1 = TRUE; }
  -> sync():
    unless (!flag0 || !flag1)
      within [TMIN~TMAX]
        commit { flag0 = FALSE; flag1 = FALSE; }
}
```

Figure 4.7: *CLEOPATRA* specification of a 2-input synchronizer.

The body of the TRA-class `sync2` specifies that `sync2` objects have a state space consisting of four states described by 2 boolean state variables, `flag0` and `flag1`, both initially set to `FALSE`. Three TETs describe the behavior of a `sync2` object. The first and second TETs record the occurence of any actions on channels `in0` and `in1` by setting to

`TRUE` the state variables `flag0` and `flag1`, respectively. The third TET is triggered by the firing of any action on any channel of the TRA. Once triggered, and if the TRA is in the state where both `flag0` and `flag1` are `TRUE`, this TET will result in the firing of an action on the output channel `sync` after, at least `TMIN` and at most `TMAX` units of time elapse. The commitment of the third TET results in the TRA returning to its initial state where both state variables are set to `FALSE`.

From the above description, it is obvious that the only way an action is fired on the output `sync` is when *at least* one action is signaled on *both* inputs of the TRA in an interval of time at least `TMIN` and at most `TMAX` units of time back.

The correspondence between the `sync2` TRA specification in $\mathcal{CLEOPATRA}$ and its formal definition is straightforward. In particular, the sextuple $(\Sigma, \sigma_0, \Phi, \Theta, \Lambda, \Upsilon)$ of the `sync2` TRA can be constructed using Lemma 8. This is shown in Figure 4.8. Notice that, in the $\mathcal{CLEOPATRA}$-specification of Figure 4.7, a start channel – namely `init` – is implicitly assumed. In Figure 4.8, `init` is used in the definition of the computational steps of `sync2` to appropriately initialize the state of the TRA.

- $\diamond$ $\Sigma = \{\texttt{init}, \texttt{in0}, \texttt{in1}\} \cup \{\texttt{sync}\} \cup \{\}$.
- $\diamond$ $\sigma_0 = \texttt{init} \in \Sigma$.
- $\diamond$ $\Pi(\texttt{init}) = \{\}, \Pi(\texttt{in0}) = \{\}, \Pi(\texttt{in1}) = \{\}$, and $\Pi(\texttt{sync}) = \{\}$.
- $\diamond$ $\Theta = \{\texttt{FF}, \texttt{FT}, \texttt{TF}, \texttt{TT}\}$.
- $\diamond$ $\Lambda = \{(\texttt{FF}, \texttt{init}(), \texttt{FF}), (\texttt{FT}, \texttt{init}(), \texttt{FF}), (\texttt{TF}, \texttt{init}(), \texttt{FF}), (\texttt{TT}, \texttt{init}(), \texttt{FF}),$
  $(\texttt{FF}, \texttt{in0}(), \texttt{FT}), (\texttt{FF}, \texttt{in1}(), \texttt{TF}), (\texttt{FT}, \texttt{in0}(), \texttt{FT}), (\texttt{FT}, \texttt{in1}(), \texttt{TT}),$
  $(\texttt{TF}, \texttt{in0}(), \texttt{TT}), (\texttt{TF}, \texttt{in1}(), \texttt{TF}), (\texttt{TT}, \texttt{in0}(), \texttt{TT}), (\texttt{TT}, \texttt{in1}(), \texttt{TT}),$
  $(\texttt{TT}, \texttt{sync}(), \texttt{FF})\}$.
- $\diamond$ $\Upsilon = \{(\texttt{in0}, \texttt{sync}, [\texttt{TMIN}, \texttt{TMAX}], \{\texttt{FF}, \texttt{FT}, \texttt{TF}\}),$
  $(\texttt{in1}, \texttt{sync}, [\texttt{TMIN}, \texttt{TMAX}], \{\texttt{FF}, \texttt{FT}, \texttt{TF}\})\}$.

Figure 4.8: Formal TRA specification of a 2-input synchronizer.

4.2.2 Completeness

As we mentioned before, not every TRA object is describable in *CLEOPATRA*. In this section, we identify classes of TRA objects that are not expressible in *CLEOPATRA*.

Lemma 9 TRA *objects with input computational steps that depend on state information are not expressible in CLEOPATRA.*

Proof: In *CLEOPATRA*, the only possible computational step (state transition) resulting from the occurence of an input event is that of recording the signaled action in one of the state variables of the TRA object. The value assigned to such a state variable cannot depend on state information.

■

For example, in the formal TRA specification of the up/down counter given in Figure 3.6, the state transitions dictated by `cmd` actions, namely `cmd(UP)` and `cmd(DOWN)`, cannot be expressed in *CLEOPATRA*; they induce transitions that are state-dependent.

Our decision not to allow state dependent input transitions in *CLEOPATRA* is motivated by the fact that improper TRA specifications are easily verifiable if input transitions do not make use of state information. In particular, the verification process reduces to checking whether two different TET triggers use the same state variable to record the value of the triggering action, thus, giving rise to potential conflicts.[13]

Allowing state dependent input transitions in an input enabled model is not a realistic decision. In particular, such transitions involve computations dependent on the state of the TRA. To be realistic these computations must be carried, not by input actions, but by actions under the local control of the TRA in question. Restricting *CLEOPATRA* specifications to exclude state dependent input transitions makes these specifications realistic, and thus implementable. This argument motivates the exclusion of another class of TRA objects, namely clairvoyant TRAs.

[13]Refer to Definition 2 and Definition 4.

Lemma 10 TRA *objects with input computational steps that affect different components of the* TRA *depending on the input actions are not expressible in* $\mathcal{CLEOPATRA}$.

Proof: In $\mathcal{CLEOPATRA}$, the only possible computational step (state transition) resulting from the occurence of an input event is that of recording the signaled action in a predetermined subset of the state variables of the TRA object. This subset cannot depend on the value signaled.

$\blacksquare$

Another class of TRA objects that cannot be expressed in $\mathcal{CLEOPATRA}$ are those possessing *Zeno executions*, namely executions with an infinite number of events occuring in a finite stretch of time.

Lemma 11 TRA *objects that exhibit behaviors allowing Zeno executions are not expressible in* $\mathcal{CLEOPATRA}$.

Proof: A TRA object possesses a Zeno execution only if its set of time-constrained causalities is infinite. This follows immediately from the fact that only constant intervals (independent from the state of the TRA) are allowed in the specification of the time-constrained causalities. Since $\mathcal{CLEOPATRA}$ admits only a finite number of TETs, and thus time-constrained causalities, in the description of a TRA class, it follows that Zeno executions are impossible to specify in $\mathcal{CLEOPATRA}$.

$\blacksquare$

Chapter 5

TRA-based Verification

Verification is the process of establishing the correctness of a system by proving that it preserves certain desired properties. In this chapter, we present three verification techniques based on modular, functional, and hierarchical system decomposition. Using modular decomposition, each property of the entire system is verified separately. Using the functional decomposition, the problem to be solved is divided into independent sub-problems, whose implementations are verified separately. Using hierarchical decomposition, implementations are verified at varying levels of details. In a typical situation, the use of a combination of all three verification methodologies might be necessary.

5.1 Modular Decomposition

One common methodology for verifying properties of a complex system is *modular decomposition*, in which one reasons about each property of the entire system separately. In this section, we lay the groundwork for using the modular decomposition technique to verify properties of TRA-based specifications. In particular, we obtain sufficient conditions for a TRA to satisfy a given property.

The input enabling property of the TRA model forbids a system specification from controlling or constraining inputs it receives from its environment.[1] As a result, such a specification can only guarantee properties that are independent from the behavior of the environment. In computer embedded applications, this seems to be the only safe and realistic approach to be adopted. In many circumstances, however, the correct operation of a system is only expected under certain restrictions on its inputs. These restrictions may be guaranteed in the context of a known installation, where the behavior of other parts of the environment is a priori certified, or may be assumed by a problem statement, where the correct behavior of a solution is required only under a set of specific conditions.

For example, consider the up/down counter $\mathcal{C}$ of Figure 3.6 and assume that, as a safety condition, $\mathcal{C}$ is required to produce at least one output action in the interval between any two inputs. Obviously, such a requirement cannot be guaranteed without restricting the behavior of the environment feeding the `cmd` signal to the counter. In particular, it can be seen that an unsafe behavior will result if two or more `UP` (or `DOWN`) actions are fired on the `cmd` channel within less than 1.9 units of time.

Now, assume that in a given installation, $\mathcal{C}$ is composed with a subsystem $\mathcal{X}$ that generates `cmd` actions at a slower rate, or a subsystem $\mathcal{Y}$ that issues a new counting request only after it receives the response of $\mathcal{C}$ to its previous request. The $\mathcal{CLEOPATRA}$ specification of both $\mathcal{X}$ and $\mathcal{Y}$ is shown in Figure 5.1. Obviously, in these restricted environments, the aforementioned safety condition can be indeed certified.[2]

[1]Except for the requirement that two events on the same channel cannot be simultaneous.

[2]In particular, both compositions $\mathcal{X} \times \mathcal{C}$, and $\mathcal{Y} \times \mathcal{C}$ satisfy the requirement that two `cmd` events will be

```
TRA-class X
  -> cmd(enum{UP,DOWN})
{
  act:
   init(),cmd() -> cmd():
     ;
}
```

```
TRA-class Y
  cnt() -> cmd(enum{UP,DOWN})
{
  act:
   init(),cnt() -> cmd():
     ;
}
```

Figure 5.1: $\mathcal{CLEOPATRA}$ specification of the installations $\mathcal{X}$ and $\mathcal{Y}$.

A useful notion for discussing the aforementioned restrictions is that of a TRA *preserving* a property. This notion was first introduced in [Lync88b] to study fair behaviors of discrete event systems using the Input/Output Automata model. In this section we generalize this notion to suit the TRA model.

A property $\mathcal{P}$ defines a possibly infinite set of sequences over a given alphabet (or signature). Properties can be defined by specifying them as TRAs, or, alternately, by describing the set of behaviors they allow. Defining a property by specifying it as a TRA has been termed in [Zave82] as the *functional* specification approach, as opposed to the *conventional* black-box approach.[3]

Definition 20 *A* TRA $\mathcal{P}$ *is said to define a property for a* TRA $\mathcal{A}$ *if and only if* $\sigma_0^{\mathcal{A}} \in \Sigma_{\text{ext}}^{\mathcal{P}}$.

A TRA $\mathcal{A}$ is said to preserve a property $\mathcal{P}$ if it is not the "first" to violate it. Once the property $\mathcal{P}$ is violated, $\mathcal{A}$ is under no obligation to behave in any specific way. That is, the TRA $\mathcal{A}$ behaves according to the property $\mathcal{P}$ until the environment, or possibly another TRA composed with $\mathcal{A}$, violates that property.

separated by at least one `cnt` event.

[3]In contrast to the conventional approach where a problem specification is formulated in terms of the required behavior, the operational approach calls for the specification of the problem by formulating a system to *solve* it. The formulated system is given in terms of implementation-independent structures that, once implemented, would generate the required behavior.

Definition 21 *A TRA $\mathcal{A}$ preserves property $\mathcal{P}$ if whenever $\beta|\Sigma^{\mathcal{P}}_{\text{ext}} \in pref(behs(\mathcal{P}))$ and $\beta\langle\pi_1, \pi_2, \ldots, \pi_m : t\rangle|\Sigma^{\mathcal{A}}_{\text{ext}} \in pref(behs(\mathcal{A}))$, then $\beta\langle\pi_1, \pi_2, \ldots, \pi_m : t\rangle|\Sigma^{\mathcal{P}}_{\text{ext}} \in pref(behs(\mathcal{P}))$, where $\pi_i \in \Pi(\Sigma^{\mathcal{A}}_{\text{out}})$, $1 \leq i \leq m$, $t \in \Re$ and β is any sequence of events over the signature $\Sigma^{\mathcal{A}}_{\text{ext}} \cup \Sigma^{\mathcal{P}}_{\text{ext}}$.*

Lemma 12 *Let $\{\mathcal{A}_i : i \in \mathcal{I}\}$ be a collection of strongly compatible TRAs. If $\mathcal{A}_i$ preserves property $\mathcal{P}$ for all $i \in \mathcal{I}$, then the composition $\prod_{i\in\mathcal{I}} \mathcal{A}_i$ preserves $\mathcal{P}$.*

Proof: The proof is by contradiction.

- ◇ Assume that $\mathcal{A}_i$ preserves property $\mathcal{P}$ for all $i \in \mathcal{I}$. Moreover, assume that the composition $\mathcal{A} = \prod_{i\in\mathcal{I}} \mathcal{A}_i$ does not preserve $\mathcal{P}$. (1)
- ◇ From (1) and Definition 21, there must exist a sequence of events β for which $\beta|\Sigma^{\mathcal{P}}_{\text{ext}} \in pref(behs(\mathcal{P}))$, $\beta\langle\pi : t\rangle|\Sigma^{\mathcal{A}}_{\text{ext}} \in pref(behs(\mathcal{A}))$, and $\beta\langle\pi : t\rangle|\Sigma^{\mathcal{P}}_{\text{ext}} \notin pref(behs(\mathcal{P}))$, where $\pi \in \Pi(\Sigma^{\mathcal{A}}_{\text{out}}), t \in \Re$. (2)
- ◇ Without loss of generality, let $\pi \in \Pi(\Sigma^{\mathcal{A}_j}_{\text{out}})$ for some $j \in \mathcal{I}$.
- ◇ From Lemma 4, we get that $\beta\langle\pi : t\rangle|\Sigma^{\mathcal{A}_j}_{\text{ext}} \in pref(behs(\mathcal{A}_j))$. The condition given in Definition 21 coupled with the assumption that $\mathcal{A}_j$ preserves $\mathcal{P}$ made in (1) necessitate that $\beta\langle\pi : t\rangle|\Sigma^{\mathcal{P}}_{\text{ext}} \in pref(behs(\mathcal{P}))$. (3)
- ◇ Statements (2) and (3) contradict each other, thus, proving the Lemma.

■

The notion of a system preserving a property is a weak version of the implementation relationship between TRAs. Following Definition 12, a system $\mathcal{A}$ strongly implements a property $\mathcal{P}$ if $behs(\mathcal{A})|\Sigma^{\mathcal{P}}_{\text{ext}} \subseteq behs(\mathcal{P})$. In other words, the TRA $\mathcal{A}$ implements the property $\mathcal{P}$ if $\mathcal{A}$ preserves $\mathcal{P}$ in all possible behaviors – independently from the environment's behavior.

The following theorem establishes sufficient conditions for a composition of TRAs to implement a property.

Theorem 3 *A set of sufficient conditions for the composition $\mathcal{A} = \prod_{i\in\mathcal{I}} \mathcal{A}_i$ to implement the property $\mathcal{P}$ is that:*

1. $\Sigma_{\text{in}}^{\mathcal{A}} \subseteq \Sigma_{\text{in}}^{\mathcal{P}}$.

2. $\mathcal{A}_i$ *preserves* $\mathcal{P}$*, for all* $i \in \mathcal{I}$.

Proof:

◇ Since $\mathcal{A}_i$ preserves $\mathcal{P}$, for all $i \in \mathcal{I}$, it follows from Lemma 12 that the composition $\mathcal{A}$ preserves $\mathcal{P}$. (1)

◇ Since any input channel of $\mathcal{A}$ is also an input channel of $\mathcal{P}$, it follows from the input-enabled property of $\mathcal{P}$ that $\mathcal{A}$ cannot have any input that $\mathcal{P}$ could refuse. (2)

◇ From (1) and (2), $\mathcal{A}$ cannot exhibit any behavior that $\mathcal{P}$ does not exhibit. Hence, $\mathcal{A}$ implements $\mathcal{P}$.

■

A special case of particular interest occurs when the composition in Theorem 3 is *closed*; a TRA is closed if it has no input channels except the start channel. A closed TRA can be thought of as specifying a system that is *environment independent.* In particular, if $\mathcal{S}$ is the TRA representing an embedded system, and $\mathcal{E}$ is a strongly compatible TRA, where: $\Sigma_{\text{in}}^{\mathcal{S}} = \Sigma_{\text{out}}^{\mathcal{E}} \cup \{\sigma_0^{\mathcal{S}}\}$, $\Sigma_{\text{in}}^{\mathcal{E}} = \Sigma_{\text{out}}^{\mathcal{S}} \cup \{\sigma_0^{\mathcal{E}}\}$, then the composition $\mathcal{S} \times \mathcal{E}$ is closed, and the TRA $\mathcal{E}$ is said to define an *installation* for $\mathcal{S}$.

In the counting example presented in Figure 3.6, let $\mathcal{P}$ be the property depicting the requirement that any two events on `cmd` will be separated by at least one event on `cnt`. Figure 5.2 shows the $\mathcal{CLEOPATRA}$ specification of $\mathcal{P}$. In particular, $\mathcal{P}$ has two states "ready for a `cmd` action", or "waiting for a `cnt` action". The computational steps of $\mathcal{P}$ are such that firing a `cnt` action makes the TRA move to the "ready" state, whereas firing a `cmd` moves it to the "wait" state. $\mathcal{P}$'s time constraints specify that an event on the `init` channel will cause later repeated firings on the `cnt` channel, which might cause a firing on the `cmd` channel unless $\mathcal{P}$ is in the "wait" state. Figure 5.3 shows the sextuple specification of $\mathcal{P}$ obtained through the construction given in Lemma 8.

According to the conditions of Definition 21, it can be easily shown that both the counter $\mathcal{C}$ and the installation $\mathcal{Y}$ preserve the property $\mathcal{P}$. From Theorem 3, it follows that the closed system resulting from embedding $\mathcal{C}$ in $\mathcal{Y}$, namely the composition $\mathcal{Y} \times \mathcal{C}$,

