SYMBOLIC APPROXIMATIONS FOR VERIFYING REAL-TIME SYSTEMS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

By
Howard Wong-Toi
March 1995

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

David L. Dill
(Principal Adviser)

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Yoav Shoham

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Moshe Vardi
(Rice University)

Approved for the University Committee on Graduate Studies:

Acknowledgements

I would like to thank my advisor, David Dill, for his tremendous guidance and support. It has been a pleasure and privilege to learn from him how to do research. He helped me focus on the important research issues in the field, and suggested finding efficient verification algorithms for real-time systems. His comments have helped to clarify many ideas in this thesis. I am also thankful for his numerous careful readings of thesis drafts. Throughout my years at Stanford, he has been always listening, concerned, and encouraging.

I thank my reading committee members Yoav Shoham and Moshe Vardi. Toward the beginning of my stay at Stanford, Yoav suggested to me solving problems in their simple forms before moving to more advanced formulations. This advice has been extremely useful in tackling tough verification problems. Moshe's comments led to a more careful examination of the lessons learnt building the verifier and have improved the thesis significantly. I also thank Gene Franklin and Hector Garcia-Molina who kindly served on my Orals committee.

I benefitted from many discussions and suggestions from colleagues. In particular, I thank Rajeev Alur for answering numerous questions about timed automata and AT&T's verifier, Nicolas Halbwachs for explaining his approximations for reactive systems and their limitations, and Alan Hu for sharing his expertize on OBDDs. Many thanks go to Gérard Hoffmann: we spent many enjoyable and fruitful hours together discussing research in general and models of discrete event control in particular. My various office mates provided a stimulating research environment at the same time as a friendly atmosphere for grad students blues, as well as answering many questions on hardware and software. For this, I thank Anuchit Anuchitanukul, David Cyrluk, Dinesh Katiyar and Elizabeth Wolf. Thanks also go to all the members of the murphi verification group.

I am grateful to Tom Henzinger for enabling me to finalize my thesis while at Cornell University. Wee Eng Koh and Xiao-Wu Su helped me with completing administrative details.

Many friends at Stanford made my stay here enjoyable. Special thanks go to my host families Andy and Mamie Poggio, and Walter and Sherry Schubert. My memories of Stanford will always include all the sisters and brothers from the Chinese Christian Fellowship at Stanford who made up my home away from home.

Many thanks are due to my wife Sarah for her unceasing love and patience, especially her patience. I also thank my parents for their years of support and encouragement of my studies.

Most of all, I am thankful in all things to God, my creator and redeemer.

Howard Wong-Toi
November 1994

Of making many books there is no end, and much study wearies the body.
Now all has been heard;
here is the conclusion of the matter:
Fear God and keep his commandments,
for this is the whole duty of man.

Ecclesiastes 12:12b,13

Abstract

Real-time systems are appearing in more and more applications where their proper operation is critical, *e.g.* transport controllers and medical equipment. However they are extremely difficult to design correctly: one must consider the sequencing and co-ordination of events in concurrent processes, as well as the times they occur. One approach to this problem is the use of formal description techniques and automatic verification. Unfortunately automatic verification suffers from the state-explosion problem and is computationally expensive even without real-time. The addition of timing information makes the problem much harder. This thesis proposes a state-based approximation scheme as a heuristic for reducing the effort required in verification.

We first describe a generic iterative approximation algorithm for checking safety properties of a transition system. It exploits the fact that not all the details of a system need be considered to prove it correct. Successively more accurate approximations of the reachable states are generated until it can be determined whether the specification is satisfied or not. The algorithm automatically decides where the analysis needs to be more exact, and uses state partitioning to force the approximations to converge towards a solution. In the case of finite-state systems, the method is complete.

The algorithm is used to verify that systems with hard real-time bounds satisfy timed safety properties. State approximations are performed over both timing information and control information. We also approximate the system's transition structure. Case studies include some timing properties of the MAC sublayer of the Ethernet protocol, the tick-tock service protocol, and a timing-based communication protocol where the sender's and receiver's clocks advance at variable rates.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Many computer-related systems are time-critical: they may depend on timing information for their correct operation, or their specifications may require certain tasks to be performed within specific time bounds. Typical examples include embedded systems, communication protocols, and transportation controllers. In many of these applications, correct operation is imperative. Failures may result in financial disaster, system shut-downs, physical harm, or in some cases even the loss of lives. However, it is generally accepted that it is a difficult task to specify and reason about the timing behavior of concurrent systems. It is easy for an *ad hoc* analysis, or even extensive simulation, to miss crucial cases which lead to errors.

One approach to this problem is to develop mathematically formal methods for system verification. The idea is to be able to *prove* that the system is correct rather than to assume it is because no bugs have been discovered so far. In this framework, a potential system implementation can be modeled formally and analyzed against a specification early in the design cycle. Logical design bugs can be removed before they percolate down to lower levels of implementation. As an implementation is refined, it can be verified against its higher-level description. The major drawback of this approach is that formal specifications quickly become too complex to analyze manually as the size of the system increases. With today's computer-based applications growing ever larger and larger, there is a need for computer assistance in the verification process. Indeed one strategy is to use a fully automatic verification technique. Here, the user supplies a verification tool with a formal system description and a specification for it, and then waits for the verifier to check, without any further human assistance, whether the system is correct or not.

Up until recently, verification methodologies had abstracted away the times at which events occur, and concentrated on the logical sequencing of actions. While such an abstraction is often useful, it is clearly not acceptable when the specification includes timing properties. Over the last few years, numerous formalisms have been proposed for describing the real-time behavior of concurrent systems, by either extending existing techniques or developing whole new theories. Indeed, the automatic verification problem for some classes of finite-state real-time systems has been solved, in theory [Dil89, AH89, AD90, Lew90, ACD90, LV92, HNSY92]. In many cases, there are known algorithms that are theoretically optimal in the worst case. However, from a practical standpoint, these algorithms are computationally infeasible on realistic examples. They have to deal with an extremely large number of reachable states, as well as taking into account the times at which they are reached. Algorithms are typically exponential in the size of the untimed part of the system description, and also exponential in the system's timing information. So while a verification engineer has a large choice of models to describe her system formally, she is left with no practical tools to verify the system is correct. Our goal is to address this shortcoming by using heuristic techniques to make automatic verification of real-time systems computationally feasible.

1.2 Approximation

This thesis describes an efficient automatic approximation scheme which has been applied to the verification of timed safety properties. It is based on the observation that usually not all of a real-time system's timing information is necessary to establish its correctness. The basic idea is perform symbolic simulation of the system's execution traces while simultaneously checking whether they violate the specification. The simulation however is only approximate. The set of reachable states is approximated from above and from below. If the overapproximation contains no *violating states*, *i.e.* states where an error has been detected, the system is successfully verified. If the underapproximation contains violating states, the system is not correct.

Taking approximations instead of computing the exact set of reachable states can be computationally advantageous. Firstly, the size of the symbolic approximation may be far smaller than the representation of the exactly reachable states. Secondly, the time required to generate an approximation may be less than for performing precise reachability analysis.

Approximation, however, is not always accurate enough to determine whether the system satisfies its specification. There is the possibility of false negatives (if the overapproximation contains violating states, these may or may not be truly reachable states of the system) and false positives (the underapproximation may not include violating states which are reachable). Thus the result from approximating may be inconclusive.

Our algorithm tackles this problem by iteratively refining the approximations so that they converge towards the truly reachable states. It is *complete* for finite-state systems, in that it always decides exactly whether the system is correct. We also prove completeness for the class of real-time systems we verify.

The key idea behind the iterative scheme is to limit where approximations are taken. This is achieved by partitioning the state-space into different regions, where states in the same region are believed to behave similarly. Approximation of reachable states is carried out within each region. When it is discovered that states in the same region have sufficiently different outgoing behaviors, the partitioning is refined. This successive refinement leads to progressively more accurate approximations.

1.3 Contributions

The main contributions of this thesis are a generic framework for iterative approximations for safety verification, an efficient approximation algorithm for real-time systems, and the demonstrated automatic verification of non-trivial real-time systems, including a model of real-time systems with skewed clocks.

The iterative algorithm proposed in this thesis solves the safety verification problem. It is flexible enough to apply to many different problem domains. At the barest level, the algorithm designer needs to provide a symbolic system representation, including set representations and next-state operators, and two approximating operators. If desired, he can also add any number of his own heuristics to speed convergence.

The algorithm itself uses dynamic refinement of approximations, rather than statically determined convergence. This means that it attempts to determine automatically which parts of the state-space need to be analyzed more carefully, and where approximations can be more liberal. It is easily parameterizable to begin approximating as finely or loosely as desired. There is a limited capacity for user-supplied information to be exploited, by instructing the program where to start approximating more aggressively. Both backwards and forwards reachability information is utilized, whereas most verification methodologies choose one direction only. This is possible since we can take a quick analysis in one direction, and then combine that with information from the other, rather than being bogged down in an exact analysis in only one direction, or attempting to compute exact reachability in both directions at the same time. While the main algorithm is based on state approximations, the theory also allows next-state relations to be approximated. The algorithm is shown to terminate over finite-state systems.

The algorithm is applied to real-time verification, using both state approximations and transition relation approximations. Our method is the first to benefit substantially from combining symbolic representations of control information and timing information. We also provide a natural and efficient handling of *urgency* semantics, where urgent events are events which must take place as soon as they are enabled.

We use our tool to automatically verify several non-trivial real-time systems. We verify some timing properties of an abstracted Ethernet MAC sublayer protocol. We also introduce a new subclass of linear hybrid automata that models systems where clocks advance at variable linearly-bounded rates. Using this model, we verify a recently published audio control protocol which uses Manchester-encoded bit streams. Communication is between processes which have a fixed error tolerance in their clock speeds. We automatically prove that for arbitrary length messages, all bits are received correctly and in a timely fashion. The performance of our tool compares favorably to the symbolic real-time verifier KRONOS developed by Sifakis et al [NSY92a] at VERIMAG, France. Finally we describe our experience with developing verification methodologies for real-time systems.

1.4 Real-time systems

Recently there have been many formal description techniques proposed for describing real-time systems and their timing properties. We outline the model we use (timed safety automata [NSY92a]), and then compare it briefly with other approaches. Our concern is not so much with a specification technique as the algorithm required to verify correctness, so we concentrate more on the formalisms that lend themselves to automatic verification.

Discrete vs continuous time

A major dividing line in the methods is how they model time, as either a discrete entity, or as continuous. In a discrete time framework, events occur only at discrete clock ticks. In the continuous time model, events may occur at any real-valued time. The main advantages and disadvantages of each approach are listed below.

- discrete: In this framework, it is easy to incorporate time into many existing untimed models, specification languages, and implementations. A discrete notion of time is accurate for some classes of processes, such as synchronous hardware.

- continuous: The continuous time model is more natural and accurate, especially since it can be shown that in some cases the time domain cannot be discretized sufficiently finely for an accurate semantics [Alu91]. However introducing continuous time involves new models, more complex semantics, and more complex reasoning.

Timed safety automata operate in continuous time. We later show that the savings due to performing discrete computation may be minimal compared to continuous, since there are no known symbolic methods for discrete time which outperform those for continuous time models.

1.4.1 Timed safety automata

We use *timed safety automata* (TSAs) to describe real-time systems and their specifications [HNSY92, NSY92a]. They operate with finite-state control. Time is modeled through the addition of a finite set of fictitious clocks [AD90, AH94]. Each clock records the exact time which has elapsed since its last reset. Timing conditions are expressed by constraints on when events may occur. Following the introduction of timed automata by Alur and Dill [AD90] there have been many variants described in the literature. The particular version we use is taken from Nicollin et al [NSY92a] and augmented with urgency semantics. Local progress is enforced by constraining the amount of time which can pass while control rests in a location. However, these automata have no means of expressing unbounded fairness information.

1.4.2 Other formalisms

Real-time logics

Temporal logics [Pnu77, Pnu86, CES83] have met wide success in reasoning about untimed reactive systems. Naturally, these logics are a good starting point for developing logics that can reason directly about a system's timed behaviors. Properties are expressed using formulas such as "$p \Rightarrow \Diamond_{<3} q$" to mean that when p is true, q will eventually be true within 3 time units. See [AH92] for a excellent survey of logics for real time. Of the logics interpreted over dense models, only MITL of Alur et al [AFH91] is known to be decidable. A number of decidable logics use a fictitious clock as a global integer variable to record the current time, for instance RTTL [Ost92], XCTL [HLP90], and RTCTL [EMSS89]. Essentially all discrete time extensions to decidable logics are decidable.

Process algebras

A process algebra is a calculus with operations for building more complex processes from simple ones [Mil80, Hoa85]. Typically the simplest processes are single events, and there are operations for sequential composition, parallel composition, hiding of events, synchronization, and non-deterministic choice. Algebraic laws state that some processes are equal to others; for example, the choice operator may be commutative. Time is usually introduced into a process algebra through a mechanism to explicitly model the passing of time. It is may be in the form of a unary delay operator [Yi90] or a special process. For example "$\Delta(t).P$" may be used to represent the process which delays for t time units and then behaves like P. There may also be other operators such as a timeout operator, which states that a process executes for some fixed amount of time and then behaves like another. Examples of such timed process algebras are Timed CSP [RR88], TCCS [Yi90], and ATP [NSV90].

Duration calculus

The calculus of durations [CHR91] is an extension to interval temporal logic which allows reasoning about the durations of states within an interval, without explicit mention of absolute time. A duration formula $\int P = 5$ is true for an interval $\mathcal{I}$ when $\int_{\mathcal{I}} P = 5$, and the formula $\int P \leq 20 \int Q$ intuitively means that Q holds over the interval at least $1/20$ of the time that P holds. In addition to the usual boolean operations on formulas, there is a chop operation denoted $(D_1; D_2)$ which is true over an interval whenever it can be partitioned into two consecutive parts, the first of which satisfies D_1 while the second satisfies D_2. Formulas may be interpreted over discrete time or continuous time. In general, the calculus is undecidable. However, Chaochen et al [CHS93] have identified decidable fragments: allowing only primitive formulas of the form $\lceil P \rceil$, which assert that P holds almost everywhere over an interval, is decidable for both the dense time and discrete time versions, and admitting formulas $l = k$, which express that the interval is of length k, maintains decidability for the discrete time calculus only.

Other state based approaches

Lewis' *state-diagrams* [Lew90] are very similar to timed automata. The enabling conditions on transitions are based on delays between pairs of events, rather than delays since individual events occurred. The primary advantage of timed automata is that they have a simpler definition and semantics.

Ostroff's *timed transition machines (TTMs)* [Ost92] and the *timed transition systems* of Henzinger et al [Hen91] are timed extensions of Manna and Pnueli's fair transition systems. Each transition is associated with a lower time bound and an upper bound. An execution is timing consistent if every transition which fires has been continuously enabled no less than its lower time bound and no more than its upper bound, and no transition is continuously enabled for longer than its upper bound without firing. *Timed I/O automata* [LA90] correspond to the analogous extension to I/O automata. The finite-state versions of all these machines can be modeled by timed safety automata, except that unbounded fairness cannot be expressed, nor is there any structure to model the input/output events of timed I/O automata. However, it should be noted that all timing aspects of these transition systems can be captured.

Various real-time extensions have been proposed for Petri nets. Time bounds may be placed on the lives of tokens [Van93] or enabled transitions [MF76], or delays may be associated with transitions [Ram74]. Again, timed automata are generally as expressive as all the finite-state versions of these nets.

There are other state-based formalisms which allow more general modeling of real-time systems. Hybrid systems model finite-state systems augmented with continuous variables that evolve according to differential equations. They can be used to model skewed clocks, drifting clocks, and interrupted clocks, as well as analog variables such as pressure and temperature.

We choose to use timed automata because of their simplicity, expressiveness, and algorithmic solutions to verification problems. Many other formalisms are no more expressive (at least not as far as representing timing information) and can be compiled into timed automata, or they have undecidable verification problems.

1.4.3 Verification

We briefly survey the verification techniques associated with the formalisms above. Most algorithmic verification is no easier than verification using timed automata. In fact, many of the verification problems which have algorithmic solutions can be solved by the same techniques required for verifying timed automata. Therefore we consider the practical verification of timed automata a major issue in real-time verification.

Timed automata

Alur et al [AD90, ACD90] show how timed automata may be analyzed by first constructing a finite quotient graph called a *regions graph*. Its equivalence classes are in some sense a bisimulation of the system's states. Typically an analysis problem for a timed automaton is reduced to its untimed counterpart over the regions graph. The finiteness of the regions graph allows numerous analysis problems to be solved, including bisimulation equivalence, automata emptiness, model-checking of TCTL formulae, reachability, and controller synthesis [Cer93, ACD90, HNSY92, CY92, WTH91]. Unfortunately, the regions graph is exponential in the number of time-keeping elements in the system, and also in the size of the timing constants used. The main problem which this thesis tackles is reachability, which is known to be PSPACE-complete. This exponential blow-up makes automatic verification of real-time systems particularly challenging. Previous approaches to tackle this state-explosion are described in the next section.

Logics

One way logics can be used to verify timed systems is by proving the validity of the formula $\phi \Rightarrow \psi$, where ϕ defines the system and ψ its specification. However most temporal logics over dense models are undecidable. For the decidable logics, the complexity of the decision procedure is typically one exponential more than for its untimed version – the same blow-up we encounter moving from untimed automata to timed automata. Furthermore there has been no work that we know of for developing heuristic decision procedures for these logics.

Model-checking is an alternative verification method where the system is given as a proposed model to be checked against the logical specification. For some logics the complexity of model-checking is better than for validity, and in the cases of XCTL [HLP90] and TCTL [ACD90] it is PSPACE-complete. It is in theory then no easier than reachability of timed automata.

Temporal proof systems may also be used to establish the validity of temporal formulae. However our main interest here is in automatic methods, rather than human-generated proofs.

Process algebras

Correctness of process algebras is usually defined in terms of a process equivalence (where processes have similar behavior according to some criterion such as observable traces) or preorder (where an implementation is intended to refine a specification). Verification consists of either constructing proofs using the algebraic laws associated with the operators, or by compiling process algebraic terms into graphs which are then tested for equivalence or simulation preorders. For timed process algebras, the graphs for the algebraic terms are essentially timed automata [Cer93, NSY92b]. So yet again, verification reduces to analysis of timed automata.

Duration calculus

For some restricted subclasses of the duration calculus, the sets of satisfying behaviors are regular sets [CHS93]. Skakkebæk et al [SS93] discuss a verification strategy and implementation based on converting duration calculus formulas into regular expressions and checking for emptiness.

Petri nets

The usual way Petri nets are used for verification is by performing a reachability analysis and testing for whether a marked place is reached. The Petri net formalism of Rokicki [Rok93] models timed circuits and uses a notion of conformance as its correctness criterion. Again, the finite-state versions of these nets (*i.e.* those with a bounded number of markings, or k-safe), could be analyzed by the same reachability techniques used for timed automata. However more direct methods have been advocated, and are described in the next section.

Other state-based approaches

Timed transition systems are proven correct by using a temporal proof system. Lynch et al [LA90, LV92] study the use of mappings and simulations between timed I/O automata to establish that one implements another. Neither of these two approaches is designed for automatic verification.

1.5 Related work

We describe previous attempts at tackling the state-space explosion encountered when verifying real-time systems. Most closely related are other approximation methodologies designed specifically for real-time systems [AIKY93, BSV93]. Other approaches directly using the timed automaton formalism include building minimal regions graphs [ACH+92, ACD+92], symbolic model-checking [HNSY92], and reachability graphs [KL94]. We also outline some related work on Petri net reachability analysis [BM83, YTK91, Rok93].

Finally, we give a brief comparison with similar work in the domain of abstract interpretations [CC92], and mention some other fields where state based approximation has been used.

1.5.1 Iterative approximations

The iterative method we propose is not the only viable iterative scheme for approximating the behavior of a real-time system. We know of two other iterative approximation schemes which converge to an answer to the correctness problem. Alur et al [AIKY93] and Balarin et al [BSV93] describe approximation algorithms which use a different methodology from that advocated here. Their approach assumes that not many timing constraints in the system are necessary for its correct operation. Based on this premise, they initially attempt to verify the system based only on logical constraints, *i.e.* ignoring all timing information. When a potential violating trace is detected, timing constraints are used to determine how the untimed sequence is not timing consistent, if possible. An untimed automaton which eliminates the false negative based on the effect of these timing constraints is then added into the system. Alur et al add the minimized regions graph for the constraints, and Balarin et al add subprocesses which monitor difference constraints between clocks. The algorithms generate additional useful information about the system: if the system can be successfully verified, we know that the only constraints necessary for correctness are those that have been iteratively added by the algorithm. Other constraints can be ignored. Also Alur et al's algorithm uses a clever rounding of the timing constants in order to keep the regions graphs for each approximation small. This feature also provides parametric information about system correctness. The drawback of these algorithms is that while they approximate the system description (by dropping constraints) they still require exact computation of the regions graph for each abstracted system.

By comparison, our algorithm performs its approximations based on state information. It maintains all timing constraints on transitions, but then discards information from the states which are reached. Refinement of our approximations is primarily state based, rather than transition based, although local approximation of the time-passage transition is also performed. Our algorithm is general enough to allow approximations over control information. It can also easily be applied to systems other than real-time systems.

Ostroff [Ost92] uses formulas in real-time temporal logic to describe forward and backwards heuristic approximations. Since his underlying model is not finite-state, and his specifications are more expressive, his method does not automatically decide whether a property holds or not. Instead he shows how heuristics can be used to provide helpful hints to a human attempting a proof of a property.

1.5.2 Analyzing timed automata

Alur et al [ACH+92, ACD+92] approach the state-explosion problem of the regions graph by building a minimal regions graph instead of the full graph. Nodes in the minimal graph are clustered equivalence classes from the regions graph. While this leads to far fewer nodes in the generated graph, experience shows that even these graphs can easily exceed available memory.

Our iterative approximation scheme bears resemblance to the minimization algorithms of Lee and Yannakakis [LY92, YL93]. A closer study of the relationship could lead to improved algorithms. Lee and Yannakakis' algorithms cleverly partition the reachable states of an implicitly defined system into the minimal number of bisimulation equivalence classes, while here we are interested only in reachability. However their marking of points may be considered an underapproximation of the reachable states, and the potentially reachable blocks an overapproximation. The role of the separating classes of our approximation algorithm is similar to the splitting of blocks in minimization. We are only interested in reachability, not bisimulation equivalence and so we need not partition the state-space as finely. In addition, we make use of backward reachability information.

Kang and Lee [KL94] have recently proposed an alternative approach to solving the reachability problem for timed automata. Rather than build a regions graph (where states are partitioned according to the values of their timers), they generate a reachability graph where relative delay information is encoded on the transitions. A state is reachable if the constraints on a path to it in the reachability graph are satisfiable.

Symbolic model-checking [HNSY92, NSY92a] involves iteratively computing the set of timed states of the system which satisfy each subformula of its TCTL specification. In this sense, the computation is very much driven by the specification, and involves mainly backwards reachability. This computation is performed symbolically using the same representation for timing information which we use. However the analysis is exact. The model-checking framework is more expressive than the reachability problem we consider.

1.5.3 Petri nets

Berthomieu and Menasche [BM83] show how the reachability problem for safe time Petri nets is decidable. Their symbolic representation of timing information by difference constraints between timers is essentially the same as that of Dill [Dil89]. Yoneda et al [YTK91] exploit the concurrency of transition firings to generate difference constraints which correspond to several possible firing sequences, rather than considering each sequence individually. Although Rokicki's description language is *orbital nets* [Rok93], a Petri-net formalism, his algorithms also compute reachability using constraint matrices. He builds *processes* whose linear executions correspond to multiple interleavings of events. When there is a lot of concurrency in the system, this technique reduces the number of interleavings he must consider and the number of constraint matrices needed to store the reachable states.

1.5.4 Abstract interpretation

Abstract interpretation is a well-studied theory of semantic approximation [CC77, Cou90, CC92]. The approximations described in this paper can be viewed as a combination of abstraction, operation on an abstract domain, and concretization. A similar idea to that of iterating forward and backwards passes, using overapproximations only, to refine the set of reachable states on paths to violating states has been suggested in an abstract interpretation framework for type-checking flowchart programs [KU80], and for analyzing logic programs [CC92].

Halbwachs [Hal93b] successfully applied abstract interpretation to synchronous reactive systems, demonstrating its effectiveness in reducing the computational effort required for analysis. His approximations are taken over discrete variables, and he uses polyhedra for describing the reachable variable values. He does not consider approximations over control information. Moreover no means of refining approximations is given, so if a verification attempt fails, there is no way to tell if it is due to a real error, or simply inaccurate approximation.

The full algorithm presented here is the first which uses both underapproximations and overapproximations, and for finite-state systems, automatically determines precisely whether there are reachable violating states.

1.5.5 Other applications of approximation

Approximation techniques have been used in many fields other than verification. We briefly describe the approaches most closely related to this thesis.

Approximate methods for logical inference have been studied in artificial intelligence. Levesque [Lev84, Lev89] introduced the idea of *limited inference* to model an agent's "shallow" reasoning process based on simple inference rules. Kautz and Selman [SK91] advocate *knowledge compilation* of propositional theories into *Horn approximations*. Their idea is that an intractable theory may be reduced to a stronger (or weaker) Horn theory, allowing efficient reasoning over the Horn theories. If the Horn theories do not answer the logical inference problem, the method resorts to the exact theory. Cadoli [Cad92] describes a method which does allow both sound and complete approximations in a framework that incrementally iterates toward an exact answer. Roughly speaking, more accurate approximations are obtained by increasing the number of literals that are semantically consistent. However his methodology provides no semantically based means of dynamically choosing *how* the approximations are to be refined.

Various state based approximations are based on the idea of divide-and-conquer. Typically a 2-dimensional or 3-dimensional state-space is partitioned using hierarchical data-structures called quad-trees or oct-trees. In the case of quad-trees, the root node represents a two-dimensional space. Each internal node has 4 leaves, representing neighboring sets which partition the node. The quad-tree is built dynamically, with new nodes created whenever a leaf node needs to be analyzed more carefully. The idea is to work efficiently with large chunks of the state-space as much as possible, subdividing a node only when necessary. This method has been used successfully in a number of different fields, such as path-planning for robots [AKH88], image processing [MM88], and VLSI layout design [HF90]. With the exception of robot path-planning, the problem domains admit varying degrees of accuracy (*e.g.* many image resolutions are acceptable) in their solutions and the chief concern is with obtaining a good approximation with low computational expense, as opposed to using approximation as an efficient means of finding an exact answer. In addition, our problems involve more complex state-spaces, a combination of n-dimensional spaces for timing information, where n is the number of clocks in the system, and a discrete component for the control information. This state-space complexity does not allow an easy and effective application of the quad-tree approach.

1.6 Outline of thesis

In the remainder of this chapter, we provide some introductory notation, describe the framework we use for verifying safety properties, and explain how symbolic computation can speed up verification.

Chapter 2 describes the main approximation algorithm for a generic safety verification problem. In the next chapter, we describe in more detail the model of real-time systems we consider, and its safety verification problem. The next two chapters outline how the approximation algorithm can be applied to the verification of real-time systems, firstly approximating over only timing information, then over the control information as well[1]. Case studies appear in the following chapter. Chapter 7 describes a class of hybrid systems which can be verified exactly via a reduction to real-time systems. Chapter 8 discusses a prototype implementation, gives performance results, and describes some of the lessons we learnt in building verifiers for real-time systems. Finally, conclusions can be found in chapter 9.

[1] The main generic approximation algorithm and its application to simple timed automata without approximations of the next-state relation appears in [WTD94].

1.7 Preliminaries

1.7.1 Transition systems

We model a process P as a *transition system* $\langle S, S_0, N\rangle$ where S is the underlying state-space of the system, $S_0 \subseteq S$ is a set of *initial states*, and $N \subseteq S \times S$ is a *next-state relation*. A transition system describes a directed graph in the usual way. We sometimes write $s \rightarrow s'$ and $N(s, s')$ to mean that $(s, s') \in N$. For a set of states A, we abuse notation and simply use $N(A)$ to mean the set of successors of A, *i.e.* $N(A) = \{y \in S \mid \exists\ x \in A \text{ s.t. } N(x, y)\}$. An *execution trace* of the system is any infinite sequence of states $s_0, s_1, s_2, \ldots$ such that $s_i \in S$ and $(s_i, s_{i+1}) \in N$ for $i \geq 0$. A *partial trace* is a finite sequence $s_0, s_1, \ldots, s_n$ such that $s_i \in S$ and $(s_i, s_{i+1}) \in N$ for $0 \leq i \leq n - 1$. A trace is initialized iff its first state lies in S_0. The transition system is *non-deadlocking* iff every initialized partial trace of the system is extensible to an infinite execution trace.

A state s' is said to be *forward reachable* from s in P iff there is a path in the graph for P from s to s'. In this case, the state s is called an *ancestor* of s', and s' is a *descendant* of s. A state s is *backwards reachable* from s' iff there is a path in P from s to s'. We define the set of states *reach*$(\mathcal{S})$ to be the states which are forwards reachable from an initial state.

An equivalence relation $\approx$ over the states of the system is a *bisimulation* iff whenever $s_1 \approx s_2$ and $s_1 \rightarrow s_1'$ then there exists a state s_2' such that $s_2 \rightarrow s_2'$ and $s_1' \approx s_2'$.

1.7.2 Safety verification problem

The problem we are interested in solving is called the *safety verification problem*. Intuitively, a process is correct iff it never does anything "bad".

A common framework for verification uses trace inclusion as its correctness condition. The process P is modeled by a formal language $L(P)$ describing the possible infinite execution traces of the system. Its specification is also given as a language L_S of infinite traces, and it represents a maximal set of permissible traces. The process is said to be correct iff $L(P) \subseteq L_S$. In the automata-theoretic approach [VW86], correctness can be decided by checking for emptiness of an automaton representing $L(P) \cap \overline{L_S}$. We consider a special case of this approach, which we call the safety verification problem.

A *(safety) verification problem* $\mathcal{VP} = (S, S_0, N, V)$ consists of a process $P = \langle S, S_0, N\rangle$ together with a set of violating states $V \subseteq S$ which indicate that the process has violated some user-specified safety property. The process is said to be correct iff no violating states are reachable from S_0.

The trace inclusion problem can be expressed in the form of a safety verification problem when the process is non-deadlocking and the specification is a *safety property*. The specification language L_S is a safety property iff it is a closed language, *i.e.* whenever an infinite string w has infinitely many prefixes which are prefixes of strings in L_S, then w is also in L_S. Intuitively, to verify a safety property, we may simulate the execution traces of a non-deadlocking process P together with a monitor which enters a violation state precisely when P does something undesirable (the partial trace so far leaves the prefix set of the specification). Because P is non-deadlocking, all partial traces are extensible to infinite traces, and so this violating partial trace can be extended to an infinite violating trace. Thus verification by automata-emptiness reduces to reachability in this case.

If the system is finite-state, it is theoretically possible to enumerate explicitly all reachable states in the state-space, via, for example, a depth-first search. This technique correctly answers the verification problem. However in many cases the state-space is simply too large to be fully explored, or it may even be infinite. This thesis proposes a symbolic state-space approximation technique for reducing the effort required to solve safety verification problems. It is applicable to both finite-state and infinite state-spaces, but termination is only guaranteed over finite spaces.

1.8 Symbolic verification

The use of various symbolic techniques in finite-state verification has led to great success in recent years. The main feature of symbolic verification algorithms is their ability to express information about sets of states succinctly without having to refer explicitly to every set element. Reasoning about a system is done by reasoning about sets of states instead of individual elements. A symbolic algorithm may be computationally advantageous compared to an explicit enumeration technique if the number of set operations required by the symbolic algorithm is small by comparison. The obvious drawback is that computation over sets can be expensive. A symbolic technique which performs a small number of expensive algorithmic steps may do more work overall than an explicit technique which uses a large number of fast operations.

However the potential benefits of symbolic techniques are numerous. In many cases symbolic computation over sets has been shown to be far faster than explicit state-by-state analysis [Bry92, BCM$^+$90, CK91, McM92, FKM91, HWT92b, PD94]. Symbolic representations of sets may also be far smaller than explicitly storing individual states. In fact, memory usage is often a more critical resource than time. In addition, symbolic representations may allow infinite state spaces to be represented.

In order for a symbolic technique to be useful, we require

- a verification algorithm which can be expressed in terms of sets of states, and
- an efficient representation of sets of states.

An efficient symbolic representation of sets should ideally meet the following criteria:

- the representation of a "typical" set encountered by the algorithm should be small.
- there should be fast operations on sets of states for all operations required by the particular algorithm, *e.g.*
  - computing successors of a set of states
  - computing predecessors of a set of states
  - computing intersection
  - computing set difference and complementation

- computing union
- testing equality and emptiness

All the above characteristics are relative compared to the cost of performing explicit analysis, *i.e.* storing and performing computation on all individual set elements. Notice too that the efficient state representation need only be applicable over those sets of states encountered by the algorithm.

We should note at this point that performing computation over sets is only a heuristic technique. Many of the verification problems studied are PSPACE-complete, and the use of symbolic techniques will not overcome the inherent complexity of the problem in the worst-case scenario. However, in practice, some algorithms whose complexity is actually exponentially worse than an explicit enumeration technique perform extremely well on real examples.

In the symbolic methodology, safety properties for the verification problem $\mathcal{VP} = (S, S_0, N, V)$ can be verified by performing the following iterative fixpoint computation:

$$\begin{aligned} F_0 &= S_0 \\ F_{i+1} &= F_i \cup N(F_i) \\ F &= \lim_i F_i \end{aligned} \tag{1.1}$$

The specification is satisfied iff $F \cap V \neq \emptyset$. We assume the limit always exists and is obtained after a finite number of iterations. Note that this assumption is always true when the underlying system has a finite state-space. This algorithm requires the computation of the next-state operator over sets, the union of sets, tests for equality and emptiness, and an intersection operator. While such symbolic verification can often outperform explicit analysis, there are still many situations when even the symbolic representations of states are simply too large and complex. Thus this thesis proposes using only approximate symbolic analysis.

Chapter 2

Approximation

This thesis is built upon the simple observation that it is not always necessary to consider all the details of a system in order to make useful conclusions. In particular, the iterative approximation algorithm for real-time systems is designed to exploit the fact that often not all timing information is relevant to the property being verified. The key idea is to divide the state-space into separate regions, and to perform state approximation within each region. The algorithm is fully automatic, and is guaranteed to terminate correctly whenever the underlying system has a finite equivalence structure (*e.g.* a finite state-space). Furthermore it makes efficient use of both backwards and forwards reachability information. As presented here it is specific to the task of verifying safety properties.

The algorithm is presented in a general framework: while it was developed specifically for verifying timing properties, it is applicable to a wide variety of systems. In chapters 4 and 5, we show how it can be applied to the verification of real-time systems in particular.

2.1 Fundamental approximation algorithm

The technique of *approximation* can be used to extend the usefulness of symbolic analysis. Here we investigate the symbolic approximation of the set of reachable states. Such state-based overapproximations for verifying hard real-time systems were first investigated by N. Halbwachs [Hal93b], and this work is inspired by his success.

The basic idea is to replace the exact "union" on sets of states in equation 1.1 with either of

- an (overapproximating) "join" operator $\sqcup$, satisfying the soundness condition:

$$\text{for all } A, B : A \cup B \subseteq A \sqcup B \tag{OA\_1}$$

  or

- an (underapproximating) "plus" operator $\mathrel{|\!\!\rhd}$ satisfying the soundness condition:

$$\text{for all } A, B : A \subseteq A \mathrel{|\!\!\rhd} B \subseteq A \cup B \tag{UA\_1}$$

  and the nonemptiness condition:

$$\text{for all } A : A \neq \emptyset \text{ implies } \emptyset \mathrel{|\!\!\rhd} A \neq \emptyset \tag{UA\_2}$$

Notice that the second axiom for an underapproximating operator is asymmetric, and that neither operator need be commutative nor associative. The set $A \mathrel{|\!\!\rhd} B$ is referred to as the *expansion* of A with B. Observe that it is not necessarily larger than A.

We thus have two approximation algorithms, one for overapproximating (figure 2.1) and one for underapproximating (figure 2.2), each obtained by performing the fixpoint computation with the appropriate approximating operator. Both take as input a safety verification problem, and return the boolean variable verified_correct. The function disjoint() returns the boolean value for whether its operands are disjoint or not.

2.1.1 Correctness

When computing the fixpoint using the overapproximating operator $\sqcup$, it is clear all the truly reachable states of the system are contained in the approximating set F. It

```
Fundamental_Overapprox(S,S0,N,V)
Last_Over := S0;
Over := S0;
converged := FALSE;
while (not converged) do
   Next_states := N(Last_Over);
   Last_Over := Over;
   Over := Over ⊔ Next_states;
   converged := (Last_Over = Over);
endwhile
verified_correct := disjoint(Over,V);
```

Figure 2.1: Fundamental overapproximation

```
Fundamental_Underapprox(S,S0,N,V)
Last_Under := S0;
Under := S0;
converged := FALSE;
while (not converged) do
   Next_states := N(Last_Under);
   Last_Under := Under;
   Under := Under ⊳ Next_states;
   converged := (Last_Under = Under);
endwhile
verified_correct := disjoint(Under,V);
```

Figure 2.2: Fundamental underapproximation

is also easy to see that using the underapproximating operator $\triangleright$ gives a set which is contained in the set of truly reachable states.