```
TRA-class P
  -> cmd(enum{UP, DOWN}), cnt()
{
  state:
   enum{Ready,Wait} f;
  act:
   init(),cnt() -> cnt():
     commit { f = Ready; }
   cnt() -> cmd():
     unless(f == Wait)
       commit { f = Wait; }
}
```

Figure 5.2: $\mathcal{CLEOPATRA}$ specification of the property $\mathcal{P}$.

- ◇ $\Sigma_{\text{in}} = \{\texttt{init}\}$, $\Sigma_{\text{out}} = \{\texttt{cmd}, \texttt{cnt}\}$, and $\Sigma_{\text{int}} = \phi$.
- ◇ $\sigma_0 = \texttt{init} \in \Sigma$.
- ◇ $\Pi(\texttt{init}) = \{\}$, $\Pi(\texttt{cmd}) = \{\texttt{UP}, \texttt{DOWN}\}$, and $\Pi(\texttt{cnt}) = \mathcal{Z}$.
- ◇ $\Theta = \{\texttt{Ready}, \texttt{Wait}\}$.
- ◇ $\Lambda = \{\ (\theta, \texttt{init}(), \theta), (\texttt{Ready}, \texttt{cmd}(c), \texttt{Wait}), (\theta, \texttt{cnt}(i), \texttt{Ready}) :$ where $\theta \in \Theta$, $c \in \Pi(\texttt{cmd})$, and $i \in \Pi(\texttt{cnt})\}$.
- ◇ $\Upsilon = \{\ (\texttt{init}, \texttt{cnt}, [0, \infty], \phi), (\texttt{init}, \texttt{cmd}, [0, \infty], \phi),$ $(\texttt{cnt}, \texttt{cnt}, [0, \infty], \phi), (\texttt{cnt}, \texttt{cmd}, [0, \infty], \{\texttt{Wait}\})\}$.

Figure 5.3: TRA-specification of the property $\mathcal{P}$.

implements $\mathcal{P}$. The same conclusion, although correct,[4] cannot be reached using Theorem 3 for the composition $\mathcal{X} \times \mathcal{C}$ since, in general, $\mathcal{X}$ does not preserve $\mathcal{P}$. This stems from the fact that the conditions in Theorem 3 are sufficient and not necessary conditions.

5.2 Functional Decomposition

System verification using functional decomposition is strongly tied to system implementation using the divide-and-conquer approach. In particular, one way of verifying an implementation is by dividing the problem to be solved into independent sub-problems, and verifying the implementation of each sub-problem separately.

Lemma 13 *Let $\mathcal{A}$ be the composition $\prod_{i\in\mathcal{I}} \mathcal{A}_i$. If β is a sequence of events from the set $\Pi(\Sigma^{\mathcal{A}}_{\text{ext}}) \times \Re$ then $\beta \in behs(\mathcal{A})$ if and only if $\beta|\Sigma^{\mathcal{A}_i}_{\text{ext}} \in behs(\mathcal{A}_i)$ for every $i \in \mathcal{I}$.*

Proof: Directly from the definition of the composition operation and the input enabled property of TRAs.

■

Theorem 4 *Let $\{\mathcal{A}_i : i \in \mathcal{I}\}$ and $\{\mathcal{P}_i : i \in \mathcal{I}\}$ be two collections of strongly compatible TRAs. If $\mathcal{A}_i$ implements $\mathcal{P}_i$, for every $i \in \mathcal{I}$, then the composition $\mathcal{A} = \prod_{i\in\mathcal{I}} \mathcal{A}_i$ implements the composition $\mathcal{P} = \prod_{i\in\mathcal{I}} \mathcal{P}_i$.*

Proof: Let $\beta \in behs(\prod_{i\in\mathcal{I}} \mathcal{A}_i)$. From Lemma 13 we get: $\beta|\Sigma^{\mathcal{A}_i}_{\text{ext}} \in behs(\mathcal{A}_i)$ for every $i \in \mathcal{I}$. Since $\mathcal{A}_i$ implements $\mathcal{P}_i$, we get: $\beta|\Sigma^{\mathcal{A}_i}_{\text{ext}} \in behs(\mathcal{P}_i)$, for every $i \in \mathcal{I}$. Furthermore, since $\mathcal{A}_i$ is an implementation of $\mathcal{P}_i$, Lemma 1 necessitates that $\Sigma^{\mathcal{A}_i} \subseteq \Sigma^{\mathcal{P}_i}$. Thus, $\beta|\Sigma^{\mathcal{P}_i}_{\text{ext}} \in behs(\mathcal{P}_i)$. From Lemma 13, we get: $\beta \in behs(\mathcal{P})$, which implies that $\mathcal{A}$ implements $\mathcal{P}$.

■

[4] The proof will be given when we discuss hierarchical decomposition.

5.3 Hierarchical Decomposition

Another methodology for the verification of complex systems is *hierarchical decomposition*, in which one reasons about the entire system at varying levels of abstractions and details. This verification approach is analogous to the *stepwise refinement* implementation approach. In this section, we lay the groundwork for using the hierarchical decomposition technique to verify properties of TRA-based specifications.

The idea behind hierarchical decomposition is to prove that a given TRA implements a second, that the second implements the third, and so on until the final TRA is shown to implement the required specifications. The transitivity of the implementation relation guarantees that the first TRA, indeed, implements the specifications. This leads to the following easy-to-prove Lemma.

Lemma 14 *The implementation relation is transitive.*

Proof: Directly from the definition of the implementation relationship. ■

In the remainder of this section, we derive a set of sufficient conditions for the (strong) implementation of a TRA by another. The idea is to come up with a mapping Ψ between the states and intentions of the two TRAs and show that any possible status succession in the implementing TRA corresponds to some possible succession in the specification TRA. Figure 5.4 illustrates that correspondence.

Our approach in establishing a mapping between a specification and its implementation is similar to the possibilities mappings proposed in [Lync88b, Lync88a] and the prophecy mappings proposed in [Merr89], except that it is complicated here by the need to preserve the timing constraints of the specification TRA.[5] The following theorem establishes a set of sufficient conditions.

[5]As a matter of fact, an alternative way of proving this theorem is to construct the Input/Output Automata equivalent to the TRAs $\mathcal{A}$ and $\mathcal{B}$ [Best90b] and prove the implementation relationship using Lynch's possibilities mapping [Lync88b].

Theorem 5 *A set of sufficient conditions for a* TRA $\mathcal{A}$ *to (strongly) implement another* TRA $\mathcal{B}$ *is that both of the following conditions are satisfied:*

1. – $\Sigma_{\text{in}}^{\mathcal{A}} = \Sigma_{\text{in}}^{\mathcal{B}} = \Sigma_{\text{in}}$, *and*
   – $\forall \sigma_i \in \Sigma_{\text{in}} : \Pi^{\mathcal{A}}(\sigma_i) = \Pi^{\mathcal{B}}(\sigma_i)$.

2. *There exist two mappings:* $\Psi_\Theta : \Theta^{\mathcal{A}} \rightarrow 2^{\Theta^{\mathcal{B}}}$, *from the set of states of* $\mathcal{A}$ *to the power set of states of* $\mathcal{B}$, *and* $\Psi_I : I^{\mathcal{A}} \rightarrow 2^{I^{\mathcal{B}}}$, *from the set of intentions of* $\mathcal{A}$ *to the power set of intentions of* $\mathcal{B}$, *such that the following conditions hold:*

   a. $\Psi_I(I_\phi^{\mathcal{A}}) = \{I_\phi^{\mathcal{B}}\}$,

   b. *Let* (θ_i, I_i) *be a reachable status of the* TRA $\mathcal{A}$, *and let* (θ'_i, I'_i) *be a reachable status of the* TRA $\mathcal{B}$, *where* $\theta'_i \in \Psi_\Theta(\theta_i)$, *and* $I'_i \in \Psi_I(I_i)$. *If for some* $m > 0$, $(\theta_i, I_i) \overset{\langle \pi_1, \pi_2, \ldots, \pi_m : t\rangle}{\longmapsto} (\theta_j, I_j)$ *is a possible status succession of* $\mathcal{A}$, *then there exists an extended status succession*[6] *for* $\mathcal{B}$ *of the form* $(\theta'_i, I'_i) \overset{\alpha}{\longmapsto} (\theta'_j, I'_j)$, *such that:*

      i. $\alpha | \Sigma_{\text{ext}}^{\mathcal{B}} = \langle \pi_1, \pi_2, \ldots, \pi_m : t\rangle | \Sigma_{\text{ext}}^{\mathcal{A}}$.
      ii. $\theta'_j \in \Psi_\Theta(\theta_j)$, *and* $I'_j \in \Psi_I(I_j)$.

Proof:

◇ The first of the above conditions guarantees that the necessary conditions of Lemma 1 are met.

◇ To conclude the proof of the theorem, we need to show that the satisfaction of the second condition is sufficient to insure a strong implementation. Basically, we have to show that the condition guarantees that any possible behavior of $\mathcal{A}$ can be generated by $\mathcal{B}$. We do so by proving two lemmas. Lemma 15 shows that partial executions of $\mathcal{A}$ result in behaviors that can be exhibited by partial executions of $\mathcal{B}$. This concludes the proof of the theorem for infinite behaviors. For finite behaviors, the proof is concluded using Lemma 16, which establishes a correspondance between the fixed points of $\mathcal{A}$ and those of $\mathcal{B}$.

■

[6]See Figure 5.4 for an illustration.

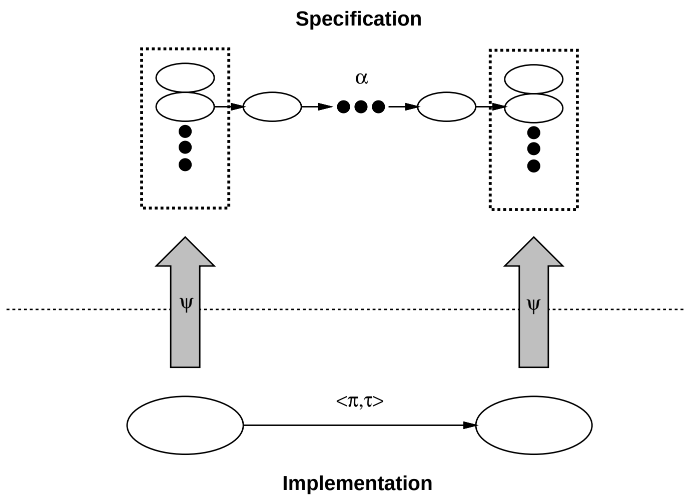


Figure 5.4: Ψ-mapping between specification and implementation.

Lemma 15 *Let $\mathcal{A}$ and $\mathcal{B}$ be two* TRA*s for which the conditions of Theorem 5 are satisfied. If e is a partial execution of $\mathcal{A}$, then there exists a partial execution e' of $\mathcal{B}$ such that:*

1. *The external behaviors exhibited throughout the partial executions e and e' are identical,*
2. *The status (θ, I) of $\mathcal{A}$ at the end of e and the status (θ', I') of $\mathcal{B}$ at the end of e' are such that: $\theta' \in \Psi_\Theta(\theta)$, and $I' \in \Psi_I(I)$.*

Proof: We prove the above lemma by induction on l the length (number of status transitions) of any partial execution of $\mathcal{A}$.

<u>Base</u> $(l = 0)$

◇ By definition, a 0-length partial execution of $\mathcal{A}$ will necessarily consist of just an initial status $(\theta_0^{\mathcal{A}}, I_\phi^{\mathcal{A}})$, for some initial state $\theta_0^{\mathcal{A}} \in \Theta^{\mathcal{A}}$.

◇ Let $\theta_0^{\mathcal{B}} \in \Theta^{\mathcal{B}}$ be a state of $\mathcal{B}$ such that $\theta_0^{\mathcal{B}} \in \Psi_\Theta(\theta_0^{\mathcal{A}})$.

◇ Obviously, both $(\theta_0^{\mathcal{A}}, I_\phi^{\mathcal{A}})$, and $(\theta_0^{\mathcal{B}}, I_\phi^{\mathcal{B}})$ are 0-length partial executions of $\mathcal{A}$ and $\mathcal{B}$, respectively, which exhibit identical external behaviors – namely the empty behavior. This concludes the proof for the first part of the Lemma for the base case.

- $\diamond$ The proof of the second part of the Lemma follows directly from the assumption that $\theta_0^{\mathcal{B}} \in \Theta^{\mathcal{B}}$ and from condition (2.a) of Theorem 5, which guarantees that $I_\phi^{\mathcal{B}} \in \Psi_I(I_\phi^{\mathcal{A}})$.

Induction[7] $(l > 0)$

- $\diamond$ Assume that the Lemma is valid for $(l-1)$-long partial executions. That is, for any $(l-1)$-long partial execution u of $\mathcal{A}$ there exists a corresponding partial execution u' of $\mathcal{B}$ such that: if α and α' are the schedules corresponding respectively to u and u', then $\alpha|\Sigma_{\text{ext}}^{\mathcal{A}} = \alpha'|\Sigma_{\text{ext}}^{\mathcal{B}}$. Also, assume that the status (θ, I) of $\mathcal{A}$ at the end of u and the status (θ', I') of $\mathcal{B}$ at the end of u' are such that $\theta' \in \Psi_\Theta(\theta)$ and $I' \in \Psi_I(I)$.

- $\diamond$ Now consider any l-long partial execution e of $\mathcal{A}$. Such a partial execution can be rewritten as the catenation of w and $\langle \pi_{l-1} : t_{l-1}\rangle(\theta_l, I_l)$, where w is a partial execution of length $l-1$ given by: $w = (\theta_0, I_0)\langle \pi_0 : t_0\rangle(\theta_1, I_1)\langle \pi_1 : t_1\rangle \ldots \langle \pi_{l-2} : t_{l-2}\rangle(\theta_{l-1}, I_{l-1})$.

- $\diamond$ Our induction assumption guarantees the existence of a partial execution w' of $\mathcal{B}$ such that the behaviors of $\mathcal{A}$ and $\mathcal{B}$ throughout w and w' are identical. (3)

- $\diamond$ Our induction assumption, also, guarantees that the status $(\theta'_{l-1}, I'_{l-1})$ at the end of w' is such that $\theta'_{l-1} \in \Psi_\Theta(\theta_{l-1})$ and $I'_{l-1} \in \Psi_I(I_{l-1})$. (4)

- $\diamond$ Now consider the status succession $(\theta_{l-1}, I_{l-1}) \overset{\langle \pi_{l-1}:t_{l-1}\rangle}{\longmapsto} (\theta_l, I_l)$. From statement (4), and from condition 2(b) of Theorem 5, there exists a schedule α_l that takes $\mathcal{B}$ from status $(\theta'_{l-1}, I'_{l-1})$ to a status (θ'_l, I'_l) such that:

  - $(\alpha_l|\Sigma_{\text{ext}}^{\mathcal{B}}) = (\langle \pi_{l-1} : t_{l-1}\rangle|\Sigma_{\text{ext}}^{\mathcal{A}})$, and (5)
  - $\theta'_l \in \Psi_\Theta(\theta_l)$ and $I'_l \in \Psi_I(I_l)$. (6)

- $\diamond$ Statements (3) and (5) prove the validity of the first part of the lemma. Statement (6) proves the second part. This concludes the proof of the induction step and the lemma.

$\blacksquare$

[7]For the sake of clarity, this proof is constructed assuming that the behavior of $\mathcal{A}$ does not contain simultaneous events.

Lemma 16 *Let $\mathcal{A}$ and $\mathcal{B}$ be two* TRA*s for which the conditions of Theorem 5 are satisfied. If e is a finite legal execution for the* TRA *$\mathcal{A}$, then there exists a finite legal execution e' for the* TRA *$\mathcal{B}$ such that the external behaviors exhibited throughout the executions e and e' are identical.*

Proof:

- ◇ Let (θ, I) be the status of $\mathcal{A}$ at the end of the execution e. Since e is finite, it follows that (θ, I) is a stable status.[8] Thus, $I = I_\phi^{\mathcal{A}}$. (1)
- ◇ Lemma 15 establishes the existence of a partial execution e' for $\mathcal{B}$, which exhibits an external behavior identical to that exhibited by e. (2)
- ◇ Also, Lemma 15 guarantees that the status (θ', I') of $\mathcal{B}$ at the end of the partial execution e' satisfies the condition that $I' \in \Psi_I(I)$. From (1), it follows that $I' \in \Psi_I(I_\phi^{\mathcal{A}})$. (3)
- ◇ From statement (3) and condition (2.a) of Theorem 5, it follows that $I' = I_\phi^{\mathcal{B}}$ is the empty intentions vector. Therefore, (θ', I') is a stable status, and the partial execution e' of $\mathcal{B}$ is also a finite legal execution. (4)
- ◇ The proof of the Lemma follows directly from (2) and (4).

■

As an example, let us focus once more on the counting example of Figure 3.6 and the installation $\mathcal{X}$ that we described before. Recall that, using modular decomposition, we were unable to verify that the composition $\mathcal{X} \times \mathcal{C}$ preserves the property $\mathcal{P}$ stating that any two events on `cmd` are separated by at least one event on `cnt`. Using the sufficient conditions of Theorem 5, such a proof can be constructed.

In particular, consider the TRA sextuples for the installation $\mathcal{X}$ and the closed system $\mathcal{X} \times \mathcal{C}$ shown in Figure 5.5 and Figure 5.6, respectively. Now, consider the mappings $\Psi_\Theta : \Theta^{\mathcal{X}\times\mathcal{C}} \rightarrow 2^{\Theta^{\mathcal{P}}}$ and $\Psi_I : I^{\mathcal{X}\times\mathcal{C}} \rightarrow 2^{I^{\mathcal{P}}}$ given below:

[8] This follows directly from the first condition of Definition 8 for legal executions.

- ◇ $\Psi_\Theta((Q,\theta_i)) = \{\texttt{Ready},\texttt{Wait}\}$, where $i \in \mathcal{Z}$.
- ◇ $\Psi_I([\{\delta_0\},\{\delta_1\},\{[t_l,t_h]\},\{[t'_l,t'_h]\}]) =$
  $\{[\{\delta_0\},\{\delta_1\},\{[t_l,t_h]\},\varepsilon],[\{\delta_0\},\{\delta_1\},\{[t_l,t_h]\},\{[t'_l,t'_h]\}]\}$, if $t'_l > t_h$,
  $\{[\{\delta_0\},\{\delta_1\},\{\delta_2\},\{[t_l,t_h]\}]\}$, otherwise.

- ◇ $\Sigma_{\text{in}} = \{\texttt{init}\}$, $\Sigma_{\text{out}} = \{\texttt{cmd}\}$, and $\Sigma_{\text{int}} = \phi$.
- ◇ $\sigma_0 = \texttt{init}$.
- ◇ $\Pi(\texttt{init}) = \mathcal{Z}$, $\Pi(\texttt{cmd}) = \{\texttt{UP},\texttt{DOWN}\}$.
- ◇ $\Theta = \{Q\}$.
- ◇ $\Lambda = \{(Q,\texttt{init}(),Q),(Q,\texttt{cmd}(c),Q)\}$, where $c \in \Pi(\texttt{cmd})$.
- ◇ $\Upsilon = \{(\texttt{init},\texttt{cmd},[4.9,5.1],\phi),(\texttt{cmd},\texttt{cmd},[4.9,5.1],\phi)\}$.

Figure 5.5: TRA-specification of the installation $\mathcal{X}$.

- ◇ $\Sigma_{\text{in}} = \{\texttt{init}\}$, $\Sigma_{\text{out}} = \{\texttt{cmd},\texttt{cnt}\}$, and $\Sigma_{\text{int}} = \phi$.
- ◇ $\sigma_0 = \texttt{init}$.
- ◇ $\Pi(\texttt{init}) = \mathcal{Z}$, $\Pi(\texttt{cmd}) = \{\texttt{UP},\texttt{DOWN}\}$, and $\Pi(\texttt{cnt}) = \mathcal{Z}$.
- ◇ $\Theta = \{(Q,\theta_i) : i \in \mathcal{Z}\}$.
- ◇ $\Lambda = (\bigcup_{i,j\in\mathcal{Z}}\{((Q,\theta_i),\texttt{init}(j),(Q,\theta_j))\}) \cup (\bigcup_{i\in\mathcal{Z}}\{((Q,\theta_i),\texttt{cmd(UP)},(Q,\theta_{i+1}))\}) \cup$
  $(\bigcup_{i\in\mathcal{Z}}\{((Q,\theta_i),\texttt{cmd(DOWN)},(Q,\theta_{i-1}))\}) \cup (\bigcup_{i\in\mathcal{Z}}\{((Q,\theta_i),\texttt{cnt}(i),(Q,\theta_i))\})$.
- ◇ $\Upsilon = \{(\texttt{init},\texttt{cnt},[1.9,2.1],\phi),(\texttt{cnt},\texttt{cnt},[1.9,2.1],\phi),(\texttt{init},\texttt{cmd},[4.9,5.1],\phi),$
  $(\texttt{cmd},\texttt{cmd},[4.9,5.1],\phi)\}$.

Figure 5.6: TRA-specification of the composition $\mathcal{X} \times \mathcal{C}$.

The mapping Ψ_Θ reflects the fact that there is no direct correspondence between the states of $\mathcal{X}\times\mathcal{C}$ and that of $\mathcal{P}$; any state of $\mathcal{X}\times\mathcal{C}$ maps to any state of $\mathcal{P}$. The mapping Ψ_I, however, reflects how the combined time constraints of $\mathcal{X}$ and $\mathcal{C}$ interrelate to guarantee the implementation of $\mathcal{P}$. In particular, the mapping Ψ_I implies that in the composition $\mathcal{X}\times\mathcal{C}$, an intention to fire an action on the `cmd` channel necessitates a similar intention in $\mathcal{P}$ only if `cnt` cannot be guaranteed to fire before `cmd`. The proof is concluded by checking that the mappings Ψ_Θ and Ψ_I satisfy the conditions of Theorem 5, a straightforward process.

Chapter 6

TRA-based Validation

Validation is the process of building the confidence of customers in the specification of a system via simulation, prototyping, and testing. In this chapter, we report on our work in developing a compiler that allows CLEOPATRA specifications to be executed in simulated time for validation purposes. In this respect, we introduce the compiler-dependent ingredients of CLEOPATRA and illustrate the use of the language in the simulation of various reactive systems.

6.1 *CLEOPATRA*: A Simulation Language

We have developed a compiler that transforms *CLEOPATRA* specifications into an event-driven simulator for validation purposes. We have used the *CLEOPATRA* compiler to simulate a variety of systems. In particular, we used it extensively to specify and analyze sensori-motor robotics applications [Best90d, Best90c] and to simulate complex behaviors of autonomous creatures [Best91a].

6.1.1 Data Types

In *CLEOPATRA*, data types are used in conjunction with state variables and channels. The type of a state variable determines the set of values that the state variable can assume. The type of a channel determines the signaling range of that channel. For state variables, *CLEOPATRA* admits all the fundamental types (`int`, `char`, `double`, ... *etc.*) and the derived types (arrays, structures, unions, pointers, ... *etc.*) of the C language.[1] For channels, it admits both fundamental and derived types of C with the exception of pointers. This is necessary to preserve the privacy respect principle, which requires state information and local actions to be invisible from outside a TRA.

CLEOPATRA is a strongly-typed language with respect to TRA inclusion, thus disallowing the composition of any TRA-classes that are not I/O compatible. In *CLEOPATRA*, implicit type conversion for channels is restricted. A type can be converted to another, only if the domain of the former is included in that of the latter. For example, a channel of type `double` can be converted to a channel of type `int`, but not vice versa. Communication between TRAs, via channels, is done with a *call by value* semantics. This includes array values. Explicit type conversion is possible for state variables,[2] but not for channels.

[1]*CLEOPATRA* also introduces new fundamental types (e.g. `string`, `unit`, `bool`).

[2]The privacy respect principle can be violated if unrestricted use of type conversion is allowed. For instance, by converting an `integer` to a pointer, a TRA object might be able to access state information of another TRA. Current *CLEOPATRA* processors do not detect such malignancies.

6.1.2 The `main` TRA-class

In *CLEOPATRA*, any TRA-class with no input channels represents a stand-alone (closed) system whose behavior is independent from the outside world; it is a world of its own. One such TRA-class, namely `main`, is singled out by *CLEOPATRA* to represent the entire system being specified. For embedded systems, a typical `main` TRA-class will simply be the composition of a programmed system, representing the control system, and an external interface, representing the environment. For example, the `main` TRA-class shown in Figure 6.1 represents the *CLEOPATRA* specification of the closed process control system shown in Figure 6.2.

The execution of a *CLEOPATRA* stand-alone system is started by instantiating an object from the TRA-class `main` at time[3] 0 and, thereafter, committing only the legal transitions dictated by the system specification and the semantics of the TRA model. Figure 6.3 shows the values signaled on the `x` and `z` channels over time.

6.1.3 Object Instantiation

The instantiation of an object from a given TRA-class entails allocating space for the object's status (state variables and intentions) and firing its initiating event. The result of firing the initiating event of an object is to modify the state variables and intentions of that object (if necessary), thus obtaining its initial status, and to instantiate any included objects that do not have explicit `init` channels. Objects with explicit `init` channels are instantiated only when their initiating events are fired.

For example, to start the execution of the system specified in Figure 6.1, a `main` object is instantiated at time 0. This will result in the instantiation of a `world` object, a `control` object, and two `monitor` objects at the same time. The instantiation of the `world` object will, similarly, result in the instantiation of a `plant` object and a `user` object.

[3]The start time of the simulation can be explicitly specified.