Proposition 2.1 *Given a verification problem (S, S_0, N, V), if the fundamental overapproximation algorithm terminates, then*

- *the resulting overapproximation Over contains reach(S, S_0, N).*

- *if the output verified_correct has value true, then the system is correct.* □

Proposition 2.2 *Given a verification problem* (S, S_0, N, V)*, if the fundamental underapproximation algorithm terminates, then*

- *the resulting underapproximation Under is contained in reach*(S, S_0, N).
- *if the output verified_correct has value false, then the system is not correct.* □

2.1.2 Advantages

The computational benefit of using approximation depends critically on the approximating operators and the sets they act on. Advantages accrue when the approximating operations are much faster than exact union, and there are fewer iterations overall. One way to exploit this feature is to introduce the notion of *approximating sets*, a subdomain of the power set of states over which the symbolic next-state relation, intersection, and the approximating operators are closed. In many cases, approximating sets can be chosen to ensure that applying the approximating operators is computationally inexpensive. Using approximating sets with compact representations can lead to great reductions in the space required to perform the fixpoint computation.

A further advantage may arise when the domain of approximating sets has only small chains of increasing sets. In this case, computing the fixpoint iterations may converge faster, and in any event, there is a smaller upper bound on the number of iterations required.

2.1.3 Disadvantages

There is a price paid for only approximating as opposed to using exact computation. The approximation may not correctly determine whether the system meets its specification. Furthermore it is possible that computing the approximation is more work than finding the set of exactly reachable states.

Before explaining the potential disadvantages of approximation, we first define some terms. A *false negative* is said to occur when a method reports the system is

	Overapproximation	Underapproximation
Correctness	Possible false negatives	Possible false positives
Computation	May search many unreachable states	May iterate more times than exact computation

Figure 2.3: Potential disadvantages of approximation

not correct, when in fact it is. A *false positive* occurs when a method reports the system is correct, when in fact it is not.

When the overapproximation incorrectly includes violating states which are not truly reachable, a false negative may arise. In addition, a single overapproximation does not provide enough information to confirm any true negative, *i.e.* to say for sure that a system really does violate its specification. From a computational point of view, the overapproximation may waste effort searching through parts of the state-space which are not really reachable.

When an underapproximation fails to include any of the violating states which are truly reachable, a false positive may arise. Analogous to the case of overapproximation, there is no means of confirming any positive results when the system is correct. A potential computational disadvantage is that finding the underapproximating fixpoint may involve more iterations that exact computation, since not necessarily all successor states are added at each iterative step. These disadvantages are summarized in figure 2.3.

Individually computing both an underapproximation and an overapproximation solves the problem of the lack of confirmed negatives in overapproximating and confirmed positives in underapproximating, but it may still yield inconclusive answers when the two approximations report different results.

2.1.4 Example

As a simple example let us overapproximate the reachable states of the following system.

Example 2.3 *[Basic overapproximation] Process P's state-space consists of all pairs of integer values for the variables x and y. Initially, x=y=0, and there is one violating state: x=6,y=0. The next-state relation is determined from the program:*

```
while (x<5 & y<5) do
   <x,y> := <x+1,y+1>;
while TRUE do
   <x,y> := <x,y>;
```

The set of truly reachable states is $\{\langle i, i\rangle \mid i \in [0,5]\}$, and so the system is correct. Following Cousot's interval analysis [Cou78], we choose as approximating sets the set of rectangles, i.e. *sets of points of the form:*

$$\{(x,y) : l_x \leq x \leq u_x \wedge l_y \leq y \leq u_y\}$$

where all bounds are integers or infinity, denoted ∞. Such a set will be denoted $[l_x, u_x] \times [l_y, u_y]$. The join operator acting on A and B returns the smallest rectangle containing both A and B. Computing the fixpoint iterations gives

$$\begin{array}{rcll} F_i & = & [0,i] \times [0,i] & \textit{for } i = 0..5 \\ F & = & [0,5] \times [0,5] & \end{array}$$

The overapproximation F does not include the violating state so the system is verified. Suppose, however, the violating state were $(2,0)$. By the second iteration the overapproximation would include $(2,0)$ and a false negative would be reported. □

2.2 Simple variations

Before explaining the full iterated algorithm which determines exactly whether the system is correct, we first introduce some basic variations on the simple approximation scheme in the last section. These variations will be combined in the full algorithm appearing in the next section.

2.2.1 Backwards reachability

While the previously described approximations proceeded forward through the state-space, it is also possible to approximate while performing *backwards* traversals. The system is correct iff the initial states are not backwards reachable from the violating states. Thus backwards graph traversal gives rise to the following verification scheme:

$$\begin{aligned} B_0 &= V \\ B_{i+1} &= B_i \cup N^{-1}(B_i) \\ B &= \lim_i B_i \end{aligned} \tag{2.4}$$

The system is correct iff $B \cap S_0 \neq \emptyset$. Naturally, we can replace the exact union operator in equation 2.4 with the $\sqcup$ and $\trianglerighteq$ operators to approximate the backwards reachable states.

In the remainder of this thesis we assume backwards approximations refer to approximations of the states backwards reachable from the violating states.

2.2.2 Iterated overapproximations

Information from a forward overapproximation can be used to refine the computation of a backwards approximation, and vice versa, leading to a scheme of iteratively refined overapproximations. Observe that every system state lying on a violating execution trace satisfies two properties: it is both *forward reachable* from the initial states and *backwards reachable* from the violating states. In analyzing the reachable state-space, we need only consider states which potentially fulfill both these properties. Thus, in a forward traversal, we may disregard states which are not backwards reachable from the violating states.

Figure 2.4 outlines an iterative scheme of alternately computing forward and backwards overapproximations, where the last computed overapproximation in the opposite direction is used to narrow the scope of the states considered during the current approximation. Overapproximations are repeatedly computed until either the system is verified correct, or the forward and backwards approximations are the same, in which case a (potentially false) negative is reported. The function Opposite_Dir maps FORWARDS to BACKWARDS, and vice versa. Given a set of states *OverRev* representing a superset of the reachable states in the reverse direction, the function Approx_Within limits the next overapproximation so it never goes outside *OverRev*. Thus for a set of initial states *Start*, and a known backwards overapproximation *B_Over*, the function call Approx_Within(*B_Over*,*Start*,N,FORWARDS) returns an overapproximation obtained by computing the following limit:

$$\begin{aligned} F_0 &= \mathit{Start} \cap \mathit{B\_Over} \\ F_{i+1} &= (F_i \sqcup N(F_i)) \cap \mathit{B\_Over} \\ F &= \lim_i F_i \end{aligned}$$

The function Approx_Within works similarly when computing a backwards overapproximation relative to the previous forwards overapproximation. The current (resp. last) overapproximations are stored in the array Over (resp. Last_Over), and the arrays Start and End indicate the sets of starting and ending states for violating traces viewed in the indexed direction.

2.2.3 Separating classes

We now describe a mechanism which enables more accurate approximations. The false negative of example 2.3 could be explained as due to poor approximation: the approximation was too "loose". A good goal would be to use more accurate approximations. One way to do this would be to have the join operator result in the smallest enclosing *convex polyhedron* rather than a rectangle, *i.e.* improve the accuracy of the approximation by using more expressive approximating sets. However, we are really interested in a methodology which will allow approximations to become *successively* more accurate as necessary. The method we use is based on the simple idea of localizing the approximations: we use state-space partitioning to limit the application of the approximating operators when it is suspected that joining states will result in too crude an approximation. The mechanism divides the state-space into different *separating classes*. An approximation is then a *set* of sets, each of which lies entirely

```
Iterated_Overapprox

Start[FORWARDS] := End[BACKWARDS] : = S0;
Start[BACKWARDS] := End[FORWARDS] := V;
Last_Over[BACKWARDS]  := S;
Last_Over[FORWARDS]  := ∅;
dirn  := FORWARDS;
verified_correct  := FALSE;
iterations_done  := FALSE;
while (not verified_correct and not iterations_done) do
  Over[dirn]  :=
       Approx_Within(Last_Over[Opposite_Dir(dirn)],Start[dirn],N,dirn);
  iterations_done  := (Over[dirn] = Last_Over[dirn]);
  verified_correct  := Disjoint(Over[dirn],End[dirn]);
  Last_Over[dirn]  := Over[dirn];
  dirn  := Opposite_Dir(dirn);
endwhile
```

Figure 2.4: Iterated overapproximations

within a separating class. The approximation operators are only applied within any given class. In computing an approximation, we apply the next-state relation to each set of states in the current approximation, and intersect the result with each separating class. The approximating operators are then applied only across sets from within the same separating class. In effect, the approximation is always localized within any given separating class[1].

Approximating structures

We delay until the next section a detailed explanation of how to find a good set of separating classes based on avoiding joins which might lead to false negatives and false positives. For now, we concentrate on how separating classes enable more accurate approximations. Formally, we define a *separating structure* $\mathcal{C}$ for a set of states D

[1] The basic approximation algorithm may be viewed as having all states lie in one large separating class, S.

to be a tuple of distinct (but not necessarily disjoint) sets $\langle C_i \rangle_{i \in I}$ which cover D, *i.e.* $\bigcup_{i \in I} C_i = D$. We refer to the components of a tuple as its elements. The elements of a separating structure are called *separating classes.* An *approximating structure* $\mathcal{A}$ with respect to $\mathcal{C}$ is a tuple of sets of sets $\langle \{A_{ij}\}_{j \in J_i} \rangle_{i \in I}$ where each $A_{ij} \subseteq C_i$. Reference to $\mathcal{C}$ will be omitted when the meaning is clear. We say that the set A is in (or appears in) $\mathcal{A}$ iff $A = A_{ij}$ for some i and j. The i-th component of $\mathcal{A}$ is the set of sets $\{A_{ij}\}_{j \in J_i}$ and can be thought of as a set of approximating sets for the reachable states lying within C_i. We say the *base elements* of an approximating structure are those states found in any of the individual sets of the structure, *i.e.* s is a base element of $\mathcal{A}$ iff $s \in \bigcup_{i \in I, j \in J_i} A_{ij}$. For any approximating structure $\mathcal{A}$, let $\cup\mathcal{A}$ denote its base elements. A state *appears* in an approximating structure $\mathcal{A}$ iff it is one of its base elements.

Operations on approximating structures

Instead of using a single approximating set as an estimate for the set of reachable states, we now use approximating structures respecting $\mathcal{C}$. Applying the next-state relation N to an approximating structure $\mathcal{A} = \langle \{A_{ij}\}_{j \in J_i} \rangle_{i \in I}$ yields the structure $N_{\mathcal{C}}(\mathcal{A})$ whose i-th component is the set of sets

$$\{N(A_{i'j}) \cap C_i \mid i' \in I \text{ and } j \in J_{i'}\}$$

The join operator is defined relative to a separating structure $\mathcal{C} = \langle C_i \rangle_{i \in I}$. Its operands are approximating structures respecting $\mathcal{C}$. Intuitively the join is done independently in each component, where each approximating set is the result of joining sets in its operands. A set of sets $\{D_l\}_{l \in L}$ is said to be a *join-combination* of a set of sets $\{A_j\}_{j \in J}$ iff

- for each $l \in L$, $D_l = \bigsqcup_{i=1..m} A_{j_i}$ where each index j_i is in J, and,
- for each $j \in J$ there exists an $l \in L$ such that $A_j \subseteq D_l$.

The set of sets $\{D_l\}_{l \in L}$ is said to be a join-combination of two sets of sets $\{A_j\}_{j \in J}$

and $\{B_k\}_{k\in K}$ iff it is a join-combination of their union $\{A_j\}_{j\in J} \cup \{B_k\}_{k\in K}$. An approximating structure is a join of two approximating structures $\mathcal{A} = \langle\{A_{ij}\}_{j\in J_i}\rangle_{i\in I}$ and $\mathcal{B} = \langle\{B_{ik}\}_{k\in K_i}\rangle_{i\in I}$ iff its i-th component is a join-combination of $\mathcal{A}_i$ and $\mathcal{B}_i$. For simplicity, we may write $\mathcal{A} \sqcup \mathcal{B}$ to refer to any join of $\mathcal{A}$ and $\mathcal{B}$, rather than introducing notation for relations over triples $\langle\mathcal{A}, \mathcal{B}, \mathcal{C}\rangle$ of approximating structures to indicate that $\mathcal{C}$ is a join of $\mathcal{A}$ and $\mathcal{B}$.

We can also obtain underapproximations in a similar way. We first extend the $\trianglerighteq$ operator to sets of states and then to approximating structures. Intuitively, expanding an approximating structure $\mathcal{A}$ with $\mathcal{B}$ is the result of expanding each component $\mathcal{A}_i$ with $\mathcal{B}_i$. The expansion over components consists of taking sets in $\mathcal{B}_i$ and adding them via the $\trianglerighteq$ operator to the sets in $\mathcal{A}_i$. The set of sets $\{D_l\}_{l\in L}$ is said to be an *expansion* of the set of sets $\{A_j\}_{j\in J}$ with $\{B_k\}_{k\in K}$ iff it is the result of taking each set A_j and expanding it with some number of sets $B_{j,1}, B_{j,2}, \ldots, B_{j,K_j}$ in such a way that each set B_k is added to some A_j, *i.e.* for every $k \in K$ there are j and l such that $B_k = B_{j,l}$. Formally, for every $l \in L$ there is an index $j_l \in J$ selecting a set A_{j_l}, and a sequence of indices $k_{l,1}, k_{l,2}, \ldots, k_{l,m_l} \in K$ selecting some sets in $\{B_j\}_{j\in J}$ to be added to A_{j_l} such that

- every D_l results from expansions to A_{j_l} by the sets $B_{k_{l,1}}, \ldots, B_{k_{l,m_l}}$, *i.e.* for every l, $D_l = (\cdots((A_{j_l} \trianglerighteq B_{k_{l,1}}) \trianglerighteq B_{k_{l,2}}) \cdots \trianglerighteq B_{k_{l,m_l}})$, and,
- every set A_j is preserved, *i.e.* for every A_j there is a set D_l such that $A_j \subseteq D_l$, and,
- every set B_k is added, *i.e.* for every $k \in K$, there is some index $k_{l,m}$ equal to k.

An approximating structure is an expansion of the approximating structure $\mathcal{A}$ with $\mathcal{B}$ iff its i-th component is an expansion of $\mathcal{A}_i$ with $\mathcal{B}_i$. Again we avoid unwieldy notation involving relations and informally write $\mathcal{A} \trianglerighteq \mathcal{B}$ to indicate some expansion of $\mathcal{A}$ with $\mathcal{B}$.

Finally we define the separation of a set with respect to a separating structure. Given a set of states A and an approximating structure $\mathcal{C}$, $A \downarrow \mathcal{C}$ is the approximating structure whose i-th component is $A \cap C_i$. The algorithms for computing overapproximations and underapproximations using separating structures appear in figures 2.5

```
Separating_Classes_Overapproximation((S,S_0,N,V),C)
Last_Over := S_0 ↓ C;
Over := Last_Over;
converged := FALSE;
while (not converged) do
   Next_states := N_C(Last_Over);
   Last_Over := Over;
   /* the join operator returns a legal join */
   Over := Over ⊔ Next_states;
   converged := (Last_Over = Over);
endwhile
verified_correct := disjoint(∪Over,V);
```

Figure 2.5: Separating classes overapproximation

and 2.6.

Proposition 2.4 *If the overapproximating algorithm using separating classes (figure 2.5) terminates, then*

- $\cup Over \supseteq reach(\mathcal{S})$.
- *if the boolean output verified_correct has value true, then the system is correct.* □

Proposition 2.5 *If the underapproximating algorithm using separating classes (figure 2.6) terminates, then*

- $\cup Under \subseteq reach(\mathcal{S})$.
- *if the boolean output verified_correct has value false, then the system is not correct.* □

Example 2.6 *[Separating Classes: Overapproximation] Consider again the system in example 2.3, with violating state* $(2, 0)$. *In an effort to show that the reachable states do not include* $(2, 0)$, *we use approximating sets to partition the state space so that the*

```
Separating_Classes_Underapproximation((S,S0,N,V),C)

Last_Under := S0 ↓ C;
Under := Last_Under;
converged := FALSE;
while (not converged) do
   Next_states := N_C(Last_Under);
   Last_Under := Under;
   /* the addition operator returns a legal expansion */
   Under := Under ⊳ Next_states;
   converged := (Last_Under = Under);
endwhile
verified_correct := disjoint(∪Under,V);
```

Figure 2.6: Separating classes underapproximation

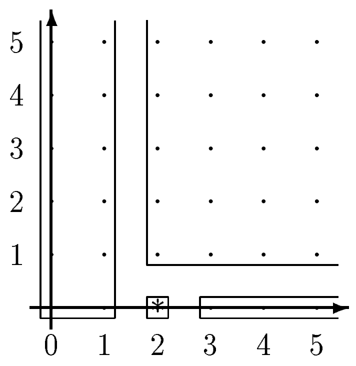


Figure 2.7: Separating classes example

violating state is separate from the rest. Thus we may choose as a separating structure $\mathcal{C} = \langle C_1 = [0,1] \times [0,\infty], C_2 = [2,\infty] \times [1,\infty], C_3 = [2,2] \times [0,0], C_4 = [3,\infty] \times [0,0] \rangle$*. We adopt a simple policy for choosing a join of a set of sets:* $\{A\} \sqcup \{B\} = \{A \sqcup B\}$[2]*.*

Then we iterate from $\mathcal{A}_0 = \langle \{[0,0] \times [0,0]\}, \{\}, \{\}, \{\} \rangle$*, giving first* $\mathcal{A}_1 = \langle \{[0,1] \times$

[2] This policy ensures every component of every approximating structure generated is either a single set or the empty set.

$[0,1]\}, \{\}, \{\}, \{\}\rangle$. *To compute $\mathcal{A}_2$ we first find*

$$\begin{aligned} T &= N_{\mathcal{C}}(\mathcal{A}_1) \\ &= N(\mathcal{A}_1) \downarrow \mathcal{C} \\ &= ([1,2] \times [1,2]) \downarrow \mathcal{C} \\ &= \langle [1,1] \times [1,2], [2,2] \times [1,2], \{\}, \{\} \rangle \end{aligned}$$

giving

$$\begin{aligned} \mathcal{A}_2 &= T \sqcup \mathcal{A}_1 \\ &= \langle \{([1,1] \times [1,2]) \sqcup ([0,1] \times [0,1])\}, \{([2,2] \times [1,2])\}, \{\}, \{\} \rangle \\ &= \langle \{[0,1] \times [0,2]\}, \{[2,2] \times [1,2]\}, \{\}, \{\} \rangle \\ \mathcal{A}_3 &= \langle \{[0,1] \times [0,3]\}, \{[2,3] \times [1,3]\}, \{\}, \{\} \rangle \\ \mathcal{A}_4 &= \langle \{[0,1] \times [0,4]\}, \{[2,4] \times [1,4]\}, \{\}, \{\} \rangle \\ \mathcal{A}_5 &= \langle \{[0,1] \times [0,5]\}, \{[2,5] \times [1,5]\}, \{\}, \{\} \rangle \\ \mathcal{A} = \lim_i \mathcal{A}_i &= \mathcal{A}_5 \end{aligned}$$

The base elements of the approximating structure $\mathcal{A}$ do not include the violating state, and the system is correctly verified. $\square$

Note that the iterated approximation method mentioned in subsection 2.2.2 is a special application of using the result Over[dirn] of each previous forward (or backwards) pass in a separating structure $\langle$Over[dirn]$\rangle$ for the next pass in the opposite direction.

So far the discussion has been about using separating classes for forwards approximations, but the algorithm applies perfectly well to backwards approximation as well.

2.3 Full approximation algorithm

The full approximation algorithm iterates with increasingly accurate underapproximations and overapproximations, both in the forward and backwards directions.

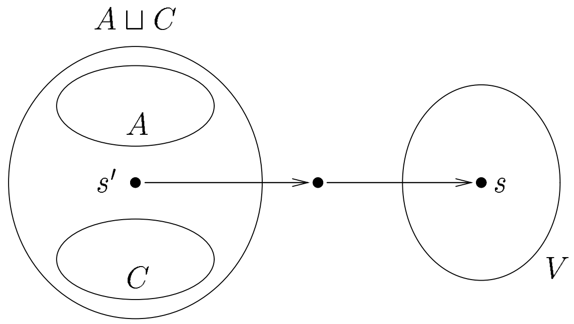


Figure 2.8: Conditions leading to false negatives

Approximations are computed with respect to successively finer separating structures which are dynamically generated by the algorithm. Whenever the algorithm terminates, it gives a true answer to the verification problem, *i.e.* there is no possibility of termination with false negatives or false positives. If the system's state-space is finite, the algorithm is guaranteed to terminate, and thus always determines whether the specification is satisfied or not.

Iterative convergence

Many iterative approximation schemes can be designed with this kind of progress property, namely that successive approximations are more accurate, and termination is guaranteed over finite state-spaces. For example, we need only ensure that the final iteration is the full exact computation gained from the separating structure where every state forms a class of its own. We can design iterative schemes where the approximation is performed with a fixed sequence of successively finer separating structures. For instance, an algorithm which uses a given partitioning of a (finite) state-space into at least 2^i disjoint separating classes at the i-th traversal will guarantee a confirmed answer to the verification problem after logarithmically many iterations. While this approach may be successful in some cases, it is generally difficult to choose in advance which separating structures should be used in order to achieve efficient verification. We propose instead an iterative approximation scheme which automatically discovers where approximations can be taken more freely, and where the analysis needs to be more exact. The user provides only an initial separating structure, and then the algorithm uses information from previous approximations to generate suitable refined separating structures. The scheme is therefore dynamically based, and adapts itself to the particular problem being solved, rather than being statically determined.

Conditional joining

The refinement procedure is based on some simple observations about how false negatives and false positives arise. It uses a notion of *conditional joining* to determine which parts of the state-space should be kept separate, and thus placed in different separating classes. The additional conditions we describe for joining sets are easily detectable and lead to increased accuracy of the approximations only in those parts of the state-space which are likely to lead to false positives or negatives. Suppose a false negative is obtained while performing a forward overapproximation of the reachable states. It must be a consequence of some join in the midst of computing the approximation. Figure 2.8 shows how false negatives occur: at some point a join caused an ancestor state s' of s to be included in the approximation although s' and s are not truly reachable[3]. If all such joins could be avoided, there would be no false negatives in the approximation. However, it is not easy to use this criterion to decide whether to join two sets or not, since we cannot predict whether a state s' is a predecessor of any violating states. There is a clear trade-off in the amount of effort spent in determining whether s' is a predecessor of a violating state and a possibly inaccurate approximation as a result of unwisely joining sets A and D.

Quick decision strategy

The strategy we propose is to use simple and fast checks on whether to join sets. Any mistakes which are made when sets are joined when they should not have been can be detected and corrected in a later approximation. The advantage of this approach is that sets are joined unless there is very strong reason not to, and so the approximating structures are kept small, and the computations of each approximation are fast. For

[3]It may be that $s = s'$.

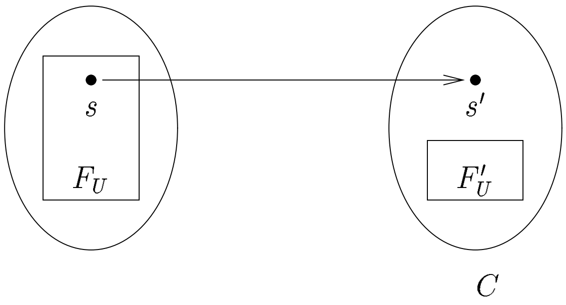


Figure 2.9: Conditions leading to false positives

false negatives then, we concentrate on cases where it is clearly foolish to join sets. If the state s' lies in the previous backwards underapproximation, then joining A and D in figure 2.8 will lead to a negative being introduced by this approximating step, since s' definitely has a path to a violating state. However, there is no particular reason to believe that s' is really reachable, since it is only included in the approximation because of a join operation and we have not constructed a path to it. Thus there is every chance that this negative will be a false negative. This discussion suggests avoiding all joins where the operands A and D contain no states in the previous reverse direction's underapproximation, but their join does.

There is a similar condition based on the occurrence of false positives. It is also simple to detect, and results in refining the approximations in areas of the state-space where false positives are likely to originate. Suppose we are computing a forwards underapproximation. Figure 2.9 shows how the propagation of the reachable states is stalled at s, and its successor s' is omitted from the underapproximation. Clearly s' is truly reachable. Let us first examine the conditions leading to s' not appearing in the underapproximation. Since its predecessor s is in the underapproximation, there is some stage of the underapproximation algorithm when all the successors of s, including s', are considered for inclusion in the underapproximation. If at this point, the underapproximation does not include any states in the same separating class as s', then some states would immediately be added to the underapproximation, by the nonemptiness for the underapproximating operator (*i.e.* $\emptyset \unrhd A \neq \emptyset$ for nonempty sets

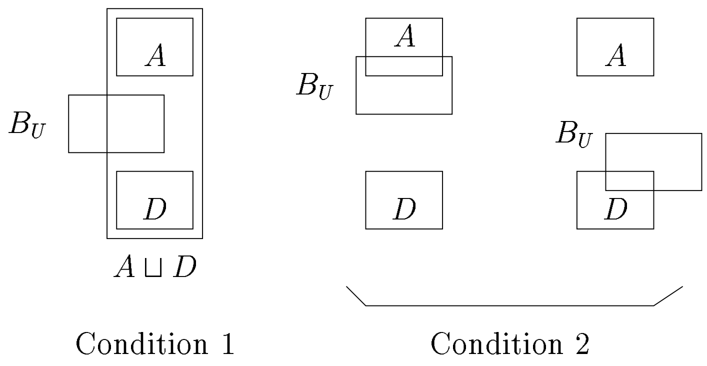


Figure 2.10: Violating conditions for permissible joins

A). Because we know s' is not in the underapproximation, it follows that the underapproximation must include some other states in s''s separating class C. One way to increase the likelihood s' appears in the next underapproximation is to use separating classes to separate it from all states in the underapproximation which lie in C. These separating classes are created dynamically by the next *backwards overapproximation* which will avoid joining sets within C if one of its operands contains states in F'_U while the other does not.

2.3.1 Conditional joins

The usual algorithm using separating classes would *always* join two sets A and D whenever they lie within the same separating class. Following the discussion above, we now provide more restrictive conditions under which such joins should be performed. The conditions given below apply when performing forward reachability. Symmetric conditions apply for backwards reachability and are not explicitly stated here. Let A and D be two sets lying within the same separating class. Let B_U be the set of states in the previous backwards underapproximation which are contained in that separating class. A join between A and D is said to be *permissible* unless either of the following two conditions hold:

1. both A and D are disjoint from B_U but $A \sqcup D$ is not, or

2. D is disjoint from B_U and A is not, or A is disjoint from B_U and D is not.

Condition 1 corresponds to a situation leading to a false negative, and Condition 2 to possible false positives.

We say that when sets are joined in a manner that respects the above conditions, the result is a B_U*-consistent* join, which we now formally define. The auxiliary function overlap() returns whether its two parameters have non-empty intersection, *i.e.*

$$overlap(X, Y) = \begin{cases} TRUE & X \cap Y \neq \emptyset \\ FALSE & \text{otherwise} \end{cases}$$

Given a set X, we say that a set of sets $\{D_l\}_{l \in L}$ is an X*-consistent join* of a set of sets $\{A_j\}_{j \in J}$ iff

- for each $l \in L$, $D_l = \bigsqcup_{i=1..m} A_{j_i}$ where each $j_i \in J$ and for each $i = 1..m$, $overlap(A_{j_i}, X) = overlap(D_l, X)$.
- for each $j \in J$ there exists an $l \in L$ such that $A_j \subseteq D_l$.

Corollary 2.7 *If the separating classes overapproximation algorithm of figure 2.5 is run under the restriction that all joins are X-consistent for some set X, then*

- $\cup Over \supseteq reach(\mathcal{S})$.
- *if the boolean output verified_correct has value true, then the system is correct.*

Proof: Obvious from proposition 2.4, since all X-consistent joins are joins. □

2.3.2 Refinement of approximations

As explained informally above, the approximations are successively more accurate because they are computed using finer and finer separating structures. The separating structures are derived from the most recently computed overapproximation. Their refinement is the result of using only conditional joins. In other words, if an overapproximation contains the class C the next overapproximation may have created approximating sets C_1, C_2, $\ldots$, C_k, all within C through using only conditional joins.

The next approximation will use each of these sets C_i as separating classes instead of C. The result is a more accurate approximation, because some joins which would have taken place within the class C will no longer do so since their operands now lie in different classes.

Before an approximating structure can be used as a separating structure, it must first be *flattened*, since it is a tuple of sets of sets, rather than a tuple of sets. We define the Flatten() function over approximating structures $\mathcal{A}$ such that $\mathrm{Flatten}(\langle\{A_{ij}\}_{j\in J_i}\rangle_{i\in I}) = \langle A_k\rangle_{k\in K}$ where $A_{ij} = A_{\Sigma_{l<i}|J_l|+j}$, *i.e.* every approximating set in $\mathcal{A}$ is a component of $\mathrm{Flatten}(\mathcal{A})$.

2.3.3 Sketch of algorithm

The full algorithm is sketched below. Forward overapproximations and underapproximations, and backwards overapproximations and underapproximations, are alternately computed. Each time an approximation is computed, information from the latest available approximations in the opposite direction is used. The opposite direction's overapproximation gives an upper bound on the states which need to be considered, see section 2.2.2. In addition this overapproximation also serves as a separating structure for the current overapproximation. The opposite direction's underapproximation is used to determine which joins are permissible, see section 2.3.1. Overapproximations are computed as described above, with only permissible joins. Thus an overapproximation may have several unjoined sets for each separating class of the separating structure it respects. This resulting overapproximation is used as a separating structure for the next pair of approximations. Thus the approximations are computed relative to finer and finer separating classes, resulting in successively more accurate approximations.

The forward and backward overapproximations and underapproximations are successively computed until the system is deemed correct, or a true violation is detected. Notice that in general the full algorithm need not terminate: it may generate infinitely many approximations without ever solving the verification problem. However, if the state-space is finite, or can be partitioned into finitely many equivalence classes, the algorithm is guaranteed to terminate (see Theorem 2.14). The skeleton

```
Full_Approx
Over[BACKWARDS] := original separating structure;
Under[BACKWARDS] := empty approximating structure;
confirmed_positive := FALSE;
confirmed_negative := FALSE;
dirn := FORWARDS;
Sep_Structure := original separating structure;
while ( (not confirmed_positive) and (not confirmed_negative) ) do
   Over[dirn] :=
      Over_Approx(dirn,N,Sep_Structure,Under[Opposite_Dirn(dirn)]);
   Sep_Structure := Flatten(Over[dirn]);
   Under[dirn] := Under_Approx(dirn,N,Sep_Structure);
   dirn := Opposite_Dirn(dirn);
endwhile
```

Figure 2.11: Full approximating algorithm

of the full algorithm appears in figure 2.11. The arrays Over and Under are global variables storing the current approximations in each direction, and confirmed_positive and confirmed_negative are global booleans. The algorithm starts by computing approximations in the forward direction[4]. Initially nothing is known about which states are backwards reachable, so we assume the user supplies an initial overapproximating structure whose base elements are all of S. We take the empty approximating structure as a conservative underapproximation of the backwards reachable states[5]. The functions Over_Approx() and Under_Approx() return approximations in the appropriate direction.

Pseudocode for the overapproximation algorithm appears in figure 2.12. The parameter *Opp_U* is an underapproximating structure in the opposite direction. The parameter *Sep* is the result of flattening an overapproximating structure into its corresponding separating structure. When called with parameters FORWARDS, N, A and $\mathcal{C}$, the function Successors() returns the set of successors of A via the next-state relation N, separated with respect to the structure $\mathcal{C}$, and the function call

[4]The algorithm could just as well start going backwards from the violating states instead.

[5]In fact, we could use any approximating structure whose base elements are a subset of V.

```
Over_Approx(dirn, Nsr, Sep, Opp_U)
Last_Over[dirn] := ∪ Start[dirn] ↓ Sep;
Over[dirn] := Last_Over[dirn];
converged := FALSE;
while (not converged) do
   Next_states := Successors(dirn,Nsr,Last_Over,Sep);
   Last_Over[dirn] := Over[dirn];
   Over[dirn] := consistent_join(Opp_U, Over[dirn], Next_states);
   converged := (∪ Last_Over[dirn] = ∪ Over[dirn]);
endwhile
verified_correct := disjoint(∪ Over[dirn], End[dirn]);
confirmed_positive := verified_correct;
```

Figure 2.12: Overapproximating algorithm

Successors(BACKWARDS,$N, A, \mathcal{C}$) returns its set of predecessors separated with respect to $\mathcal{C}$. The function consistent_join(), called with parameters X, $\mathcal{A}$ and $\mathcal{B}$ returns an X-consistent join of $\mathcal{A}$ and $\mathcal{B}$. The algorithm for underapproximations is similar, except that there is no need to check for consistency when applying the approximating operator.

Correctness

Theorem 2.8 *The following are true for forward and backwards traversals:*

1. *The states appearing in any underapproximating structure are a subset of the truly reachable states.*

2. *The states appearing in any overapproximating structure returned by the routine Over_Approx are a superset of the truly reachable states that lie on violating paths.* □

Termination

Let $FO_i(BO_i)$ and $FU_i(BU_i)$ be the i-th forward (backward) overapproximations and underapproximations in a sequence of approximations generated by the algorithm.

```
Under_Approx(dirn, Nsr, Sep)

Last_Under[dirn] := ∪Under[dirn] ↓ Sep;
Under[dirn] := Last_Under[dirn];
converged := FALSE;
while (not converged) do
   Next_states := Successors(dirn,Nsr,Last_Under[dirn],Sep);
   Last_Under[dirn] := Under[dirn];
   /* the addition operator returns a legal expansion */
   Under[dirn] := Under[dirn] ⊳ Next_states;
   converged := (∪Last_Under[dirn] = ∪Under[dirn]);
endwhile
verified_correct := disjoint( ∪Under[dirn], End[dirn]);
confirmed_negative := not verified_correct;
```

Figure 2.13: Underapproximating algorithm

We refer to the computation of FO_i and FU_i as the i-th forwards traversal of the algorithm. We first note that when S is finite, each individual traversal will complete.

Proposition 2.9 *If S is finite, then the individual calls to Over_Approx and Under_Approx terminate.*

Proof: The **while** loop of each algorithm is only repeated when additional base elements are added to the currently computed approximation. Therefore the loop terminates since the state-space is finite. □

The argument for termination of the full algorithm consists of showing that there is well-founded ordering over the approximations generated by the algorithm, such that they are non-increasing and decreasing infinitely often.

We define a partial order over approximating structures, where

$$\mathcal{A} \prec_{base} \mathcal{B} \text{ if and only if } \cup\mathcal{A} \subset \cup\mathcal{B}$$

In addition, we say

$$\mathcal{A} \preceq_{base} \mathcal{B} \text{ if and only if } \cup\mathcal{A} \subseteq \cup\mathcal{B}$$

We also denote $\mathcal{A} \prec_{base} \mathcal{B}$ by $\mathcal{B} \succ_{base} \mathcal{A}$, and write $\mathcal{B} \succeq_{base} \mathcal{A}$ for $\mathcal{A} \preceq_{base} \mathcal{B}$. The orders are well-founded.

Proposition 2.10 *If S is finite, then there are no infinite strictly $\prec_{base}$-descending or $\prec_{base}$-ascending chains of approximating structures.* □

We now show the overapproximations are non-increasing with respect to $\preceq_{base}$.

Proposition 2.11 $BO_{i+1} \preceq_{base} FO_{i+1} \preceq_{base} BO_i \preceq_{base} FO_i$

Proof: Every base element of an approximation is also a base element of the separating structure it respects. The separating structures are obtained by flattening the previous overapproximations and flattening preserves the base elements of a structure. □

Finally we establish that if the algorithm does not terminate, then the forwards overapproximations must decrease infinitely often with respect to $\prec_{base}$. In the next proposition we first show non-termination implies that after every two traversals either the overapproximations strictly decrease, or the underapproximations strictly increase. Then the proof of proposition 2.13 shows the overapproximations must decrease infinitely often, since the underapproximations cannot increase infinitely often in a finite state-space. This contradicts the well-foundedness of $\prec_{base}$.

We first introduce some notation. Given a set of states $Y \subseteq S$, we say that a set X of states is Y*-avoiding* iff $X \cap Y \neq \emptyset$. It is Y*-touching* iff it is not Y-avoiding.

Proposition 2.12 *If the algorithm has not terminated after computing FO_{i+2}, then either $FO_{i+2} \prec_{base} FO_i$, or $BU_i \prec_{base} BU_{i+1}$.*

Proof: The proposition essentially states that in every couple of traversals some progress is made in either the overapproximations or the underapproximations. Assume the algorithm has not terminated after computing FO_{i+2}. Then by proposition 2.11 if $BO_{i+1} \prec_{base} FO_i$, it follows that $FO_{i+2} \prec_{base} FO_i$, and progress has been made in the overapproximation as required. Thus we need only consider the case where $\cup BO_{i+1} = \cup FO_i$. First observe that $BU_i \preceq_{base} BU_{i+1}$ since $\cup BU_{i+1}$ contains $\cup BU_i \cap \cup BO_{i+1}$ which equals $\cup BU_i \cap \cup BO_i$ which equals $\cup BU_i$ since

$\cup BU_i \subseteq \cup BO_i$. Hence in order to show that $BU_i \prec_{base} BU_{i+1}$ we need only demonstrate that BU_{i+1} includes some state not in BU_i.

We establish three claims that complete the proof:

1. there is at least one $\cup BU_i$-avoiding approximating set A of FO_{i+1} whose set of successors is $\cup BU_i$-touching, *i.e.* $N(A) \cap \cup BU_i \neq \emptyset$.

2. there is at least one $\cup BU_i$-avoiding approximating set B of BO_{i+1} whose set of successors is $\cup BU_i$-touching.

3. some state $b \in B$ appears in BU_{i+1} but not BU_i, and hence $BU_i \prec_{base} BU_{i+1}$.

The first claim follows from the fact that only $\cup BU_i$-consistent joins are performed at any stage of the Over_Approx routine. Since the full algorithm has not terminated, we know that the initial states used in Over_Approx are disjoint from $\cup BU_i$, and hence all approximating sets in $S_0 \downarrow BO_i$ are $\cup BU_i$-avoiding. The final converged overapproximation FO_{i+1} is not $\cup BU_i$-avoiding, or else it is also V-avoiding, and hence verified correct. Thus at some stage of the overapproximating routine a $\cup BU_i$-touching set is including in the accumulated overapproximation. Since all joins are $\cup BU_i$-consistent, no $\cup BU_i$-avoiding approximating sets are ever replaced with $\cup BU_i$-touching sets. Hence there must be some $\cup BU_i$-touching set which is first added to the overapproximation, and it must be added as a result of computing the successors of a $\cup BU_i$-avoiding approximating set. Let this set be A_0. Thus A_0 has successor states in $\cup BU_i$. The overapproximating algorithm may join other sets to A_0, but only if the join is $\cup BU_i$-consistent, so there is always an approximating set that is $\cup BU_i$-avoiding and contains A_0. This argument establishes the first claim above. Let the approximating set thus found be called A.

The second claim states that BO_{i+1} also has such a set. We know that some state $a \in A$ has a successor $a' \in \cup BU_i$. We have already shown that $\cup BU_i \subseteq \cup BU_{i+1}$ and so $\cup BU_i \subseteq \cup BO_{i+1}$. In particular, $a' \in \cup BO_{i+1}$. When the overapproximation algorithm computes the predecessors of an approximating structure containing a', it obtains a structure $\mathcal{B}$ with at least one set B_0 containing a. Thus when $\mathcal{B}$ is joined to the current backwards overapproximation under construction, there is some approximating set containing a which is a subset of both B_0 and the class A, since approximating sets in FO_{i+1} are used as separating classes in computing BO_{i+1}. The converged backwards overapproximation also contains some set $B \subseteq A$ which contains a. Because the class A is $\cup BU_i$-avoiding, so is B, and the claim is established.

Finally, for the third claim, we need to show that some state $a' \in A$ is in BU_{i+1}. The state a has been chosen so that $N(a)$ includes elements of $\cup BU_i$. Let $B' = N(a) \cap \cup BU_i \neq \emptyset$. While computing the underapproximation BU_{i+1}, the routine Under_Approx at some stage considers all predecessors of some approximating set containing some $b \in B'$. These predecessors B'' would include the state $a \in A$. Since B is a separating class used in this computation, $B \cap B''$ is a set in the approximating structure for the predecessors being considered now. Suppose the underapproximation under construction already included some states in A. Then we are done since $\cup BU_i$ does not, and since the underapproximation algorithm never discards base elements, it follows that $BU_i \prec_{base} BU_{i+1}$. So suppose not. But in this case the algorithm would then include some set of states in B by the second axiom for underapproximating operators, namely that $\emptyset \trianglerighteq X \neq \emptyset$. It follows that $BU_i \prec_{base} BU_{i+1}$. □

Proposition 2.13 *Given a finite state-space, if the algorithm generates infinitely many forwards overapproximations, then infinitely many of them are strictly decreasing with respect to* $\prec_{base}$.

Proof: Suppose the forwards overapproximations are not infinitely often decreasing. Then by proposition 2.11 the base elements of the forwards overapproximation must converge to some set $\cup FO$. Suppose this occurs after k traversals. From this point on, the backwards underapproximations are non-decreasing, since $\cup BU_i \subseteq \cup FO$ and $\cup BU_i \cap \cup FO \subseteq \cup BU_{i+1}$ for $i > k$. Hence they cannot increase infinitely often since they are contained within a finite set. Thus by proposition 2.12 the forwards overapproximations are infinitely often decreasing. □

Theorem 2.14 *Given a finite state-space S, the full approximation algorithm of figure 2.11 terminates.*

Proof: The well-founded ordering $\preceq_{base}$ over the forwards overapproximations is non-increasing and strictly decreasing infinitely often, and so the algorithm must terminate. $\square$

Theorem 2.15 *Given a finite state-space S, the full approximation algorithm terminates with the correct answer to the verification problem.*

Proof: Immediate from theorems 2.8 and 2.14. $\square$

2.3.4 Additional splitting

The full algorithm can easily be modified to allow additional splitting of classes. This feature enables the program to use various heuristics to accelerate convergence, other those outlined above for conditional joins.

Additional splitting may be safely performed between traversals. In the algorithm given above, each successive traversal of the algorithm uses a separating structure derived from the previous overapproximation. However it is always possible to refine this separating structure without losing soundness, or completeness over finite-state systems. If the separating structure used instead of the previous overapproximation has the same base elements as the overapproximation, correctness is maintained. Furthermore, if it is also finer than it (wrt $\preceq_{sp}$ defined below) the property of termination is maintained.

We define a notion of splitting one approximating structure into another. Intuitively, $\mathcal{A}$ is the result of some splitting of $\mathcal{B}$ iff it is obtained by taking some sets in $\mathcal{B}$ and splitting them into nontrivial parts.

$$\mathcal{A} \preceq_{sp} \mathcal{B} \text{ if and only if } \begin{cases} \forall A \in \mathcal{A}, \exists B \in \mathcal{B} \text{ such that } A \subseteq B, \text{ and} \\ \forall B \in \mathcal{B}, B = \cup\{A \in \mathcal{A} \mid A \subseteq B\} \end{cases}$$

We let Split() be any function which, given input approximating structure $\mathcal{B}$, returns some Flatten($\mathcal{A}$) for which $\mathcal{A} \preceq_{sp} \mathcal{B}$.

Proposition 2.16 *Replacing the Flatten function with the Split function in the full* *approximation algorithm maintains the properties of termination over finite-state systems, and correctness.*

Proof: Correctness is obvious, since the base elements are maintained and thus still form an overapproximation.

An examination of the termination proof over finite-state systems reveals that termination depends on successive overapproximations containing sets which do not contain elements in the underapproximations, but which have successor states which do. See the proof of proposition 2.12. Suppose X is such a class as required by the proof of termination, *i.e.* X is a set in FO_i that is $\cup BU_i$-avoiding but its successors are not. Suppose then that $s \in X$ is not in $\cup BU_i$ but has a successor state which is. Splitting a class X into several classes $X_1, X_2, \ldots, X_k$ which partition X ensures that there will always be a class among the X_i which contains s and is $\cup BU_i$-avoiding. □

Alternative termination conditions

An alternative dynamic method for refining the separating structure used for each iteration is to separate states appearing in the underapproximation from those which do not. This technique may be seen as a special case of allowing additional splitting. The potential disadvantage of this approach is that classes may get fragmented quickly, and it requires use of the difference or negation operator. In particular, for the real-time systems we consider we do not have a space-time efficient means of computing the difference between approximating sets.

2.3.5 Generating debugging traces

An important, and often overlooked, factor in any algorithm for verification is the ability to generate useful debugging information when a system violation is detected. Here we briefly describe how the underapproximations can be used to generate debugging traces, and some of the limitations associated with them.

In its most general form, the algorithm as it stands does not guarantee violating paths will be obtained every time a violation is detected. However, the underapproximation algorithm can easily be used to generate a graph whose nodes are sets of states with an edge between nodes whenever there is an edge between elements of the two sets. If a violation is detected, the graph contains violating states. From this graph it is possible to generate a *pseudo-trace* $A_1, A_2, A_3, \ldots, A_k$ where all states in A_1 are initial, all states appearing in any A_i are reachable from the initial states, and between any two successive sets A_i and A_{i+1} in the sequence there is at least one edge from a state in A_i to a state in A_{i+1}. Notice however that there is no guarantee at all that there is even a path $a_1, a_2, \ldots, a_k$ in $\langle S, S_0, N\rangle$ such that $a_i \in A_i$. In many cases however, this kind of debugging information can be useful.

There are a number of ways to generate real violating paths. One could use exact analysis over that part of the state-space covered by the underapproximation until a violating state is reached.

Another method is to use a restricted form of underapproximating operator that enables real violating traces to be extracted. The idea is to build a graph whose nodes are sets of states with edges between sets whenever there is an edge to every element in the second set from some element of the first set. We say a graph with sets of states in S as nodes is a $\overleftarrow{\exists}\forall$-setgraph for $\langle S, S_0, N\rangle$ iff whenever $A \rightarrow B$, for every $b \in B$ there is some $a \in A$ such that $a \rightarrow b$. Every trace in a $\overleftarrow{\exists}\forall$-setgraph corresponds to a trace in the underlying transition system.

Proposition 2.17 *Given a $\overleftarrow{\exists}\forall$-setgraph G for the transition system $\langle S, S_0, N\rangle$, for every path $A_1, A_2, \ldots, A_k$ in G, there is a path $a_1, a_2, \ldots, a_k$ in $\langle S, S_0, N\rangle$ such that $a_i \in A_i$.* $\square$

Thus we need only guarantee that the underapproximation builds an $\overleftarrow{\exists}\forall$-setgraph. An easy way to achieve this is to restrict the underapproximating operator so that $A \mathrel{|\!\!\triangleright} B = A$ whenever A is non-empty. However this results in a very weak underapproximating operator. In order to compensate for the weak operator, we may restrict the expansion operator over sets so that the underapproximation advances sufficiently. We propose using the expansion operator which always returns a set of sets which is maximal, up to a certain limit on its size. Given a sets of sets $\{A_i\}$ and $\{B_j\}$, we say that any subset of $\{A_i\} \cup \{B_j\}$ which contains every A_i, has at most k members, and is maximal is an expansion of $\{A_i\}$ with $\{B_j\}$. The larger the value of k, the closer the approximation is to being exact, at a cost of time and space.

2.3.6 Further features

All the algorithms described in this section are flexible enough to allow the user to specify an initial separating structure. Hence the algorithms can be run approximating as aggressively (loosely), or as accurately (tightly), as desired. The user can also use her own understanding of the system to determine which parts of the state-space to analyze more accurately, and over which states rough approximations are adequate.

An additional advantage of this approximation scheme is that it utilizes both forwards and backwards reachability information. Empirical experience with finite-state verification has shown that in some instances performing reachability in one direction is easy while the other is prohibitively expensive. Rather than having to commit to an expensive exact forward or backwards analysis, or perform both simultaneously, the approximation algorithm can quickly compute an approximation in one direction, and then the other. Thus information from both traversals may be combined relatively quickly before the analysis becomes more exact.

2.4 Approximating next-state relations

We conclude this chapter by showing how approximations can be made over not only the accumulated set of reachable states, but also over the individual image computations. In the description above, the exact next-state relation is used to compute the successors of a set of states. However, it is not always easy to find the exact set of successor states for a given approximating set. Furthermore, the set of successors may not be a single approximating set, but rather a large number of approximating sets. We later explain in subsection 5.1.2 how this situation occurs for the real-time systems we verify, where we find it necessary to approximate next-state relations.

This section outlines how next-state relations can be approximated. It also provides sufficient conditions for the approximation algorithm to terminate over finite-state systems.

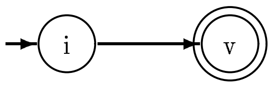
An underapproximation of the next-state relation N, usually denoted $\underline{N}$, is any relation such that $\underline{N} \subseteq N$. Similarly, a relation $\overline{N}$ is an overapproximation of N iff $N \subseteq \overline{N}$. These relations induce relations over sets of states in the natural way, *i.e.* $\underline{N}(A) = \{t \mid \exists\ s \in A \text{ such that } \underline{N}(s,t)\}$. Since we are mainly concerned here with relations over sets of states, we further define a *set-underapproximation* of the set-relation induced by N as any relation $\underline{N}^s$ over sets of states such that for every set $A \subseteq S$, $\underline{N}^s(A) \subseteq N(A)$. *Set-overapproximations* are similarly defined. Set-approximating next-state relations are usually referred to simply as approximations of N.

2.4.1 Correctness

The following propositions state that it is sound to replace N with an overapproximation in the overapproximation algorithms, and with an underapproximation in the underapproximating algorithms. As a point of clarification, the algorithms for backwards overapproximation do not use $(\overline{N})^{-1}$, but rather an overapproximation $\overline{N^{-1}}$ of the inverse relation N^{-1}. Similarly N^{-1} should be replaced by some $\underline{N^{-1}}$ in the underapproximating algorithms.

Proposition 2.18 *The overapproximating algorithms (for fundamental overapproximation (figure 2.1), for iterated approximations (figure 2.4), for separating classes (figure 2.5), and within the full approximation algorithm (figure 2.12)), when run with N replaced by an overapproximating relation $\overline{N}$ ($\overline{N^{-1}}$) in the forwards (backwards) direction yield converged overapproximations whose base elements are a superset of the states lying on violating paths.* □

Proposition 2.19 *The underapproximating algorithms (for fundamental underapproximation (figure 2.2), for separating classes (figure 2.6), and within the full approximation algorithm (figure 2.13)), when run in the forwards (backwards) direction*



$$\overline{N} = \overline{N^{-1}} = N$$
$$\underline{N} = \underline{N^{-1}} = \{\}$$

Figure 2.14: Non-termination example

with N replaced by an underapproximating relation $\underline{N}$ ($\underline{N^{-1}}$) yield converged underapproximations whose base elements are a subset of the states forwards (backwards) reachable from S_0 (V). □

Combining propositions 2.18 and 2.19 gives soundness for the full approximation algorithm.

Proposition 2.20 *If the full algorithm terminates when N is replaced by $\overline{N}$ ($\overline{N^{-1}}$) in the overapproximating routines, and by $\underline{N}$ ($\underline{N^{-1}}$) in the underapproximating routines, it gives a correct answer to the verification problem.* □

2.4.2 Non-termination

The following examples illustrate that even if N is approximated for just overapproximations (or underapproximations) and the approximation operators actually return the exact union, termination is not guaranteed even for finite-state systems.

Example 2.21 *Consider the verification problem (S, S_0, N, V) for the 2-state system with $S = \{i, v\}$, $S_0 = \{i\}$, $V = \{v\}$, and $N = \{(i, v)\}$, shown in figure 2.14. Let us use exact operators as our approximating operators,* i.e. *we assume that $\trianglerighteq$ and $\sqcup$ are exact over the sets we consider. Suppose we approximate N with the overapproximation $\overline{N} = \overline{N^{-1}} = N$, and the underapproximation $\underline{N} = \underline{N^{-1}} = \{\}$. The initial separating structure must separate i from v, and is thus taken to be $\langle\{i\}, \{v\}\rangle$. The first iteration of the forwards overapproximation yields the approximating structure $\langle\{\{i\}\}, \{\{v\}\}\rangle$. The forward underapproximation is $\langle\{\{i\}\}, \{\}\rangle$. The system contains an unconfirmed* *violation. The backwards iterations yield the same overapproximation, and the underapproximation $\langle\{\{\}\}, \{\{v\}\}\rangle$. Continued iterations of the algorithm result in no change, so the violation will never be detected.* □

Example 2.22 *Consider a system where S, S_0, and V are as above but with $N = \{\}$. Take $\underline{N}$ to be N, and the overapproximation $\overline{N}$ to be $\{(i, v)\}$. In this case we iterate with exactly the same approximations as before, and never discover that the system is correct.* □

Proposition 2.23 *The full algorithm, with N replaced by $\overline{N}^s$ in the overapproximating routines, and by $\underline{N}^s$ in the underapproximating routines, need not terminate even over finite-state systems.* □

2.4.3 Termination

We outline methods which guarantee the full approximation algorithm terminates over finite-state systems, even when the next-state relation is approximated.

Convergence to exact relations

The first strategy proposed is to use a sequence of approximations to N rather than a fixed approximation. Let $\overline{N}_1, \overline{N}_2, \ldots$ be overapproximations of N that are converging towards N, *i.e.* $\overline{N}_i \supseteq \overline{N}_{i+1} \supseteq N$. It is easy to that if $\overline{N}_i$ is used in place of N on the i-th traversal of the full approximation algorithm, then correctness is maintained. A sequence of next-state relations increasing towards N may also be used soundly for underapproximating N. If the approximate next-state relations converge to the exact relation N, then the algorithm terminates over finite-state systems.

We define a straightforward ordering on set-approximating next-state relations as follows:

$$N_1^s \preceq N_2^s \text{ iff } \forall A \subseteq S, N_1^s(A) \subseteq N_2^s(A)$$

Proposition 2.24 *Given sequences of decreasing overapproximating relations for N and sequences of increasing underapproximating relations for N both of which eventually converge to exactly N, the full approximation algorithm terminates correctly* *over finite state-spaces if the i-th approximating relations are used instead of N in computing the i-th approximations.*

Proof: Running the full approximation algorithm as described with overapproximating relations $\overline{N_i}$ and $\overline{N_i^{-1}}$ and underapproximating relations $\underline{N_i}$ and $\underline{N_i^{-1}}$ is sound, by repeated applications of propositions 2.18 and 2.19, and then proposition 2.20. The algorithm cannot run forever since after j traversals the approximating next-state relations converge to the exact relation, from which point the algorithm is guaranteed to terminate. In other words, the computation can be viewed as taking place in two distinct phases, each of which will terminate. Any computation using approximate next-state relations up to the j-th traversal may be regarded as a preliminary restriction of the state-space to states potentially lying on violating paths. Computation from the j-th traversal on may be regarded as running the full approximation algorithm with the exact next-state relation. $\square$

Exact application of approximate relations

The second strategy suggested is to use a set-approximating next-state relation which is exact when applied to a subclass of approximating sets. Rather than guaranteeing *a priori* that a sequence of approximating relations converges to the exact relation, we can use a fixed approximating relation, and instead ensure that it is eventually only ever applied to approximating sets over which it is exact. This strategy is the one we use for verifying real-time systems. Let $Dom_0 \subseteq Dom$ be a subset of the domain of approximating sets. A set next-state relation N' *exactly matches* a set next-state relation N over Dom_0 iff for all sets $A \in Dom_0$, $N'(A) = N(A)$.

Proposition 2.25 *If the full algorithm is run with set-underapproximating relations and set-overapproximating relations which are exact over the domain of all sets appearing in the initial separating structure and all subsets of those sets, then the algorithm terminates over finite state-spaces.*

Proof: It is sufficient to establish that the approximating relations are exact over all sets to which they are applied. First observe that at any stage of the full algorithm, every separating class is the subset of one of the classes in the original separating structure. All approximating sets lie within some separating class, and hence are subsets of some initial separating class. Thus, by the assumption in the statement of the proposition, the next-state relation is exact over all sets it operates on. □

Theorem 2.26 *Given a finite state-space and a well-founded ordering, if the full approximation algorithm is run with set-underapproximating relations $\underline{N}$ and $\underline{N^{-1}}$ and set-overapproximating relations $\overline{N}$ and $\overline{N^{-1}}$ such that the separating structures generated are non-increasing, and at each traversal, either the most recent separating structure $\mathcal{C}$ is strictly less than the previous one, or $\overline{N}$, $\overline{N^{-1}}$, $\underline{N}$ and $\underline{N^{-1}}$ exactly match N over the domain of separating sets appearing in $\mathcal{C}$, then the algorithm terminates.*

Proof: Assume the algorithm generates infinitely many approximations without terminating. If there is are infinitely many approximations which are strictly decreasing, then the algorithm must terminate. Suppose then that this is not the case, and that eventually all adjacent overapproximations have the same set of base elements. By assumption, $\underline{N}$, $\underline{N^{-1}}$, $\overline{N}$ and $\overline{N^{-1}}$ all exactly match N over the current separating classes, and then by proposition 2.25, the algorithm terminates. □

A natural candidate for the well-founded ordering is $\preceq_{base}$. However, it is often difficult to guarantee that the successive approximations are decreasing infinitely often with respect to this ordering. We introduce another ordering for which it is easy to modify the algorithm so that the approximations decrease as required.

Let

$$\mathcal{A} \prec_{set} \mathcal{B} \text{ if and only if } \begin{cases} \forall A \in \mathcal{A}\ \exists B \in \mathcal{B} \text{ such that } A \subseteq B, \text{ and} \\ \exists B \in \mathcal{B} \text{ such that } \nexists A \in \mathcal{A} \text{ with } B \subseteq A \end{cases}$$

Proposition 2.27 *Over a finite state-space, there are no infinite chains of approximating structures which are strictly $\prec_{set}$-descending or strictly $\prec_{set}$-ascending.*

Proof: Over a finite state-space, there are only finitely many approximating structures, so we need only show that $\prec_{set}$ admits no cycles. By definition, if $\mathcal{A} \prec_{set} \mathcal{B}$, then some set $B \in \mathcal{B}$ has no superset in $\mathcal{A}$. If $\mathcal{B} \prec_{set} \mathcal{C}$, then there is some set $C \in \mathcal{C}$

such that $B \subseteq C$. The set C cannot have any superset in $\mathcal{A}$ or else $\mathcal{A}$ would contain a superset of B. By induction it is impossible for there to be a cycle $\mathcal{A} \prec_{set} \mathcal{B} \prec_{set} \cdots \mathcal{A}$, since every set is a superset of itself. $\square$

Corollary 2.28 *Given a finite state-space, if the successive overapproximations are strictly decreasing with respect to* $\prec_{set}$ *up until the approximating relations are exactly matching, then the algorithm terminates.* $\square$

It is easy to see how to obtain from an overapproximation $\mathcal{A}$ a separating structure $\mathcal{C}$ which has the same base elements, but such that $\mathcal{A} \prec_{set} \mathcal{C}$. We need only take any non-zero number of approximating sets in $\mathcal{A}$ which are not contained in any other approximating set in $\mathcal{A}$, and let $\mathcal{C}$ be result of replacing each with nontrivial parts which partition it. Since the replaced approximating sets do not have supersets in $\mathcal{A}$, it follows that $\mathcal{A} \prec_{set} \mathcal{C}$.

Proposition 2.29 *The successive overapproximations of the full algorithm will be decreasing with respect to* $\prec_{set}$ *if the following alteration is made to the algorithm: whenever the algorithm generates overapproximations whose base elements are not a strict subset of the base elements in the separating structure used in its computation, use as the next separating structure one obtained by splitting as described above.* $\square$

These results suggest a policy for ensuring termination when using approximate next-state relations over finite-state systems. Classes in the separating structures can be split whenever "sufficient" progress is not made in successive overapproximations, up until the approximate relations are exactly matching.

Chapter 3

Real-Time Systems

3.1 Introduction

Computerized controllers are appearing more and more in embedded systems as the cost, size, development time and power requirements of computerized systems plummets. In these systems, the computer interacts with physical processes for which time is an important factor. Thus the design of these controllers must consider not only the sequencing and coordination of events, but also the times at which they occur. Any formal methodology for specifying and reasoning about such interactive systems must include an accurate model of timed behavior.

In this chapter we review a formalism for modeling real-time systems: timed automata, and show how they can specify timed safety problems.

3.2 Timed automata

3.2.1 Time-stamped traces

The domain of time is the set of *non-negative* reals, simply denoted $\mathbb{R}$. Given an alphabet Σ, a *timed-stamped trace* is a sequence of pairs in $\Sigma \times \mathbb{R}$

$$\langle \sigma_0, t_0 \rangle, \langle \sigma_1, t_1 \rangle, \langle \sigma_2, t_2 \rangle, \ldots$$

such that

- (weak monotonicity): $t_i \leq t_{i+1}$ for all $i \geq 0$

An infinite time-stamped trace is *divergent* iff

- (divergence): for all $k \in \mathbb{R}$, there exists an i such that $t_i > k$.

Note that timed-stamped traces may be finite or infinite, and that several events may occur in sequence with the same time-stamp.

3.2.2 Timed traces

A *timed trace* is an alternative view of a time-stamped trace. Rather than noting the time of every event, we instead model explicitly the passage of time (if any) between events. Let Δ_T be the set of time-passage events

$$\Delta_T = \{\delta_t \mid t \in \mathbb{R}\}$$

Given an alphabet Σ disjoint from Δ_T, a *timed trace* consists of a sequence of events taken from $\Sigma \cup \Delta_T$. Events from Σ take place instantaneously, while events from $\mathbb{R}$ represent the passage of time. It is easy to see that every time-stamped trace can be modeled as a timed trace, and vice versa.

An infinite timed trace is *divergent* iff its corresponding time-stamped trace is. To express this explicitly, we define a duration function over $\Sigma \cup \Delta_t$ as follows:

$$dur(e) = \begin{cases} t & \text{if } e = \delta_t \in \Delta_T \\ 0 & \text{if } e \in \Sigma \end{cases}$$

Then an infinite timed trace $\tilde{e}$ is divergent iff the sum of event durations is unbounded, *i.e.* for all $k \in \mathbb{R}$ there exists a j such that $\Sigma_{i=0..j} dur(e_i) > k$.

3.2.3 Timed safety automata

We recall the definition of *timed safety automata (TSAs)* as a means of specifying timed transition systems and their properties [HNSY92]. There are many variants of timed automata; the one we use most closely resembles those of [NSY92a, HNSY92]. Timed safety automata are a form of finite-state automata with finitely many real-valued clocks. Each clock records the exact amount of time which has elapsed since its last reset. Each transition has an enabling condition depending on the values of the clocks. Transitions occur instantaneously and may include the resetting of clocks.

Each enabling condition is expressed as a non-empty set of points in $\mathbb{R}^n$, where n is the number of clocks in the automaton. We assume the clocks have been ordered so that the values of all the clocks may be expressed as a vector of real values. The transition is enabled whenever the n-vector of clock values lies in its enabling set. Enabling conditions are restricted to be sets definable as a conjunction of constraints of the form $x \sim k$ where x is a clock and $\sim \in \{<, \leq, =, \geq, >\}$. For convenience, we may refer to an enabling condition as either a set of points or the logical formula defining it. The domain of all enabling conditions is called $\mathcal{E}n$. We define a set of *reset actions* $\mathcal{A}(n)$, which are functions from $\mathbb{R}^n$ to $\mathbb{R}^n$ corresponding to the resetting of some of the clocks to 0. For each $a \in \mathcal{A}(n)$, there is a set of indexes $I_a \subseteq \{1 \dots n\}$ such that

$$\forall \vec{x} \in \mathbb{R}^n, \forall i = 1, \dots, n, \quad a(\vec{x})_i = \begin{cases} 0 & \text{if } i \in I_a \\ (\vec{x})_i & \text{otherwise} \end{cases}$$

The enabling conditions on events express precisely that they are enabled to take place: they do not stipulate that the event must occur at all. However in many real-time systems, we need to model the fact that an event is *guaranteed* to occur within a certain time bound. This situation is modeled in a timed safety automaton by giving *safety invariants* for each location, thereby specifying upper bounds on how long time may progress. For example, if an event is guaranteed to occur at control location q at time $x = 5$, the invariant at q should require "$x \leq 5$". This condition expresses that time cannot pass beyond $x = 5$ without an event occurring.

Definition 3.1 *A* timed safety automaton (TSA) *G is a tuple $\langle \Sigma, Q, Q_{init}, C, T, Inv \rangle$ where*

- *Σ is a finite set of events, disjoint from Δ_T,*
- *Q is a finite set of control locations,*

- $Q_{init} \subseteq Q$ *is a set of initial locations,*
- $C = \{x_1, \ldots, x_n\}$ *is a finite set of clocks,*
- $T \subseteq Q \times \Sigma \times \mathcal{E}n \times \mathcal{A}(n) \times Q \times \{0, 1\}$ *is a* proper *transition relation, defined below,*
- $Inv \in (Q \rightarrow IZ)$ *is an invariant assignment mapping control locations to the domain* IZ *of safety invariant zones defined below.*

Transition relations

An edge $e = (q, \sigma, \phi, a, q', urg)$ in the timed automaton's transition relation corresponds to a transition from control location q labeled with event σ. It is enabled iff the values of the clock variables satisfy ϕ. The transition is instantaneous and the reset action a is applied to the clock values. The resulting control location is q'. The transition is said to be *urgent* iff $urg = 1$. An urgent transition must occur as soon as it is enabled, unless another instantaneous event occurs and disables it. In other words, no time may pass while an urgent event is enabled. There is an added restriction that all urgent events are never constrained by a timing condition with strict lower bounds. This restriction ensures that the time when an urgent event first becomes enabled because of time passing is well-defined. For example, if an urgent event has enabling condition $x > 3$, and the value of x is currently 2, then it would be impossible for time to pass incrementing x beyond 3, and yet having the urgent event occur as soon as $x > 3$ since there is no *first* value of x which is strictly greater than 3. Formally, then, we first define the vector $\vec{t}$ to be the n-vector with all components equal to t, *i.e.* $\vec{t} = \langle t, t, \ldots, t \rangle \in \mathbb{R}^n$.

A transition is *proper* iff it is non-urgent, or it is urgent and its enabling timer values form a set Z which is topologically closed in the downwards direction, *i.e.* for all points $\vec{x}$, if there exists an $\epsilon > 0$ such that $\vec{x} + \vec{\delta} \in Z$ for all $0 < \delta < \epsilon$, then $\vec{x} \in Z$. This set closure condition is equivalent to saying that the enabling condition can be defined without using any strict lower bound constraints on the absolute values of any clocks. A transition relation is proper iff all its transitions are proper.

Safety invariants

The domain IZ of *invariant zones* is defined to be the set of all *predecessor closed time zones.* A *time zone* Z is any convex polyhedron of $\mathbb{R}^n$, consisting of all solutions of a system of linear inequalities where each inequality is of one of the following forms:

- $x \le k$, $x < k$, $x \ge k$, $x > k$, where x is a clock and k is an integer constant
- $x - y \le k$, $x - y < k$, where x and y are clocks and k is an integer constant.

Let $\mathcal{Z}(n)$ be the set of zones of $\mathbb{R}^n$. The set of *time successors* of a zone Z is the set

$$Z_{\nearrow} = \{\, \vec{y} \ \mid \exists\, \vec{x} \in Z, t \in \mathbb{R}, \text{such that } \vec{y} = \vec{x} + \vec{t}\,\}$$

The set of *time predecessors* of a zone Z is the set

$$Z_{\swarrow} = \{\, \vec{y} \ \mid \exists\, \vec{x} \in Z, t \in \mathbb{R}, \text{such that } \vec{y} + \vec{t} = \vec{x}\,\}$$

Finally, a time zone Z is *predecessor closed* iff it includes all its time predecessors. An equivalent definition is that it can be defined without using any lower bound constraints on the absolute values of clocks.

Semantics

We are now ready to define operational semantics for a timed safety automata G in terms of a transition system $\langle S, S_0, N\rangle$. A *timed-state* of the system is a pair $s = \langle q, \vec{x}\,\rangle$, where $q \in Q$ is a control location and $\vec{x} \in \mathbb{R}^n$ a vector of clock values. The set S consists of all timed-states.

The set S_0 of initial states is the set of all timed-states whose control component is an initial location in G, and whose clocks values are all equal to 0, as given by

$$S_0 = \{\langle q, \vec{0}\,\rangle \mid q \in Q_{init}\}$$

For each transition $e = (q, \sigma, \phi, a, q', urg) \in T$, let

$$N_e \ = \ \{(\langle q, \vec{x}\,\rangle, \langle q', \vec{x}\,'\rangle) \mid \vec{x} \in \phi,\ \vec{x}\,' = a(\vec{x}\,),\ \text{and } \vec{x}\,' \in Inv(q')\}$$

For each $t \in \mathbb{R}$, we define

$$N_{\delta_t} = \{(\langle q,\, \vec{x}\,\rangle, \langle q,\, \vec{x} + \vec{t}\,\rangle) \mid \vec{x} + \vec{t} \in \mathit{Inv}(q), \text{ and } \forall\, 0 \leq t' < t, \forall e \in T,$$
$$e \text{ is urgent implies } N_e(\{\langle q,\, \vec{x} + \vec{t'}\rangle\}) = \emptyset\}$$

In other words, time may increase at a uniform rate over all clocks, provided the control location's safety invariant is satisfied, and no urgent events are enabled. The next-state relation for all time-passage events is then $N_\delta = \bigcup_{t \in \mathbb{R}} N_{\delta_t}$. The next-state relation of the transition system is

$$N = \bigcup_{e \in T} N_e \cup N_\delta$$

The transition system induced by the timed safety automaton G is referred to as $\langle S_G, S_{0,G}, N_G \rangle$.

Because a transition system is unlabeled, we find it convenient for our discussion of timed systems to first define some familiar language-theoretic terms for timed automata. A *run* of the TSA G for the timed trace $e_0, e_1, e_2, \ldots$ is any infinite sequence of timed states $s_0, s_1, s_2, \ldots$ which is a path in $\langle S_G, S_{0,G}, N_G \rangle$ such that for all $i \geq 0$, either

- $e_i = \sigma$ and $(s_i, s_{i+1}) \in N_\sigma$, where $N_\sigma = \bigcup \{N_e \mid e \text{ is labeled with } \sigma\}$, or,
- $e_i = \delta_t$ and $(s_i, s_{i+1}) \in N_{\delta_t}$.

Such a run may be represented pictorially as

$$s_0 \xrightarrow{e_0} s_1 \xrightarrow{e_1} s_2 \xrightarrow{e_2} \cdots$$

The *language* accepted by G is defined as the set of all divergent timed traces for which $\langle S_G, S_{0,G}, N_G \rangle$ has a run starting in $S_{0,G}$.

Graphical conventions

Automata are depicted graphically by labeled, directed graphs. Locations are represented by circular nodes, and transitions by labeled edges. Reset actions of transitions

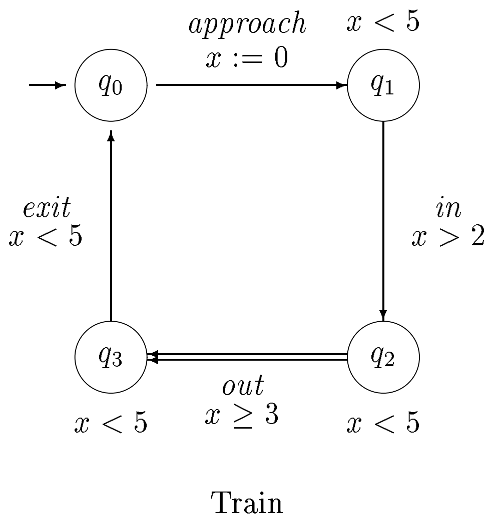


Figure 3.1: TSA for a train

are denoted by explicit assignments to 0. Urgent actions are denoted by double-lined arrows. Small incoming arrows mark any initial locations. Safety invariants are written next to the locations they apply to.

Example 3.2 *The automaton in figure 3.1 represents a train approaching a control intersection. While TSAs do not distinguish between input and output events, it is convenient here to think of the train as sending an* approach *signal to the controller. The train then enters the intersection (the* in *event) at least 2 time units later. The safety invariant* $x < 5$ *forces execution to leave location* q_1 *before the clock* x *reaches 5. We can infer that the* in *event must occur, and that it does so within 5 time units of the approach, because there is only one event leaving location* q_1. *Upon entering* q_2, *if the value of* x *is at least* 3, *then the urgent event* out *must occur right away, otherwise it will occur exactly* 3 *time units after the approach.* □

Simple timed automata

We introduce a special subclass of timed safety automata called *simple timed automata (STAs).* These are sufficient for modeling some but not all aspects of timed safety automata, and are particularly useful for analyzing systems which do not depend on the eventuality of timed events. A simple timed automaton is a timed safety automaton with no urgent events, and where all safety invariants are trivial, *i.e.* $Inv(q) = \mathbb{R}^n$ for all $q \in Q$. These automata have no means of forcing control to leave any given location, and therefore cannot model events which are guaranteed to occur. In particular they cannot express bounded liveness properties such as "$y = 5$ within 3 seconds". However their simplified semantics permits faster verification.

3.3 Modeling real-time systems

This section discusses process composition using timed safety automata, and how we can guarantee *non-Zenoness*, *i.e.* ensuring timed safety automata do not represent systems for which time cannot progress without bound.

3.3.1 Process composition

Most systems consists of a number of interacting processes. For a clear and compact description, each component can be represented with a separate timed safety automaton, and their parallel execution modeled by their automaton *composition*. For simplicity, we interchangeably use the term *real-time process* to refer to both the process being modeled and its timed safety automaton representation.

The composition operation uses interleaving semantics, with synchronization over shared events. Note however that a straightforward language semantics of a real-time process is not compositional, because of the treatment of urgent events, whose enabling conditions depend on external components.