```
#include "sysTRA.cleo"

#define TAU 1
#define DLY 5


TRA-class user(double EPOCH)
  -> x(double)
{
  act:
   init(),x() -> x(random(0,1)):
     within [0.8*EPOCH~1.2*EPOCH]
       ;
}

TRA-class plant(double GAIN)
  y(double) -> z(double)
{
  state:
   double drive = 0, val = 0 ;

  act:
   y(drive) -> :
     ;
   init(), z() -> z(val):
     within [0.9*DLY~1.1*DLY]
       commit {
         val = val + GAIN*drive ;
       }
}
```

```
TRA-class world()
  y(double) -> x(double), z(double)
{
  include:
   user(300) -> x() ;
   plant(1.5) y() -> z() ;
}

TRA-class control()
  x(double), z(double) -> y(double)
{
  state:
   double s = 0, f = 0;

  act:
   x(s), z(f) -> y(s-f):
     within [0.95*TAU~1.05*TAU]
       ;
}

TRA-class main() ->
{
 internal:
  -> x(double),y(double),z(double)
 include:
  world y() -> x(), z() ;
  control x(), z() -> y() ;
  fmonitor("x.dat") x() -> ;
  fmonitor("z.dat") z() -> ;
}
```

Figure 6.1: The `main` TRA-class.

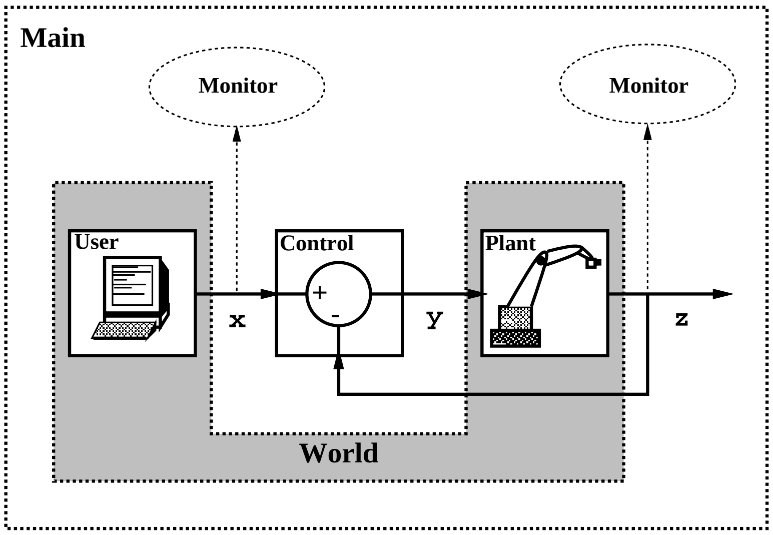


Figure 6.2: A stand-alone process control system.

6.1.4 System-defined TRA-classes

A library of system-defined TRA-classes is available for debugging and performing I/O in *CLEOPATRA*. For example, in the specification of the TRA-class `main` given in Figure 6.1, the TRA-class `fmonitor` is used to record the action values signaled on the `x` and `z` channels in files `x.dat` and `z.dat` respectively. Upon its instantiation, a given `fmonitor` object creates a sequential file, named after its only argument, where, thereafter, every action signaled on its input channel is recorded along with the time of its occurrence.

System-defined TRA-classes are themselves specified in *CLEOPATRA*. In particular, the specification of the TRA-class `fmonitor` is shown in Figure 6.4.[4] System-defined TRA-classes are different from regular, user-defined TRA-classes in that they have access to *global* information known only to the simulator. For instance, `fmonitor` objects have access to the simulator's "real" clock, `_clk`, whereas user-defined TRA-classes have to maintain their own *local* clocks, if needed.

[4]The `do` clause used in the specification of `fmonitor` shown in Figure 6.4 is a variant of the `commit` clause. It instructs the *CLEOPATRA* processor to commit the TET as early as possible within the specified time bounds. If no time bounds are specified, this amounts to committing the TET *before* any future events.

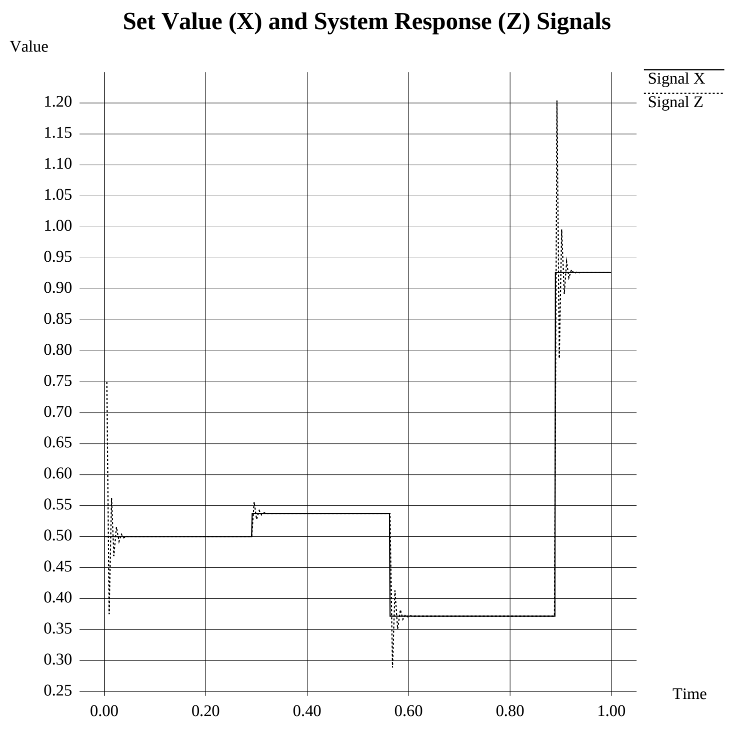


Figure 6.3: Simulated behavior of an underdamped process control system.

```
TRA-class fmonitor(string FILENAME)
  signal(double) ->
{
  state:
   double x ;
   file f ;
  init:
   f = fopen(FILENAME,"w") ;
  act:
   signal(x) -> :
     do { fprintf(f,"%f %f\n", _clk, x) ; }
}
```

Figure 6.4: The `fmonitor` system-defined TRA-class.

6.1.5 Compatibility with C

C functions can be called from within a *CLEOPATRA* specification. To maintain the semantics of the TRA formalism, however, only functions with no side effects should be used.[5] In other words, C function should be restricted to act as pure operations on the state variables of an object. It should not reach beyond the boundaries of the state space of that object.[6] Also, it should not alter the structure of the state space of the object in any way.[7] An example of the use of a C-function is illustrated in the description of the `user` TRA-class of Figure 6.1 where the function `random()` is called periodically to generate a random set value.

Most of the C preprocessor utilities are available in *CLEOPATRA*. This includes simple and parameterized macro definition and invocation, constant definition, and nested file inclusion.[8] For example, in the *CLEOPATRA* specification of the stand-alone process control system shown in Figure 6.1, system-defined TRA classes are included using the `#include` directive, and constants are defined using the `#define` directive.

6.1.6 Compilation and Execution

Figure 6.5 shows the different stages involved in the compilation and execution of specifications written in *CLEOPATRA*. At the heart of this process is a one-pass preprocessor, written in C, which parses user-defined *CLEOPATRA* specifications, augmented with system-defined TRA classes,[9] and generates an equivalent C simulator. The C simulator consists of three components. The first is a header (`.h`) file, which includes type definitions for the state space of the various TRA classes in the specification. The second is a schema (`.s`) file, which includes definitions for the state transition functions of the various TETs. The third is the code (`.c`) file, which includes the simulator initialization and control structure along with the instantiation code for the various TRA classes, including `main`. The final step of this

[5] Currently, *CLEOPATRA* processors do not check for that condition.

[6] By using pointers to other object spaces, for example.

[7] By using memory management routines, for example.

[8] Current *CLEOPATRA* processors do not admit conditional compilation.

[9] System-defined TRA classes are mainly for i/o and debugging purposes.

process involves the invocation of the C compiler to produce an executable simulator. Figure 6.6 illustrates a typical session, in which the *CLEOPATRA* compiler `ccleo` is invoked to process the file `process-ctrl.cleo` containing the specification of the stand-alone process control system shown in Figure 6.1.

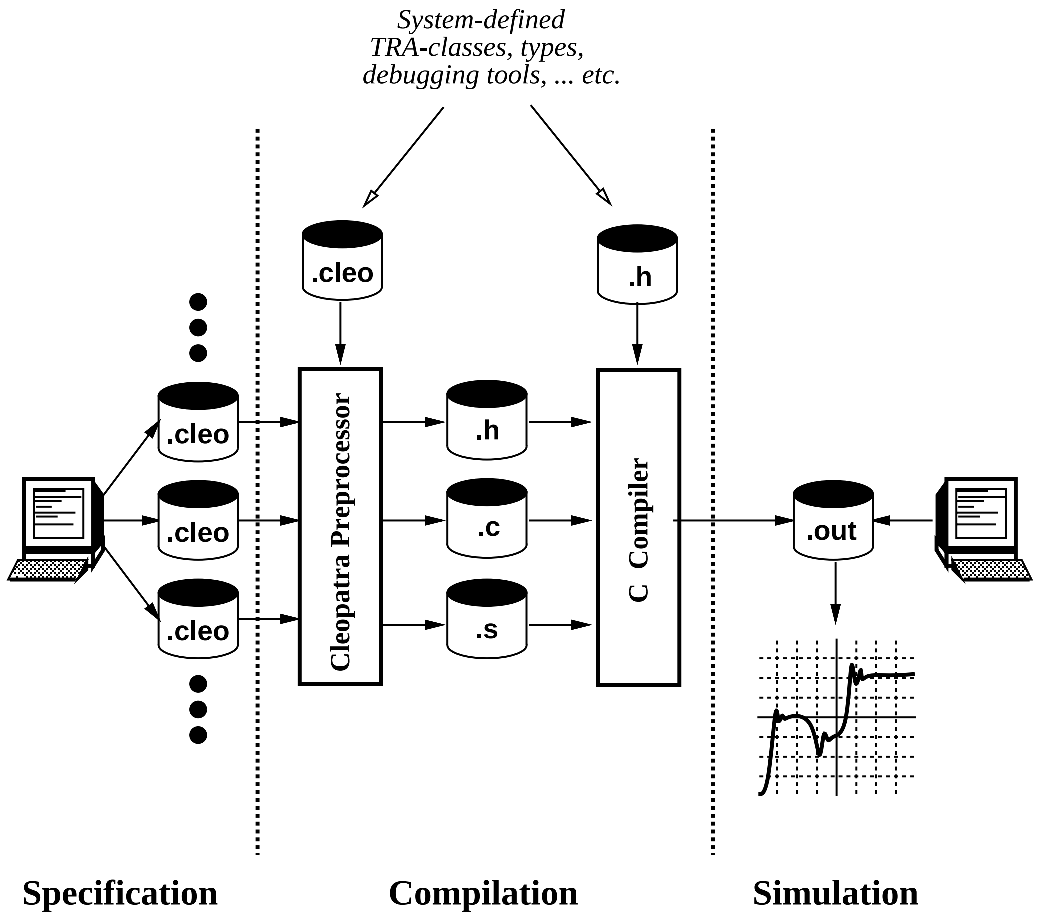


Figure 6.5: Compilation and simulation of *CLEOPATRA* specifications.

The simulator has proven to be quite efficient. This is due primarily to the causal and compositional nature of the TRA model, which tend to localize the computation triggered by the occurrence of an event within the boundaries of few TETs. The number of simulated events per second (seps) depends on a number of factors: the average channel fan-out, the average number of TETs per TRA, and the complexity of the event-driven computation. It does not depend, however, on the size of the state space or on the amount of TRA nesting. For an application with a fan-out of 1 and an average of 2.4 TETs per TRA, and an O(1) event-driven computational complexity, the compiled CLEOPATRA specifications executed at a rate of almost 19,500 seps.[10] The performance of a simulator for the same application hand coded directly in C performed only slightly better. Namely, it executed at a rate of almost 20,000 seps. The performance of the simulator degrades considerably when extensive i/o and tracing operations are performed.[11]

```
% ccleo process-ctrl
TRA-class fmonitor(string FILENAME)
  init(unit), signal(double) -> ;
TRA-class user(double EPOCH)
  init(unit) -> x(double) ;
TRA-class plant(double GAIN)
  init(unit), y(double) -> z(double) ;
TRA-class world()
  init(unit), y(double) -> x(double), z(double) ;
TRA-class control()
  init(unit), x(double), z(double) -> y(double) ;
TRA-class main()
  init(unit) -> `z(double)', `y(double)', `x(double)' ;

Cleopatra preprocessing completed.
C compilation completed.

% process-ctrl
CPU time = 1366612 usec   # of events = 5486   SEPS = 4014.3069

%
```

Figure 6.6: A typical CLEOPATRA compilation and execution session.

[10] All simulations were performed on a SPARCstation SLC^{TM}workstation.

[11] This is the case in the simulation shown in Figure 6.6, where an almost 5-fold decrease in efficiency can be attributed to the use of the `fmonitor` TRA-class.

6.2 Simulation of Reactive Behaviors in CLEOPATRA

In this section, we illustrate the use of CLEOPATRA as a simulation language for various reactive and control real-time systems. We single out four levels of real-time control *Servo*, *Selective*, *Teleo-selective*, and *Intelligent.* [Best91a]. One common aspect shared between these types of control is that, at any given point in time, a unique *reactive* behavior – certified to preserve the safety, liveness and responsiveness of the embedded system – is followed. This notion of continuous reactivity is crucial to embedded systems [Nils90].

6.2.1 Servo Control Systems (Basic Behaviors)

A *servo control* system consists of a *unique behavior* that underscores a tight coupling[12] between its input and output signals. Designers of servo systems are often concerned with questions of stability, transient and steady state responses,[13] compensation, ... *etc.* Power steering is an example of a servo control system. It has one behavior, to keep the steering wheel (input) and the front wheels (output) tightly coupled.

CLEOPATRA can be used to specify and simulate low level controls representing basic (servo-controlled) behaviors of embedded systems.[14] Examples include position and velocity feedback linear and non-linear control systems, as well as, complex systems involving asynchronous digital circuits and systems. This can be especially advantageous to model available resources (for example, actuators and sensors) and include their properties and capabilities in the overall analysis of the specified behavior.

To exemplify the use of CLEOPATRA in basic behavior specification, consider the CMOS `nand` gate in Figure 6.7 and its CLEOPATRA specification in Figure 6.8. The gate has two inputs `x` and `y`, and one output `z`. Actions signaled on `x` and `y` are assumed to be restored logical values [Puck88]. Actions signaled on `z`, however, are real voltages that depend on the driving voltages, the pull-up and pull-down resistances, and the load.

[12] Coupling is done using either open-loop or closed-loop (feedback) systems.

[13] For example overshoot, settling time, steady state errors ... *etc.*

[14] The feedback control used in the system shown in Figure 6.2 is an example of such basic behaviors.

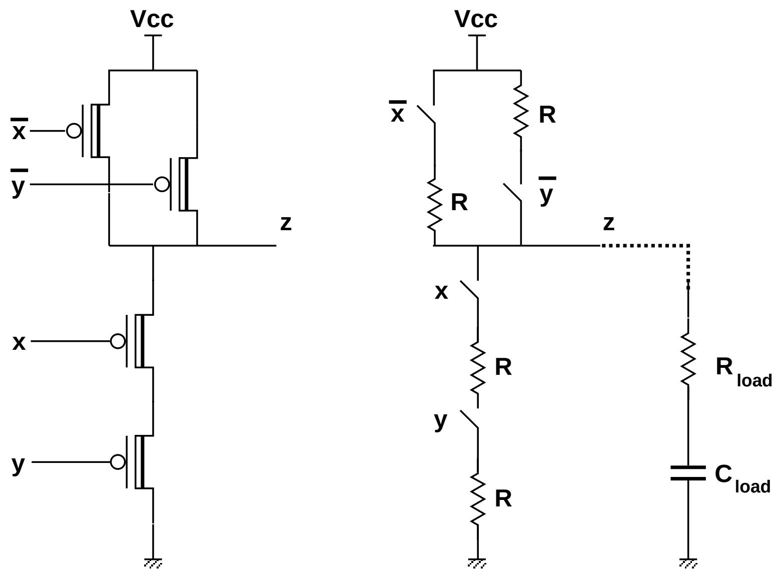


Figure 6.7: CMOS `nand` gate and a switching circuit approximation.

```
TRA-class nand(double R, R_load, C_load)
  x(bool),y(bool) -> z(double)
{
  state:
   bool x_val = 0, y_val = 0;
   double z_val = 0, ;
  act:
   x(x_val),y(y_val) -> :
     ;
   init(),z() -> z(z_val):
     within[0.9*DLY ~ 1.1*DLY]
       commit{
         if (x_val && y_val)
           z_val = z_val*exp(-TAU/(C_load*(R_load+2*R))) ;
         else
           if (x_val || y_val)
             z_val = z_val*(1-exp(-TAU/(C_load*(R_load+R)))) ;
           else
             z_val = z_val*(1-exp(-TAU/(C_load*(R_load+R/2)))) ;
       }
}
```

Figure 6.8: *CLEOPATRA* specification of the `nand` gate.

The specification of the TRA-class `nand` given in Figure 6.8 has four parameters, `R`, `R_load`, and `C_load`, denoting the equivalent resistance of a CMOS transistor, the load resistance, and the load capacitance, respectively. Periodically, every `DLY` units of time ($\pm$ 10%) the `nand` gate produces a value on the output channel `z` and updates its state[15] by either discharging the capacitance via the pull-down path or charging it via one pull-up path or both pull-up paths, depending on the values last signaled on the input channels `x` and `y`.

6.2.2 Selective Control Systems (Subsuming Behaviors)

Using *selective control*, the behavior of the system is selected from a fixed number of competing behaviors based on stimuli from the environment (the state of the world) in order to achieve a *unique goal*. Temperature control is an example of a reactive control system where either a cooling or heating behavior is selected depending on the ambient temperature. The cooling/heating behaviors might themselves be servo controlled. Examples of reactive control systems include Brooks' subsumption architecture [Broo86] and Brockett's Motion Description Language (MDL) [Broc88b].

In [Broo86], Rodney Brooks proposes the *subsumption architecture* as a methodology for specifying and building complex control systems. This architecture suggests the use of a vertical decomposition of the control system into a number of parallel independent task-achieving behaviors. Each one of these layers of behaviors is made out of smaller units called *modules*. Each module has a *finite state controller* and a certain number of inputs and outputs for communicating with other modules. There are two distinguished inputs for each module: *reset* and *inhibit*. Reset forces the finite state controller to go back to its initial state. Inhibit prevents the module from producing its output. Another special form of interaction, the subsumption, allows a module from a higher layer to overwrite the output of a module from a lower layer. The higher layer is called a dominant behavior, whereas the lower layer is called an inferior behavior. Subsumption allows control systems to be patched up by allowing smarter (or higher priority) behaviors to take over from default behaviors whenever appropriate.

[15]The state of the nand gate can be thought of as being the voltage of the load capacitance `C_load`.

The subsumption architecture can be supported easily and effectively using the TRA model. A module is simply a TRA with its inputs (outputs) being the input (output) signals of the TRA. The reset and inhibit inputs can be easily implemented as always-enabled input signals. In particular, the reset signal should make the TRA return to an initial state, whereas the inhibit signal should make it go to a specific state in which all output actions are disabled. The subsumption interaction between layers can be modeled by a simple TRA, the `subsumption` TRA. This TRA will have as input signals the output of a dominant module and that of an inferior module. Its output signal will be identical to the inferior input signal as long as the dominant input signal is absent.[16] Otherwise, it will be identical to the dominant input. Obviously, the *subsumption*-TRA is just an implementation of a two-level priority scheme. It can be easily extended[17] to model a static multi-level priority scheme, or any other priority scheme, which would then represent a hierarchy of dominant and inferior behaviors. Figure 6.9 shows a possible specification of the TRA-class `subsumption`.

```
typedef enum{0,1,X} tristate;