Given two real-time processes denoted $P' = \langle \Sigma', Q', Q'_{init}, C', T', Inv' \rangle$ and $P'' = \langle \Sigma'', Q'', Q''_{init}, C'', T'', Inv'' \rangle$, with disjoint sets of clocks, their composition is defined by the real-time process $P = \langle \Sigma, Q, Q_{init}, C, T, Inv \rangle$, where

- $\Sigma = \Sigma' \cup \Sigma''$
- $Q = Q' \times Q''$

- $Q_{init} = Q'_{init} \times Q''_{init}$
- $C = C' \cup C''$
- T consists of all tuples $\langle\langle q'_1, q''_1\rangle, \sigma, \psi, a, \langle q'_2, q''_2\rangle, urg\rangle$ such that either
  - $\sigma \in \Sigma'$ and $\sigma \notin \Sigma''$ and there is a transition $\langle q'_1, \sigma, \psi, a, q'_2, urg\rangle$ in T', and $q''_2 = q''_1$, or,
  - $\sigma \in \Sigma''$ and $\sigma \notin \Sigma'$ and there is a transition $\langle q''_1, \sigma, \psi, a, q''_2, urg\rangle$ in T'', and $q'_2 = q'_1$, or,
  - $\sigma \in \Sigma' \cap \Sigma''$ and there are transitions $\langle q'_1, \sigma, \psi', a', q'_2, urg'\rangle$ in T', and $\langle q''_1, \sigma, \psi'', a'', q''_2, urg''\rangle$ in T'' such that
    - $\psi = \psi' \wedge \psi''$, and,
    - $I_a = I_{a'} \cup I_{a''}$, and,
    - $urg = 1$ iff either $urg' = 1$ or $urg'' = 1$
- $Inv(\langle q', q''\rangle) = Inv'(q') \wedge Inv''(q'')$

3.3.2 Non-Zenoness

A machine model of a timed system is *non-Zeno* iff every finite execution can be extended to an divergent infinite one [HNSY92]. A timed safety automaton P_1 is called *time progressive* with respect to a set of processes $\{P_2, \ldots, P_k\}$ iff it satisfies the following conditions:

- (immediate progress): every control location q has either
  1. no upper bound in its safety invariant, *i.e.* $Inv(q)$ includes all its time successors, or
  2. for every $\vec{x} \in Inv(q)$, there is a transition labeled σ leaving location q, such that
     - for some δ, $\vec{x} + \vec{\delta}$ satisfies its timing enabling condition, and

- the transition is guaranteed to be enabled in the product because all other processes sharing the event σ never disable σ regardless of which timed-states they are in, *i.e.* for every $i \geq 2$, if $\sigma \in \Sigma_i$, then for every control location $q' \in Q_i$ and point $\vec{x}\,'$ there is an outgoing transition from q' labeled σ satisfied by $\vec{x}\,'$.

- (time-progressive cycles): for every cycle of transitions in P there is a positive constant $\delta > 0$ such that it is impossible to traverse the cycle without at least δ time passing.

Theorem 3.3 *Given a set of real-time processes $\mathcal{P} = \{P_1, P_2, \ldots, P_k\}$, if each P_i is time progressive with respect to $\mathcal{P} \setminus P_i$, then $P_1 \parallel P_2 \parallel \cdots \parallel P_k$ is non-Zeno.*

Proof: We show that every finite timed run can be extended to an infinite divergent one. Consider the timed-state $s = \langle q, \vec{x}\,\rangle$ at the end of the finite run. We extend the run inductively as follows.

If time can progress without bound in the current control location, we are done, since we can repeatedly take events δ_t for any fixed positive t, yielding a divergent run.

Suppose otherwise. If a transition is enabled, take it, leading to timed-state s_1. Otherwise, we may add a time passage event δ_t until a transition t is enabled. The following reasoning shows this can always be done. By the immediate progress property, for every $i \geq 1$ for which P_i's control location has a nontrivial safety invariant there is a δ_i such that a transition is enabled in P_i after δ_i time units and the safety invariant in P_i still holds. Let δ_t be the smallest such δ_i. Then we can safely add δ_t time units to the global state without violating any safety invariants, and there is an event enabled at $\langle q, \vec{x} + \vec{\delta_t}\,\rangle$. After adding this time passage event to the run, the transition t is fired.

Repeating the above procedure results in a path either leading to a control location with no upper bound in its invariant, or a path involving infinitely many labeled transitions. The first case obviously gives a divergent run, and in the second case, the run must pass through infinitely many cycles, giving a divergent run because each cycle takes at least a fixed non-zero number of time units, because of the time-progressive cycles condition. □

Finally, we note that simple timed automata are always non-Zeno, since arbitrary amounts of time may pass while control remains in any fixed location.

Theorem 3.4 *Simple timed automata are non-Zeno.* □

3.3.3 Example

We consider a simple version of the well-known timed mutual exclusion protocol due to Fischer. A similar example appears in [AL92, SBM92].

This is an n process algorithm, where each process uses timing constraints on its actions to ensure mutual exclusion. Each process has a unique process identifier i and 4 operating states. They synchronize their actions through the shared variable X. From location q_0 a process may advance to location q_1 at any time provided X has value 0. It may delay here for up to Δ_B seconds before setting the value of X to i. It simultaneously advances to location q_2, from which it may enter its critical section as long as it does so after at least δ_c seconds and the value of X is still i. Upon leaving its critical section, it reinitializes X to 0.

The timed safety automata for the case of two processes are given in figure 3.2. The conditions on the value of the global variable X are maintained by the special process called VARIABLE-X whose states encode the current value of the global variable. In other words, if this process is at control location q_i then X equals i. Because each process can independently read and write the value of the variable X, we need to create separate events for each process. If not, the events could only occur when they were synchronized across all processes. Thus Process 1's alphabet has events *start1* for starting the protocol, *setX1* for moving from state q_1 to q_2 and setting X to 1, *enter1* for entering its critical section, and *P1setX0* for leaving its critical section and reassigning the global variable x to 0. Whenever a process has an event for writing the value of X, the process for the variable X shares that event, and its effect in VARIABLE-X reflects the written value. Constraints on each process's behavior are expressed by disallowing certain process events when the value of X

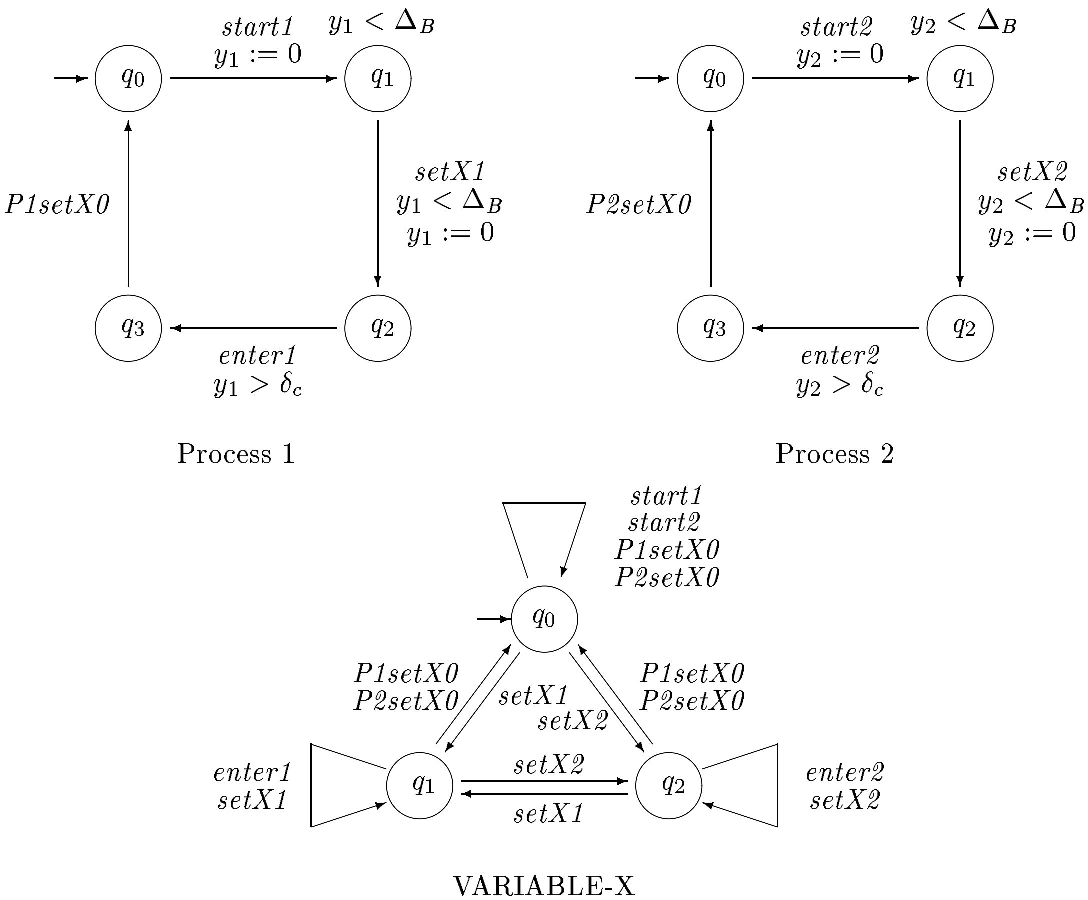


Figure 3.2: Automata for mutual exclusion protocol

would prohibit it. For example, the lack of a *start1* action out of locations q_1 and q_2 indicates Process 1 cannot start the protocol if X equals 1 or 2.

The clock y_i is used to express the timing conditions on transitions. Notice that the safety invariants at locations q_1 of each contending process force the process to proceed to the next step of the algorithm: it cannot delay in q_1 forever. However, there is no similar invariant forcing a process to eventually enter its critical section in location q_3.

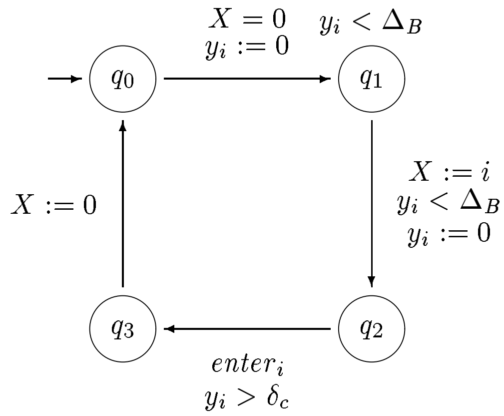


Figure 3.3: Real-time process i for mutual exclusion protocol

Non-Zenoness

The composed system is non-Zeno since each process P_i satisfies both the immediate progress property (since the safety invariant at q_1 implies the enabling constraint on the event leaving q_1) and the time-progressive cycles condition (since at least Δ_B time units pass on each cycle through P_i). Thus by theorem 3.3 their composition is non-Zeno.

Graphical shorthand

For simplicity and clarity of exposition, we allow an abbreviated automaton representation which handles discrete-valued variables over finite domains. We write $X := k$ within a process P to mean that it executes a write event of the variable X, assigning it the value k. It is understood that the process for the variable X will include transitions modeling the effect of the P's write event. Similarly, read events may appear as $X = k$ in a process, with the corresponding event enabled in the process for X from the location for value k. In this case the automaton model for the variable X need not be explicitly shown.

For example, an automaton for the i-th process in the Fischer's mutual exclusion algorithm appears in figure 3.3. By convention, variables are written in upper case to help distinguish them from clocks.

3.4 Safety verification

A methodology for verifying timed safety properties of a non-Zeno real-time system is the following:

1. Describe the real-time system to be verified as a non-Zeno timed safety automata A.
2. Describe the complement of the specification as a timed automaton D with a specially marked violation state, *i.e.* all violating traces have a run in the automaton leading to its violation state.
3. Form the product G of D and A.
4. Test whether the violating state in D is reachable in G.

This procedure is equivalent to checking for emptiness of the language $L(A) \cap \overline{L(\mathit{Spec})}$. In many instances, the automaton for the complemented specification may be obtained by first constructing a *deterministic* timed safety automaton for the specification, then taking its *completion.* The idea behind the completion automaton is that every trace not in the specification induces a run leading to the violating state. Because a violation corresponding to time exceeding a safety invariant is not detectable as a labeled event, we need to add a new event α to signal this has happened. The completion *compl*(A) of the automaton A is a timed automaton with a specially marked trap state which has incoming edges for every potential transition not enabled in A, including those which correspond to allowing time to pass beyond any safety invariants. Let $W(q,\sigma) = \mathit{Inv}(q) \;\cap\; \bigcup\{\phi' \mid \exists q', a' \text{ such that } (q,\sigma,\phi',a',q') \in T\}$ be the set of points within q's safety invariant for which q has an enabled transition labeled σ. The action a_{ID} is the null reset action. A constraint is *maximal* in a set Y iff it is contained in Y and not contained in any other enabling constraint within Y.

The definition uses maximal sets because the direct complements are not time zones, and so are not permissible as timing constraints on transitions. The completion of deterministic automaton A is defined as $compl(A) = \langle \Sigma', Q', Q_{init}, C, T', Inv' \rangle$ where

- $\Sigma' = \Sigma \cup \{\alpha\}$, where the event $\alpha \notin \Sigma$ signifies time has exceeded a location's safety invariant.
- $Q' = Q \cup \{q_{viol}\}$, where q_{viol} is a special violation state.
- $Inv'(q) = \mathbb{R}^n$ for all control locations $q \in Q'$.
- $T' = T_0 \cup T_1 \cup T_2 \cup T_3$, where
  - $T_0 = \{(q, \sigma, \phi', a, q') \mid (q, \sigma, \phi, a, q') \in T \text{ and } \phi' = \phi \cap Inv(q)\}$, representing transitions in A with the implicit constraint that safety invariants be satisfied made explicit,
  - $T_1 = \{(q, \sigma, \phi, a_{\text{ID}}, q_{viol}) \mid \phi \text{ is maximal in } \overline{W(q, \sigma)}\}$, representing all events for which A has no transition.
  - $T_2 = \{(q, \sigma, \phi, a_{\text{ID}}, q_{viol}) \mid \phi \subseteq \overline{Inv(q)}, \text{ and } \phi \text{ is maximal in } \overline{Inv(q)}\}$, representing events which may occur when the safety invariant at q does not hold.
  - $T_3 = \{q_{viol}, \sigma, TRUE, a_{\text{ID}}, q_{viol})\}$

Example 3.5 *Figure 3.4 shows a deterministic automaton A and its completion. The alphabet of A is $\Sigma = \{a, b\}$. The safety invariant on location q_0 is removed. In order to correctly constrain the event a leading to location q_1 the conjunct $x \leq 5$ is added to its enabling condition. If a b event ever occurs in location q_0 it is a violation. Furthermore any event occurring beyond q_0's safety invariant indicates that no outgoing event has taken place in timely fashion, and again control enters the q_{viol} state. Strictly speaking, the enabling condition $x \neq 3$ is syntactically illegal since it does not represent a time zone, but we use it as shorthand for two transitions for b, one enabled when $x < 3$ and one when $x > 3$.* □

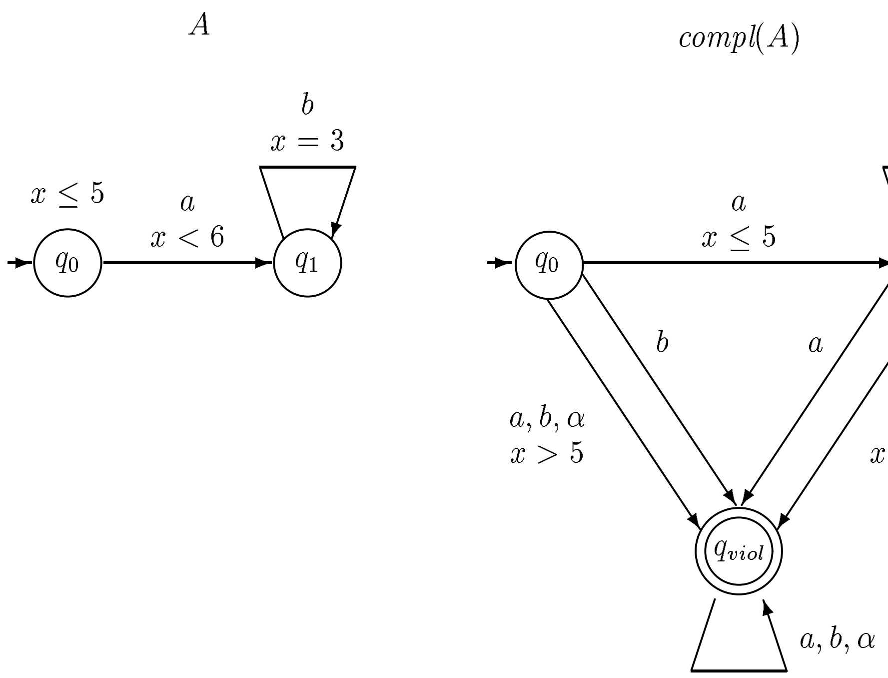


Figure 3.4: Automaton completion

The projection of a timed trace $\tilde{e}$ over Σ onto a subalphabet $\Sigma' \subseteq \Sigma$ is denoted $\mathit{proj}(\Sigma')(\tilde{e})$ and is defined as the trace obtained by deleting all events in $\Sigma \setminus \Sigma'$ from $\tilde{e}$.

Proposition 3.6 *Given a deterministic timed safety automaton A, a timed trace τ is not in $L(A)$ iff there exists a trace τ' such that $\mathit{proj}(\Sigma)(\tau') = \tau$ and $\mathit{compl}(A)$'s run for τ' enters the trap state.*

Proof: (Sketch) Since A is deterministic, by construction so is $\mathit{compl}(A)$. Furthermore $\mathit{compl}(A)$ has a run for every timed trace over Σ.

If the timed trace τ is not in $L(A)$, then there must be some point at which either time passes beyond the current safety invariant or an event occurs for which there is no enabled transition in A. We show that both cases cause $\mathit{compl}(A)$'s run to enter the violation state for a trace whose projection is τ. In the first case, the safety invariant in A is violated. If this happens after i events in τ, then $\mathit{compl}(A)$

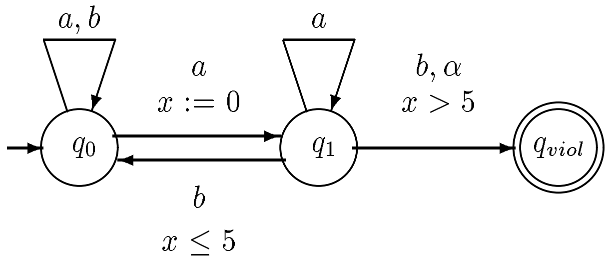


Figure 3.5: Bounded liveness specification

can mimic A over the first i events of τ using transitions in T_0, take a transition in T_2 via the added event α to the violation location, then follow transitions in T_3 for the remainder of τ. In the second case, an illegal labeled transition occurs in A. If this happens at the i-th event in τ, then *compl*(A) can mimic A over the first $i-1$ events of τ using transitions in T_0, take a transition in T_1 to the trap location, then follow transitions in T_3 for the remainder of τ. In both cases, it is easy to see that *compl*(A)'s trace τ' projects onto τ.

The reverse direction of the equivalence is similar and omitted. □

We note that for the purposes of safety verification the self-loops on the violation location can be dropped. This is because it is not necessary to continue the run for a trace not in $L(A)$ once it is known that *compl*(A) has a corresponding run to the q_{viol} location.

Example 3.7 *Bounded liveness is a common form of specification property. Figure 3.5 shows how an automaton can specify the property "every a event is followed by a b event within 5 time units."* □

Example 3.8 *Fischer mutual exclusion: The automata of the processes in the Fischer mutual exclusion algorithm were given in figure 3.2. We verify the untimed safety property that no two processes are ever in their critical sections at the same time. This property is expressed by the automaton of figure 3.6. As an alternative, we observe that if all the processes are symmetric, we can test for the error condition*

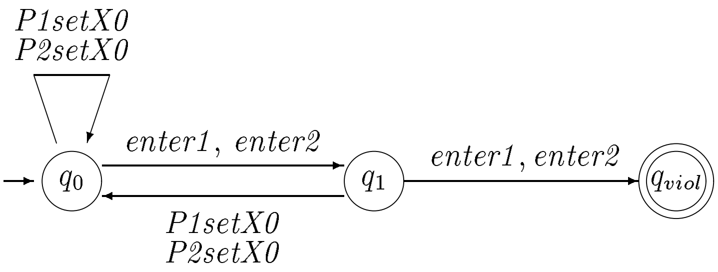


Figure 3.6: Mutual exclusion specification

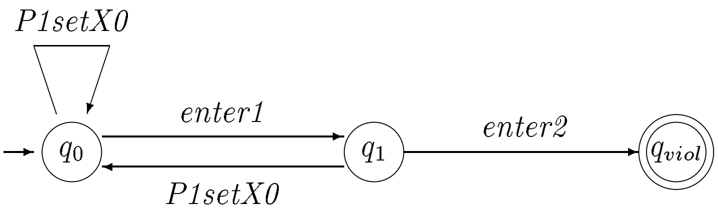


Figure 3.7: Mutual exclusion specification

resulting from Process 1 entering its critical section, followed by Process 2 entering its critical section, as shown in figure 3.7. □

3.4.1 Decidability

This section is a restatement of results by Alur, Courcoubetis, and Dill [ACD90, AD90], who show that the state-space of an timed automaton can be divided into a finite number of equivalence classes sufficient for deciding whether a particular control location is reachable. We briefly describe the equivalence relation, which gives a bisimulation over the transition system induced by a timed automaton. It essentially distinguishs the critical integral values of the clocks and the ordering of their fractional parts. We assume that every clock appears in some enabling condition, and define K_i to be the largest constant which clock x_i is ever compared to. For any $r \in \mathbb{R}$, let $\lfloor r \rfloor$ denote the integral part of r and *fract*(r) the fractional part, *i.e.* *fract*$(r) = r - \lfloor r \rfloor$. We first define the equivalence relation $\approx_{AD}$ on n-vectors as $\vec{x} \approx_{AD} \vec{x}\,'$ if and only if

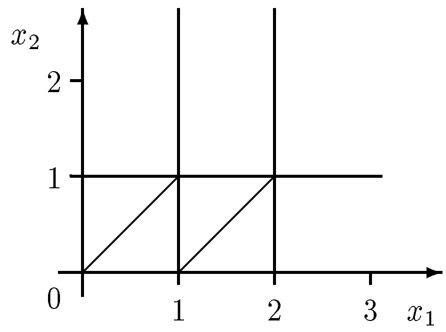


Figure 3.8: Detailed Alur-Dill regions

1. $\forall i = 1..n$, if $x_i \leq K_i$ or $x'_i \leq K_i$ then
   (a) $\lfloor x_i \rfloor = \lfloor x'_i \rfloor$
   (b) $\mathit{fract}(x_i) = 0$ iff $\mathit{fract}(x'_i) = 0$
2. $\forall i, j = 1..n$, if $x_i \leq K_i$ and $x_j \leq K_j$, then

$$\mathit{fract}(x_i) < \mathit{fract}(x_j) \text{ iff } \mathit{fract}(x'_i) < \mathit{fract}(x'_j)$$

We extend this equivalence relation from points in $\mathbb{R}^n$ to timed-states as follows: $\langle q, \vec{x} \rangle \approx_{AD} \langle q', \vec{x}' \rangle$ iff $\vec{x} \approx_{AD} \vec{x}'$ and $q = q'$. The equivalence classes are called *detailed regions.*

Example 3.9 *The detailed regions induced by the two clocks x_1 and x_2 with $K_1 = 2$ and $K_2 = 1$ are all the intersection points, open line segments, and open faces in figure 3.8.* □

The following three theorems are due to Alur and Dill.

Theorem 3.10 *For a timed safety automaton $\mathcal{A}$, the number of equivalence classes of the relation $\approx_{AD}$ is $O(|Q| \cdot |C|! \cdot 2^{|C|} \cdot \Pi_{x \in C}(2K_x + 2))$.* □

A relation $\approx$ is said to be a *labeled bisimulation* over a set of timed-states with respect to the relations N_σ and N_δ, given in subsection 3.2.3, iff for all s_1, s_2, $s_1 \approx s_2$ implies

1. for all s'_1, if $N_\delta(s_1, s'_1)$ then there exists a timed-state s'_2 such that $N_\delta(s_2, s'_2)$ and $s'_1 \approx s'_2$.
2. for all σ, for all s'_1, if $N_\sigma(s_1, s'_1)$ then there exists a timed-state s'_2 such that $N_\sigma(s_2, s'_2)$ and $s'_1 \approx s'_2$.

Theorem 3.11 *The relation $\approx_{AD}$ is a labeled bisimulation over the timed-states.* □

A reachability analysis can be performed over the equivalence classes, instead of over the individual timed-states. We construct a *set-graph*, a graph whose nodes are sets of states. There is an edge in the set-graph from set A to set B whenever there exists an edge in the underlying transition system from some state $a \in A$ to some state $b \in B$. The nodes of the set-graph are the detailed regions, and because these form a bisimulation, a class is reachable in the set-graph iff some element of it is reachable in the underlying timed transition system.

Theorem 3.12 *The timed safety verification problem is decidable.* □

We note however that the problem is PSPACE-complete. It is exponential both in the number of clocks and the size of the timing constants. Reachability over modular untimed systems is already a hard problem. But the addition of timing information is comparable to adding extra processes, and makes real-time verification in practice much harder than analyzing untimed systems. This difficulty motivated the search for effective heuristics for timing verification to be viable on real examples.

Chapter 4

Verifying Real-Time Systems – Part I

4.1 Introduction

The approximation algorithm can be applied to real-time systems represented by timed safety automata. The first four sections of this chapter show how to perform forward and backward symbolic reachability on timed automata. Sets of timed states are symbolically represented using *rounded regions*. We define these sets of states in section 4.4 and review a description of an efficient data-structure for them, *difference bounds matrices* due to Dill [Dil89]. We show how to perform the successor, predecessor, and intersection operations.

The rest of the chapter describes how approximation is applied to verify real-time systems. It provides the approximating operators, discusses termination, and demonstrates the algorithm over toy examples.

In this chapter, we describe approximations over the timing component only of the state-space. We delay until the next chapter a discussion of how approximations can be performed simultaneously over both the control information and timing information.

Symbolic representation of timing information

We use the *rounded regions* of an automaton as the domain of approximating sets. Recall that the states of a real-time system are pairs of the form $\langle q, \vec{x}\,\rangle$, where q is a TSA location and $\vec{x}$ is a vector of clock values. In this chapter we consider only sets of states which share the same control location, namely sets of the form $\langle q, Z\rangle$ where Z is a *rounded (time) zone*. The algorithm for approximating reachable states is obtained in a straightforward way, except for two considerations, namely rounding (to ensure the algorithm terminates) and the use of a disjunctive next-state relation (to ensure that each next-state computation is closed for the approximating sets).

4.2 Time zones and bounds

Successful symbolic verification of real-time systems depends on effective manipulation of sets of timed states. The rounded zones we use in our approximating sets are a subclass of time zones. Time zones have an efficient representation due to Dill [Dil89] called *difference bounds matrices (DBMs)*. Difference bounds matrices have a canonical form for which there are $O(n^3)$ algorithms for finding intersections, time successors, time predecessors, images and preimages of events [Dil89, ACD$^+$92, Rok93], where n is the number of clocks in the systems.

Recall that a time zone $Z \in \mathcal{Z}(n)$ is a (possibly unbounded) polyhedron defined by integer constraints on clocks and clock differences. If we identify a new fictitious clock variable x_0 with the constant value 0, these constraints can be represented uniformly as bounds on the difference between two clock values. For instance, $x > 5$ can be expressed as $x - x_0 > 5$. Furthermore we can restrict attention to upper bounds without loss of generality. More precisely, each inequality can be re-expressed in one of the following forms:

$x_i - x_j < k$ or $x_i - x_j \leq k$, for some integer k,

To describe these inequalities in a uniform fashion we introduce the domain of *bounds*. Let $\mathbf{Z}^- = \{\ldots -3^-, -2^-, -1^-, 0^-, 1^-, 2^-, \ldots\}$ where n^- represents a value "infinitesimally different from n". A bound is any element of $\mathbf{Z} \cup \mathbf{Z}^- \cup \{-\infty, \infty\}$.

Each bound is intended to represent an upper bound on a real value. We take both "$x \leq n^-$" and "$x < n^-$" to mean "$x < n$" and similarly "$x \geq n^-$" and "$x > n^-$" stand for "$x > n$". We define an ordering $\prec$ on bounds as the smallest ordering induced by the usual ordering over $\mathbf{Z} \cup \{-\infty, \infty\}$ and $n - 1 \prec n^- \prec n$. The relation $\preceq$ is defined over bounds as $b_1 \preceq b_2$ iff $b_1 \prec b_2$ or $b_1 = b_2$.

Bounds can be added, with the exception that $-\infty$ cannot be added to ∞. Bounds in $\mathbf{Z}$ and $\mathbf{Z}^-$ are finite, and the value of the bounds n and n^-, denoted $v(n)$ and $v(n^-)$ respectively, is n. The result of computing $b + b'$ is

$$\begin{cases} (v(b) + v(b')) & \text{if } b \text{ and } b' \text{ are in } \mathbf{Z} \\ (v(b) + v(b'))^- & \text{if } b \text{ and } b' \text{ are both finite, and at least one is in } \mathbf{Z}^- \\ -\infty & \text{if } b \text{ or } b' \text{ is } -\infty \\ \infty & \text{otherwise} \end{cases}$$

4.3 Difference bounds matrices

A *difference bounds matrix* (DBM) for $\mathbb{R}^n$ is an $(n+1) \times (n+1)$ matrix of bounds, with rows and columns indexed from 0 to n. The DBM A with entries a_{ij} represents the polyhedron consisting of all points that satisfy the inequalities $x_i - x_j \leq a_{ij}$ for each i and j. Clearly every time zone can be described by a DBM. However there are many DBMs defining the same zone, because some of the upper bounds need not be tight. For example, the time zone Z in figure 4.1 represented by the system of inequalities

$$\begin{aligned} x_1 &< 2 \\ x_1 &\geq 1 \\ x_2 &\leq 5 \end{aligned}$$

can be represented by any matrix

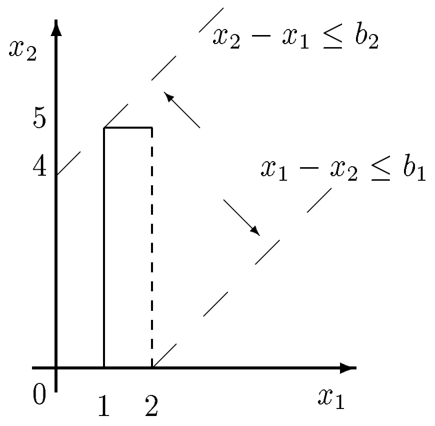


Figure 4.1: Time zone Z

$$\begin{array}{ccc} 0 & -1 & 0 \\ 2^- & 0 & b_1 \\ 5 & b_2 & 0 \end{array}$$

where $b_1 \not\prec 2^-$ and $b_2 \not\prec 4$.

4.3.1 Canonical form for DBMs

The key idea in performing operations on zones is to represent them as canonical DBMs. A constraint $x_i - x_j \leq b$ is said to be *tight* for a time zone Z iff there is no bound $b' \prec b$ such that all of Z satisfies $x_i - x_j \leq b'$. The canonical matrix, denoted cf(Z), has all entries representing tight constraints. Dill [Dil89] showed that this matrix can be computed from an arbitrary matrix for Z by applying an all-pairs shortest path algorithm. This representation therefore leads to easy tests for equality and emptiness of time zones.

```
procedure time_successors(A,B)
input  DBM A;    /* DBM for Z */
output DBM B;    /* DBM for time successors */
B := A;
for i := 1 to n do
   B[i][0] := ∞;
endfor
```

Figure 4.2: Pseudocode for finding time successors

4.3.2 Operations on time zones

We demonstrate how operations on time zones can be computed over their DBM representations.

Intersection

The intersection of two time zones Z and Z' is a time zone. It can be computed from their DBMs. Intuitively we take the conjunction of all the inequalities for both zones by taking the lower of the two bounds for each pair of clock differences. Let A and A' be DBMs for Z and Z'. The zone $Z \cap Z'$ is represented by the matrix B where for all i and j, $b_{ij} = min\{a_{ij}, a'_{ij}\}$, where the minimum min of two bounds is defined using the ordering $\prec$ over bounds.

Time successors

The set of time-successors $Z_{\nearrow}$ of the time-zone Z is obtained from Z by removing all inequalities of the form $x \leq k$ or $x < k$, since these upper bounds restrict time passing indefinitely. The pseudo-code of figure 4.2 describes how this operation can be performed on a canonical DBM. The result is a canonical DBM for $Z_{\nearrow}$.

```
procedure reset(A,I_a,B)

input   DBM A, reset I_a;     /* DBM for Z, reset index set I_a */
output  DBM B;                /* DBM for a(Z) */

B := A;
for x_i ∈ I_a do
   /* disregard constraints involving clock x_i */
   for j := 1 to n do
      B[i][j] := ∞;
      B[j][i] := ∞;
   endfor
   /* enforce clock reset, i.e. x_i = 0 */
   B[i][0] := 0;
   B[0][i] := 0;
endfor
```

Figure 4.3: Pseudocode for computing resets

Time predecessors

Similar to the computation of time successors, we may replace all lower bounds on clocks with 0. However in this case, canonical input does not in general imply the output will be canonical.

Reset actions

In order to find the set of timed successors under an instantaneous transition, we need to compute the image of the transition's reset action. Let a be a reset action with corresponding index set I_a. Then $a(Z)$ is the projection of Z onto the axes for variables in I_a. It can be found by first ignoring all constraints on variables in I_a, and then taking the subset for which all variables in I_a equal 0. Pseudo-code for this operation appears in figure 4.3.

Inverse images of reset actions

We also need to compute the inverse image of the transition's reset action. Let a be a reset action with corresponding index set I_a. The set $a^{-1}(Z)$ consists of all timer vectors $\vec{x}$ such that $a(\vec{x}) \in Z$. It is the union $\bigcup_{\vec{y} \in Z} \{\vec{x} \mid a(\vec{x}) = \vec{y}\}$ which is the same as $\bigcup_{\vec{y} \in Z \cap Z_a} \{\vec{x} \mid a(\vec{x}) = \vec{y}\}$, where Z_a is the zone where all clocks in I_a are equal to 0. In other words, it is the set of all clock vectors $\vec{x}$ for which there exists a vector $\vec{y} \in Z \cap Z_a$ which agrees over all clock variables not in I_a.

Thus the inverse $a^{-1}(Z)$ is computed by first finding the possible image of a within Z (this is done by setting to 0 the bounds on the absolute value of each clock in I_a and canonicalizing), and then taking the inverse projection of the reset variables (by making all bounds relating to clocks in I_a trivial).

4.4 Rounded time zones

This section explains why using arbitrary time zones would not guarantee termination in reachability algorithms. We then define a restricted form of time zone called the *rounded time zone*, which is used in our approximating sets.

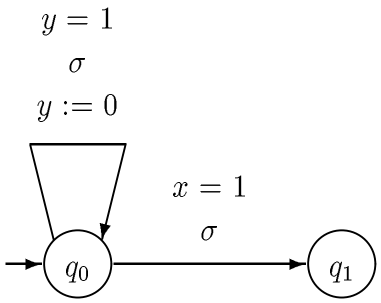
Decidability of the timed safety verification problem follows from the finiteness of the Alur-Dill equivalence relation. A naive verification algorithm could explicitly enumerate all the reachable equivalence classes. A more practical algorithm may choose to use symbolic enumeration, by considering *sets* of equivalence classes at a time. Time zones are a natural candidate for a symbolic representation of sets, because operations on them can be performed efficiently. If the time zones encountered by an algorithm were always Alur-Dill equivalence classes, or the exact union of classes, the algorithm would terminate. This, however, is not always the case. Consider a simple set-reachability algorithm that generates a set-graph where each node is a time zone, and every successor set of every node also appears in the graph. Such a graph can be generated using a simple reachability algorithm as in figure 4.4. The algorithm will terminate if and only if the cardinality of $\{N^k(S_0) \mid k \geq 0\}$ is finite. However for timed systems, the algorithm may generate time zones with successively larger finite bounds.

```
procedure set_reachability

input  ⟨S, S_0, N⟩;      /* a transition system */
output G;                /* a set-graph as described above */

vertices(G) := {};
edges(G) := {}
initial(G) := {S_0};
stack := emptystack;
push(S_0, stack);
while (not empty(stack)) do
  A := pop(stack);
  B := N(A);
  if (B ≠ {}) then
    if (B ∉ vertices(G)) then
      vertices(G) := vertices(G) ∪ B;
      push(B, stack);
    endif
    edges(G) := edges(G) ∪ ⟨A, B⟩;
  endif
endwhile
```

Figure 4.4: Set reachability algorithm

Example 4.1 *The set-reachability algorithm applied as above to the two-state automaton A_1 in figure 4.5 would not terminate. The algorithm would successively generate sets with points $(i, 0)$ after each self-loop on q_0. The reachable time zones for q_0 are shown in the figure.* □

One way to use symbolic representations of timed states and still maintain termination properties is to replace each time zone generated with the set of Alur-Dill equivalence classes that it intersects. The problem with this strategy is two-fold: firstly, finding the set of intersecting equivalence classes may be expensive, and secondly, the classes may not be representable by a small number of time zones. For instance, in a 3-clock automaton with $K_1 = 1$, $K_2 = 2$, and $K_3 = 3$, the classes intersecting the time successors of the singleton time zone consisting of the origin require at least 3 time zones to be represented, *e.g.* $(x_1 = x_2 = x_3 \leq 1)$, $(1 < x_1 \wedge 1 < x_2 = x_3 \leq 2)$,



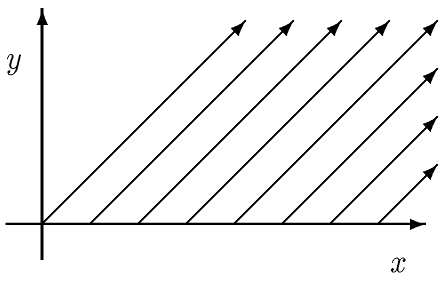


Figure 4.5: Automaton A_1, causes nontermination without rounding

and $(1 < x_1 \wedge 2 < x_2 \wedge 2 < x_3)$.