TRA-class subsume(double DELAY)
  dominant(tristate),inferior(tristate) -> behavior(tristate)
{
  state:
   tristate d_val = X, i_val = X ;
  act:
   dominant(d_val) -> behavior(d_val):
     before DELAY
       unless(d_val == X && i_val != X)
         commit { i_val = X ; }
   inferior(i_val) -> behavior(i_val):
     before DELAY
       unless(d_val != X)
         ;
}
```

Figure 6.9: *CLEOPATRA* specification of the subsumption TRA.

[16] Absent can be interpreted in a number of different ways. For example, it can mean the absence of any actions for more than a given time interval.

[17] by implementing the r-level priority scheme with a binary tree of *subsumption*-TRAs.

The subsumption architecture is suitable for the specification of task-achieving behaviors that can be *statically* organized as a hierarchy of dominant and inferior behaviors. It cannot deal with applications with *dynamically* changing priorities. In particular, if the priority of a behavior depends on the task (or goal) to be achieved, and if such a goal is dynamically changing, then this behavior can be dominant in some situations and inferior in others. Rather than dominant and inferior behaviors, such systems are described in terms of *competing behaviors*.

6.2.3 Teleo-selective Control Systems (Competing Behaviors)

Using *teleo-selective* control, the behavior of the system is dynamically selected from amongst a fixed number of competing behaviors based on stimuli from the environment and motivations to achieve one of a *set of pre-determined goals*.

The TRA framework is ideal for the specification of systems with competing behaviors. Examples of TRA behavioral specification of such systems were given in [Best90b]. The use of the TRA model in the specification and simulation of these systems is similar to the use of Nilsson's action networks [Nils88], and Maes' situated agents [Maes90]. TRA specifications, however, allow (potentially automated) analysis to be performed on behaviors. For instance, given a finite-state TRA description, it is possible to obtain a finite-state description of all of its possible behaviors, and thus, proving assertions about these behaviors. This can be done using techniques similar to those suggested in [Lewi89, Alur90]. In addition, the TRA model provides a vehicle for efficient simulation and implementation using *CLEOPATRA*. Silicon compilation of *CLEOPATRA* specifications for simple behaviors is also a possibility [Frie91].

As an example of TRA-specification of competing behaviors, consider the specification of *Buggy*, a bug-like autonomous creature. Buggy has two actuators that allows it to move in 2-D and three sensors that allow it to find food, locate predators, and detect floor cracks within a limited neighborhood. Buggy has two potentially competing behaviors, searching for food along cracks, and keeping itself away from predators (or obstacles). Buggy has only one goal: to survive. This requires both eating (to avoid starvation) and escaping from predators (to avoid being crushed). Buggy's urge to find food increases as time elapses and no food is found. On the other hand, Buggy's fear from predators increases as its distance from them decreases. At any point in time, the behavior that is more important to Buggy's survival subsumes the other.

Buggy's behavior was specified using *CLEOPATRA*. Figure 6.2.3 shows one of Buggy's simulated behaviors in a circular room with two cracks. In this behavior one can identify some of Buggy's basic behaviors. In particular, when Buggy's sensors fail to detect any cracks or obstacles in its immediate neighborhood (due to the limited range of these sensors), Buggy's behavior is basically to *wander* randomly. The pace of this wandering behavior (speed and rate of direction change) depends on the state of Buggy – its hunger and fear levels. Other basic behaviors of Buggy include approaching a crack, following a crack, and running away from obstacles. In addition to the basic behaviors of Buggy, one can also identify a number of *emergent behaviors*. An emergent behavior is a behavior that is not specified explicitly; it emerges from the composition of other basic behaviors. For example, in Figure 6.2.3, two behavioral patterns can be easily singled out. The first is a hesitant behavior, in which, driven by hunger and fear, Buggy switches back and forth between approaching a crack to find food and running away from it to escape from the nearby rotating predator. The second is a routine behavior, in which Buggy reaches a limit-cycle of approaching a crack, following it, and running away from it.

6.2.4 Intelligent Control Systems (Intelligent Behaviors)

Using *intelligent* control, the behavior of a system is selected from amongst a number of fixed and superimposed *synthesized behaviors* based on stimuli from the environment and motivations to achieve a run-time goal. The process of synthesizing a behavior is carried out by a planning agent (process) *outside* the sensing/acting loop based on a *perceived* model of the world and a set of goals to be achieved. Figure 6.11 shows the architecture of an intelligent control system.

The success of the planning process, which entails a safe progress towards achieving the planner's goals, depends heavily on the existence of an accurate description of the behavior of the world (external environment). This can be achieved in two different ways. On the one hand, a cognition agent might be able, through observation and learning, to

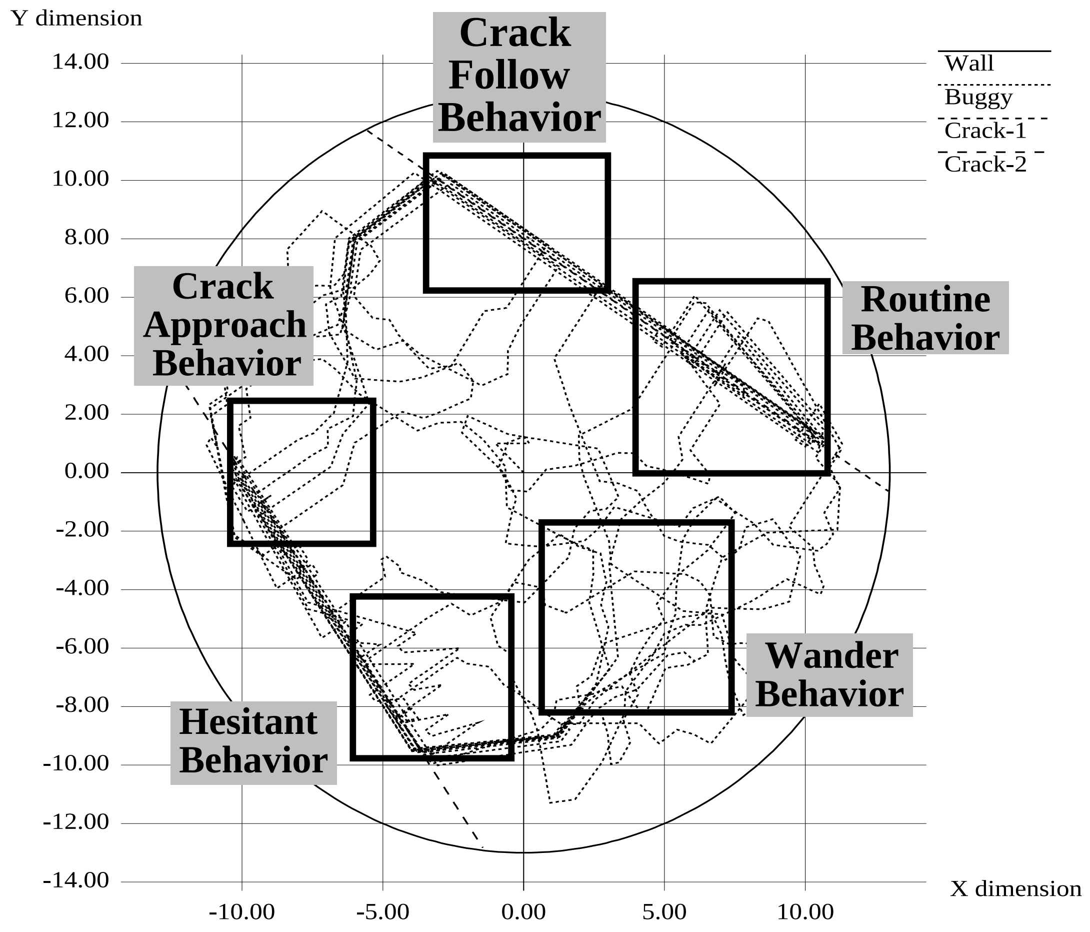


Figure 6.10: Basic and emergent behaviors of Buggy in a typical simulation.

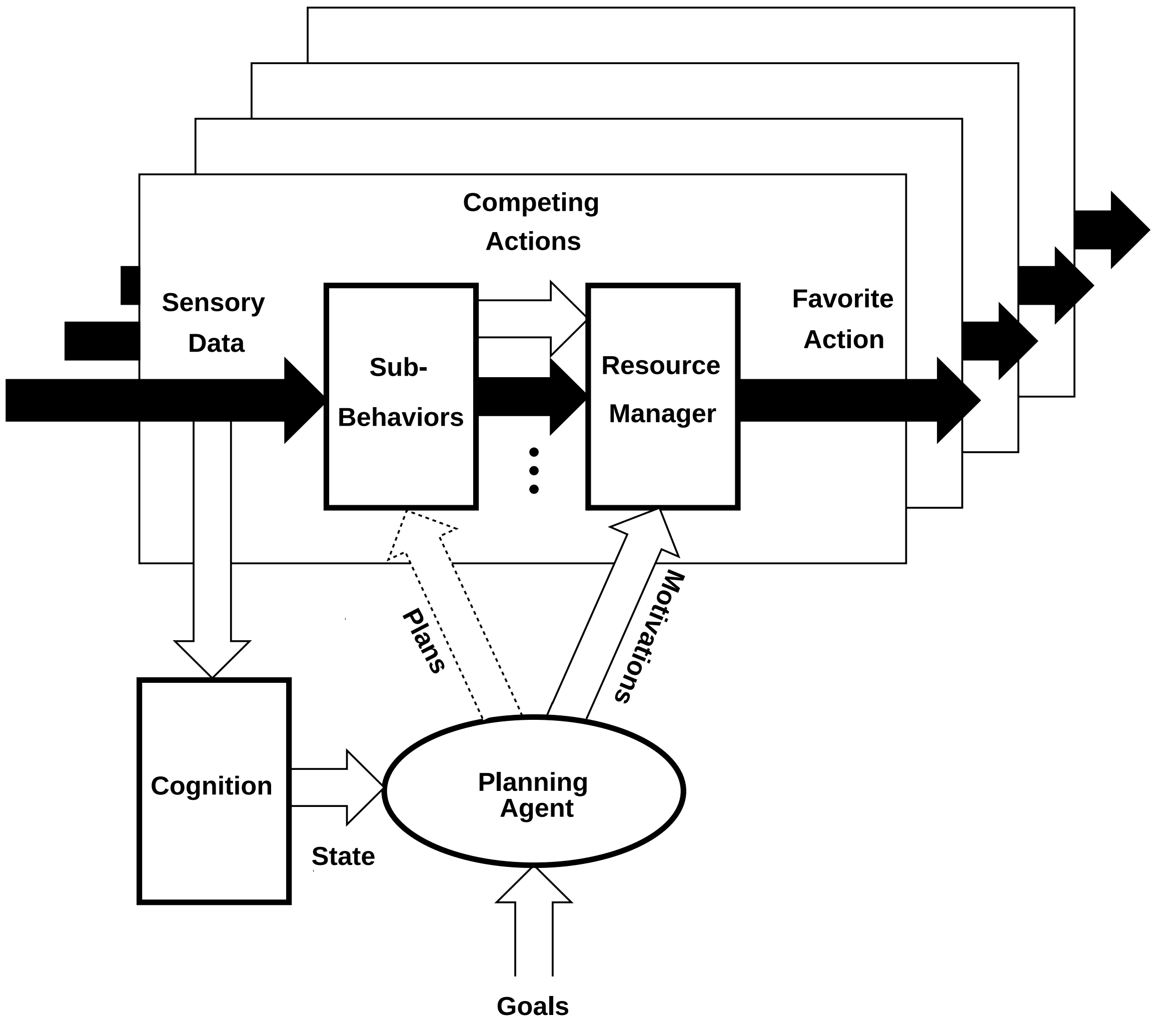


Figure 6.11: Interaction between behavioral planning and real-time control

provide the planning agent with a *perceived* behavioral specification of the world. The potential inaccuracy of this process – due, for example, to a rapidly changing environment – dictates that perceived world specifications be used only in conjunction with the plan synthesis process. On the other hand, the verification process can rely only on invariants about the world's behaviors, which are asserted and guaranteed by the system designers. We call these assertions *hardcore specifications.*[18]

Most of the current research in real-time planning falls in the first three levels of control systems discussed thus far, namely, servo, selective, and teleo-selective, where plans are pre-compiled (hardwired) into the sensing/acting loop. During run-time, the system is executing a pre-compiled plan. Current attempts at building intelligent embedded systems [Kael86, Lyon91b, Lyon91a] are mostly concerned with the dynamic, unpredictable, and often hostile nature of the environment. The consideration of safety issues expressed as timing constraints has been minimal. This is primarily due to the lack of a computational model powerful enough to capture timing properties relevant to real-time specification and verification, and expressive enough to serve as a framework for planning. In [Best91a], we have proposed the use of the TRA model as a vehicle for planning in real-time systems. In particular, it is suitable as a formalism for the representation of actions, both for planning and for verification purposes.

[18]In embedded systems, it is usually assumed [Leve91] that an external mechanism is responsible for securing the safety of the system. Examples of such mechanisms include human intervention, using limit switches, ... *etc.*

Chapter 7

TRA-based Implementation

The rational behind proposing the TRA *formalism is that it can serve as the backbone of a development methodology for embedded real-time applications. In previous chapters, we discussed various aspects of this methodology, namely specification, verification, and validation. In this chapter, we focus on the potentials of the* TRA *model as a vehicle for implementation purposes. In particular, we report on some initial work we did in developing a compiler for the real-time execution of* CLEOPATRA *specifications of robotics applications.*

To close the gap between formality and practicality, the development cycle of embedded applications has to be supported in its entirety. This requires that system implementation – and not only specification, validation and verification – be addressed. In this chapter, we describe on-going and future research in that direction.

7.1 *CLEOPATRA*: A Programming Language

For software processes, the distinction between an executable specification and an implementation is vague. This suggests that specification languages can themselves be used as programming languages. For real-time applications, this is true only if programs can be compiled to execute in real-time rather than in simulated time. Currently, we are developing a compiler for real-time programs written in *CLEOPATRA*.

Figure 7.1 illustrates the various component of a *CLEOPATRA*-based implementation environment. The target machine for the compiler is a distributed VME-based dedicated shared-memory 68030 single-board computers running real-time operating system kernels.[1] *CLEOPATRA* program development, debugging, and monitoring is to be done using a standard Unix-based workstation environment linked to that target.

Compilers for real-time languages like *CLEOPATRA* are complicated by the fact that they are required to verify the *feasibility* of the compilation process. In other words, in addition to checking syntax and semantics, such compilers have to establish that, given a specific hardware configuration, the compiled code will observe all the timing constraints specified in the source code. A simpler approach to address that problem would be to generate code that raises exceptions whenever a violation of a specified time constraint is detected during execution. This is similar to raising exceptions as a result of run-time errors in conventional (non-real-time) languages. Due to its simplicity, we have elected to follow the latter approach in our initial implementation. Meanwhile, we intend to investigate and develop efficient verification algorithms that would potentially lead to the adoption of the former approach.

[1] Possibilities include the Lynx™ and VxWorks™ operating systems.

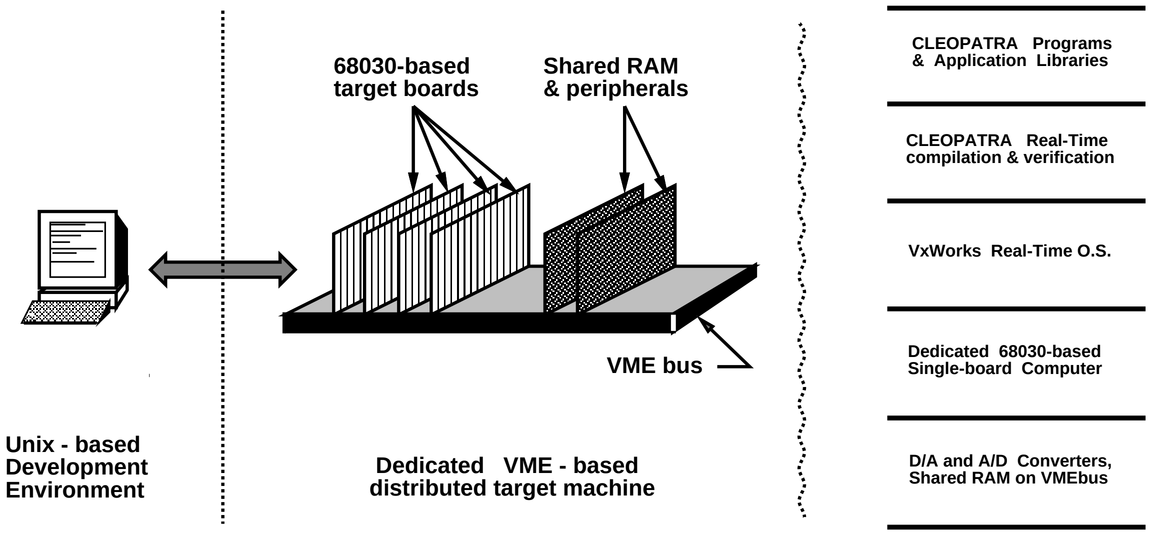

68030-based target boards
Shared RAM & peripherals
VME bus
Unix - based Development Environment
Dedicated VME - based distributed target machine
CLEOPATRA Programs & Application Libraries
CLEOPATRA Real-Time compilation & verification
VxWorks Real-Time O.S.
Dedicated 68030-based Single-board Computer
D/A and A/D Converters, Shared RAM on VMEbus

Figure 7.1: Components of a *CLEOPATRA*-based implementation environment

7.2 TRA-based Development of Robotics Applications

In order for a language to be instrumental in implementing practical systems, it must be geared towards a specific application through the development of appropriate libraries and verification tools. Our intention is for *CLEOPATRA* to target robotic applications. Robotics applications are good representatives for "real" embedded systems. The tasks involved therein are diverse[2] (vision, motion control, high-level planning, ...*etc.*) and make use of very different resources (special purpose image processors, tailor made controllers and drivers, tightly- and loosely-coupled computer networks, massively parallel architectures, ...*etc.*) In addition, the interaction between these tasks is non-trivial and highly time-constrained. Being able to model, and even implement, such complex systems in a single framework is both challenging and attractive.

[2]Refer to Figure 7.2 for a typical experiment.

A robot system will typically have associated with it a number of sensing subsystems. If these sensing subsystems are *active*[3] they will each be issuing motor requests related to the sensory processing algorithm that they are performing. In a general robot, however, the manipulative systems will also be required to perform duties not related to the acquisition of sensory information, but aimed at influencing the robot's environment in a purposive manner. Thus, both of these sensory and manipulatory subsystems, will be competing for a limited resource, that of the positional degrees of freedom of the robot. Such a competition has to be managed.