The approach we take is to use *rounded time zones* instead. Rather than replacing a time zone Z with the union of all the classes it intersects, we round it off by adding some but not necessarily all states which lie within the union. Such rounding preserves the correctness of the algorithm. The potential disadvantage of this approach is that there are more rounded time zones than zones. The advantages are that the rounded time zone is easy to compute, and the result is by definition a single time zone, rather than a union of separate time zones. We will see that for the example above, the rounded zone for the time successors is the zone of time successors itself.

4.4.1 Rounded time zones

In this subsection we define the rounding operation on zones. Since there are only finitely many rounded time zones, symbolic analysis over rounded time zones is guaranteed to terminate. We first provide an equivalent definition of the Alur-Dill partitioning relation $\approx_{AD}$ in terms of the constraints. Equivalence classes are determined by a set of *primary constraints* which are always applied, and also *secondary constraints*, only some of which may be relevant depending on which particular primary constraints are satisfied. We then show how to refine this relation into *constraint zones*, where both primary and secondary constraints are always relevant. *Rounded time zones* are defined to be time zones which are the exact union of constraint zones.

Alternative definition for Alur-Dill classes

We now give an equivalent definition for $\approx_{AD}$. We say $\vec{x} \approx_r \vec{x}\,'$ iff

1. they satisfy the exact same subset of *primary constraints*, of the form:

   $x_i \leq k$, $x_i < k$, $x_i \geq k$, $x_i > k$, where x_i is a clock and $k \leq K_i$ is an integer constant

   and,

2. if they satisfy any of the above constraints of the form $x \leq k$ or $x < k$ for *both* x_i and x_j, then they also satisfy the exact same subset of *secondary constraints*, of the form:

   $x_i - x_j \leq k$, $x_i - x_j < k$, where x_i and x_j are clocks and $-K_j \leq k \leq K_i$ is an integer constant.

Proposition 4.2 *The equivalence relations $\approx_r$ and $\approx_{AD}$ are the same,* i.e. $\vec{x} \approx_r \vec{x}\,'$ *iff* $\vec{x} \approx_{AD} \vec{x}\,'$.

Proof: The first set of constraints in the definition of $\approx_r$ determines whether x_i is less than or equal to K_i, and if so its exact integral part, and whether its fractional part is equal to zero. Thus if $\vec{x}$ and $\vec{x}\,'$ satisfy the same set of constraints, then they also share the same integral parts, and both are either exact integers or not.

We claim the second set of constraints is sufficient to determine the relative ordering of the fractional parts of two clocks, whenever both clocks are sufficiently small. Suppose $x_i \leq K_i$ and $x_j \leq K_j$. Then $-K_j \leq x_i - x_j \leq K_i$, since clock values are always positive. Now observe the condition of the Alur-Dill equivalence can be reexpressed in terms of the constraints.

- $\mathit{fract}(x_i) < \mathit{fract}(x_j)$ iff $x_i - x_j < \lfloor x_i \rfloor - \lfloor x_j \rfloor$.

Each of these two conditions is determined by the constraint sets of the definition of $\approx_r$. $\square$

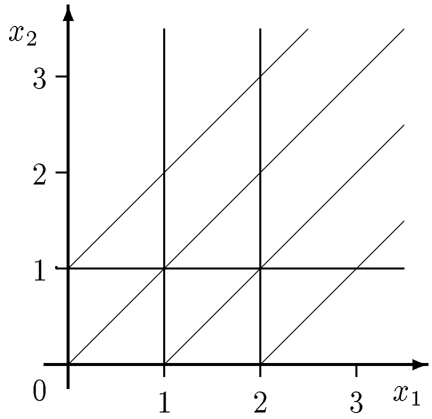


Figure 4.6: Constraint zones

Constraint zones

We define a third equivalence, which partitions the timer-valuations into *constraint zones*. These constraint zones are finer than the regions obtained from the above definition, and are used in describing *rounded regions*. We say $\vec{x} \approx_{cz} \vec{x}\,'$ iff they satisfy the exact same subset of *legal constraints*, of the form:

- $x_i \leq k$, $x_i < k$, $x_i \geq k$, $x_i > k$, where x_i is a clock and $k \leq K_i$ is an integer constant,
- $x_i - x_j \leq k$, $x_i - x_j < k$, where x_i and x_j are clocks and $-K_j \leq k \leq K_i$ is an integer constant.

The legal constraints are precisely the primary and secondary constraints used in defining the relation $\approx_r$. Observe that in contrast to $\approx_r$, the secondary constraints are always used in partitioning classes, regardless of which primary constraints hold. Notice that if we define $K_0 = 0$ for the fictitious clock x_0 whose value is always 0, then all legal constraints are of the form

- $x_i - x_j \preceq b$, where x_i and x_j are clocks and b is a bound value such that $-K_j \leq b \leq K_i$.

The constraint zones induced by the two clocks x_1 with $K_1 = 2$ and x_2 with $K_2 = 1$ are shown in figure 4.6. Pictorially the difference between these regions and the Alur-Dill regions (see figure 3.8) is the extension of the diagonals for the secondary difference constraints.

Proposition 4.3 *The relation $\approx_{cz}$ refines the relation $\approx_r$.* □

Let a *rounded zone* be any time zone which is the union of constraint zones.

Proposition 4.4 *Rounded zones are closed under intersection.*

Proof: Clearly the intersection of rounded zones is a time zone, so we need only show that it is the union of constraint zones. Constraint zones are disjoint, so since every rounded zone is the union of constraint zones, so must be its intersection. □

We define the function *round* to map any time zone to the intersection of all rounded zones which include it, *i.e.*

$$round(Z) = \bigcap\{Z' \mid Z \subseteq Z' \text{ and } Z' \text{ is a rounded zone}\}$$

Corollary 4.5 *For any time zone Z, $round(Z)$ is a rounded zone.* □

Lemma 4.6 *A time zone Z is a rounded zone iff it is definable as the conjunction of a set of legal constraints,* i.e. *there exists a set of legal constraints Θ such that $Z = \{\,\vec{x} \mid \vec{x}$ satisfies every constraint in $\Theta\}$.*

Proof: if: Let Z be defined by the set of legal constraints Θ. We show how Z can be partitioned into constraint zones. Each legal constraint $x_i - x_j \leq b$ is equivalent to the disjunction of legal constraints, $b'' \leq x_i - x_j \leq b'$ for each bound $b' \leq b$, where b'' is the nearest bound strictly lower than b', provided $b'' \leq x_j - x_i$ is a legal constraint, and $-\infty$ otherwise. Taking the conjunction of the disjuncts for each constraint in Θ defines Z in such a way that each product term defines a constraint zone.

only if: Let Z be a rounded zone. Then consider for each pair x_i, x_j, the set of all maximal bounds appearing in constraints $x_i - x_j \leq b$ used in tightly defining each of the constraint zones contained in Z. Each of these bounds corresponds to a legal constraint. Let Z' be the time zone defined by this set of legal constraints. We establish that Z is exactly Z'.

The zone Z' contains Z since its defining bounds are all greater that those appearing in the constraints defining each constraint zone in Z. Furthermore all bounds are tight. They cannot be lowered or else some points in Z would be excluded. Thus Z' is the smallest time zone containing Z, and hence is equal to Z. □

An array entry in a DBM is called *legal* iff it corresponds to a legal constraint. In other words, its integer bounding value is neither too small nor too big. Illegal constraints and entries are defined analogously.

Theorem 4.7 *The time zone for round(Z) can be represented by the DBM B obtained from the canonical DBM A for Z where all illegal entries have their bounds rounded up to the nearest legal bound value,* i.e.

$$b_{ij} = \begin{cases} a_{ij} & \textit{if } -K_j \leq a_{ij} \leq K_i \\ -K_j & \textit{if } a_{ij} < -K_j \\ \infty & \textit{if } a_{ij} > K_i \end{cases}$$

Proof: Let R be the time zone *round*(Z). Let Z_B be the time zone represented by B. Since Z_B is a rounded zone including Z, it follows that $R \subseteq Z_B$.

To see that $Z_B \subseteq R$, first observe by lemma 4.6 that R is definable by a set Θ of legal constraints. Let A_R be the matrix for R whose ij-th entry is

$$a^R_{ij} = \begin{cases} k & \text{if } x_i - x_j \leq k \text{ is in } \Theta \\ \infty & \text{otherwise} \end{cases}$$

Since A is contained in R and A is canonical, it follows that $a_{ij} \preceq a^R_{ij}$ because otherwise there would be a point in Z which satisfies $x_i - x_j \leq a_{ij}$ but not $x_i - x_j \leq a^R_{ij}$.

The rounding process replaces two kinds of entries, in either case with some $b_{ij} \preceq a^R_{ij}$ from which it follows that $Z_B \subseteq R$ as required.

Case 1: $a_{ij} < -K_j$

Then $b_{ij} = -K_j$. If $x_i - x_j \leq a^R_{ij}$ is a defining constraint in R, then $-K_j \preceq a^R_{ij}$

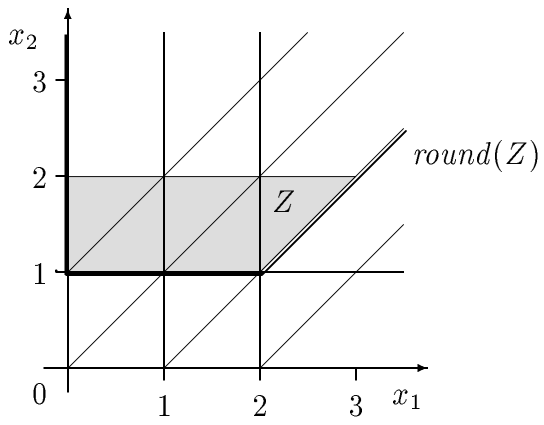


Figure 4.7: Rounded regions example

since it must be a legal constraint. This implies $b_{ij} \preceq a^R_{ij}$.

If there is no defining constraint along the $x_i - x_j$ diagonal then $a^R_{ij} = \infty$, in which case it is clear that $b_{ij} \preceq a^R_{ij}$.

Case 2: $a_{ij} > K_i$

We have that $K_i < a_{ij} \preceq a^R_{ij}$ and since all elements of A_R are either ∞ or legal constraint entries, it follows that a^R_{ij} must be ∞, so replacing a_{ij} with $b_{ij} = \infty$ does not affect containment. □

The rounded zone for Z defined by $1 \leq x_2 \leq 2$ and $x_1 - x_2 \leq 1$ is shown in figure 4.7. In section 4.4, we considered an automaton with $K_1 = 1$, $K_2 = 2$, and $K_3 = 3$. Three time zones are used to represent the time successors of the origin if we use Alur-Dill equivalence classes. However, the successor set is represented exactly by one rounded time zone.

Theorem 4.8 *For every time zone Z, round(Z) intersects the same regions as Z,* i.e. *for every $s \in round(Z)$ there is a state $s' \in Z$ such that $s \cong s'$.*

Proof: The proof must show that rounding Z is sound, *i.e.* it introduces no states whose Alur-Dill equivalence classes are not already represented in Z.

We first show it is sound to replace a single illegal constraint from a time zone which lies entirely within a detailed time zone. From this we infer that replacing all constraints from such a time zone is also sound. Given this fact, the result follows for an arbitrary zone Z, since Z is the union of zones Z_i which lie in distinct detailed regions, and if a defining constraint is illegal in Z_i then there is a similar defining illegal constraint in Z. In other words, the effect of replacing illegal constraints in Z is the same as replacing illegal constraints from each Z_i. Thus we need only establish the first claim, namely that we can soundly exchange a single illegal constraint from a zone contained in a detailed zone.

Let Z be such a zone, contained in the detailed zone D, and let $\theta : x_i - x_j \leq b$ be an illegal constraint in the canonical DBM representation of Z. The constraint θ is said to be a *defining* constraint for Z iff it is essential in the definition of Z, *i.e.* iff removing θ from the constraints in the DBM results in a different zone from Z. If θ is not a defining constraint, then replacing it with a weaker constraint in the rounding process has no effect, so we need only consider defining constraints.

For an illegal defining constraint θ, we consider four cases.

- $\theta = x_i \leq k$, for some $k > K_i$.

  Then θ is replaced by the trivial constraint $x_i < \infty$ in the rounding process. Since Z is contained in a detailed zone and θ is tight, it must be the case that $x_i > K_i$ for all points in Z. Thus D includes as a defining constraint $x_i < \infty$, and so replacing θ with $x_i < \infty$ in Z's DBM results in a region contained in D.

- $\theta = x_j \geq k$, for some $k > K_j$.

  Then θ is replaced by the constraint $x_j > K_j$ in the rounding process. Since Z is contained in the detailed zone D and θ is tight, it must be the case that $x_j > K_j$ is a constraint in D, since there are no critical constraints of form $x_j \geq k$ for any $k \leq K_j$. Thus replacing θ with $x_j > K_j$ in Z's DBM results in a region contained in D.

- $\theta = x_i - x_j < k$, for some $k > K_i$.

  Then θ is replaced by the constraint $x_i - x_j < \infty$ in the rounding process.

Suppose all points in D satisfy some primary constraint $x_i \leq b$ for some bound $b \leq K_i$. Then since θ is a tight constraint for Z, this contradicts containment in D. Therefore no points in D satisfy any primary constraints of the form $x_i \leq b$. Thus D is not defined by any secondary constraints of the form $x_i - x_j \leq b'$, and hence discarding the constraint θ from Z results in a zone contained in D.

- $\theta = x_i - x_j < k$, for some $k < -K_j$.

  Then θ is replaced by the constraint $x_i - x_j < -K_j$ in the rounding process.

  Suppose all points in D satisfy some primary constraint $x_j \leq b$ for some bound $b \leq K_j$. Then since θ is a tight constraint for Z, this contradicts containment in D. Therefore no points in D satisfy any primary constraints of the form $x_j \leq b$. Thus D is not defined by any secondary constraints of the form $x_i - x_j \leq b'$, and hence relaxing the constraint θ in Z results in a zone contained in D. $\square$

4.4.2 Augmenting next-state relations

We now formally justify the use of rounded regions. Given a verification problem $\mathcal{VP} = (S, S_0, N, V)$, a bisimulation $\approx$ respects $\mathcal{VP}$ iff every equivalence class is either entirely in V or disjoint from V. The set next-state relation $\tilde{N} : 2^S \to 2^S$ is said to be a $\approx$*-set-augmentation* of N for $\mathcal{VP}$ iff

1. $\approx$ is a bisimulation respecting $\mathcal{VP}$, and
2. $\tilde{N}$ augments N, *i.e.* for all sets $A \subseteq S, N(A) \subseteq \tilde{N}(A)$, and
3. for all $A \subseteq S$, for all $s \in \tilde{N}(A)$, there exists $t \in N(A)$ such that $s \approx t$.

Proposition 4.9 *Given a verification problem $\mathcal{VP}$, a bisimulation $\approx$ respecting $\mathcal{VP}$, and a set next-state relation $\tilde{N}$ which is a $\approx$-set-augmentation of N, (S, S_0, N, V) is correct iff $(S, S_0, \tilde{N}, V)$ is correct.*

Proof:

If (S, S_0, N, V) is incorrect, then so is $(S, S_0, \tilde{N}, V)$ since $N \subseteq \tilde{N}$.

On the other hand, if $(S, S_0, \tilde{N}, V)$ is incorrect, we can show that (S, S_0, N, V) is also incorrect by constructing a violating path in the original graph as follows. Suppose $t_0, t_1, t_2, \ldots, t_k$ is a violating path in $(S, S_0, \tilde{N})$ with $t_k \in V$. Let $s_0 = t_0$. Since $\tilde{N}$ is a $\approx$-augmentation of N there is a state s_1 that is bisimilar to t_1 and a successor state of s_0 via N. We inductively continue the construction of a path $s_0, s_1, \ldots, s_k$ in (S, S_0, N) where each $s_i \approx t_i$. Now since $\approx$ respects V and $t_k \in V$ it follows that s_k is in V, and hence (S, S_0, N, V) is also incorrect. $\Box$

Lemma 4.10 *The rounded regions are closed under the operations* $N_e^{rd} = round \circ N_e$ *and* $N_\delta^{rd} = round \circ N_\delta$. $\Box$

Lemma 4.11 *The states reachable with* N_e^{rd} *and* N_δ^{rd} *are bisimilar to those reachable by using* N_e *and* N_δ*, and thus* $\bigcup N_e^{rd} \cup N_\delta^{rd}$ *is a* $\approx_{AD}$*-set-augmentation of* $\bigcup N_e \cup N_\delta$. $\Box$

Theorem 4.12 *The verification problem with next-state relation* $\bigcup N_e \cup N_\delta$ *reduces to that over* $\bigcup N_e^{rd} \cup N_\delta^{rd}$. $\Box$

Theorem 4.13 *The set-reachability algorithm applied to a timed safety automaton with* N_e^{rd} *replacing* N_e *and* N_δ^{rd} *replacing* N_δ *terminates correctly.*

Proof: The result follows from the above theorem because there are only finitely many rounded zones. $\Box$

Thus we may use the rounded next-state relations to decide the verification problem for timed safety automata.

4.5 Approximation of real-time systems

4.5.1 Overapproximation

The overapproximation operator for verifying real-time systems is defined over time zones as the zone that results from rounding the smallest enclosing time zone:

$$A \sqcup B = round(enclose(A, B))$$

$$\mathit{enclose}(A, B) = \min\{Z'' | Z'' \text{ a time zone and } A \cup B \subseteq Z''\}$$

The smallest enclosing region is called the *prejoin* of A and B, and is well-defined since time zones are closed under intersection.

Proposition 4.14 *If A and B are represented by canonical DBMs with the same name, their prejoin is represented by the DBM D whose entries are the pairwise maxima of entries in A and B,* i.e. $d_{ij} = \max\{a_{ij}, b_{ij}\}$.

Proof: Let D be the time zone represented by the matrix of the same name. It includes A and B since all bounds in D are no tighter than in A and B.

To see that it is the smallest time zone containing both A and B, first observe that all bounds in A and B are as tight as possible. If any d_{ij} is tighter than a_{ij} say, then D cannot contain all of A since A contains points for which $x_i - x_j = a_{ij}$ but which are disallowed in D. Thus no bounds in D can be further tightened, and so D represents the smallest possible time zone enclosing A and B. □

The overapproximation operator is extended in the expected way to regions, *i.e.*

$$\langle q, Z\rangle \sqcup \langle q', Z'\rangle = \begin{cases} \langle q, Z' \sqcup Z''\rangle & \text{if } q = q' \\ \text{undefined} & \text{otherwise} \end{cases}$$

Proposition 4.15 *The set of rounded regions is closed under the overapproximation operator.* □

4.5.2 Underapproximation

We define the operator over single approximating sets, and the extension to sets of approximating sets follows from the discussion in section 2.2.3. The underapproximating $\trianglerighteq$ operator is defined as:

$$\langle q, Z\rangle \trianglerighteq \langle q', Z'\rangle = \begin{cases} \langle q', Z'\rangle & \langle q, Z\rangle \subseteq \langle q', Z'\rangle \\ \langle q, Z\rangle & \text{otherwise} \end{cases}$$

Proposition 4.16 *The operator defined above is an underapproximating operator.*

Proof: It clearly satisfies the correctness property UA_1. The non-emptiness property holds by the first condition in the definition. □

Proposition 4.17 *The set of rounded regions is closed under the underapproximating operator.* □

The plus operator over sets of approximating sets (see subsection 2.2.3) is restricted so that it is maximal up to a limit of k underapproximating sets per separating class. In other words the result of expanding a set $\{A_{ij}\}$ of approximating sets with another set $\{B_{ij'}\}$ is a superset of the original set, with as many $B_{ij'}$ added as possible, provided there are at most k sets in the extension.

4.5.3 Disjunctive next-state relation

The algorithm in chapter 2 assumes that the result of applying the next-state relation to an approximating set A yields an approximating set B. This approximating set B is then split across the separating classes into further approximating sets B_i, each of which is then joined to the existing approximating structure, one separating class at a time. However, the next-state relation of a timed safety automaton does not yield a single region, but rather a disjunction of regions, since the next-state relation is itself a disjunction of relations, each of which may yield different regions.

Such a situation can easily be handled by a modified approximation algorithm, by computing the next-state relation in parts. Suppose the next-state relation N is the disjunction of k relations N_i, and for each N_i is closed over the domain of approximating sets. Instead of computing $N(A)$ we consider each $N_i(A)$ in turn. The result after k computations, and applications of the approximating operators, has the same effect as computing the successors as a set of k approximating sets, and then performing the approximating operators in one step.

Theorem 4.18 *The modified algorithm for disjunctive next-state relations terminates correctly over finite state systems.*

Proof: (Sketch) Correctness is obvious since computing the next-state relation in several steps does not affect whether the algorithms correctly overapproximate or underapproximate. The necessary modifications to proofs of propositions supporting the termination theorem 2.14 are tedious but straightforward. □

4.5.4 Urgent events

We assume that urgent events have no timing constraints associated with them[1]. Urgent events could be modeled by adding safety invariants with upper time bound 0 on a clock which is reset on entering a state in which an urgent event is enabled. However, it is more effective to handle them directly. Marks can be placed on control locations in the automaton where urgent events are enabled. Rather than resetting a clock on entering the control location, the next-state relation is altered to disallow time passing in this state. The immediate advantage of this strategy is that we reduce the number of clocks in the system, which increases the speed of verification. Further benefits are discussed in the next chapter.

4.6 Proof of termination

The termination proof of the previous chapter applied to finite-state systems only. We show now that the algorithm also terminates for the verification of timed safety automata, essentially because the algorithm uses the finite domain of rounded regions for approximating sets.

Let $X = \{X_\alpha\}_{\alpha \in J}$ be a partition of S. We define the set

$$Y = \{Y_i \mid Y_i = \bigcup_{j \in J'} X_j \text{ for some } J' \subseteq J\}$$

to be the sets which are the union of blocks in the partition. A verification problem $\mathcal{VP} = (S, S_0, N, V)$ is said to be *separated* by the partition $X = \{X_\alpha\}_{\alpha \in J}$ of S iff

1. $S_0 \in Y$,

[1]It is possible to convert any timed safety automata into this form. A proper transition relation has no strict lower bounds in the enabling conditions of urgent events, so a location with outgoing urgent events can be divided into separate locations, each representing a zone where the urgent event is either enabled or disabled.

2. $V \in Y$

3. Y is closed under the next-state relation N, and the approximation operators $\sqcup$ and $\trianglerighteq$.

It is *separable* iff it is separated by some partition X. The problem $\mathcal{VP}$ is said to be *finitely separated* by X iff it is separated by X and X is finite. The term *finitely separable* is similarly defined.

Proposition 4.19 *If the verification problem* (S, S_0, N, V) *is* finitely separable *by* X*, and the domain of approximating sets includes all elements of* X*, then the full approximation algorithm of chapter 2 terminates, and correctly decides the verification problem.* □

Lemma 4.20 *The transition system induced by a timed safety automaton is finitely separable by the detailed rounded regions.* □

Proposition 4.21 *The full approximation algorithm applied to* $(S, S_0, \tilde{N}, V)$*, where the* $\approx$*-augmentation* $\tilde{N}$ *is* $N_\delta^{rd} \cup \bigcup N_e^{rd}$ *defined in section 4.4.1, terminates, and correctly decides the verification problem for timed safety automata.* □

4.7 Examples

We illustrate the algorithm with a couple of examples.

Separating classes and conditional joining

First consider the automaton A_1 in figure 4.8. An enabling condition of form $(x, y) = (1, 1)$ represents the constraint $x = 1 \wedge y = 1$. The violation location is q_3. The approximation algorithm finds the initial forwards overapproximation and underapproximation, and the backwards overapproximation. After these computations the algorithm halts with the system verified correct. Figure 4.9 shows the truly reachable states and the resulting approximating structures. The forward overapproximation starts by adding the time successors to the origin. The successors of the set $\langle q_0, (x = y)\rangle$

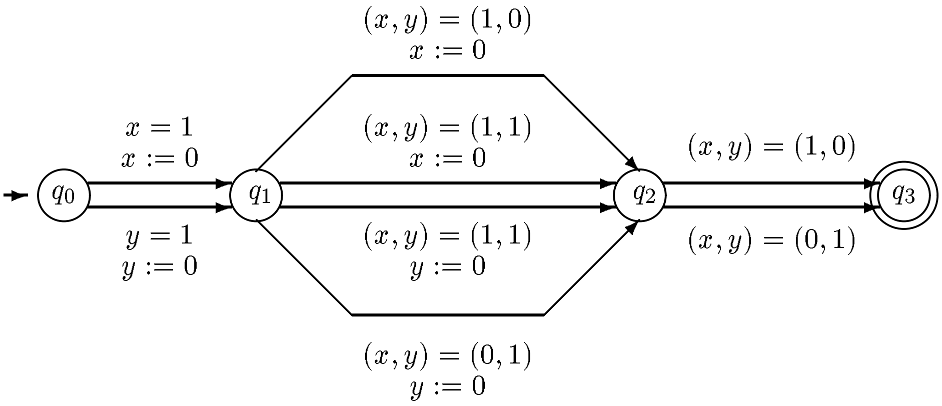


Figure 4.8: Timed safety automaton A_1

are the sets $\{\langle q_1, (1,0)\rangle\}$ and $\{\langle q_1, (0,1)\rangle\}$, each obtained by individually following a transition from q_0. Suppose the former set is added first to the overapproximation. The state-space is partitioned according to control location, so when the second set is added to the approximation, it is joined to the first, giving $\langle q_1, (x \leq 1 \wedge y \leq 1)\rangle$. Adding time successors to this set gives the region shown in figure 4.9. Joining the successor sets out of q_1 yields $\langle q_2, (x \leq 1 \wedge y \leq 1)\rangle$. Completing the approximation gives the states depicted on the second row of the diagram above.

The underapproximation is taken to consist of up to two approximating sets per separating class. The successor states of $\langle q_0, (x = y)\rangle$ are the individual sets $\langle q_1, \{(0,1)\}\rangle$ and $\langle q_1, \{(1,0)\}\rangle$. Since the approximation allows up to 2 sets per separating class, both are included. Adding time successors to $\langle q_1, \{(0,1)\}\rangle$ leads to the ray $\langle q_1, (y = x + 1)\rangle$. Since it includes the underapproximating set $\langle q_1, \{(0,1)\}\rangle$, it replaces the latter set in the underapproximation.

The backwards overapproximation starts with the violating states at location q_3. Suppose the transition enabled on $(x, y) = (1, 0)$ is considered first. Then the backwards overapproximation includes the set $\langle q_2, \{(1,0)\}\rangle$. The other transition into q_3 results in adding $\langle q_2, \{(0,1)\}\rangle$ to the approximation. These two sets at location q_2 are not joined together because doing so would violate condition 1 for permissible joins,

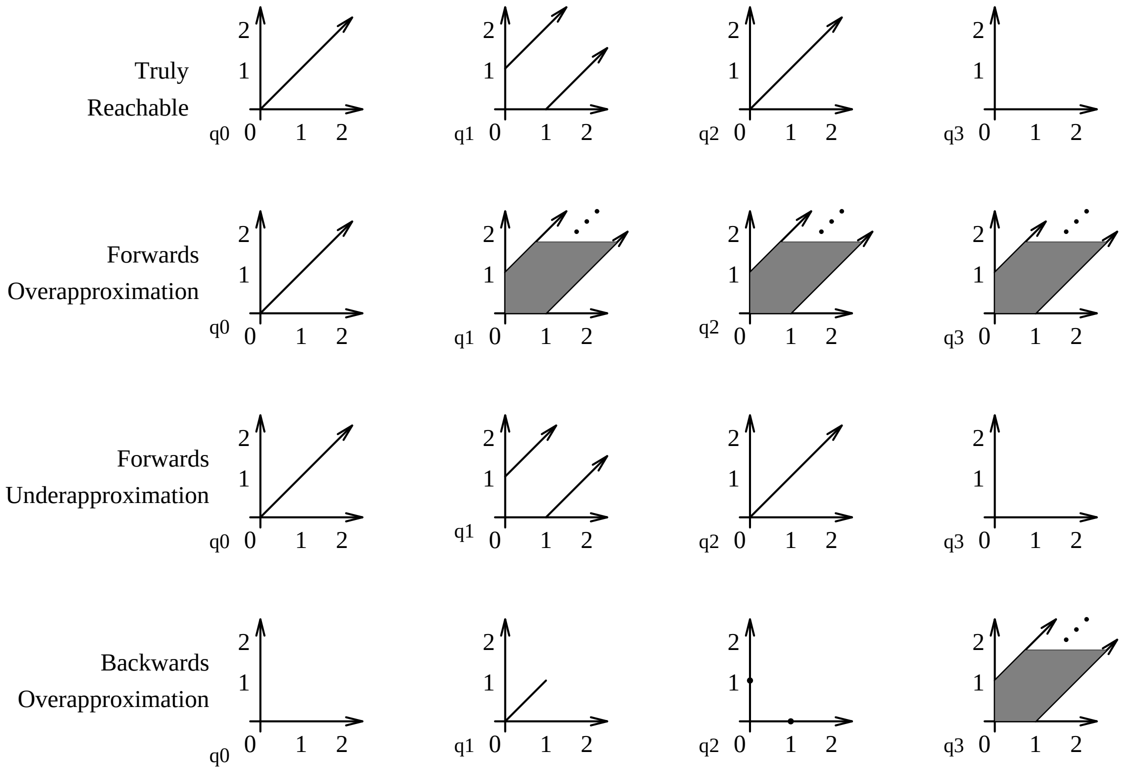


Figure 4.9: Approximations for A_1

i.e. their join $\langle q_2, (x \leq 1 \wedge y \leq 1)\rangle$ includes the forwards underapproximation whereas neither of the original sets do. Computation of the approximation completes without including the initial state, and so the system is correctly verified.

Rounding and urgent events

Our second example, shown in figure 4.10, illustrates rounding and the treatment of urgent events. The truly reachable states and the first forward overapproximation only are shown in figure 4.11. Time may pass without bound while control remains in location q_0. At any time less than 1, control may pass to location q_1. The urgent event is instantly enabled, leading to location q_2. Now time is allowed to pass in location q_2. The clock y may be reset whenever it reaches 1, and control may move to the location q_3 when $x = 2$.

The first forwards overapproximation begins by adding all time successors to the

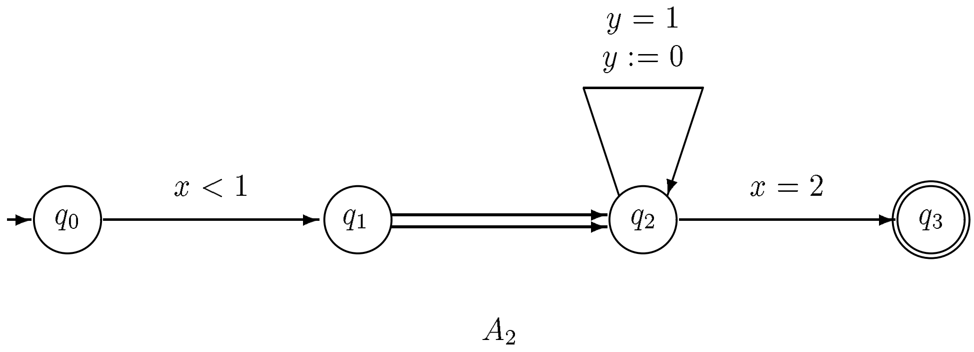


Figure 4.10: Timed safety automaton A_2

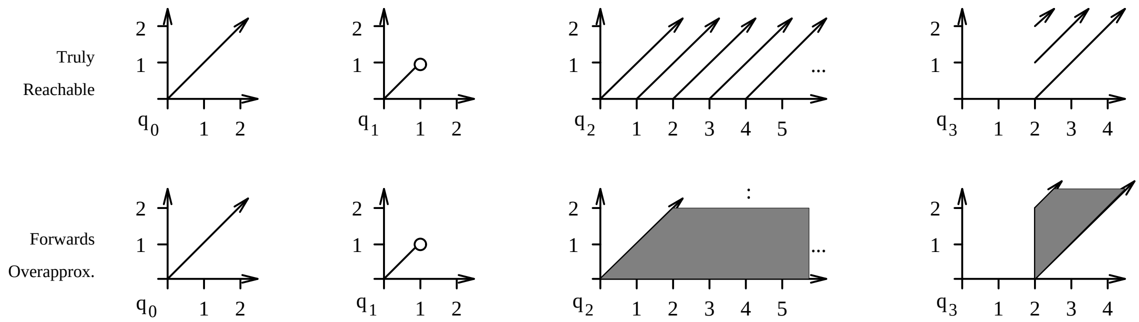


Figure 4.11: Approximations for A_2

zero vector at location q_0, giving the approximating set $\langle q_0, (x = y)\rangle$. This set is joined to the initial approximating set in the separating class for q_0, namely $\langle q_0, (x = y = 0)\rangle$, resulting in $\langle q_0, (x = y)\rangle$ since it contains the initial set. The transition from q_0 to q_1 leads to states $\langle q_1, (x = y < 1)\rangle$. Because there is an urgent event out of q_1, no time successors are added to this set. Following the urgent event leads to $G_1 = \langle q_2, (x = y < 1)\rangle$, to which the time successors $G_2 = \langle q_2, (x = y)\rangle$ may be added. The self-loop adds states $G_3 = \langle q_2, \{(1, 0)\}\rangle$ for which the prejoin with G_2 yields $G_4 = \langle q_2, (0 \leq x - y \leq 1)\rangle$ which is a rounded region.

Following the transition to q_3 leads to $H_1 = \langle q_3, (x = 2 \wedge 1 \leq y \leq 2)\rangle$. This is a rounded region, even though at first appearance it appears to be defined using the illegal constraint $y \leq 2$, which should then be discarded in the rounding process.

Notice however that the constraints $y - x \leq 0$ and $x = 2$ are legal constraints which imply $y \leq 2$. Thus the rounding operation leaves H_1 unaffected. Similarly, time successors can be added to H_1 giving $H_2 = \langle q_3, (x \geq 2 \wedge 0 \leq x - y \leq 1)\rangle$. A further self-loop on G_4 followed by adding time successors yields $G_5 = \langle q_2, (0 \leq x - y \leq 2)\rangle$, while the transition from q_2 to q_3 results in $H_3 = \langle q_3, (x \geq 2 \wedge 0 \leq x - y \leq 2)\rangle$ which is a rounded region. Adding time successors to H_3 leaves it unchanged. The effect of rounding can be seen when considering the next self-loop at q_2. The immediate successors of the self-loop from G_5 are $\langle q_2, (y = 0 \wedge 1 \leq x \leq 3)\rangle$. The prejoin of these successors with G_5 is $G_6 = \langle q_2, (0 \leq x - y \leq 3)\rangle$. The constraint $x - y \leq 3$ is illegal, since the constant exceeds $K_x = 2$. Removing $x - y \leq 3$ from G_6's DBM and replacing it with the trivial bound $x - y \leq \infty$ results in the rounded region $G_7 = \langle q_2, (0 \leq x - y)\rangle$, and no further states are then added to the overapproximation.

Chapter 5

Verifying Real-Time Systems – Part II

5.1 Symbolic representation of control locations

In many realistic real-time systems, large state-spaces arise not only from the complexity of timing information, but also from having numerous control locations. The algorithm shown in the last chapter employed a single control location per approximating set. If there are many reachable control locations, the algorithm will have to store a large number of approximating sets. We counter this problem by clustering together information across different control locations. The last chapter showed how to use approximating sets of the form $\langle q, Z \rangle$, where q is a control location and Z a rounded time zone. We generalize this to allow approximating sets of the form $\langle A, Z \rangle$ where A is a *set* of locations and Z is as before a rounded time zone. Thus control information may be represented symbolically as well as the timing information. This technique may dramatically reduce the number of approximating sets which need to be considered.

We first define the approximating operators. Later we show how the algorithm is modified to allow approximation of the next-state relation as well as approximation of the state-space. These modifications are necessary for efficiently handling the issues of urgent events and safety constraints in the passage of time.

5.1.1 Combining domains for approximation

We first show how different operators over different components of the state-space can be combined. Suppose a state-space is the cross-product of two domains, *e.g.* $S = S_0 \times S_1$. Approximating operators over the domains S_0 and S_1 can be combined to give an approximating operator over S. For convenience, we use $\langle A, B\rangle$ to denote the cross-product $A \times B$ of A and B.

Overapproximating

Given approximating sets $\langle A, B\rangle$ and $\langle A', B'\rangle$ for $S_0 \times S_1$, and overapproximating operators $\sqcup_1$ and $\sqcup_2$ over approximating sets for S_0 and S_1 respectively, we define their combination $\sqcup$ such that

$$\langle A, B\rangle \sqcup \langle A', B'\rangle = \langle A \sqcup_1 A', B \sqcup_2 B'\rangle$$

Let D be the domain consisting of the pairs of approximating sets for S_0 and S_1.

Proposition 5.1 *The operator $\sqcup$ defined above is an overapproximating operator over D.*

Proof: The operator is closed over D since each of the component operators is. Furthermore, $\langle A, B\rangle \subseteq \langle A \sqcup_1 A', B \sqcup_2 B'\rangle$ since $A \subseteq A \sqcup_1 A'$ and $B \subseteq B \sqcup_2 B'$. The argument for $\langle A', B'\rangle$ is analogous. $\square$

Underapproximating

Given approximating sets $\langle A, B\rangle$ and $\langle A', B'\rangle$ and underapproximating operators $\trianglerighteq_1$ and $\trianglerighteq_2$ over approximating sets for S_0 and S_1 respectively, we define their combination $\trianglerighteq$ as

$$\langle A, B\rangle \trianglerighteq \langle A', B'\rangle = \begin{cases} \langle A', B'\rangle & \text{if } \langle A, B\rangle \subseteq \langle A', B'\rangle \\ \langle A \trianglerighteq_1 A', B\rangle & \text{if } B \subseteq B' \text{ and } A \not\subseteq A' \\ \langle A, B \trianglerighteq_2 B'\rangle & \text{if } A \subseteq A' \text{ and } B \not\subseteq B' \\ \langle A, B\rangle & \text{otherwise} \end{cases}$$

Proposition 5.2 *The operator* $\trianglerighteq$ *defined above is an underapproximating operator over* D*.*

Proof: The operator is clearly well-defined and closed.

Containment is obvious for the first and last cases. By symmetry, we need only explain the second case. If $B \subseteq B'$, then since $A \trianglerighteq_1 A' \subseteq A \cup A'$, it follows that $\langle A \trianglerighteq_1 A', B\rangle \subseteq \langle A \cup A', B\rangle = \langle A, B\rangle \cup \langle A', B\rangle \subseteq \langle A, B\rangle \cup \langle A', B'\rangle$.

Finally, the non-emptiness condition is satisfied because of the first case. $\square$

Real-time operators

To specify the operators used for approximating real-time systems, we need only provide the operators over the two domains of control locations, and timer vectors. These operators can be combined as outlined above. We use the same operators as before over timer vectors. For simplicity, we use the exact union operator over sets of control locations, *i.e.* $A \sqcup_1 A' = A \cup A'$.

5.1.2 Computing successors

In the case of a single control location per approximating set, it is easy to compute the exact set of successors of an approximating set for any transition. The set of successors is itself an approximating set. When the approximating sets include sets of control locations, it is still straightforward to compute successors under instantaneous transitions. Timing information for the successors is independent of the control locations: if a transition is taken, its reset action must be applied to all timer values, regardless of the incoming or outgoing control locations. However, computing the exact set of time successor states is more complicated, because now the control locations affect timing information: urgent events and safety invariants may restrict how long time can pass. Consider the problem of efficiently finding the set of time successor states for the approximating set $\langle A, Z\rangle$. Each location $q \in A$ has a potentially different safety invariant, so the number of approximating sets in the time successors of $\langle A, Z\rangle$ may be as large as the size of A. If this were the case, it would defeat the purpose of grouping together information about different control locations in the same approximating set. Furthermore, time is not permitted to pass in those control locations with urgent events enabled. We tackle this problem by allowing approximations of the next-state relation, as discussed in section 2.4

5.2 Approximating real-time systems

In this section we describe the full algorithm advocated for verifying real-time systems using approximations over control information and timing information. The next-state relation for time-passage events is both underapproximated and overapproximated. The algorithm requires the use of additional splitting between traversals to ensure termination.

5.2.1 Approximating next-state relations

The algorithm proceeds exactly as described above in section 5.1, except that the next-state relation is approximated for the passage of time. Exact computation is performed for instantaneous events. Recall that the next-state relation for a timed safety automaton is the disjunction

$$N = \bigcup_{e \in T} N_e \cup N_\delta$$

where the time-passage relation is

$$N_\delta = \bigcup_{t > 0} N_{\delta_t}$$

We assume as before that urgent events are constrained only by control locations, and are independent of timing information. Let $Urg(Q)$ be those control locations for which there is some outgoing urgent event. For each $t \in \mathbb{R}$, we observe that

$$N_{\delta_t} = \{(\langle q, \vec{x}\,\rangle, \langle q, \vec{x} + \vec{t}\,\rangle) \mid q \in Q \setminus Urg(Q), \vec{x} + \vec{t} \in Inv(q)\}$$

The relation N_δ is not closed over approximating sets. In general the successors $N_\delta(\langle A, Z\rangle)$ form a set of approximating sets, one for control locations in $Urg(Q)$, and up to one each for every different safety invariant for the locations in A.

$$N_\delta(\langle A, Z\rangle) = \langle A \cap \mathit{Urg}(Q), Z\rangle \cup \bigcup_{q \in A \setminus \mathit{Urg}(Q)} \langle q, Z_{\nearrow} \cap \mathit{Inv}(q)\rangle$$

Notice that this successor set need not be represented with one approximating set for every location in $A \setminus \mathit{Urg}(Q)$, since the approximating sets with locations sharing the same safety invariant will share the same time zone, and thus can be combined into approximating sets of the form $\langle \{q \in A \setminus \mathit{Urg}(Q) \mid \mathit{Inv}(q) = \mathit{Inv}(q')\}, Z_{\nearrow} \cap \mathit{Inv}(q')\rangle$. However, the number of such sets can still be prohibitively large, especially since the timed safety automaton often represents the product of several parallel processes, so there may be exponentially many[1] different safety invariants for a set of control locations.

So while it is possible to use an exact next-state relation, we prefer to approximate the time successors using an overapproximating (set) next-state relation, and an underapproximating (set) next-state relation which returns exactly one approximating set rather than the list of approximating sets which would be returned by an exact computation.

The overapproximating relation $\overline{N_\delta}$ for the forwards relation N_δ is defined as

$$\overline{N_\delta}(\langle A, Z\rangle) = \begin{cases} \langle A, Z\rangle & \text{if } A \subseteq \mathit{Urg}(Q) \\ \langle A, Z_{\nearrow} \cap \mathit{Inv}(q)\rangle & \text{if } A \not\subseteq \mathit{Urg}(Q) \text{ and } \forall q' \in A, \mathit{Inv}(q) = \mathit{Inv}(q') \\ \langle A, Z_{\nearrow}\rangle & \text{otherwise} \end{cases}$$

The underapproximating relation $\underline{N_\delta}$ for the forwards relation N_δ is defined as

$$\underline{N_\delta}(\langle A, Z\rangle) = \begin{cases} \langle A, Z_{\nearrow} \cap \mathit{Inv}(q)\rangle & \text{if } A \cap \mathit{Urg}(Q) = \emptyset \text{ and} \\ & \quad \forall q' \in A, \mathit{Inv}(q) = \mathit{Inv}(q') \\ \langle A, Z\rangle & \text{otherwise} \end{cases}$$

[1]There may be a different invariant for every control location, and so the number of invariants may be exponential in the number of processes.

The approximating relations used for computing backwards reachable states are similar. The overapproximating relation $\overline{N_\delta^{-1}}$ for the relation N_δ^{-1} is defined as

$$\overline{N_\delta^{-1}}(\langle A, Z\rangle) = \begin{cases} \langle A, Z\rangle & \text{if } A \subseteq \mathit{Urg}(Q) \\ \langle A, (Z \cap \mathit{Inv}(q))_{\swarrow}\rangle & \text{if } A \not\subseteq \mathit{Urg}(Q) \\ & \qquad\text{and } \forall q' \in A, \mathit{Inv}(q) = \mathit{Inv}(q') \\ \langle A, Z_{\swarrow}\rangle & \text{otherwise} \end{cases}$$

The underapproximating relation $\underline{N_\delta^{-1}}$ for the relation N_δ^{-1} is defined as

$$\underline{N_\delta^{-1}}(\langle A, Z\rangle) = \begin{cases} \langle A, (Z \cap \mathit{Inv}(q))_{\swarrow}\rangle & \text{if } A \cap \mathit{Urg}(Q) = \emptyset \\ & \qquad\text{and } \forall q' \in A, \mathit{Inv}(q) = \mathit{Inv}(q') \\ \langle A, Z\rangle & \text{otherwise} \end{cases}$$

Let the domain of sets $\mathcal{Q}$ be defined as $\{\langle q, \mathbb{R}^n\rangle \mid q \in Q\}$.

Proposition 5.3 *1. The overapproximating relation $\overline{N_\delta}$ ($\overline{N_\delta^{-1}}$) is an overapproximation of the next-state relation N_δ (N_δ^{-1}).*

2. Furthermore, $\overline{N_\delta}$ and $\overline{N_\delta^{-1}}$ exactly match N_δ over sets for which all control locations share the same safety invariant and urgency information, and hence exactly match over $\mathcal{Q}$. □

Proposition 5.4 *1. The underapproximating relation $\underline{N_\delta}$ ($\underline{N_\delta^{-1}}$) is an underapproximation of the next-state relation N_δ (N_δ^{-1}).*

2. Furthermore, $\underline{N_\delta}$ and $\underline{N_\delta^{-1}}$ exactly match N_δ over sets for which all control locations share the same safety invariant and urgency information, and hence exactly match over $\mathcal{Q}$. □

To guarantee termination, it would suffice to show that whenever the routine Over_Approx is run with the approximate next-state relations $\overline{N_\delta}$,$\overline{N_\delta^{-1}}$, $\underline{N_\delta}$, and $\underline{N_\delta^{-1}}$, successive overapproximations are strictly decreasing with respect to $\prec_{base}$. Unfortunately, however, this is not the case. We use instead the policy introduced in section 2.4 of additional splitting to force the overapproximations to decrease with respect to $\prec_{set}$ until they eventually refine the partition $\mathcal{Q}$. Then since the approximating relations are exactly matching over $\mathcal{Q}$, termination follows.

5.2.2 Algorithm for real-time systems

The algorithm uses approximate next-state relations $\overline{N} = N_e \cup \overline{N_\delta}$, $\underline{N} = N_e \cup \underline{N_\delta}$, $\overline{N^{-1}} = N_e^{-1} \cup \overline{N_\delta^{-1}}$, and $\underline{N^{-1}} = N_e^{-1} \cup \underline{N_\delta^{-1}}$.

By propositions 5.3 and 5.4 and theorem 2.26, convergence is guaranteed if the approximating sets are forcibly refined until control locations share the same urgency information and invariants. Thus to ensure termination we may chose any maximal class for which the next-state relations are not exactly matching, and refine it by separating locations with different timing characteristics. We prefer to choose the classes for splitting in a way that will likely result in faster convergence of the approximations. The classes chosen for splitting are those sets A which are not adequately covered by the underapproximation. The idea is that in order to accelerate convergence, the underapproximation should increase as quickly as possible towards the overapproximation, while the overapproximation should decrease as fast as possible towards the underapproximation. By further dividing a separating class C by distinguishing states in the underapproximation from those not, we simultaneously force the overapproximations to be more accurate within C, and provide a means for the underapproximation to include more states in C.

Real-time approximation algorithm

The algorithm appears in figure 5.1. An overapproximation $\mathcal{A}$ is not merely flattened and used directly as the separating structure for the next traversal. Instead it is refined via the function Refine_Maximal() into a structure $\mathcal{C}$ such that $\mathcal{C} \prec_{set} \mathcal{A}$. The result of calling Refine_Maximal with overapproximation $\mathcal{A}$ and underapproximation $\mathcal{B}$ is a separating structure obtained from $\mathcal{A}$ by replacing every maximal set A with two disjoint parts:

1. $A_1 = (Q_1 \times \mathbb{R}^n) \cap A$

2. $A_2 = (Q_2 \times \mathbb{R}^n) \cap A$

```
RT_Approx
Over[BACKWARDS] := original separating structure;
Under[BACKWARDS] := empty approximating structure;
confirmed_positive := FALSE;
confirmed_negative := FALSE;
dirn := FORWARDS;
while ( (not confirmed_positive) and (not confirmed_negative) ) do
  Sep_Structure :=
    Refine_Maximal(Over[Opposite_Dirn(dirn)],Under[Opposite_Dirn(dirn)]);
  Over[dirn] :=
    Over_Approx(dirn,N̄,Sep_Structure,Under[Opposite_Dirn(dirn)]);
  Sep_Structure := Flatten(Over[dirn]);
  Under[dirn] := Under_Approx(dirn,N̲,Over[dirn]);
  dirn := Opposite_Dirn(dirn);
endwhile
```

Figure 5.1: Real-time approximating algorithm

where $A_1 \cup A_2 = A$. We define $proj(Q)(W)$ to be the set of all control locations found in the set of timed-states W. The splitting of A may be obtained by separating control locations by any one of the following criteria:

1. $Q_1 = proj(Q)(A \cap \cup\mathcal{B})$, or,
2. $Q_1 = Urg(Q) \cap proj(Q)(A)$, or
3. $\overline{N}_\delta(A_1) \neq A_1$

The first condition corresponds to separating out those control locations that have timed-states in the underapproximation already from those that do not. This policy is the one suggested in the discussion above. The second and third conditions reflect attempts to decrease the next overapproximation, by separating out control locations for which the passage of time could result in fewer timed-states being encountered, *i.e.* for some subset A' of A_1, $\overline{N}_\delta(A') \subset A_1$. Such sets of locations Q_1 may be obtained by splitting according to the safety invariants associated with control locations, or by separating the locations which have urgent outgoing events.

Simple timed automata

If we are verifying simple timed automata, there is no need to approximate the next-state relation N_δ. The syntax of simple timed automata does not allow urgent events, nor safety invariants, and so finding the exact sets of time successors and predecessors is always efficient, *i.e.* $N_\delta(\langle A, Z \rangle) = \langle A, Z_{\nearrow} \rangle$, and $N_\delta^{-1}(\langle A, Z \rangle) = \langle A, Z_{\swarrow} \rangle$.

5.2.3 Properties of algorithm

Proposition 5.5 *Each overapproximation FO_i from the algorithm above will either be strictly decreasing with respect to $\prec_{set}$,* i.e. *$FO_i \prec_{set} FO_{i-1}$, or the approximate next-state relations will be exactly matching over FO_i.*

Proof: Suppose the approximate relations are not exactly matching. Then by the definitions of the approximating relations there must be a set A for which the control locations differ for either safety invariants or urgent events, *i.e.* $\exists q, q' \in proj(Q)(A)$ such that $Inv(q) \neq Inv(q')$, or $\exists q, q' \in proj(Q)(A)$ such that $q \in Urg(Q)$ and $q' \notin Urg(Q)$. There must be a maximal set containing this set A, for which the control locations also differ in this regard. This set will be split causing FO_i to be computed with respect to a separating structure strictly less than FO_{i-1}, and hence $FO_i \prec_{set} FO_{i-1}$. A similar argument holds for backwards approximations. □

Proposition 5.6 *The algorithm above terminates for real-time systems.*

Proof: The result follows from proposition 5.5, the fact that the approximating relations are exact over $\mathcal{Q}$ and theorem 2.26. □

5.3 Ordered binary decision diagrams

The success of using approximations over control information depends heavily on having an efficient representation for sets of control locations. We therefore conclude this chapter by reviewing an effective symbolic representation for Boolean functions, the *ordered binary decision diagram* (OBDD) due to Bryant [Bry86]. This representation is used by our implementation for describing sets of control locations. In hardware verification and protocol verification, OBDDs have enabled successful formal verification well beyond the range of a traditional explicit implementation [BCM+90]. In addition it has been used for a variety of other problems involving manipulation of system's state-space, including the synthesis of supervisory controllers [HWT92b], logic synthesis [FKM91], sensitivity analysis and test generation [CB89], and logical databases [MC91].

Before defining OBDDs, we first describe how an untimed transition systems can be expressed and verified using Boolean functions.

5.3.1 Relations and Boolean functions

A transition system can be viewed as a set of tuples, which can in turn be expressed as Boolean functions. Operations on sets of states of a transition system can be expressed as manipulations on Boolean functions. This section makes this encoding more explicit. If Q is the set of locations of a transition diagram or automaton, let $Q' = \{q' | q \in Q\}$ be a set of locations representing the same locations in the next state of execution. If the alphabet of edge labels is Σ, a next-state relation for the transition function can be expressed as a set of tuples δ in $Q \times \Sigma \times Q'$. The sets of initial locations and final locations of an automaton can be thought of as 1-tuples.

Any set of n-tuples $T \subseteq X_1 \times X_2 \times \cdots \times X_n$ can be expressed by its characteristic function, *i.e.* a Boolean function $f : X_1 \times X_2 \times \cdots \times X_n \mapsto \{0, 1\}$ where $f(t) = 1$ iff $t \in T$. We can assume each X_i domain is Boolean. If X_i has more than two elements we can replace it by $\lceil log(|X_i|) \rceil$ Boolean domains giving a binary-encoding of its elements. It follows then that next-state relations, predicates describing initial states, final states, can all be expressed as Boolean functions over Boolean domains.

5.3.2 Ordered binary decision diagrams

An ordered binary decision diagram essentially encodes a Boolean function as a binary decision tree with the added restriction that the decisions are performed in a fixed order. In addition, common subtrees are shared for efficiency, thus resulting in

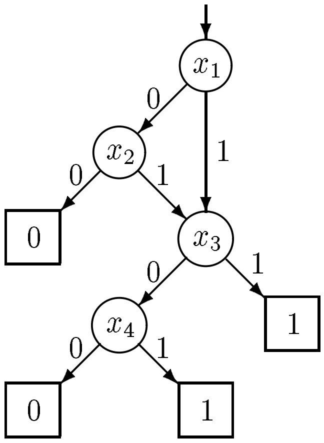


Figure 5.2: OBDD for the Boolean function $(x_1 \vee x_2) \wedge (x_3 \vee x_4)$

a directed acyclic graph (DAG). The value of the function for a particular variable assignment can be read by traversing the tree starting from the root, at each node branching according to the value of the variable labeling that node. Figure 5.2 shows a DAG representing $f = (x_1 \vee x_2) \wedge (x_3 \vee x_4)$. The variable assignment $(x_1 = 0, x_2 = 1, x_3 = 1, x_4 = 1)$ leads to a node marked 1. Thus f is true under this variable assignment. Notice that the value of x_4 is irrelevant. The path followed symbolically represents the two variable assignments $(x_1 = 0, x_2 = 1, x_3 = 1, x_4 = 0)$ and $(x_1 = 0, x_2 = 1, x_3 = 1, x_4 = 1)$.

Canonical form

However a Boolean function does not have a unique representation as a DAG. An OBDD is a DAG satisfying the additional constraint that the occurrence of variables on every path from the root to a leaf obeys a given total order. The DAG in Figure 5.2 is in fact a OBDD with variable ordering $x_1 < x_2 < x_3 < x_4$. Bryant showed that for any total order on the variables, every Boolean function is represented by a *unique* OBDD respecting that order. The advantage of having such a canonical form is that logical tests on Boolean functions given as OBDDs is easy: *logical equivalence* can be determined in linear time, and *satisfiability* and *validity* can be tested in constant time. For example, a formula is valid iff its OBDD representation is the same as that for TRUE.

Operations on OBDDs

Bryant also gave efficient algorithms to perform standard Boolean operations on OBDDs. The complexity of finding the OBDD for the logical AND, OR, or NOT of two OBDDs is bounded by the product of the sizes of the two OBDDs. To compute the successor states of a set of states, we need the additional operation of *quantification*. The existential quantification formula $\exists x_i[f]$ can be read as "(f holds when x_i is FALSE) OR (f holds when x_i is TRUE)". We use Bryant's restriction algorithm for $f|_{x_i=0}$ and $f|_{x_i=1}$ to implement $\exists x_i[f]$ as $f|_{x_i=0} \vee f|_{x_i=1}$.

Computational issues

The main advantage of using OBDDs to represent Boolean functions is that they are often far smaller than an explicit truth table representation. This fact can lead in practice to greatly improved performance but does not alter the exponential worst-case complexity *per se*. Thus the use of OBDDs is merely a heuristic to reduce the size of representing a Boolean function. Bryant has shown that there is no variable ordering that avoids an exponential representation of a multiplier. There is therefore no guarantee that implementations based on OBDDs will outperform those using explicit data structures. In the field of finite-state verification however, numerous researchers have already reported substantial improvements due to OBDDs [CK91, BCM$^+$90] over particular problem domains.

Finally, we note that typically an OBDD's size depends crucially on the chosen variable ordering. Intuitively a small OBDD will result when the function's value can be correctly determined from the remaining variable values and only a small amount of intermediate information about the variables already seen. It is generally desirable for a variable ordering to bunch together variables that are highly interdependent. For example, suppose $W_1 = \{x_{11}, \ldots, x_{1n_1}\}, \ldots, W_m = \{x_{m1}, \ldots, x_{mn_m}\}$ is a partition of the variables of f, and $f = f_1 \wedge \cdots \wedge f_m$ where each f_i depends only on variables in W_i. Let the size of the OBDD for g be denoted $|g|$. Then $|f| = O(|f_1| + \cdots + |f_m|)$ under the variable ordering $x_{11} < \cdots < x_{1n_1} < \cdots < x_{m1} < \cdots < x_{mn_m}$. Such a scenario can arise when composing loosely coupled components in a product system.

This reduced complexity is a substantial gain over an explicit representation that would be exponential in the number of components. However with a bad choice of variable ordering, the OBDD representation could also be exponential. Hence some understanding of the nature of the problem is needed to select a good variable ordering. This thesis does not explore this issue or exploit any of the advantages obtainable from clever variable orderings.

Chapter 6

Case Studies

We give some examples of real-time systems described as timed automata. We also provide automata for several timing properties used as specifications. Throughout the chapter we provide hints for describing various aspects of timing behavior. We conclude with a discussion of the limitations of using timed automata as a representation language. The performance of our verifier on the following examples can be found in chapter 8.

6.1 Examples

6.1.1 Train-gate controller

Our first example is one which appears frequently in the literature: an automatic controller which opens and closes a gate at a railway track intersection [LS85, Alu91]. The system consists of three components: a train, a gate, and their controller. The automata modeling the system's components are shown in Figure 6.1. Whenever a train enters the intersection, it sends an *approach* signal at least 2 seconds in advance to the controller. The controller also detects the train leaving the intersection, and this event occurs within 5 seconds after it started its approach. The gate responds to *lower* and *raise* commands by moving *down* and *up* respectively within certain time bounds. The controller sends a *lower* command to the gate exactly 1 second after

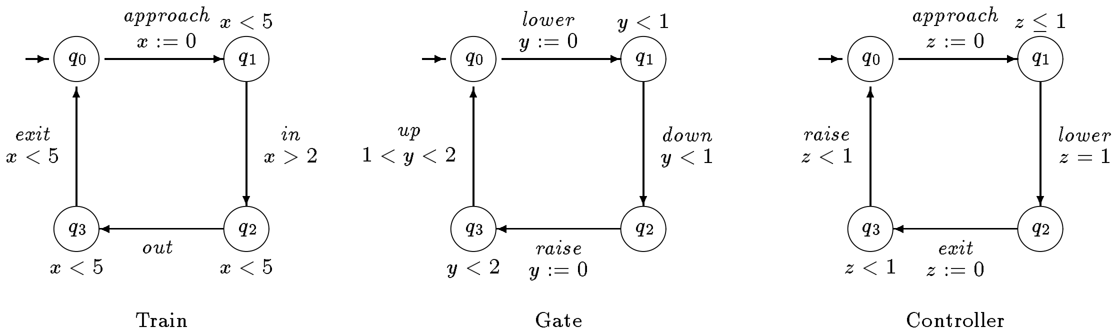


Figure 6.1: Automata for train-gate controller example

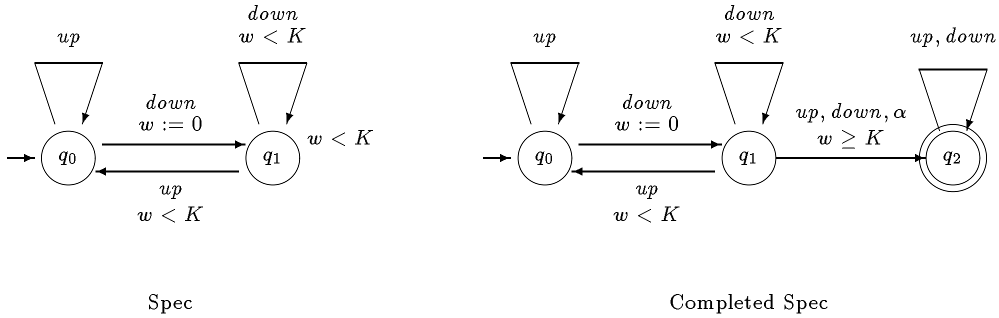


Figure 6.2: Real-time safety specification

receiving an approach signal from the train. It commands the gate to raise within 1 second of the train's exit from the intersection.

We verify a simple real-time safety property, namely that whenever the gate goes down, it is moved back up within a certain upper time bound K. In other words, the gate is never down for as long as K seconds. See the Spec automaton in Figure 6.2. It is deterministic and its completion is expressed by the same automaton with the added location q_2 which is marked as violating. The timing conditions on the edges from q_1 to q_2 are the complement of the existing edges for each event. In this case they happen to be the same for both *down* and *up* events. We do not need to add edges from q_0 to q_2 since both events are already enabled at all times in q_0. Whenever the specification constant is greater than or equal to 7 the specification is satisfied.

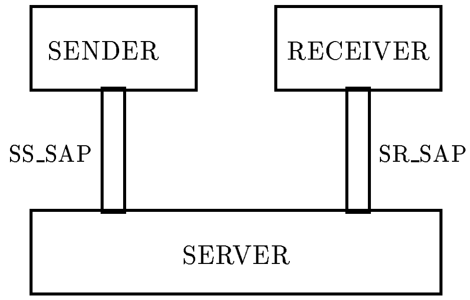


Figure 6.3: Tick-Tock protocol block diagram

From our experience, it is surprisingly easy to specify incorrectly such bounded liveness properties by forgetting the transition labeled α which indicates the deadline has been missed. This omission will only catch error traces where the gate does not go up within K time units *and* does go up or down sometime later. It detects events occurring too late, but does not notice the error if no further events occur.

6.1.2 Tick-Tock protocol

The Tick-Tock protocol [LLD94] has been proposed as a test-bed for evaluating the success of formalisms for specifying real-time systems. The protocol describes three processes: a sender, a receiver, and a service component. The service entity has been modeled as timed automata by Daws et al [DOY94], who verify the component against various properties expressed in TCTL, a real-time temporal logic. Here we show how some, but not all, of the properties they verify can be modeled as timed safety automata. Thus in some cases their timing verification problems can be reduced to timed safety verification as outlined in chapter 3.

System description

The role of the server component is to provide buffered transmission of data from the sender to the receiver, as depicted in figure 6.3. Communication is through data cells passed one at a time through Service Access Points (SAPs). The sender provides cells to the service at the SAP referred to as the SS_SAP. The server then passes them reliably on to the receiver at the SR_SAP. The only way a cell is lost is if the receiver is not ready to receive an offer from the server. Here we do not model the full protocol, where the service may also crash. The behavior of the server satisfies the following timing constraints:

Isochronism: The server offers to accept cells from the sender only at regular instants, π time units apart. At most one cell is received at any time, and the exchange is considered to be instantaneous.

Transmission delays: The server always delays between τ_{min} and τ_{max} time units between receiving a cell at SS_SAP from the sender and then delivering the cell to an internal buffer.

Spacing between deliveries: There must be a time delay of at least α time units between deliveries.

Immediate acceptance: The server offers the receiver a cell at SR_SAP as soon as delivery to its internal buffer is completed. If the receiver does not accept the cell, it is lost.

The description of the server is given by Daws et al as the product of the automata in figure 6.4. Note that the delivery of each cell is meant to take place as soon as it is enabled, modeled by the urgent event in the transmission delay automaton. The service may offer to buffer up to n SS_SAP cells at any given time. This situation is modeled by n different transmission delay components, each with events labeled by a unique identifier. The transmission delays are modeled by the product of all the delay cell components. However this process has events tagged with an identifier i signifying that it comes from the i-th delay cell. As far as the other processes are concerned, it is irrelevant which cell provides the buffering, so the events are abstracted in the delay component before composing them with the other processes, *e.g.* all Del_i events are abstracted to Del[1].

[1]Alternatively, the transitions for delivery and SS_SAP events which occur in other processes could be replicated, one for each delivery or SS_SAP exchange.

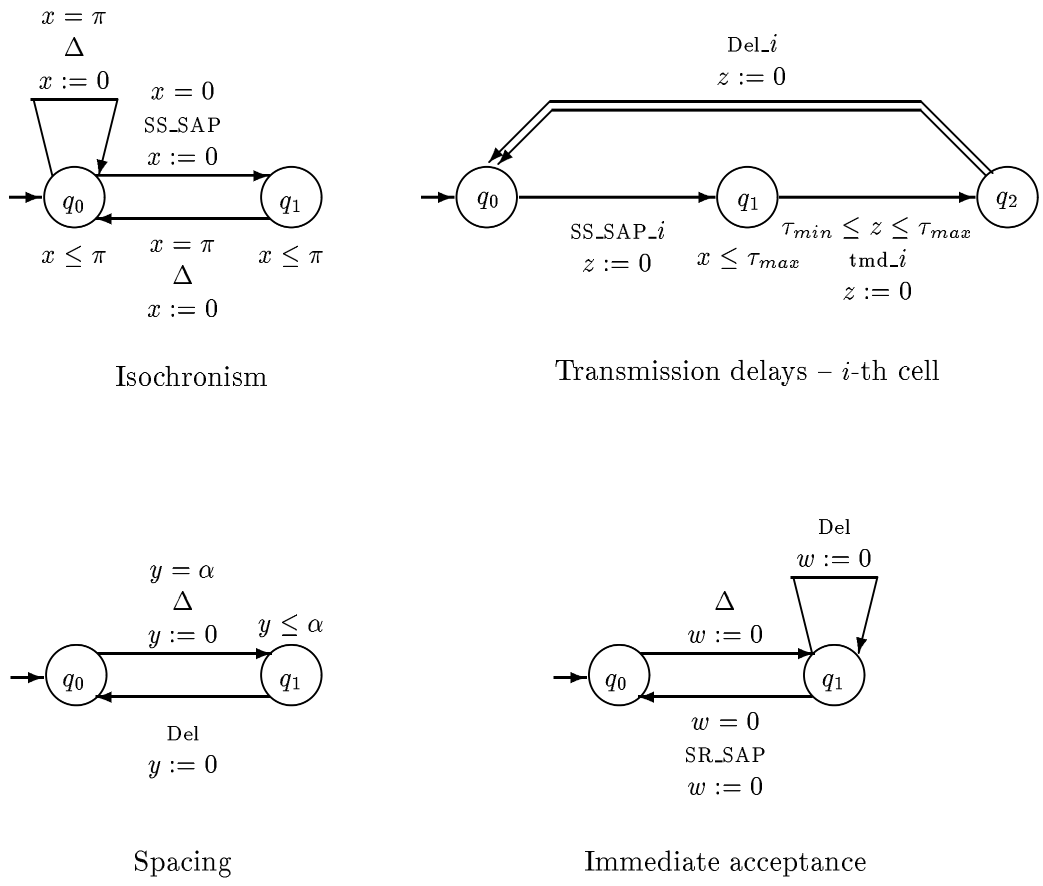


Figure 6.4: Tick-Tock service entity

Specification properties

Each component in the model of the server places a restriction on the server's behavior. However it does not guarantee that the service will be offered in a timely manner. For instance the isochronism requirement states that SS_SAP exchanges may occur at most at regular punctual instances separated by π time units, but in fact the server may not be ready to accept an SS_SAP because transmission may be delayed while waiting for delivery to occur.

Following Daws et al, we verify the server against the following three categories of timing properties.

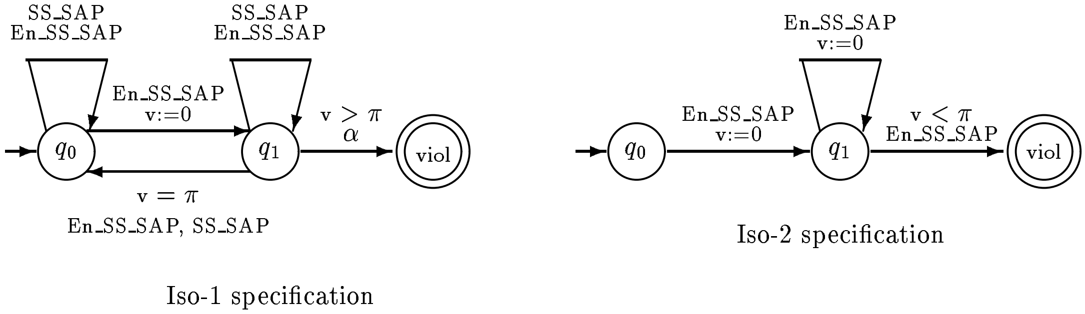


Figure 6.5: Isochronism specification processes

Isochronism:

Iso-1: Whenever an SS_SAP event is enabled, it is also enabled exactly π time units later.

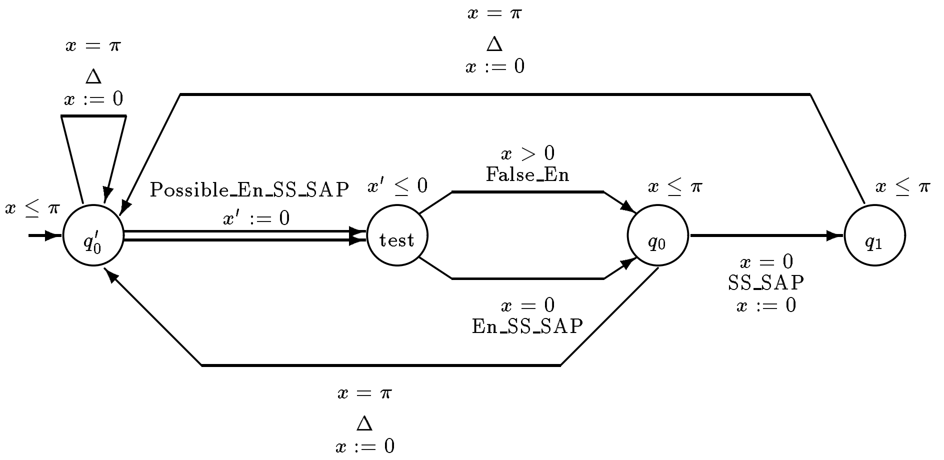
Iso-2: Whenever an SS_SAP event is enabled, it is never enabled again before π time units have passed.

Transmission delays: After a successful SS_SAP exchange, an offer at SR_SAP must occur within $[\tau_{min}, \tau_{max}]$ time units.

Spacing between deliveries: Whenever an event is enabled at SR_SAP, there is a delay of at least α time units before it is enabled again.

The specifications for properties Iso-1 and Iso-2 appear in figure 6.5. The development of these specification automata is explained in more detail below. Notice that the property Iso-1 asserts that a particular event *must* occur within a certain time interval, whereas the second property states that a particular event should *not* occur. In general, properties of the second sort are easier to specify.

Most properties are assertions about whether SAPs are *enabled* in a timely fashion or not. However the language of timed automata has no direct means of expressing that an event is enabled. We handle this by adding additional events, such as En_SS_SAP which is enabled in each component precisely when SS_SAP is ready for communication. In figure 6.4 this would result in self-loops at locations q_0 labeled



Revised isochronism process

Figure 6.6: Isochronism component indicating urgent enabling at SS_SAP

En_SS_SAP in the isochronism process, and En_SS_SAP_i in all the delay cell processes for which the transmission delay product abstracts the events to En_SS_SAP before composing with the other processes. The conditions on the En_SS_SAP events match those for the SS_SAP events. Thus in testing the second isochronism specification, the negated property asserts that a premature enabling event occurs. See figure 6.5.

Verifying the first isochronism property, which asserts that SS_SAP can take place when $v = \pi$, is not so straightforward. The En_SS_SAP event must be urgent in the automaton for the isochronism property. This is because in order to correctly check whether SS_SAP really is enabled, we need the event En_SS_SAP to occur without fail whenever it is. Otherwise SS_SAP may be enabled, with the En_SS_SAP event enabled but not occurring, leaving the impression that time passes by without the event being enabled. However, we run into two difficulties. Firstly, an event may occur at precisely the time En_SS_SAP would occur, thereby disabling En_SS_SAP. We circumvent this through a specification which checks not only for the En_SS_SAP event, but also for events which may explicitly disable it, *e.g.* the event SS_SAP itself may occur instead. Secondly, our verification tool does not allow urgent events to be subject to timed enabling conditions, *e.g.* the constraint $x = 0$ in the isochronism process. We handle this shortcoming by introducing additional control locations which are used to check whether the urgent event satisfies its timing condition, as in figure 6.6. Thus an urgent event Possible_En_SS_SAP is allowed to occur out of location q_0' regardless of the value of the clock x. The 0 upper time bound on reset clock x' at location *test*, forces control to immediately pass to q_0, signaling either that the event truly is enabled (when $x = 0$) or that this excursion into the *test* location does not correspond to a real enabling at SS_SAP (when $x > 0$). Thus to verify the property Iso-1 we replace the isochronism component in figure 6.4 with that of figure 6.6, and add self-loops on the delay cells labeled Possible_En_SS_SAP instead of En_SS_SAP.

Model-checking over TCTL formulae is strictly more expressive than our safety verification paradigm. In particular we cannot even model in our framework the following properties which Daws et al verify:

Isochronism:

Iso-3: An SS_SAP event is never continuously enabled for any non-zero length of time.