As an example of competing requests management, consider a robot whose active perception system requires it to move around a block in order to see what it occludes, and whose manipulative system requires it to stand still so that it can grasp a nearby object, which would be, otherwise, out of reach. It can be seen by this simple example that one cannot decouple the motor activities requested by the active perception system from the motor activities requested by the manipulative system. Thus any system that is developed for controlling motor activities in a robot must take into account both the perceptual goals and the manipulative goals of the machine and produce motions which address these goals in an integrated and orderly manner.

Another crucial problem in sensori-motor robotics activities is that of coordination. For example, a vision system might be required to synchronize its sampling with the robot motion. In particular, it might require the robot to remain stand still at a given coordinate for a specific period of time in the future to grab a frame.

One can think of the motor units of the robot as a limited resource that must be shared between the active perception and manipulative systems. The motion control operating system must arbitrate and/or coordinate between the conflicting requirements of these two systems in a way which allows the goals of the two systems to be attained. In [Best90d], we suggested a methodology based on the TRA model that allows one to schedule the motor commands sent to the various actuators in the robot in a manner appropriate to the robot goals.

[3]Active perception implies usage of the robots manipulative systems to move about and interact with the environment in ways that serve the sensory processes [Aloi87, Bajc85, Bajc88].

In [Clar91], an experiment that adopts the TRA framework was proposed. The experiment involves the coordination of motor requests to perform manipulative tasks using directed-vision feedback. The testbed for the experiment (Figure 7.2) consists of a six-degrees of freedom (American Cimflex) industrial robot connected to a dedicated (Merlin) controller. The controller consists of six parallel MC6809 processors (slaves), each controlling one of the robot's actuators. A single board 68000-based computer VM02 (master) is responsible for driving these processors in real-time. The backbone of the Merlin controller is a VERSAbus which is connected via a Synergist-II VMEbus-VERSAbus translator and a BIT-3 VME-VME adaptor to the bus extender of a SUN-3 workstation (MIPS). On the same bus extender, another 68000-based single-board computer (REAL1) is running VxWorks, a real-time operating system kernel. The Unix-based MIPS workstation provides an environment for developing and debugging robotics applications, whereas REAL1 is used to run these applications in real-time. In addition, MIPS acts as a Local Area Network gateway to the other computing facilities in the robotics lab. This includes the MASPAR massively parallel computer, and the data-cube special-purpose array processor for image processing. A video-camera connected to the data-cube is mounted on the American Cimflex robot arm.

The VM02 master computer of the Merlin controller runs a High Speed Host Interface (HSHI) program that allows it to receive commands at a rate of up to 250 commands per second – a 4 milliseconds latency – to remotely control the robot from a host computer. In [Best88b] we described an interface that we designed and implemented to allow a SUN workstation to communicate with HSHI via a dual-ported shared memory piggy-backed on the Bit-3 VME-VME adaptor. A drawback of that connection was our reliance on UNIX, a non-real-time operating system. Recently, we have successfully modified our interface to work from REAL1 under the VxWorks real-time operating system. This would allow us to execute time-sensitive tasks safely.

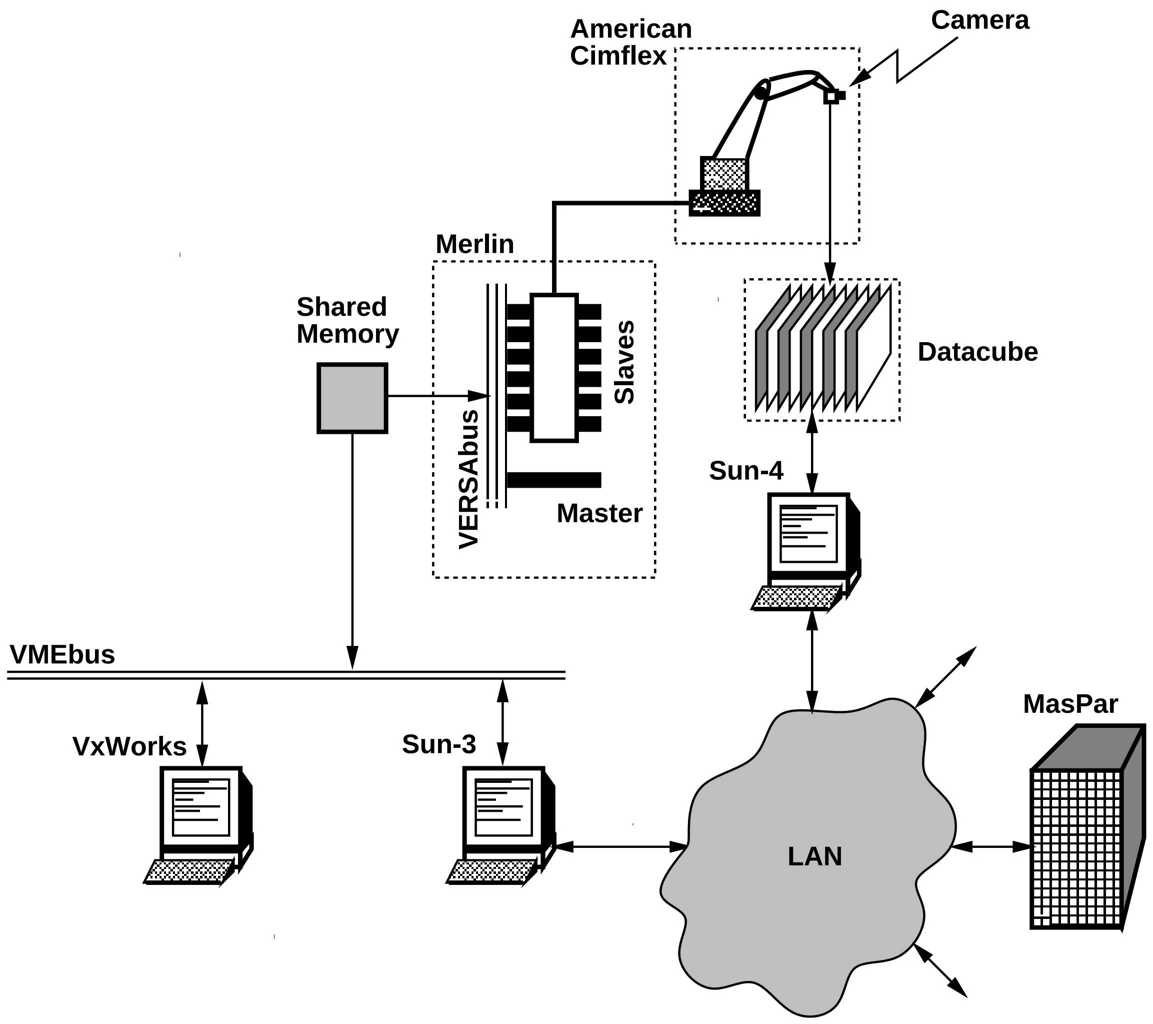


Figure 7.2: Set-up for a sensori-motor activity coordination experiment

Chapter 8

Conclusion

Current practices in building embedded systems are not based on sound scientific underpinnings. Considering the vital role that such systems are playing and will continue to play in our world, it has become imperative that a rigorous and systematic treatment that recognizes their requirements be adopted. In this thesis, we proposed such a treatment based on the Time-constrained Reactive Automata model – a novel formal model suitable for the specification, validation, verification, and implementation of embedded systems.

8.1 Summary

Predictability – the ability to foretell that an implementation will not violate a set of specification requirements – is a crucial, highly desirable property of embedded time-critical systems. The aim of this thesis is to improve the predictability of an embedded system by adopting a physically-sound formalism (the TRA model) as the basis for a programming language (*CLEOPATRA*) and a proof system. Our work differs from others in that it starts with a realistic specification, one which allows only those requirements that can be fulfilled without defying the rules of physical systems (such as causality, spontaneity, and reactivity). By limiting the expressiveness of the specification in this way, the process of finding an implementation and establishing its correctness becomes easier and – as we hypothesize – quite possible to automate.

Among the salient features of the TRA model is a fundamental notion of space and time. The payoff for this dual treatment of space and time is manifold. Requirement specifications become more accurate since they can constrain the time as well as the space coordinates of system events. Also, mappings between various levels of abstractions for compilation and verification purposes become more robust as the formalism becomes more structured. The TRA model is compositional and supports time, control, and computation non-determinism without violating the principles of causality and spontaneity. It allows the representation of both the external environment and the programmed system along with the available resources in a unique framework making it possible to prove safety and liveness properties and to study transient and steady state performances.

CLEOPATRA is a specification language based entirely on the TRA model. It features a C-like imperative syntax for the description of computation, which makes it easier to incorporate in real applications already using C; it is object-oriented, thus advocating modularity, reusability, and off-the-shelf hierarchical programming of embedded systems. *CLEOPATRA* is semantically sound. In particular, its objects can be transformed, mechanically and unambiguously, into formal TRA objects for verification purposes.

8.2 Directions for Future Research

As explained earlier in this thesis, predictability can be enhanced in a variety of ways. It can be enhanced by restricting expressiveness as was done in Real-Time Euclid, by sacrificing accuracy as was done in the Flex system, or by abstracting segmented resources as was done in the Spring kernel. The TRA-development methodology we are advocating introduces one more way of improving predictability, that of allowing only physically-sound specifications. Pursuing the ideas presented in this thesis will undoubtedly provide us with one more handle in our persistent quest for predictable systems. An interesting question to be addressed in the future would be whether this and other handles can be combined in any useful way to *guarantee* predictability.

Previous studies in modeling real-time computing systems have focussed on adding the notion of time to the formal modeling techniques of traditional systems, namely state-based, logic-based, Petri-Net-based, and process-algebra-based formalisms. An important extension of the work presented in this thesis would be to investigate the relationship between the TRA model and those computing formalisms. Particularly interesting and promising questions include: Would it be possible for properties stated as temporal logic formulae to be automatically transformed into TRAs? Could the TRA model with its strong notion of causality benefit from the safety analysis techniques developed for Petri-nets?

The boundary between an embedded computing system and its environment is inarguably the least understood and, consequently, the most dubious. Failures of embedded systems are largely due to unexpected, (thus uninspected) scenarios that arise between these systems and their environments. The main difficulty in studying the interactions between an embedded computing system and its environment is the lack of a graceful transition between computing and non-computing models. The TRA formalism is a good candidate to bridge this gap. We have found it to capture efficiently and naturally many aspects of non-computing models. For example, the notion of a TRA status, which encapsulates the state and intentions of a discrete-event system, resembles the notion of a state and its moments (derivatives) for continuous systems. This resemblance is not accidental. It is precisely a result of our insistence on physical soundness. An immediate outgrowth of this thesis is to consider the relationship between the TRA model and other non-computing formalisms. We believe that this is inevitable if an accurate and integrated view of an embedded system in its entirety is to be sought.

Our experience with the TRA development methodology in the design, simulation, and analysis of asynchronous digital circuits, sensori-motor autonomous systems, and intelligent controllers confirms its suitability for the specification, verification, and validation of many embedded and time-critical applications. Its usefulness in the implementation of such systems, although promising, is yet to be established. An fruitful direction for future research would be to automate the process of transforming TRA-based physically-sound time-critical specifications into provably-correct implementations given appropriate resources. Such research will have two complementary – experimental and theoretical – components. The experimental component would involve the development of a compiler to transform *CLEOPATRA* specifications into predictable real-time programs, given a dedicated computing platform. The theoretical component would aim at devising efficient verification algorithms that can be automated and incorporated in the *CLEOPATRA* compiler.

Finally, to be effectively evaluated, any methodology for the development of embedded systems must be used in conjunction with a specific application. A fruitful direction for future research would be to implement a TRA-based programming environment for a specific application domain.

Bibliography

[Alfo77] M. Alford. "A requirements engineering methodology for real-time processing requirements." *IEEE Transactions on Software Engineering*, SE-3:60–69, January 1977.

[Alle81] J. Allen. "An interval-based representation of temporal knowledge." In *Proceedings of the 7th International Joint Conference on Artificial Intelligence*, pages 221–226, August 1981.

[Alle83] J. Allen. "Maintaining knowledge about temporal intervals." *Communications of the ACM*, 26:832–843, November 1983.

[Alle84] J. Allen. "Towards a general theory of action and time." *Artificial Intelligence*, 23:123–154, 1984.

[Alle86] J. Allen and R. Pelavin. "A formal logic of plans in temporally rich domains." *IEEE Special Issue on Knowledge Representation*, October 1986.

[Aloi87] Y. Aloimonos, I. Weiss, and A. Bandyopadhyay. "Active vision." In *Proceedings of the 1st IEEE Conference on Computer Vision*, pages 35–54, London, Great Britain, 1987.

[Alur90] Rajeev Alur, Costas Courcoubetis, and David Dill. "Model-checking for real-time systems." In *Proceedings of the 5th annual IEEE Symposium on Logic in Computer Science*, Philadelphia, Pensylvania, June 1990. IEEE Computer Society Press.

[Auer86] Brent Auernheimer and Richard Kemmerer. "RT-ASLAN: A specification language for real-time systems." *IEEE Transaction on Software Engineering*, 12(9):879–889, September 1986.

[Baet91a] J. Baeten and J. Bergstra. "Asynchronous communication in real space process algebra." In *Proceedings of the Chalmers Workshop on Concurrency, Båstad*, 1991.

[Baet91b] J. Baeten and J. Bergstra. "Real space process algebra." In J. Baeten and J. Bergstra, editors, *Proceedings of CONCUR'91 at Amsterdam.* Springer LNCS, 1991.

[Baet91c] J. Baeten and J. Bergstra. "Real time process algebra." *Formal Aspects of Computing*, 3(2):142–188, 1991.

[Bajc85] R. Bajcsy. "Active perception versus passive perception." In *Proceedings 3rd IEEE Workshop on Computer Vision*, pages 55–59, Bellaire, CA, 1985.

[Bajc88] R. Bajcsy. "Perception with feedback." In *Proceedings of the 1988 Darpa Image Understanding Workshop*, 1988.

[Bake86] Theodor Baker and Gregory Scallon. "An architecture for real-time software systems." *IEEE Software*, 2(5):50–58, May 1986.

[Baue90] Robert Bauer. "How Real is Real-Time Unix? A review of LynxOS." *UNIX Review*, September 1990.

[Berg84] J.A. Bergstra and J.W.Klop. "Process algebra for synchronous communication." *Information and Control*, 60:109–137, 1984.

[Bern81] Arthur Bernstein and Paul Harter. "Proving real-time properties of programs with temporal logic." *Operating System Review*, 15(5), December 1981.

[Berr83] D. Berry, S. Moisan, and J. Rigault. "Esterel: Towards a synchronous and semantically sound high level language for real-time applications." In *Proceedings of the 1983 IEEE Real-Time Systems Symposium*, pages 30–37, December 1983.

[Berr85] D. Berry and L. Cosserat. "The Esterel synchronous programming language and its mathematical semantics." *Lecture Notes in Computer Science*, 197:389–448, February 1985.

[Best88a] Azer Bestavros. "The input output timed automaton." Technical Report TR-12-89, Harvard University, Department of Computer Science, DAS, Aiken Computation Lab, Cambridge, Massachusetts, November 1988. Revised October 1989.

[Best88b] Azer Bestavros. *The Michael - Merlin Connection: Programming tools for the remote control of the American Cimflex robot.* Robotics Laboratory, Harvard University, Cambridge, MA, September 1988.

[Best90a] Azer Bestavros. "ESPRIT: Executable Specification of Parallel Real-time Interactive Tasks." Technical Report TR-06-90, Department of Computer Science, Harvard University, Cambridge, MA, September 1990.

[Best90b] Azer Bestavros. "The IOTA: A model for real-time parallel computation." In *Proceedings of TAU'90: The 1990 ACM International Workshop on Timing issues in the Specification and Synthesis of Digital Systems*, Vancouver, Canada, August 1990.

[Best90c] Azer Bestavros. "TRA-based real-time executable specification using CLEOPATRA." In *Proceedings of the 10th Annual Rochester Forth Conference on Embedded Systems*, Rochester, NY, June 1990. (revised May 1991).

[Best90d] Azer Bestavros, James Clark, and Nicola Ferrier. "Management of sensori-motor activity in mobile robots." In *Proceedings of the 1990 IEEE International Conference on Robotics & Automation*, Cincinati, Ohio, May 1990. IEEE Computer Society Press.

[Best91a] Azer Bestavros. "Planning for embedded systems: A real-time prospective." In *Proceedings of AIRTC-91: The 3rd IFAC Workshop on Artificial Intelligence in Real Time Control*, Napa/Sonoma Region, CA, September 1991.

[Best91b] Azer Bestavros. "Specification and verification or real-time embedded systems using the Time-constrained Reactive Automata." In *Proceedings of the 12th IEEE Real-time Systems Symposium*, pages 244–253, San Antonio, Texas, December 1991.

[Boch82] Gregor Bochmann. "Hardware specification with temporal logic: An example." *IEEE transactions on Computers*, C-31(3), March 1982.

[Borr90] G. Borriello and T. Amon. "On the specification of timing behavior." In *Proceedings of TAU'90: The 1990 ACM International Workshop on Timing issues in the Specification and Synthesis of Digital Systems*, Vancouver, Canada, August 1990.

[Broc88a] Roger Brockett. “Dynamical systems that sort lists, diagonalize matrices and solve linear programming problems.” In *Proceedings of the 1988 IEEE Conference on Decision and Control*, December 1988.

[Broc88b] Roger Brockett. “On the computer control of movement.” In *Proceedings of the 1988 IEEE International Conference on Robotics & Automation*, Philadelphia, PA, 1988. IEEE Computer Society Press.

[Broc89] Roger Brockett. “Smooth dynamical systems which realize arithmetic and logical operations.” Internal Report, Harvard University, Cambridge, MA, 1989.

[Broo86] Rodney Brooks and Jonathan Connell. “Asynchronous distributed control system for a mobile robot.” *SPIE Proceedings*, 727, October 1986.

[Bü60] J. R. Büchi. “On a decision method in restricted second-order arithmetic.” In *Proceedings of the International Congress on Logic, Methodology, and Philosophy of Science*, 1960. Stanford University Press, 1962.

[Burn90] Alan Burns and Andy Wellings. *Real-time systems and their programming languages.* Addison Wesley Co. (International Computer Science Series), 1990.

[Caps87] P. Capsi, D. Pilaud, N. Halbwachs, and J. Plaice. “LUSTRE: a declarative language for real-time programming.” In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages*, 1987.

[Chun90] Jen-Yao Chung, Jane Liu, and Kwei-Jay Lin. “Scheduling periodic jobs that allow imprecise results.” *IEEE Transaction on Computers*, 19(9):1156–1173, September 1990.

[Clar83] E. Clarke, E. Emerson, and A. Sistla. “Automatic verification of finite-state concurrent systems using temporal logic specifications: A practical approach.” In *Proceedings of the 10th ACM Symposium on Principles of Programming Languages*, Austin, Texas, January 1983.

[Clar91] James Clark, Nicola Ferrier, and Lei Wang. “A robotics system for manipulation using directed vision feedback.” Internal report (in progress), Robotics laboratory, Harvard University, Cambridge, MA, 1991.

[Cool83] J. E. Coolahan and N. Roussopoulos. “Timing requirements for time-driven systems using augmented petri nets.” *IEEE Transactions on Software Engineering*, SE-9:603–616, September 1983.

[Cox88] Ingemar Cox, David Kapilov, Walter Kropfl, and Jonathan Shapiro. “Real-time software for robotics.” *AT&T Technical Journal*, 67(2), March/April 1988.

[Dasa85] B. Dasarathy. “Timing constraints of real-time systems: Control for expressing them, Method for validating them.” *IEEE Transactions on Software Engineering*, 11(1), January 1985.

[Eswa76] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. “The notions of consistency and predicate locks in a database system.” *Communications of the ACM*, 19(11):624–633, November 1976.

[Faus86] Antony Faustini and Edgar Lewis. “Toward a real-time dataflow language.” *IEEE Software*, 3(1):29–35, January 1986.

[Frie91] Dan Friedman and James Clark. “Silicon compilation of simple sensori-motor behaviors.” , 1991. Private communication of on-going research.

[Fu87] K. S. Fu, R. C. Gonzalez, and C. S. G. Lee. *Robotics: Control, sensing, vision, and intelligence.* McGraw-Hill Book Company, 1987.

[Gall91] B.O. Gallmeister and C. Lanier. "Early experience with POSIX 1003.4 and POSIX 1003.4A." In *Proceedings of the 12th IEEE Real-time Systems Symposium*, pages 190–198, San Antonio, Texas, December 1991.

[Gerb89a] R. Gerber, I. Lee, and A. Zwarico. "Communicating Shared Resources: A model for distributed real-time systems." In *Proceedings of the 12th IEEE Real-time Systems Symposium*, pages 68–78, Santa Monica, California, December 1989.

[Gerb89b] R. Gerber, I. Lee, and A. Zwarico. "A complete axiomatization of real-time processes." CIS, University of Pennsylvania – Submitted for publication, February 1989.

[Gerb90] R. Gerber and I. Lee. "A proof system for communicating shared resources." In *Proceedings of the 11th IEEE Real-Time Systems Symposium*, 1990.

[Ghez89] C. Ghezzi, D. Mandrioli, S. Morasca, and M. Pezzè. "A general way to put time in Petri nets." In *Proceedings of the 5th International Workshop on Software Specifications and Design*, Pittsburgh, PA, May 1989. IEEE Computer Society Press.

[Gosw88] Asis Goswami and Mathai Joseph. "A semantic model for the specification of real-time processes." Research Report 121, Department of Computer Science, University of Warwick, April 1988.

[Hare87] D. Harel. "Statecharts: A visual formalism for complex systems." *Science of Computer Programming*, 8(3):231–274, June 1987.

[Hawk88] Stephen W. Hawking. *A brief history of Time: From the Big Bang to Black Holes.* Bantam Books, April 1988.

[Henn88] Matthew Hennessy. *Algebraic theory of processes.* MIT Press, Cambridge, MA, 1988.

[Hoar85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[Holl87] M. A. Holliday and M. K. Vernon. "A generalized timed Petri-net model for performance analysis." *IEEE Transactions on Software Engineering*, SE-13, December 1987.

[Hopc79] John Hopcroft and Jeffrey Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison Wesley, 1979.

[IEEE90] IEEE. "POSIX 1003.1-1990: Standard portable operating system interface for computer environments." , 1990. Reference number ISO/IEC 9945-1: 1990(E). Institute of Electrical and Electronics Engineers, NY, NY.

[Jaha86] Farnam Jahanian and Aloysius Mok. "Safety analysis of timing properties in real-time systems." *IEEE Transaction on Software Engineering*, 12(9):890–904, 1986.

[Jaha88] Farnam Jahanian and Aloysius Mok. "Modechart: A specification language for real-time systems." *IEEE Transaction on Software Engineering*, 14, 1988.

[Kael86] Leslie Pack Kaelbling. "An architecture for intelligent reactive systems." Technical Report Technical Note 400, SRI International, 333 Ravenswood Ave., Menlo Park, CA 94025, October 1986.

[Klig86] Eugene Kligerman and Alexander Stoyenko. "Real-time Euclid: A language for reliable real-time systems." *IEEE Transactions on Software Engineering*, 12(9):941–949, September 1986.

[Lee91] I. Lee, R. Gerber, and S. Davidson. "Communicating Shared Resources: A paradigm for integrating real-time specification and implementation." In André M. van Tilborg and Gary M. Koob, editors, *Foundations of Real-Time Computing: Formal Specifications and Methods*, pages 87–110. Kluwer Academic Publishers, 1991.

[Leve87] Nancy Leveson and Janice Stolzy. "Safety analysis using Petri Nets." *IEEE Transactions on Software Engineering*, 13(3):386–97, March 1987.

[Leve91] Nancy Leveson. "Software safety in embedded computer systems." *Communications of the ACM*, 34(2), February 1991.

[Lewi89] Harry Lewis. "Finite-state analysis of asynchronous circuits with bounded temporal uncertainty." Technical Report TR-15-89, Department of computer science, Harvard University, Cambridge, MA, June 1989.

[Lewi90] Harry Lewis. "A logic of concrete time intervals." In *Proceedings of the 5th annual IEEE Symposium on Logic in Computer Science*, Philadelphia, PA, June 1990. IEEE Computer Society Press.

[Lin87] Kwei-Jay Lin, Swaminathan Natarajan, and Jane Liu. "Imprecise results: Utilizing partial computations in real-time systems." Technical report, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL, April 1987.

[Lin88] Kwei-Jay Lin and Swaminathan Natarajan. "Expressing and maintaining timing constraints in FLEX." In *Proceedings of the 9th IEEE Real-time Systems Symposium*, pages 96–105, Los Alamitos, CA, July 1988. IEEE Computer Society Press.

[Lin91] Kwei-Jay Lin and Kevin Kenny. "Building flexible real-time systems using the FLEX language." *IEEE Computer*, 24(5):70–78, May 1991.

[Liu87] Jane Liu, Kwei-Jay Lin, and Swaminathan Natarajan. "Scheduling real-time, periodic jobs using imprecise results." Technical report, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL, April 1987.

[Liu91] Jane Liu, Kwei-Jay Lin, Wei-Kuan Shih, Albert Chuang shi Yu, Jen-Yao Chung, and Wei Zhao. "Algorithms for scheduling imprecise computations." *IEEE Computer*, 24(5):58–68, May 1991.

[Lync88a] Nancy Lynch and Kenneth Goldman. "6.852 distributed algorithms lecture notes: The I/O Automata." Technical report, Laboratory of Computer Science, MIT, Cambridge, MA, Fall 1988.

[Lync88b] Nancy Lynch and Mark Tuttle. "An introduction to Input/Output Automata." Technical Report MIT/LCS/TM-373, MIT, Cambridge, Massachusetts, November 1988.

[Lync89a] N. Lynch, M. Merritt, W. Weihl, and A. Fekete. "Atomic transactions." In publication, 1989.

[Lync89b] Nancy Lynch and Hagit Attiya. "Assertional proofs for timing properties." Technical Report MIT/LCS/TM, MIT, Cambridge, Massachusetts, August 1989.

[Lync89c] Nancy Lynch and Hagit Attiya. "Using mappings to prove timing properties." Technical Report MIT/LCS/TM-412.b, MIT, Cambridge, Massachusetts, December 1989. Also in *Proceedings of the 1990 ACM Symposium on Principles of Distributed Computing*, pp. 265-280.

[Lync91] Nancy Lynch and Frits Vaandrager. "Forward and backward simulations for timing-based systems." Unpublished notes, Massachusetts Institute of Technology Laboratory for Computer Science, August 1991.

[Lyon89] Damian Lyons and Michael Arbib. "A formal model of computation for sensory-based robotics." *IEEE Transactions on Robotics and Automation*, 5(3):280–293, 1989.

[Lyon90] Damian Lyons. "A formal model for reactive robot plans." In *Proceedings of the 2nd International Conference on Computer Integrated Manufacturing*, Troy, New York, May 1990.