Immediate acceptance: An offer at SR_SAP is never continuously enabled for any non-zero length of time, *i.e.* the offer is either taken immediately or lost.

Comparison to Daws et al

Daws et al express information about the enabling of an event by using propositions stating whether the event is enabled within each participating process. The event is enabled in the server iff it is enabled in each participating process. They also explicitly form the product of the individual components, allowing them to express the urgency semantics for the Deliver event using a special clock which ensures that once delivery is enabled it occurs before any time can pass. We prefer to model such events by labeling them as urgent. This decision allows processes to be described in a simple and modular format. The correct semantics is then implemented without an added clock by simply disallowing time to pass whenever urgent events are enabled.

Chapter 8 contains a comparison of the performance of their symbolic verifier KRONOS and our approximation algorithm for those examples we can specify in the reachability framework.

6.1.3 Ethernet

We now briefly describe a more substantial example: a timed model of the Medium Access Control (MAC) sublayer of Ethernet's Data Link layer, first formally specified by Weinberg and Zuck [WZ92]. We refer the reader to their work for a full description of the protocol implemented by this sublayer. It is essentially a carrier-sense / multiple-access protocol with collision-detect (CSMA/CD), which sends and receives frames between the Logical Link Layer and the Physical Layer. A request to send a data packet causes the transmitter to listen to the channel. If the channel is not idle it waits until it is, and then sends its data packet. If collision occurs it is detected, and the transmitter sends a special jam sequence to alert other users. It waits a random time, up to a limit determined by a binary exponential backoff algorithm, and then attempts to retransmit. The logical link layer is informed whether transmission is successful or not.

The MAC sublayer consists of four different components: a frame transmitter, a deference generator, a bit transmitter and a frame receiver. Communication with processes in the Logical Link Layer above and the Physical Layer below occurs through a combination of shared variables and direct communication channels.

Our modeling of the MAC sublayer differs from the description by Weinberg and Zuck in the following ways:

- (data values): we perform no data encapsulation of the raw frames received: in fact no actual data values are sent.

- (bit transmission): the bit transmitter is modeled by signals denoting the beginning and ending of transmission of the entire sequence of bits in a frame. This is essentially the same as saying all frames consist of a single bit.

- (semantics): timed safety automata cannot capture the unbounded liveness properties expressible in the timed transition systems used by Weinberg and Zuck. Our model therefore includes some execution traces not found in theirs.

- (carrier sense/collision detection): the conditions for setting each of these variables is unspecified in [WZ92]. We assume that both the conditions for carrier sense and collision detection may be become true at any time. Furthermore, carrier sense is always true while the sender is transmitting. Whenever the condition for a change of variable value if detected, the variable changes value after an appropriate time delay.

- (retransmission delay): we set a fixed maximal number of periods to delay before attempting retransmission. The actual delay is nondeterministically chosen as any number of delay periods up to the maximum.

Our model includes six variables (number of transmission attempts, carrier sense, collision detection, transmitter waiting, deferred, counter measuring time to wait before retransmit). There are six clocks in the system. The sizes of the individual component processes are given below.

Component	STA states	STA transitions
Frame Transmitter	15	19
Deference Generator	5	5
Carrier Sense Generator	6	25
Collision Detection Generator	3	3
Bit Transmitter	7	9
Medium	3	6
Frame Receiver	3	4

We tested the protocol with three timing specifications: two lower bound properties and a bounded liveness property.

Spec A : If the transmitter is ever deferred before transmitting, then the total time before successful transmission is at least 12 milliseconds.

Spec B : If a jam is sent by the Frame Transmitter, then at least 12 milliseconds pass before successful transmission is signaled to the Logical Link Layer above.

Spec C : If there are no collisions, and the frame transmitter proceeds past the point of waiting to proceed, then transmission be successful within 40 milliseconds.

CSMA/CD

We also test our verifier on a simple CSMA/CD protocol described in [NSY92a]. This verification problem consists of two extremely simplified senders and the medium.

6.1.4 Mutual exclusion

A simple version of Fischer's mutual exclusion algorithm appears at the end of chapter 3. We also test our verifier on Alur and Taubenfeld's fast mutual exclusion algorithm [AT92] which provides a process with quicker access to its critical section in the absence of contention.

6.2 Discussion

While this thesis focusses on efficiency issues in timing verification, we comment briefly now on our experience with specifying verification problems. Although timed automata are an expressive formalism, it is not straightforward to describe systems accurately. We identify three primary sources of problems — the first two of which are generic to the shared-event automaton model.

Limited syntax

Our definition of timed safety automata provides only a basic syntax which is quite inadequate for specifying complex systems. For instance, there is no distinction between input and output events (this may lead to errors when a process is not receptive of its intended inputs, thereby unintentionally blocking the output of another process).

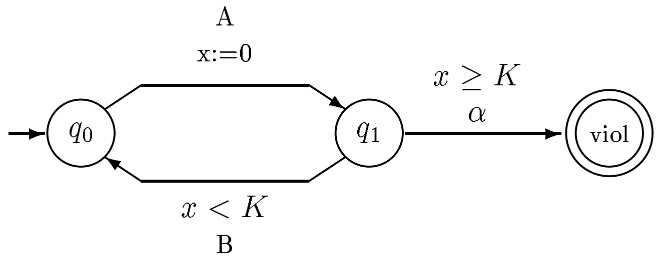


Figure 6.7: Misleading specification

There is no event abstraction mechanism. In the tick-tock protocol it would have been helpful for the timed automaton language to be able to describe how the events for different cells are abstracted into single events.

There is no built-in syntax for pointers, reading and writing variables, nor for indexing arrays. While the lack of the above features is inconvenient, we note that suitable syntactic sugar can be added to the basic model to enrich the formalism.

Machine modeling

While automata models are often convenient for small components, their lack of structure can make more complex processes difficult to understand. For example, looping constructs can have numerous branching and entry points. In the absence of liveness, the basic meaning of a transition is that it *may* occur, as opposed to representing an event which *must* occur. This makes it difficult to express clearly branching points where one of several different choices must be taken. In other words, there are no clear equivalents of while loops, for loops, if statements and case statements. There is also the frequently encountered problem of modeling processes with automata which admit too few runs because events in their composition get blocked. This cause of confusion is due to the shared-events model of composition. A common example is in specifications which are not receptive, *e.g.* the automaton in figure 6.7 does not correctly specify that every A event is followed by a B event within K time units. It disregards runs where two A's followed by a B occur in quick succession.

Forced events

A common misunderstanding is that transitions do not explicitly represent events which must occur. This confusion can lead to automaton models which permit processes to prematurely cease useful progress by resting in a location. For example, the safety invariant "$x < 5$" must be placed on all the locations q_1, q_2, and q_3 in the train component of figure 6.1. It is easy to overlook the invariant on q_2 or even q_1, but the invariant on q_3 is not enough to ensure the automaton loops back to q_0. It merely states that if control reaches q_3, then it will leave q_3 in due time — the automaton may end up in q_1 forever.

Summary

The above shortcomings suggest the need for a higher level language which enables direct reference to variables, arrays, pointers, event abstraction, input/output events, and clear constructs for looping and branching. These are primarily syntactic desiderata. On the other hand, timed safety automata are slightly limited in expressiveness too. As shown above, there are properties they cannot express, such as singularity of enabledness, and unbounded fairness constraints which would be helpful in specifying properties of the mutual exclusion algorithms.

Nevertheless we feel the array of problems we can specify to be quite large in practice, and the use of a verification tool which supplies useful debugging information is very helpful in getting system descriptions correct. We found it critical to test not only that a protocol is correct, but also that suitable changes in the timing parameters result in error traces — this strategy helps ensure the report of correctness is not merely due to modeling faults which incorrectly rule out violating traces.

Chapter 7

Hybrid Systems

Introduction

Hybrid automata [MP93, ACHH93, AHH93, NOSY93] consist of discrete state components interacting with continuously changing variables. They model the behavior of programs embedded in physical systems where the environment is changing in real-time. In the more general case, continuous variables are modeled by arbitrary differential equations, and the system's control information by discrete states. An important class of hybrid automata is that of the *linear hybrid systems*, where the continuous variables are modeled as functions whose rates of changes and reassignments are linear terms. Arbitrary linear hybrid systems are undecidable, but a number of interesting subclasses have been found which are decidable [PV94, MV94, KPSY93], or admit semi-decision procedures [OSY94].

We introduce an interesting decidable subclass of linear hybrid systems, the *skewed clock automata* (SCA), which we use to model processes whose clocks increase at variable rates. These are a subclass of the automata with *rectangular differential inclusions*, which were recently independently shown to be decidable [PV94, HPV94]. An automaton with rectangular differential inclusions has lower and upper bounds on its clock rates, which must be fixed rational numbers. Skewed clock automata add a syntactic restriction on where constraints can be placed in the automaton, the *query-reset alternation property*. The subclass is interesting in that the proof of decidability reduces the emptiness problem for SCAs to emptiness over TSAs while preserving the structure of the automaton and the number of clocks. This reduction technique provides a feasible algorithmic method for safety verification of SCAs. Henzinger et al's reduction [HPV94] applies to a much broader class of automata but doubles the number of clocks, thereby reducing its usefulness in practice.

The syntactic restriction we apply is easily checked. It is general enough to be applied to many forms of automata where clocks are used to force lower and upper bound constraints on enablement times. The alternation property essentially asserts that on every path in the automaton, for every clock, there is either a reset or a test of equality of that clock between any two queries of the clock's value (except that an upper bound query may follow another upper bound query without an intermediate reset, provided the second upper bound is no greater than the first).

We also describe a case study of a timing-based communication protocol due to Bosscher et al [BPV94]. We verify correctness for arbitrarily length bit sequences. We are also able to prove messages are received within a reasonable time, despite the fact that the statement of this timing property uses arbitrarily large constants for deadlines.

Related work

SCAs are a subclass of the automata considered by Olivero et al [OSY94]. They give abstraction mappings which preserve emptiness in only one direction for ∀TCTL formulas, and thus lead to a semi-decision procedure for their more general class of automata. Our transformation from SCAs to TSAs is the same as theirs: in our case we prove it exactly preserves the divergent runs in our automata and therefore yields a decision procedure for emptiness. Puri and Varaiya [PV94] prove decidability for a class of linear hybrid automata incomparable to SCAs. They are not restricted by the query-reset alternation property we require. Their result is very general, except that their enabling constraints and rate intervals must correspond to closed intervals, whereas we allow open intervals. Unfortunately their proof of decidability relies on discretization of the continuous space, and does not lend itself to efficient verification procedures. Recently, in work with Henzinger [HPV94], they have provided a proof of decidability that translates automata with rectangular differential inclusions directly into timed automata, but with a doubling in the number of clocks. These other approaches use a more general model of a hybrid system which allows for different bounds on clock rates at different locations, and reassignment of variables to constants other than 0. These extensions could be incorporated into SCAs but for simplicity are not included here.

Lam and Brayton [LB93] define automata with a very similar query-reset alternation property. Their property is even more restrictive than ours in that each clock may only be reset and queried once in the entire automaton. However, they allow arbitrary timing constraints. Their clocks increase at a constant rate, and the query-reset alternation is used to establish a *simple path property*, which reduces verification to reachability over paths without loops. Our automata do not necessarily satisfy the simple path property. In comparison, we use the alternation property to show that constraints on a drifting clock can be mimicked by constraints on a clock advancing at a fixed rate.

Another approach to verifying hybrid systems, one not pursued in this chapter, is to apply the approximation algorithm directly to hybrid systems. In the general case, the algorithm is not guaranteed to terminate, but the strategy is promising. Indeed Henzinger and Ho [HH94] report successful use of applying our iterated overapproximations of subsection 2.2.2 to linear hybrid systems. They also use extrapolations to speed convergence.

7.1 Skewed clock automata

A skewed clock automaton (SCA) A is a tuple $\langle \Sigma, Q, Q_{init}, C, \rho, T, Inv \rangle$ where

1. Σ is a finite set of events, disjoint from Δ_T,
2. Q is a finite set of control locations,
3. $Q_{init} \subseteq Q$ is a set of initial locations,
4. $C = \{x_1, \ldots, x_n\}$ is a finite set of clocks,

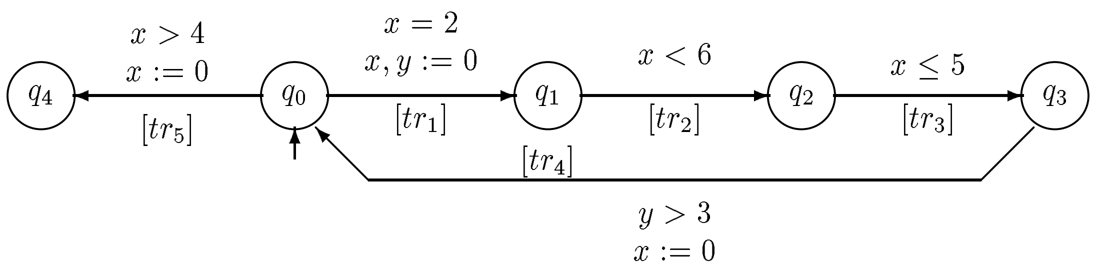


Figure 7.1: Skewed clock automaton A_1

5. ρ assigns to each clock a non-empty interval of $\mathbb{R}$ defined by positive integer endpoints. The interval $\rho(x)$ represents the range of possible rates of increase of x, and will be denoted $[dl_x, du_x]$ where dl_x and du_x are taken to be bounds in the domain $\mathbf{Z} \cup \mathbf{Z}^- \cup \{\infty\}$,

6. $T \subseteq Q \times \Sigma \times \mathcal{E}n \times \mathcal{A}(n) \times Q$ is a *query-reset alternating* transition relation, defined below, and,

7. $Inv \in (Q \rightarrow IZ)$.

We assume without loss of generality that each clock constraint is satisfiable.

For convenience we say that all clocks are reset at the initial state of any run. Before describing the query-reset alternating property, we first define the value of a clock x to be *determined* by a transition tr whenever its value on entering the successor location is uniquely determined by the enabling constraint of tr, *i.e.* x is determined by $tr = (q, \sigma, \phi, a, q')$ iff ϕ implies $x = k$ for some k. We assume without loss of generality that whenever a transition determines a clock's value, it also resets that clock. We also assume without loss of generality that the safety invariant on control location q is a conjunct in the enabling condition of every transition out of q. We now define some notation relating to queries and resets along paths. Let $\tilde{l} = l_0, l_1, \ldots, l_m$ be a path of locations and $\tilde{tr} = tr_1, tr_2, \ldots, tr_m$ a sequence of transitions such that tr_i leads from location l_i to location l_{i+1}. We define ϕ_i to be the enabling constraint associated with transition tr_i. For a given clock x, let R^x_i denote the index position of the i-th reset of clock x along the sequence of transitions, with R^x_0 set to 0. In addition, let $Q^x_{i,j}$ be the index of the j-th query of clock x along $\tilde{tr}$ occurring after the i-th reset but not after the $(i+1)$-th reset, and let $numq^x_i$ be the number of queries of x between the i-th and $(i+1)$-th resets.

Example 7.1 *Consider, for example, the SCA in figure 7.1 and its path of locations* $q_0, q_1, q_2, q_3, q_0, q_4$, *and sequence of transitions* $tr_1, tr_2, tr_3, tr_4, tr_5$. *Then* $R^x_0 = 0$, $R^x_1 = 1$, $R^x_2 = 4$, *and* $R^x_3 = 5$. *The value of* $numq^x_0$ *is* 1 *with* $Q^x_{0,1} = 1$, *and* $numq^x_1 = 2$ *with* $Q^x_{1,1} = 2$ *and* $Q^x_{1,2} = 3$, *and* $numq^x_2 = 1$ *with* $Q^x_{2,1} = 5$. □

The *query-reset alternating* property states that for every path $\tilde{l}$ of locations in the SCA A and matching sequence of transitions $\tilde{tr}$, for every clock x and $i \geq 0$, either $numq^x_i = 1$, whenever it is defined, or the last query between the i-th and $(i+1)$-th resets, *i.e.* the constraint ϕ associated with transition $tr_{Q^x_{i,numq^x_i}}$, includes an upper bound constraint of the form $x \leq b$, and for each $k < numq^x_i$, the query for the $Q^x_{i,j}$'th transition is an upper bound of form $x \leq b'$ where $b \leq b'$. Notice that in the case of multiple queries between resets the last query need not be a simple upper bound constraint: it may be of arbitrary form as long as it implies a suitable upper bound on x.

Example 7.2 *The path and sequence of transitions in example 7.1 is a query-reset alternating path. Notice that whenever a transition has an enabling constraint and a reset, the query of the enabling constraint is considered to take place before the reset. The path has alternating queries between resets, except for the two consecutive queries at transitions* tr_2 *and* tr_3. *However these queries are permissible since the first is an upper bound exceeding the second.* □

The semantics of the SCA A are given by the transition system it induces, namely $\langle S_A, S_{0,A}, N_A \rangle$, where S_A and $S_{0,A}$ are as for timed safety automata, and $N_A = N'_\delta \cup \bigcup_{e \in T} N_e$, where N_e is also as before. The time passage relations N'_{δ_t} are defined as

$$N'_{\delta_t} = \{ \langle \langle q, \vec{x} \rangle, \langle q, \vec{x}\,' \rangle \rangle \mid \forall\, i \in 1..n, (x'_i - x_i)/\delta_t \in \rho(x_i) \}$$

and $N'_\delta = \bigcup_{t \in \mathbb{R}} N'_{\delta_t}$.

Theorem 7.3 *SCAs are closed under composition.*

Proof: The syntactic query-reset alternating path restriction is preserved when composing automata, since every path in the product automaton projects to a path in each component, and repeating the same upper bounds (for a safety invariant at the same component location) along a path is permitted. □

7.2 Translation to timed safety automata

We define a transformation function K which converts a skewed clock automaton into a timed safety automaton. Note that the transformation only applies in the case where the SCA resets a clock every time its value is determined. The TSA $K(A)$ has the same control locations and transition structure as A, the only difference being that its timing constraints are transformed to reflect the different clock rates. For each SCA clock x, there is a TSA clock x'. Intuitively, x' records the amount of time which has passed since x was last reset. We assume without loss of generality that all bounds on clock rates are integer values, either strict or non-strict.

For SCA automaton $A = \langle \Sigma, Q, Q_{init}, C, \rho, T, Inv \rangle$, we define $K(A)$ to be the tuple $\langle \Sigma, Q, Q_{init}, C', T', Inv' \rangle$, where C' consists of a set of primed clocks, one corresponding to each clock in C. The transitions T' are the set of transitions $K(T)$, and Inv' are transformed invariants, both defined below via transformations on the timing constraints. The transformed constraints $K(\phi)$ express the fact that the SCA constraint could be satisfied under the TSA constraint. For uniformity of exposition, we use bounds in enabling constraints. We use an extended domain of bounds which includes r and r^- where r is a rational value. Division of bounds is defined as the expected rational division with the result being a strict bound whenever either operand is strict. The only exception to this rule is that a non-strict zero bound divided by any bound is always a nonstrict zero bound.

The transformation for basic enabling conditions of the form $x \sim b$ for a clock x and relation $\sim$ in $\{\leq, \geq\}$ is defined below. The idea is that the linearly progressing TSA clock x' in $K(A)$ records the amount of real global time since the last reset of x. Let t be the amount of time which has passed since x was last reset. For a

ϕ: in SCA A	$K(\phi)$ in TSA $K(A)$
$x \leq b$	$x' \leq b/dl_x$
$x \geq b$	$x' \geq b/du_x$

Figure 7.2: Transformation K on SCA constraints

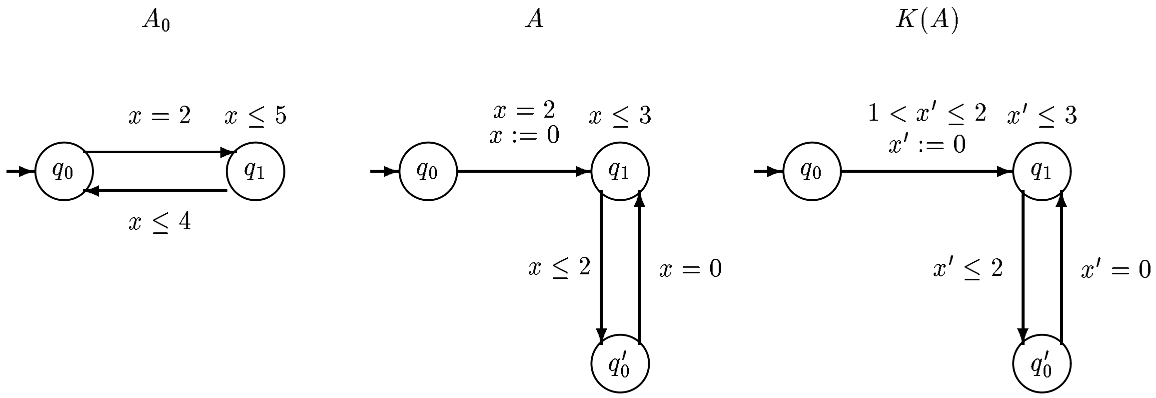


Figure 7.3: Transforming SCAs into TSAs

constraint of the form $x \leq b$ to be satisfied in the SCA, we know that at most "b/dl_x" time has passed from the time of x's last reset, since $x \leq b$ and $t \cdot dl_x \leq x$ implies $t \cdot dl_x \leq b$ which is equivalent to $t \leq b/dl_x$. Because the time since x was last reset is measured by clock x', we replace the constraint $x \leq b$ in the SCA A with $x' \leq b/dl_x$ in the TSA $K(A)$. A similar analysis for lower bounds leads to the translation table for constraints shown in figure 7.2. The transformation extends to conjunctions as $K(\phi_1 \wedge \phi_2) = K(\phi_1) \wedge K(\phi_2)$. We define $K((q, \sigma, \phi, a, q')) = (q, \sigma, K(\phi), a', q')$ where a' resets the primed versions of all clocks reset by a, and $K(T) = \{K(tr) \mid tr \in T\}$. Finally, we define Inv' such that $Inv'(q) = K(Inv(q))$ for every location q.

Example 7.4 *The SCA A_0 in figure 7.3 does not reset x after determining its value along the transition from q_0 to q_1, so we cannot apply the transformation directly* *to A_0. The SCA A accepts the same language as A_0 and avoids this problem. The rate of increase of x lies in the interval $[1, 2)$. We apply the K transformation to A yielding the TSA $K(A)$ shown in the figure. Notice that the transition from q_0' to q_1 has enabling condition $x' = 0$ derived from $0 \leq x' \leq 0$ resulting from the non-strict 0 lower bound obtained by dividing 0 by 2^-.* $\square$

Theorem 7.5 *A skewed clock automaton A has an empty language iff the timed safety automaton $K(A)$ has an empty language.*

Proof: The proof of correctness shows that nonemptiness is preserved, *i.e.* a run in the TSA $K(A)$ implies a run in the SCA A and vice versa.

SCA non-empty implies TSA non-empty

We first prove the simpler direction, namely that a run in the SCA A has a matching run in the TSA $K(A)$. The time in $K(A)$ represents the real time. Given a run in the SCA A,

$$s_0 \xrightarrow{e_1} s_1 \xrightarrow{e_2} s_2 \xrightarrow{e_3} \cdots$$

let $t_i = \Sigma_{j \leq i} dur(e_j)$. We refer to the transition which takes place from state s_i as tr_{i+1}, and let tr'_{i+1} denote $K(tr_{i+1})$. The location of s_i is referred to as q_i. The run

$$s'_0 \xrightarrow{e_1} s'_1 \xrightarrow{e_2} s'_2 \xrightarrow{e_3} \cdots$$

in the TSA is obtained as follows. The control location at s'_i is q_i. Let x'_k be the value of clock x at state s'_k, which we set as $x'_k = x'_{k-1} + t_k - t_{k-1}$ if tr_i does not reset clock x, and $x'_k = 0$ otherwise.

It is easy to see that the timed-states along the run are reset appropriately, and advance correctly for time-passage events. Thus we need only check that all queries in $K(A)$ are satisfied along the run. Consider a query of x' along the run at transition tr_{i+1} out of state s'_i. Suppose the most recent reset of x' occurred at transition tr'_r into state s'_r. Then the value of x' at s'_i is $x'_i = t_i - t_r$, since resets of x' match those of x. If tr'_{i+1} has a constraint of form $x' \leq b'$ in $K(A)$, then tr_{i+1} has a constraint of form $x \leq b$ such that $b/dl_x = b'$. By the lower bound on the rate of progress of x, and the fact that x satisfies its constraint in tr_{i+1} at s_i, we have that $(t_i - t_r) \cdot dl_x \leq x \leq b$, which implies that $(t_i - t_r) \leq b/dl_x = b'$. Since $x'_i = t_i - t_r$, it follows that $x'_i \leq b'$ as required. The argument for lower bounding constraints is similar.

TSA non-empty implies SCA non-empty

We construct a matching run in A for every run in $K(A)$. Analogous to the above argument, we use the values of the clocks in the given run to provide clock values for the constructed run. We then show that the corresponding skewed clocks satisfy their timing constraints because their mapped clocks x' in $K(A)$ do.

Consider a run

$$s'_0 \xrightarrow{e'_1} s'_1 \xrightarrow{e'_2} s'_2 \xrightarrow{e'_3} \cdots$$

in the TSA $K(A)$. Suppose $t'_i = \Sigma_{j \leq i} dur(e'_j)$. We claim that

$$s_0 \xrightarrow{e'_1} s_1 \xrightarrow{e'_2} s_2 \xrightarrow{e'_3} \cdots$$

is a run in A if $s_k = \langle q'_k, \vec{x_k} \rangle$ with control location and transitions matching those of the TSA's run and the values of each $\vec{x_k}$ determined below.

The value of each clock x is assigned independently of the other clocks. Let R^x_i, $numq^x_i$, and $Q^x_{i,j}$ be defined for the run as in the definition of the query-reset alternating path property, *i.e.* R^x_i is the index position of the i-th reset of x, $numq^x_i$ is the number of queries between the i-th and $(i+1)$-th resets, and for $1 \leq j < numq^x_i$, $Q^x_{i,j}$ the index position of the j-th query of x after its i-th reset. Clearly x_k should be assigned the value 0 whenever $k = R^x_i$ for some i. We need to define x_k between resets, *i.e.* for $R^x_i < k \leq R^x_{i+1}$. To do so, we use the last query of x before the $(i+1)$-th reset, *i.e.* the query at the $Q^x_{i,numq^x_i}$-th transition out of the $(Q^x_{i,numq^x_i} - 1)$-th state, to choose a linear rate of progress between the R^x_i-th and R^x_{i+1}-th states. We will then show that for all $1 \leq j < numq^x_i$ the queries at the $(Q^x_{i,j} - 1)$-th states are satisfied.

First we choose an appropriate rate of progress Δ which guarantees the enabling constraint ϕ at the $(Q^x_{i,numq^x_i} - 1)$-th state is satisfied. For notational convenience, we fix $M = (Q^x_{i,numq^x_i} - 1)$, and let the enabling constraint of tr'_{M+1} be ϕ', *i.e.* $K(\phi) = \phi'$. Let the value of x' at s'_M be v.

- If $v \neq 0$ and ϕ' includes an upper bound constraint of the form $x' \leq b'$ in the

SCA $K(A)$ derived from the constraint $x \leq b$ in A, we require Δ to lie in the range $[dl_x, b/v]$

- If $v \neq 0$ and ϕ' includes a lower bound constraint of the form $x' \geq b'$ in the SCA $K(A)$ derived from the constraint $x \geq b$ in A, we require Δ to lie in the range $[b/v, du_x]$

We need to show that there is a $\Delta \in \rho(x)$ satisfying the above constraints. We do this in two steps: first showing each interval above is non-empty, and then showing they must overlap. For non-emptiness, the case where $v = 0$ is obvious, so suppose $v > 0$. Consider the restriction implied by an upper bound constraint. Since the value of x' at s'_M satisfies the constraint ϕ', it follows that $x'_M = v \leq b' = b/dl_x$. Hence $dl_x \leq b/v$ since both v and dl_x are non-negative. For lower bound constraints we have that $b/du_x = b' \leq x'_M = v$, and hence $b/v \leq du_x$.

To see that the intervals overlap, first observe that ϕ is satisfiable by assumption on the structure of SCAs . Therefore when it contains constraints $b_1 \leq x$ and $x \leq b_2$ it must be that $b_1 \preceq b_2$ with $b_1 \neq b_2$ unless both represent non-strict bounds, and hence $b_1/v \preceq b_2/v$. Because $dl_x \preceq du_x$, all interval restrictions of form $[dl_x, b_2/v]$ and $[b_1/v, du_x]$ overlap as required. Thus for each i, we may fix a rate Δ_i within the prescribed ranges.

We are now ready to give the explicit values of the clock variable x over the intervals between resets, namely, for all i we set $x_k = \Delta_i.(t_k - t_{R_i^x})$ for all $R_i^x \leq k < R_{i+1}^x$.

We need to show that all queries are satisfied. Consider a query ϕ at state s_k. Then $k = Q_{i,j}^x - 1$ for some i and j. We examine two cases.

Case 1: $j = numq_i^x$.

Then the query is the last before the $(i+1)$-th reset. Let the time elapsed since that reset be $v = t_k - t_{R_i^x}$. The value Δ_i has been chosen so that for every upper bound constraint $x \leq b_2$ in ϕ, $\Delta_i \in [dl_x, b_2/v]$ if $v > 0$ and $[dl_x, du_x]$ otherwise. In either case, $x_k = \Delta_i \cdot (t_k - t_{R_i^x}) \leq b_2$.

The argument for lower bound constraints is similar.

Case 2: $j < numq_i^x$.

By the query-reset alternating property, ϕ's constraint on the clock x must be of the form $x \leq b$ where the $numq_i^x$-th constraint ϕ_2 after the i-th reset has a constraint of form $x \leq b_2$ for some $b_2 \leq b$. Since, by case 1 above, ϕ_2 is satisfied, we know that $x_k \leq x_M \leq b_2 \leq b$: in other words, since the value of x at this later query does not exceed b_2, its no greater value at s_k cannot exceed the higher bound b.

Thus all timing constraints are satisfied and $K(A)$ non-empty implies A non-empty. □

Notice that the above result also holds for SCAs augmented with urgent transitions. Such transitions can be encoded using an auxiliary clock x being reset on entry into every location where any urgent events are enabled, and having invariant $x \leq 0$ at all such locations.

7.3 Case study: Manchester bit encoding

We describe how a timing-based communication protocol using Manchester encoding [BPV94] can be verified using skewed clock automata. The protocol forms a small part of a real audio control protocol under development by Philips. Bits are encoded based on timing delays between signals, and the rates of both the sender's and receiver's clocks vary within a given tolerance. The algorithm is due to Bosscher et al [BPV94] who model the protocol using a general model of linear hybrid systems, and verify its correctness using simulation-based proof rules. They also provide a counterexample when the timing constraints are not appropriately met. We model the protocol with skewed clock automata, and specify its correctness by adding violation states which should not be reachable. It is then manually converted to a timed safety automaton representation, and then automatically verified using our approximation algorithm. We verify two properties: correctness of the bit stream that is received, and timeliness of the output.

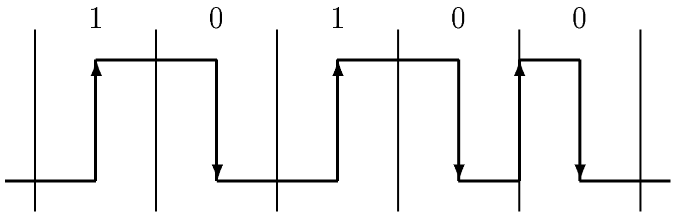


Figure 7.4: Timing diagram for Manchester encoding of 10100

7.3.1 Protocol description

Bit streams are communicated using Manchester encoding. See figure 7.4 for the encoding of 10100. The voltage on the communication bus is either high or low. A 0 bit is sent as a *down* signal from high voltage to low, and a 1 bit as an *up* signal from low to high. The time line is divided into equal length time slots, and the signals are sent in the middle of each time slot. In order to send a repeated bit, there must be an intermediate change in voltage, and this occurs at the edge of the time slot as shown in the diagram for the last two bits.

Bosscher et al [BPV94] report a number of complications in the algorithm used by Philips, partly due to the fact that there is a ±5% tolerance in the clock rates of the sending and receiving components.

1. The receiver does not know when the first time slot begins, although it does know the agreed upon width of the slots. The sender and receiver synchronize the start of transmission by requiring a low voltage whenever no bits are being sent, and starting all bit streams with a 1.

2. The receiver is not explicitly told the length of the message being sent. It must infer the bit stream is complete after a suitable lapse in receiving bits.

3. Drops in voltage are not instantaneous, and cannot be reliably detected. Therefore, the receiver must decode the message based solely on upgoing signals. Because the downgoing edge of a final 0 bit is not seen, this would create ambiguity between messages ending in 10 and in 1. This problem is solved by restricting

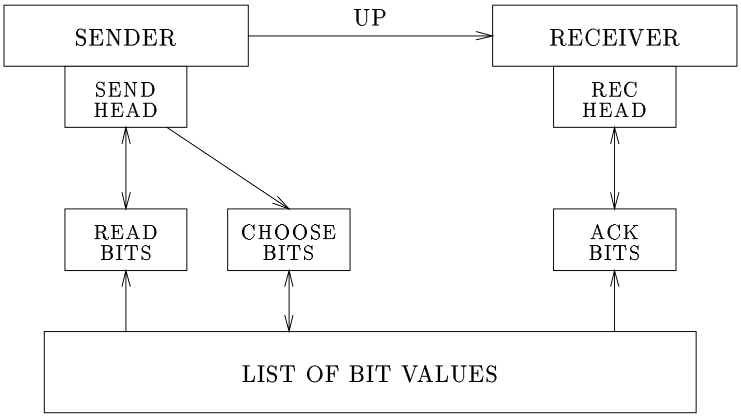


Figure 7.5: Overview of processes

bit streams to be either odd in length, or ending in 00.

4. Message collisions may occur due to several components sending at the same time.

5. There may be significant delays in communication over the bus.

The algorithm considered here ignores the last two difficulties, *i.e.* we assume there is a single sender and a single receiver and each upgoing signal is seen instantaneously by the receiver. We verify for arbitrary length bit streams that the receiver correctly receives all bits, and realizes the bit stream has finished in a timely fashion.

The sender and receiver have the same clock error tolerance of $\pm T\%$. The receiver interprets the *up* signals by rounding the times they are received to the nearest time it expects them to be sent, *i.e.* to the slot edges or to the middle of a time slot, whichever is closer. We use the constant Q to denote 1/4 the length of the bit slot.

The protocol is modeled using two primary components, the sender and the receiver, and a number of auxiliary processes for the stream of bits, pointers into the stream, and processes coordinating the reading of bit values and generation of the nondeterministic bit sequences. The overall structure of the system is shown in figure 7.5.

7.3.2 Modeling arbitrary length bit streams

We briefly indicate how we model arbitrary length bit streams. SCAs have only finite control structure, so they cannot store the value of an *arbitrarily long* input bit stream, model the message being sent and received, and then compare the received message with the input.

Generating bit sequences

Instead, we generate the bits to be sent on-the-fly, and compare each received bit with its intended value as it is received. Each correctly received bit may then be discarded. This is modeled by the reuse of bit values. The protocol is such that we need only store a small number of the most recent bits: this is because the receiver can never get too far "behind" the sender in acknowledging bits sent.

We store the most recent bit values as a list of separate processes, one per bit. Bit values are either 0, 1, or null. A null bit indicates the end of the list. Both the sender and receiver maintain pointers into the list. Each time the sender reads another bit to send it advances its pointer into the bit list and, if necessary, the next bit value(s) to send is also nondeterministically chosen. Care is taken to ensure that the resulting bit stream is legal, *i.e.* bit streams are either odd in length or end in 00. Whenever termination is chosen, the values 0, 0, and null are selected for the next three bits if an even number of bits have already been sent, and the value null is selected otherwise. Since the receiver is sometimes two bits "behind" the sender, we need to store the last 5 bit values. However for simplicity of description, we choose to model an even number of bits in order to maintain the parity of bits, and hence store 6 bit values. In addition the first bit is treated separately since it must always be 1.

Verifying timing properties

Bosscher et al prove the timing property that the bit stream is output by the receiver within $(4m+5)Q/(1-T)$ time units, where m is the length of the message. We are also able to automatically verify this property, despite the fact that it appears to be described by a timing constraint on a clock that must increase without bound.

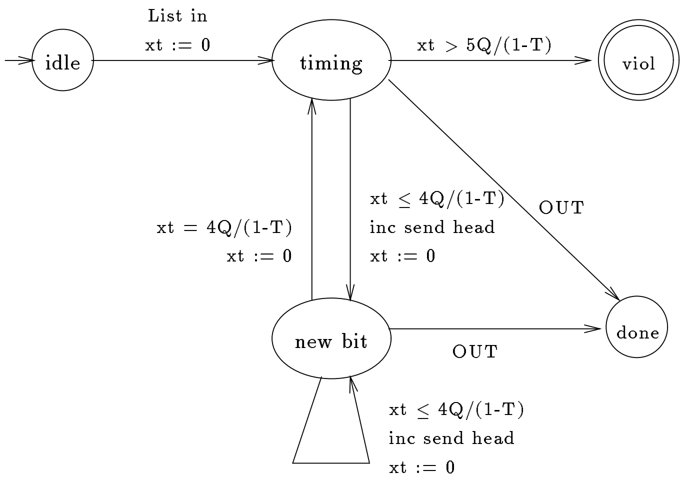


Figure 7.6: Timing specification

We achieve this by proving a stronger property, namely that whenever the sender reads a bit to send, either the next bit to be sent is read within $4Q/(1-T)$ time units, or the bit stream is output within $5Q/(1-T)$. This property implies that the output takes place within the desired time, since the deadline for output is delayed by at most $4Q/(1-T)$ time units for every bit sent. This localized property can be encoded in the timed automaton of figure 7.6 using fixed time bounds, and therefore used as input to our verifier.