[Lyon91a] D. Lyons and A. Hendriks. "Reactive planning." Technical Report Philips TR-91-016 (MS-91-023), Philips Laboratories, Briarcliff Manor, New York, April 1991. To appear in the 2^{nd} edition of the Encyclopedia of Artificial Intelligence (S. Shapiro, Editor-in-chief – John Wiley & Sons, Inc.).

[Lyon91b] D. Lyons, A. Hendriks, and S. Mehta. "Achieving robustness by casting planning as adaptation of a reactive system." Technical Report Philips TN-91-011, Philips Laboratories, Briarcliff Manor, New York, February 1991.

[Maes90] Pattie Maes. "Situated agents can have goals." *Special issue of Journal of Robotics and Autonomous vehicle control*, Spring 1990. Also, in Designing Autonomous Agents - Pattie Maes editor, MIT Press.

[Mann82] Zohar Manna and Amir Pnueli. "Verification of concurrent programs: Temporal proof principles." *Lecture notes in Computer Science*, 131, 1982.

[Merl74] P. M. Merlin. *A study of the recoverability of computing systems.* PhD thesis, Department of Information and Computer Science, University of California, Irvine, CA, 1974.

[Merl76] P. M. Merlin and D. J. Faber. "Recoverability of communication protocols: Implications of a theoretical study." *IEEE Transactions on Communication*, COM-24:1036–1043, September 1976.

[Merr89] Michael Merritt. "Completness theorems for automata." , May 1989. In *REX Workshop.*

[Mish83] B. Mishra and E. Clarke. "Automatic and hierarchical verification of asynchronous circuits using temporal logic." Technical Report CMU-CS-83-155, Carnegie-Mellon, September 1983.

[Mosz85] Ben Moszkowski. "A temporal logic for multilevel reasoning about hardware." *IEEE Computer*, 18(2), February 1985.

[Nils88] Nils Nilsson. "Action networks." In *Proceedings of the Rochester Planning Workshop: From Formal Systems to Practical Systems*, University of Rochester, Rochester, NY, October 1988.

[Nils90] Nils Nilsson and Azer Bestavros, November 1990. Private discussions.

[Papa79] Christos Papadimitriou. "The serializability of concurrent database updates." *Journal of the ACM*, 26(4):631–653, October 1979.

[Pnue77] Amir Pnueli. "The temporal logic of programs." In *Proccedings of the IEEE Annual Symposium on Foundations of Computer Science*, November 1977.

[Puck88] Douglas Pucknell and Kamran Eshraghian. *Basic VLSI design: Systems and Circuits (second edition).* Prentice Hall, 1988.

[Ramc74] C. Ramchandani. *Analysis of asynchronous concurrent systems by timed Petri nets.* PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, 1974. Project MAC Report MAC-TR-120.

[Razo83] R. Razouk. "The derivation of performance expressions for communication protocols from timed petri net models." Technical Report 211, Department of Information and Computer Science, University of California, Irvine, CA, November 1983.

[Read86] James Ready. "VRTX: A real-time operating system for embedded microprocessor applications." *IEEE Micro*, 6(4):8–17, August 1986.

[Reed88] G. M. Reed and A. W. Roscoe. "A timed model for Ccommunicating Sequential Processes." *Theoretical Computer Science*, 58:249–261, 1988.

[Rubi87] Dean Rubine and Roger Dannenberg. "ARCTIC: Programmer's manual and tutorial." Technical Report CMU-CS-87-110, Carnegie Mellon, Pittsburgh, PA, June 1987.

[Schn88] Fred Schneider. "Critical (of) issues in real-time systems: A position paper." Technical Report 88-914, Department of Computer Science, Cornell University, Ithaca, NY, May 1988.

[Schw87] Karsten Schwan, Prabha Gopinath, and Win Bo. "CHAOS – Kernel support for objects in the real-time domain." *IEEE Transactions on Computers*, 36(8):904–916, August 1987.

[Shih91] Wei-Kuan Shih, Jane Liu, and Jen-Yao Chung. "Algorithms for scheduling imprecise computations with timing constraints." *SIAM Journal of Computing*, July 1991.

[Sifa77] J. Sifakis. "Petri nets for performance evaluation." In H. Beilner and E. Gelenbe, editors, *Measuring, Modeling, and Evaluating Computer Systems (Proceedings of the 3rd Symposium, IFIP Working Group 7.3)*, pages 75–93. North-Holland, Amsterdam, The Netherlands, 1977.

[Sree90] Ramavarapu Sreenivas. *Towards a system theory for interconnected Condition/Event systems.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, September 1990.

[Stan87] John Stankovic and Krithi Ramamritham. "The design of the Spring kernel." In *Proceedings of the Real-time Systems Symposium*, pages 146–157. IEEE Computer Society Press, December 1987.

[Stan88a] John Stankovic. "Misconceptions about real-time computing." *IEEE Computer*, October 1988.

[Stan88b] John Stankovic and Krithi Ramamritham, editors. *Hard Real-Time Systems.* IEEE Computer Society Press, 1988.

[Stan89] John Stankovic and Krithi Ramamritham. "The Spring Kernel: A new paradigm for real-time operating systems." *ACM Operating Systems Review*, 23(3):54–71, July 1989.

[Stan92] John Stankovic and Krithi Ramamritham. *Advances in Hard Real-time systems.* IEEE Computer Society Press, 1992. (to appear).

[Stua91] D.A. Stuart and P.C. Clements. "Clairvoyance, capricious timing faults, causality, and real-time specifications." In *Proceedings of the 12th IEEE Real-time Systems Symposium*, pages 254–263, San Antonio, Texas, December 1991.

[Terw87a] Robert Terwilliger and Roy Campbell. "ENCOMPASS: An environment for the incremental software development." Technical Report UIUCDCS-R-86-1296, Department of Computer Science, University of Illinois at Urbana-Champaign, June 1987.

[Terw87b] Robert Terwilliger and Roy Campbell. "PLEASE: A language for incremental software development." In *Proceedings of the 4th International Workshop on Software Specification and Design*, April 1987.

[Terw87c] Robert Terwilliger and Roy Campbell. "PLEASE: Executable specifications for incremental software development." Technical Report UIUCDCS-R-86-1295, Department of Computer Science, University of Illinois at Urbana-Champaign, June 1987.

[Terw88] Robert Terwilliger and Roy Campbell. "An early report on ENCOMPASS." In *Proceedings of the 10th International Conference on Software Engineering*, April 1988.

[Tilb91a] André M. van Tilborg and Gary M. Koob, editors. *Foundations of Real-Time Computing: Formal Specifications and Methods.* Kluwer Academic Publishers, 1991.

[Tilb91b] André M. van Tilborg and Gary M. Koob, editors. *Foundations of Real-Time Computing: Scheduling and resource management.* Kluwer Academic Publishers, 1991.

[Tutt88] Mark Tuttle, Michael Meritt, and Francesmary Modugno. "Time constrained automata." MIT/LCS, November 1988.

[Wils89] Philip Wilsey. "Computer architecture specification with interval temporal logic." In *Proceedings of the 9th International Symposium on Computer Hardware Description Languages*, pages 35–45, June 1989.

[Wils90] Philip Wilsey. "The use of interval temporal logic in specifying relationships between clock phases." In *Proceedings of TAU'90: The 1990 ACM International Workshop on Timing issues in the Specification and Synthesis of Digital Systems*, Vancouver, Canada, August 1990.

[Wind89] Wind River Systems, Inc., Emeryville, CA. *VxWorks Version 4.0.2: Programmers Guide and Reference Manual*, 1989.

[Wirt77] Niklaus Wirth. "Toward a discipline of real-time programming." *Communications of the ACM*, 20(8), August 1977.

[Yann84] Mihalis Yannakakis. "Serializability by locking." *Journal of the ACM*, 31(2):227–244, April 1984.

[Zave82] Pamela Zave. "An operational approach to requirements specification for embedded systems." *IEEE Transactions on Software Engineering*, 8(3), May 1982.

[Zave84] Pamela Zave. "The operational versus the conventional approach to software development." *Communications of the ACM*, 27(2), February 1984.

[Zave86] Pamela Zave. "Salient features of an executable specification language and its environment." *IEEE Transactions on Software Engineering*, 12(2), February 1986.

[Zave88] Pamela Zave. *PAISLey User Documentation: Volume 1, Volume 2, Volume 3.* Computing Systems Research Laboratory, AT&T Bell Laboratories, Murray Hill, NJ, 1988.

Index

action *24*
  class *28*
  input *8*
  local *8*
automata
  IOA *8*
  IOTA *9*
  Mealy *23*
  TRA *23, 24*
behaviors
  basic *88*
  clairvoyant *23, 64*
  competing *92*
  emergent *93*
  intelligent *93*
  reactive *88*
  subsuming *90*
  synthesized *93*
causality principle *11, 13*
channel *24*
  capacity *26*
  class *27*
  fan-in *45*
  fan-out *45*
  readers of *45*
  switching time *26*
  writers of *45*
CLEOPATRA *51*
  completeness *60*
  data types *80*
  explicit type conversion *80*
  implicit type conversion *80*
  semantics *60*
  soundness *60*
  syntax *51, 52*
composition operation *45*
  definition *47*
composition
  parallel *49*
  sequential *49*
computation
  categories *7*
  fixed point of *56*
  imprecise *16*
  nondeterminism *31*
  time-constrained *17*
  time-dependent *17*
conflict relationship
  between actions *32*
  between channels *32*
  between transitions *32*
control
  intelligent *93*
  nondeterminism *31*
  selective *90*
  servo *88*
  teleo-selective *92*
decomposition
  functional *72*
  hierarchical *73*
  modular *67*
development requirements
  modularity *5*
  predictability *14, 4*
  validation *4*
  verification *4*
development support
  operating systems *18*
  scheduling *19*
embedded system
  installation *67*
embedded systems *2*
  aspects *2*
  components *2*
  constraints *3*
  example testbed *101*
  examples *2*
  external interface *2*
  installation *70*
  programmed system *2*
  sensori-motor applications *99*
equivalence relationship *42*
event *24*
  initiating *39*
execution
  admissible *35*
  causality requirement *38*
  consistency requirement *38*
  finite *40*
  fragment *39*

infinite *40*
legal *40*
legality requirement *38*
partial *39*
safety requirement *38*
spontaneity requirement *38*
Zeno *65*
hiding operation *43*
implementation relationship *41*
necessary condition *41*
strong *41*
weak *41*
input enabled property *28, 46, 8*
intentions
notion *36*
vector *37*
IOA model *22*
nondeterminism
sources *31*
physical
correctness *10*
properties *22*
principle
causality *22*
spontaneity *22, 23*
privacy respect principle *80*
projection operation *40*
properties
fairness *34, 8*
finite progress *8*
liveness *8*
safety *7*
timeliness *7*
property *68*
implementation *69*
preservation *68*
prototyping *7*
real space-time *10, 23, 35*
real-time
compilation *98*
renaming
function *44*
operation *44*
schedule *40*
signal *24*
signature *27*
external *27*
input *27*
internal *27*
local *27*
output *27*
simulation *7*
space
coordinates *10, 22, 35*
locality *35*
uncertainty *32*
specification
black-box approach *68*
conventional approach *68*
conventional (black-box) approach *12*
executable *4*
functional approach *68*
hardcore *96*
languages *12, 14*
logic-based *10*
operational approach *13*
Petri-net-based *11*
reusability/adaptability *5*
state-based *8*
status *37*
initial *39*
stable *39*
succession, direct *38, 39*
succession, extended succession *38, 39*
subsumption
architecture *90, 91*
TRA *91*
systems
causal *22*
closed *81*
control *88*
reactive *22, 88*
spontaneous *23*
stand-alone *81*
testing *7*
TET
after clause *57*
before clause *57*
body *55*
commit clause *57*
definition *55*
disabling condition *56*
fire section *55*
header *55*
input transition *59*
output transition *59*
state transformation schema *57*
time constraint *57*
trigger section *55*
unless clause *56*
within clause *57*
definition *51*
time constraint *28*
activation *37*
active *28*
disabled *28*

time-constrained causal relationships *28*
time
  coordinates *10, 22, 35*
  dense intervals *24*
  intervals *24*
  model *24*
  nondeterminism *31*
  perceived time *23*
  points *24*
  real time *23*
  switching *26*
TRA
  autonomy structure *34*
  behavior *40*
  closed *70*
  compatibility *45*
  computational step *28*
  control components *33*
  definition *24, 27*
  I/O compatibility *45, 80*
  installation *67, 70*
  isomorphic *44*
  operational semantics *36*
  operations on *43, 44, 45, 47*
  proper/improper *33*
  sextuple *27*
  signaling range function *28*
  signature *27*
  start channel *27*
  state space *26, 28*
  state *26, 36*
  status *37*
  strong compatibility *46*
  time constraint *28*
TRA-class
  basic/composite *53*
  body *53*
  header *53*
  parameters *53*
  start channel *53*
validation *79*
  via prototyping *4*
  via simulation *4*
verification
  empirical *5, 7*
  formal *4, 7*
  logic-based *10*
  Petri-net-based *11*
  process-algebra-based *9*
  state-based *8*