7.3.3 Sender

Figure 7.7 shows the SCA for the sending process. The sender starts execution as soon as it receives the list input signal. During transmission it looks ahead at the next bit value, and decides whether it needs to perform an intermediate transition before sending the bit signal in the middle of the time slot. Thus the sender must keep track of the current voltage value. After transmitting each signal, the sender immediately increments *Send_Head*, its pointer into the bit stream, reads the next bit to send, if any, and decides how long to wait until its next signal. Timing constraints are correctly maintained by the skewed clock x which is reset each time there is a voltage change. It is relatively straightforward to see that the query-reset alternating

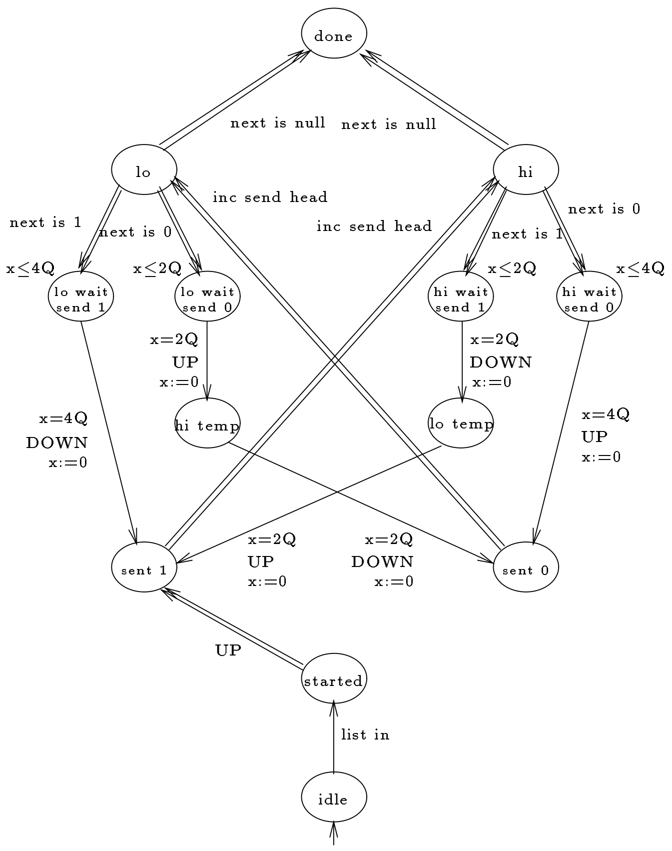


Figure 7.7: Sender

property holds. The processes for reading and generating the bit sequences appear in figures 7.8 and 7.9, and are explained in more detail below.

Reading pointer values

Our automata have no explicit means of managing pointers. Thus to determine whether the "next" bit has value 0, 1 or null, we cannot refer directly to the bit pointed to by *Send_Head*. We model this by enumerating the possible values of the head pointer and the bit values and creating separate events for each combination. However, listing the result of each possible combination in the sender process would clutter its description. So for clarity we choose instead to use a separate process

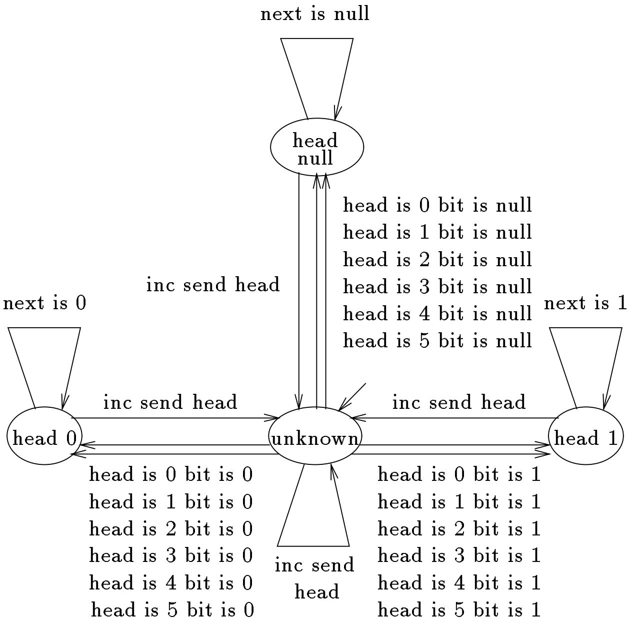
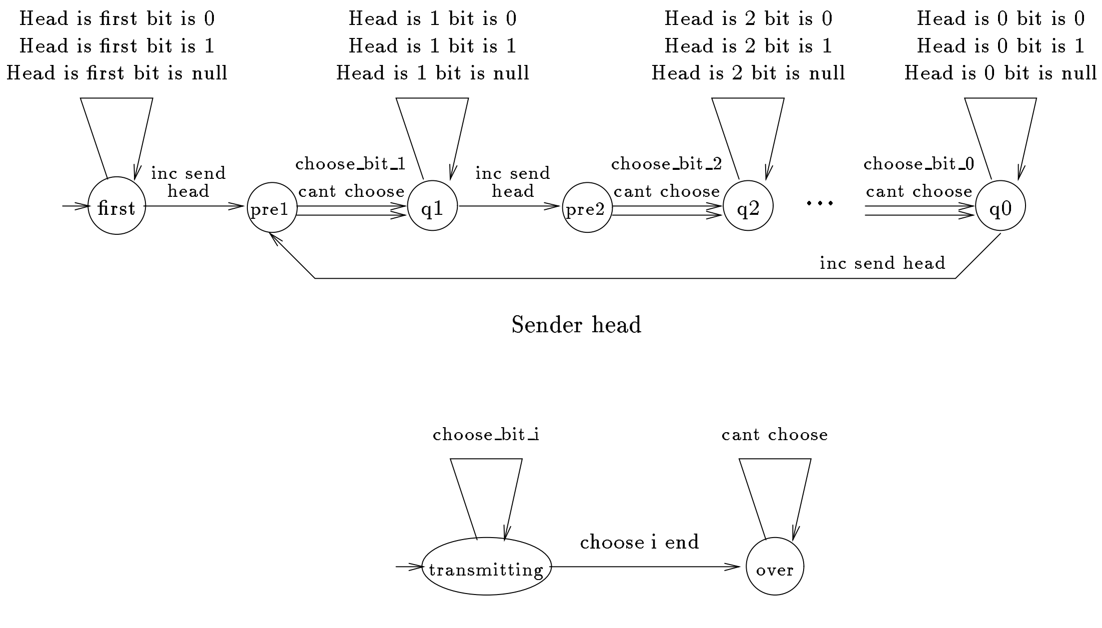


Figure 7.8: Process monitoring reading head of bit stream

Read_Bits, depicted in figure 7.8, which keeps track of the current value of the next bit. From its unknown state, it immediately determines the value of the next bit, as described above, and enters either *head_0*, *head_1*, or *head_null*. From these locations, it can freely allow the sender to read the head value. Because we want the process to be ready for the sender to read the next bit value at any time, we force control to leave the *unknown* location as soon as possible by making its outgoing transitions urgent[1]. Whenever an event occurs which may alter the value pointed to by the *Send_Head*, the *Read_Bits* process reenters its unknown location. We note that an extended syntax for timed automata would eliminate the need for the *Read_Bits* process, *e.g.* allowing abstraction of events, or pointer values. In any case, the description we give matches exactly the input for the current implementation of our verifier.

[1]It is straightforward to augment the definition of SCAs and the K-transformation with urgent events, while maintaining reducibility to TSAs.



Sender head

Process monitoring termination

Figure 7.9: Processes for generating and reading bits

Generating the bit sequence

As mentioned above, bit values are dynamically chosen each time the sender increments its pointer into the list of bits. The bit sequence may increase in length by choosing a 0 or 1 value. Alternatively, it may nondeterministically choose to terminate. However we must be careful to ensure that only valid bit sequences are generated. If an odd number of bits has been sent already, the next bit takes value null. If an even number of bits have been sent, we append 00 to the bit sequence, and so the next 3 bits are affected, taking values 0, 0 and null, respectively. In this way, all valid bit sequences may be nondeterministically generated. In addition, the system uses a process monitoring termination which keeps track of whether the list has terminated. This is necessary in order to ensure the trailing 0, 0, null sequence of bits for an even length sequence are not mistakenly overwritten with new values.

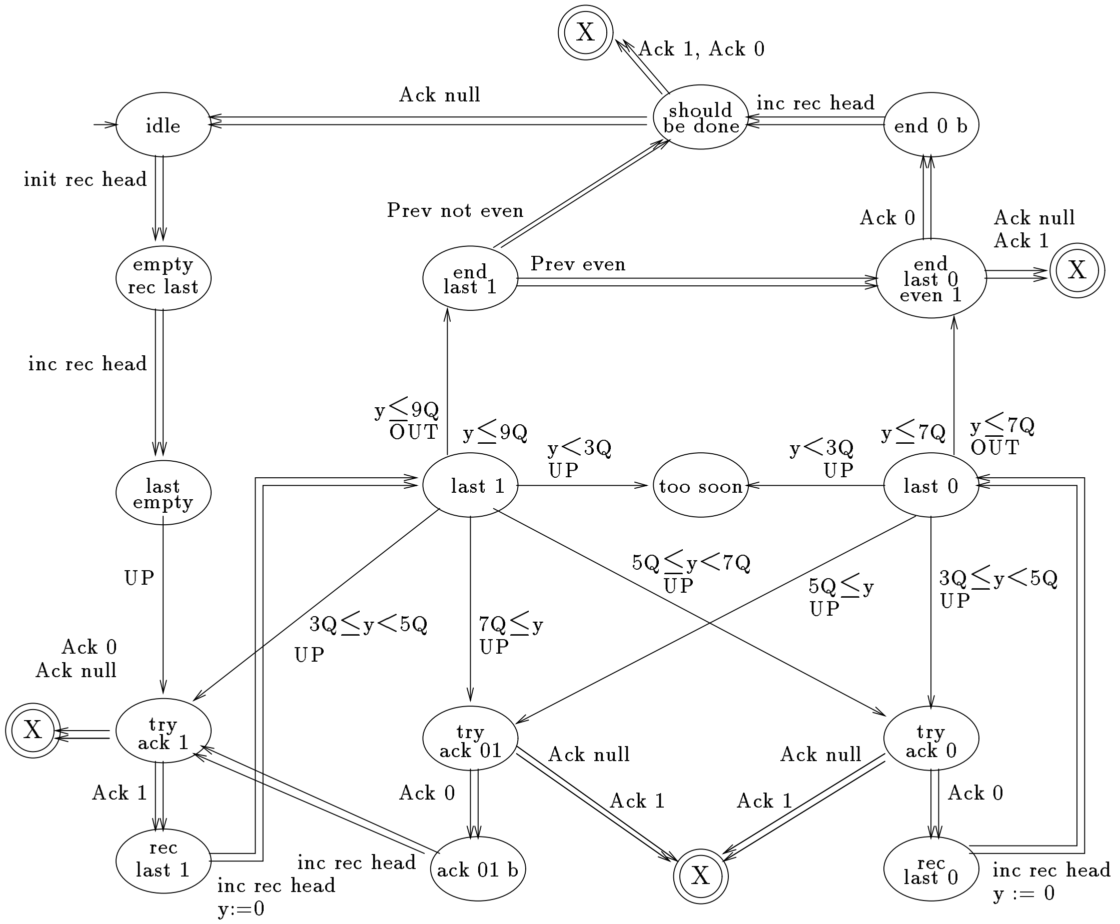


Figure 7.10: Receiver

7.3.4 Receiver

The process modeling the receiver appears in figure 7.10. Depending on the last bit received, and the delay between the upgoing signals it detects, it infers which bit values are being sent. The receiver is in two basic modes, depending on the last bit received. For each mode, there is a waiting location (*last_0* and *last_1*), where it passively rests until it detects an *up* signal. The process then decides which bits to "add" to its bit stream. After adding the bits, it uses urgent events to return to the appropriate waiting location. The list is output, if, however, an *up* signal does not appear within a reasonable time, *i.e.* within $7Q$ time units after the last signal if the

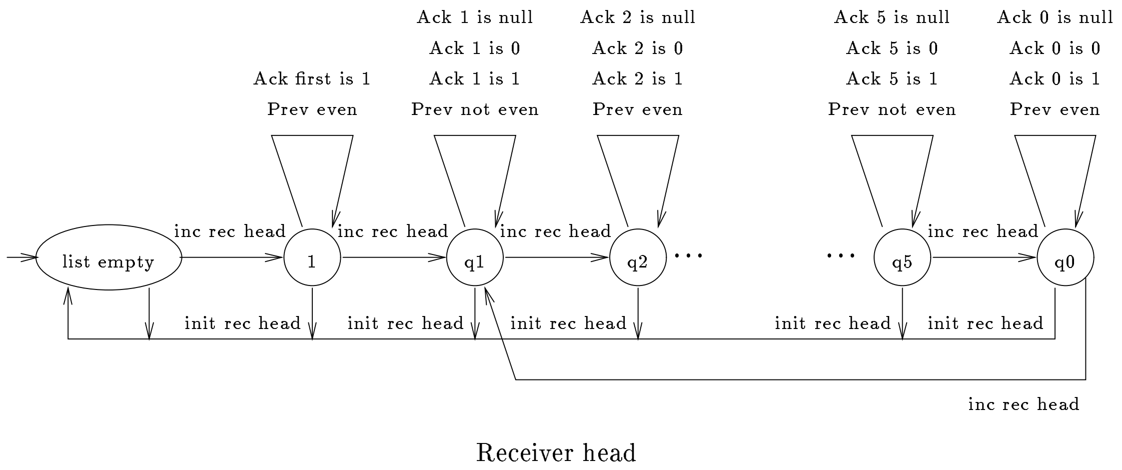


Figure 7.11: Receiver head of list pointer

signal indicating the last bit was 0, and within $9Q$ if the last signal caused the last bit added to be 1. At this point, a trailing 0 bit may be added to the list, depending on the value of the last bit received, and whether the sequence received so far is odd or even in length.

The addition of violation states, marked as nodes labeled with an X, to the receiving process enables it to serve as the specification for the correct reception of all bits. Whenever this process does not correctly receive bits, or notice the end of the bit sequence, a violation is flagged.

The details of how the monitoring works is similar to the modeling of the sender. The variable *rec_head* indexes the stream of bits, pointing to the next bit which should be received. When the receiver decides to add bits to the sequence it receives, the process actually attempts to acknowledge that these are the correct bits in the chosen bit stream. If it cannot acknowledge the correct bits, it enters the failure location. Again urgent events are used to ensure the tests for acknowledgement all happen without time passing, and control returns to one of the waiting locations.

For completeness, figures 7.11, 7.12, and 7.13 show the automata for the remaining processes: the receiver's pointer into the list of bits, the individual bits with their response to terminating the bit sequence, and the acknowledgement mechanism for

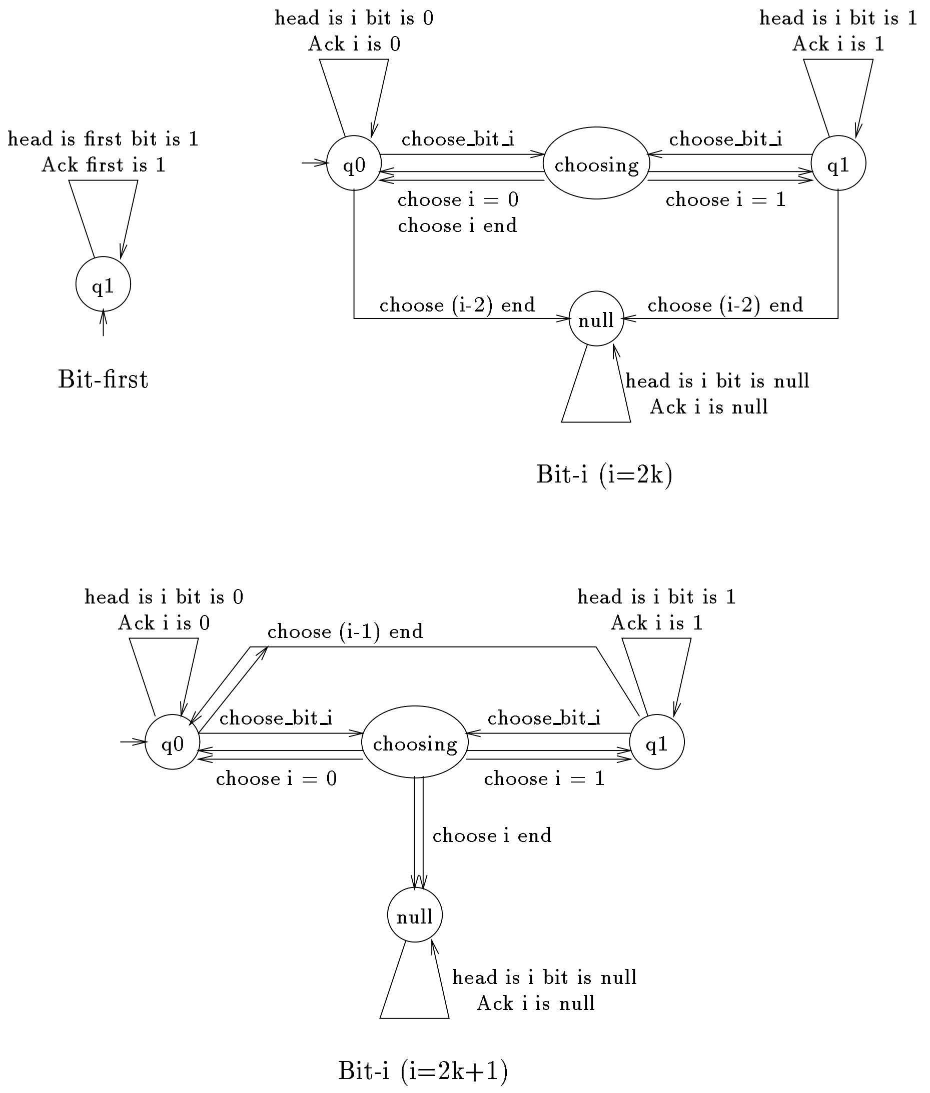


Figure 7.12: Bit processes

abstracting events for the acknowledgement of bits depending on the value of the receiver's pointer.

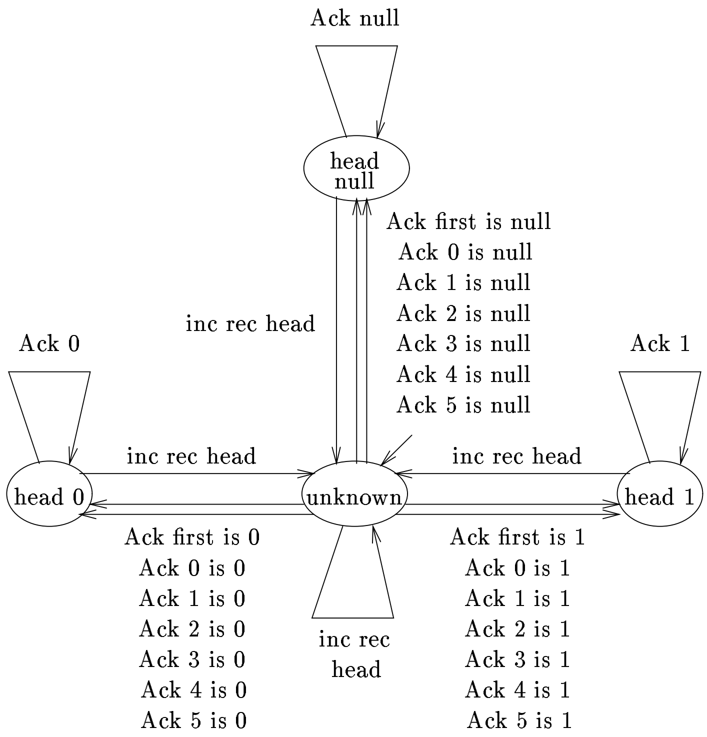


Figure 7.13: Process coordinating acknowledgements

Chapter 8

Implementation and Results

The approximation algorithm for verifying real-time systems has been implemented and tested on several examples. For the more challenging verification problems, it outperforms other symbolic verification algorithms we have implemented, as well as KRONOS, a symbolic model-checker developed elsewhere.

8.1 Implementation

Two forms of the algorithm – approximating only timing information, and approximating over both timing information and the control locations – have been implemented. Time zones are represented by DBMs, and sets of locations by ordered binary decision diagrams (OBDDs). Unless otherwise stated, the following discussion applies to the algorithm where control information is also represented symbolically.

The verification problem input is first preprocessed, and then relevant system parameters, such as the number of clocks in the system, are used in the compilation into executable code. In theory, the algorithm always terminates, but in practice it is of course limited by both time, and more importantly, space. If the program has the resources to terminate successfully, it indicates whether the system is correct or not. If the system contains a violation, a violating pseudo-trace is generated.

8.1.1 Input

The input consists of an event-based modular description of a system and its specification. It also provides the user a simple means of specifying the initial separating structure.

Problem description

Each system component is a timed safety automaton. A global automaton for the system is the composition of automata for each component. Each component automaton is described by its set of locations, event alphabet, initial location, and a listing of transitions. Components synchronize their actions through shared events. Associated with each component is an alphabet of event symbols, and an event can occur provided it is enabled in every component automaton whose alphabet includes the event.

The specification is also given as a timed safety automaton, and is included in the input as a special component. Its violating locations are labeled.

Initial partitioning

The user may specify the initial separating structure. It must be given as a partition of the timed state-space which is determined by each process's control locations. The user partitions the control locations within each component process, thereby partitioning the state-space such that two timed-states are in the same separating class precisely when their control locations are in the same block of the partitioning for every process component. In other words, given blocks $\{X^i_j\}$ as a partition of the control locations Q^i of process i, $\langle \vec{q}, \vec{x} \rangle$ is in the same separating class as $\langle \vec{q}', \vec{x}' \rangle$ if and only if for all i both $(\vec{q})_i$ and $(\vec{q}')_i$ are in the same X^i_j. We require that the partitioning respect V, and hence the specification process must have its control locations partitioned with all blocks either containing only violating locations, or containing no violating locations.

8.1.2 Implementational variations

We explain here how the algorithm implemented departs from the algorithm described in chapter 5. The reader may safely choose to skip this section without losing any understanding of the rest of the chapter.

The algorithm has been modified for the implementation in the following ways, none of which affect the correctness nor termination properties:

1. separating structures are refined on-the-fly: a separating structure may be refined during a traversal, not just after a traversal is complete. This change is the most significant variation from the algorithm described in the previous chapters. The idea is that if it can be detected that a class should be split at some later point, such as after the traversal, it might as well be split in the middle of the traversal. This anticipated split refines the approximation right away rather than waiting until the next set of traversals. However, to avoid excessive splitting in the middle of a traversal, a limit is imposed on the number of times a block may be predictively split in this way.

   In our implementation, such predictive splits are designed to decrease the overapproximation. As before, classes are split according to their control locations. A class may be split if doing so enables a finer approximation which avoids some states in the reverse direction's underapproximation which would otherwise have been included. The particular rule implemented attempts to split a class so that the time-passage events are more accurately approximated. The reason for this is that the next-state relations are only approximate over time-passage events, so a great deal of inaccuracy can potentially be introduced in the computation of time successors. The heuristic we use anticipates such problems and allows a more accurate calculation of time successors when it appears the approximation is too crude. More specifically, if the approximating set A is disjoint from the reverse direction's underapproximation Opp_U, and $\overline{N}_\delta(A)$ is not, the set A may be split by control locations into A_1 and A_2 if either or both of $\overline{N}_\delta(A_1)$ and $\overline{N}_\delta(A_2)$ is disjoint from Opp_U. This split is brought into effect by splitting the separating class containing A in the appropriate manner.

2. Identical classes may be repeated in a separating structure. Theoretically, repeated sets may not occur in a separating structure: the definition of a separating structure requires that each class be distinct. However we choose to allow repeated sets rather than performing a potentially expensive check to remove all redundant sets.

3. Disjuncts are combined: transitions which share the same timing information are combined into a single disjunct in the next-state relation regardless of the symbol they are labeled by. This strategy allows a more compact system description, and avoids repeatedly analyzing the same timing conditions. We found this strategy to be essential when analyzing larger systems, since many disjuncts do indeed share the same timing constraints. In some cases, systems with over 400 transitions are reducible to only 20 distinct disjuncts.

4. Splitting occurs over non-maximal sets as well as maximal sets. The algorithm description requires only that maximal sets be split between different traversals of the algorithm. However, we avoid the check for maximality and split also non-maximal overapproximating sets according to locations appearing in the underapproximation. As well as bypassing the check for maximality, this extra splitting has the advantage of accelerating convergence of the underapproximation by allowing the next underapproximation to include more states. For example, suppose that A has been split into A_1 and A_2 where A_1 intersects the underapproximation and A_2 does not. By separating out states in A_2, we make it easier for reachable states there to appear in the next underapproximation. For instance, if A_2 contains successors of states in the current underapproximation, by the second condition of the $\trianglerighteq$ operator, they will occur in the next underapproximation (unless their predecessors no longer occur in the next overapproximation).

8.2 Results

The approximation algorithm enables us to verify larger systems than our previous

Ex.	TA locns	Nr. Clocks	Mem (MB)	Time (s)
Fischer Mutual exclusion MX-4	1,704	4	4	23
MX-4-e	1,704	4	4	9
MX-7	120,863	7	5	126
MX-7-e	120,863	7	5	56
MX-9	3,259,136	9	9	941
MX-9-e	3,259,136	9	9	585
Fast Mutual exclusion FMX-3	17,377	3	8	144
AUDIO	83,660	2	7	489
AUDIO with timing	202,802	3	14	1077
Ethernet ETH-A	41,733	6	6	159
ETH-A-e	41,733	6	11	727
ETH-B	27,045	6	9	279
ETH-B-e	27,045	6	7	723
ETH-C	6,405	7	6	197
ETH-C-e	6,405	7	5	89
CSMA	189	4	3	3
Tick-tock protocol TT:iso-1	384	7	6	148
TT:iso-2	216	6	4	19
TT:transmission delay	432	7	7	356
TT:spacing	216	7	4	22

MX-i	Fischer mutual exclusion, i processes
FMX-i	Fast mutual exclusion, i processes
AUDIO	Audio control protocol
ETH-X	Ethernet examples, Specification X
CSMA	Carrier Sense / Multiple Access Protocol
-e	example contains error run

Figure 8.1: Results

implementations of verifiers, as well as any other automata-based automatic verifiers. It is also relatively fast. All code is in `C`, and the OBDD routines are from David Long's package. The results in figure 8.1 were obtained on a DEC 5000 with 56 MB of main memory. The number of "reachable" TSA locations refers to those locations forwards reachable in an untimed analysis of the state graph, *i.e.* assuming the enabling of events is independent of the timing conditions. We note that we are able to verify systems with over a million control locations.

		SINGLE LOCNS		SETS of LOCNS (OBDDs)	
Ex.	TA locns	# Itns	Time (s)	# Itns	Time (s)
MX-4	1,704	1	3	12	23
MX-4-e	1,704	1	26	3	9
MX-7	120,863	—	—	18	417
MX-7-e	120,863	—	—	4	37
ETH-A	41,733	—	—	1	220
ETH-A-e	41,733	—	—	7	1501
ETH-B	27,045	1	2160	1	108
ETH-B-e	27,045	—	—	3	1929

Figure 8.2: Single locations vs sets of locations

Approximating over control information

For the larger examples we considered, the implementation using a symbolic representation of sets of control locations far outperforms the one with all control locations separated, see figure 8.2. It should be noted however that performance may depend critically on the initial separating structure used.

8.3 Additional heuristics

8.3.1 Choice of initial partition

The system designer is capable of using her own knowledge of the system to aid the verification procedure, by judicious choice of an initial separating structure. Many other automatic verification techniques do not allow the user to supply useful information directly to the verification package. Typically the verification engineer must have a thorough understanding of both the system being verified, and the algorithm being used to verify it, and then devise a clever encoding of the problem which takes both into account. While optimal use of the approximation algorithm also requires knowledge of both the system being verified and the approximation technique, the features of the algorithm can be exploited without having to manipulate the system description itself. The user need only tell the algorithm where to approximate the truly reachable states more carefully, and where it can be more lax. In general, states can be kept in the same separating class whenever their outgoing behaviors are in some sense similar. The following heuristics can be used to guide the choice of an initial separating structure:

- parts of the state-space where timing information (either for outgoing constraints or incoming timer values) is similar can be clustered together.
- the state-space should be finely partitioned in areas where timing information is critical for correct operation.
- states which correspond to different branches of a critical case analysis should be separated.
- processes or variables which play a key role in the correctness of the specification should be partitioned more finely.
- the size of the initial separating structure should depend on the memory available. If a machine has only enough memory to store n approximating sets and their associated overhead, then a good heuristic is to keep the size of the initial separating structure less than $n/30$. This policy allows room for the four converging approximations, while still permitting reasonable growth due to splitting.

As an example, if the Fischer mutual exclusion protocol for six processes is verified by separating out locations based only on their specification component, the computation takes 17 traversals to complete in 343s. If a finer partition is chosen, namely splitting also according to the value of the critical controlling variable X, only 5 traversals are made and verification completes in 57s. If the additional splitting is done for process 1 rather than for the control variable X we find that 13 traversals are required in 298s.

8.3.2 Enhanced underapproximations

Experience shows that the main drawback to the performance of the approximation algorithm is due to slow convergence of the underapproximations. Slowly increasing underapproximations not only hamper the detection of violations, but also contribute to slow downward convergence of the overapproximations, since the refinement of the separating structures relies on information from the underapproximations.

In the case of real-time systems, the underapproximating operator over time zones is extremely weak. It essentially throws away information about its second operand unless it contains the first. In other words, sets of newly reachable timer vectors are discarded unless their time zone includes all timer vectors already reached within the class.

We now discuss further two techniques which help the propagation of the underapproximations.

Multiple underapproximation sets

The first strategy is to allow the underapproximating operator to return expansions which contain more than one approximating set. Allowing more approximating sets in the underapproximation results in more accurate underapproximations, but at the expense of additional memory. At one extreme we may chose to allow an underapproximation to consist of arbitrarily many approximating sets, and let the underapproximating operator return the union of its operands. In this case, computing an underapproximation will be essentially the same as performing exact set reachability. A happy compromise between a weak underapproximation and an exact computation is to allow the user to specify a fixed maximal number of approximating sets as the result of an application of the underapproximation operator.

Figure 8.3 shows how increasing the number of approximating sets can decrease the number of iterations necessary. The possible cost is more memory for storing an increased number of approximating sets.

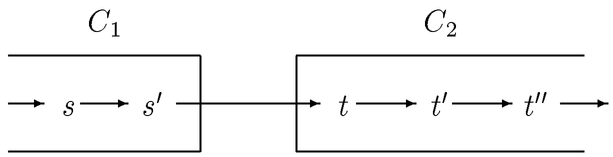
Stuttering the next-state relation

Underapproximations can be propagated throughout the state-space without the need for storing more approximating sets. First observe how a truly reachable state may be left out of the underapproximation. Suppose $s \in S$ is in the underapproximation but one of its successors s' is not. The non-emptiness condition for underapproximating

	1 U/A SET			3 U/A SETS			5 U/A SETS			7 U/A SETS		
	Itns	Time	Mem	Itns	Time	Mem	Itns	Time	Mem	Itns	Time	Mem
Ex.	#	s	MB	#	s	MB	#	s	MB	#	s	MB
MX-4	14	32	3.8	4	10	3.7	4	11	3.8	2	5	3.6
MX-4-e	3	6	3.7	2	6	3.7	2	4	3.6	2	4	3.6
MX-7	17	389	5.4	6	160	4.9	6	198	5.0	4	126	4.9
MX-7-e	3	69	4.8	2	67	4.7	2	55	4.5	2	56	4.6

Figure 8.3: Multiple underapproximating sets

operators implies the only way this can occur is if every separating class containing s' already contains other states in the underapproximation. If the only separating class containing s' also contains s, it may well be that no other successors of s appear in the underapproximation at all. Propagation of the underapproximation may be "stalled" at s.



This nonextension of states in the underapproximation can be partially solved by computing successors using an iterated (or stuttered) next-state relation, *i.e.* N can be replaced in the underapproximating algorithms with some N^k, where N^k is the result of composing N with itself $k-1$ times.

8.3.3 Untimed analysis

Sometimes computing even the first approximation of the reachable timed-states is expensive. A preliminary untimed analysis may be able to prune large parts of the state-space from consideration. For example, it may be that many control locations are forward reachable from the initial states, but not backwards reachable, and in this case the first forwards overapproximation will explore numerous control locations unnecessarily. A simple untimed backwards reachability analysis would rule out many of these control locations.

We propose the following procedure to cope with such situations: first analyze the state-space by ignoring all timing information, and then begin approximating the reachable timed-states over the reduced state-space. The untimed analysis should return a superset of the control locations which may possibly lie on violating paths. This phase considers the timed automaton as a simple (untimed) finite-state automaton, with an edge between two control locations whenever there is a transition between them. The analysis may be either exact or itself approximate, but it must yield an overapproximation of the control locations on violating paths. Indeed, this first untimed analysis may be considered to be a special case of overapproximating with an approximate next-state relation which disregards timing constraints and clock resets.

In our implementation, the untimed analysis consists of an exact forwards untimed traversal of the states, followed by an untimed backwards reachability analysis from the violating locations which are encountered.

The disadvantage of performing this untimed analysis is that it may itself be expensive to perform, and indeed may not even complete.

We note in passing that this untimed analysis may be sufficient to prove the system is correct, in which case either the system does not depend on timing information for correctness, or there is a description error in the input.

Observe that the algorithms of Alur et al [AIKY93] and Balarin et al [BSV93] also begin with an untimed analysis, and iteratively restrict the untimed traces by adding untimed components to rule out paths which are not possible because of timing constraints. In contrast, we use the untimed analysis merely as a special preliminary procedure to narrow the search space for our state-based approximations.

8.4 Performance comparison to other tools

Meaningful comparison with other implementations is difficult. Firstly there are not many verifiers which handle dense-time semantics. Secondly, those which do are often still undergoing development. Thirdly, and perhaps most prohibitive, is the fact each tool uses at least slightly different formalisms for describing real-time processes, and for specifying timing properties.

We compare our approximation algorithm against our previous implementations based on set-reachability and minimization, and against the symbolic model-checker KRONOS, recently been made publicly available by Sifakis et al [NSY92a, HNSY92, DOY94] at IMAG in France.

8.4.1 Reachability and minimization

The approximation algorithm represents a significant practical improvement over a couple of previously published algorithms we have experimented with. In this subsection, we describe our previous implementations and compare their performance. Many of the ideas behind the approximation scheme advocated here arose from experience with these other verifiers, and we discuss some of these issues in more detail in section 8.5.

Set-reachability

A basic set-reachability algorithm is given in figure 4.4 of section 4.4. It can be used in a straightforward way to solve the timed safety verification problem, since it computes exactly which regions have states which are reachable from the initial states. It can also be easily modified to prove stronger properties involving fairness.

One problem with explicitly enumerating all nodes in the regions graph, is that many different regions need to be examined, and the size of the graph generated depends crucially on the size of the timing constraints used. Set-reachability does much better locally, since all successors of a single transition can be added in a single step. For example, sets of time successors can be clustered together in a single DBM. In systems with simple looping structures this algorithm may be quite effective, but in more complex examples, a single control location can be entered along different transitions, each with different timing constraints. When the algorithm follows these transitions, many new sets may be generated. Of all the algorithms we provide comparative data for, this one is the least efficient in practice, as shown by the data in figure 8.4 appearing later in this section.

Minimization

An approach to circumventing the size of the regions graph is to build instead a minimal representation of the reachable part of the graph. The algorithms of Bouajjani et al [BFH$^+$92] and Lee and Yannakakis [LY92] *simultaneously* minimize and generate a superset of the reachable subgraph of a transition system. We implemented a variation of the algorithm of Bouajjani et al applied to timed automata [ACH$^+$92, ACD$^+$92]. We refer the reader to their publications and only sketch the ideas behind their algorithm. The algorithm starts with a transition system and an initial partition of its states. A class X is *stable* if whenever a state $s \in X$ has a successor in a class X', all states in s's class have successors in X'. The algorithm continually refines the partition by splitting *reachable* classes which are not stable with respect to the other classes.

Lee and Yannakakis's minimization algorithm [LY92] is similar to the one above. They specify an explicit strategy for choosing which class of the partition to split next. Their selection strategy guarantees an upper bound on the running time which is quadratic in the size of the minimal graph, provided there is a finite minimal graph and a means of detecting termination. The idea is to search forward to find classes which need to be split, and to give every class a fair chance of being split. Classes are marked with reachable points, and consequent splitting is done "around" this reachable point, thereby ensuring that all splitting is done on reachable classes. Yannakakis and Lee [YL93] also discuss how the algorithm can be applied efficiently to minimize a real-time system.

The most straightforward use of the minimization algorithm for safety verification would be to generate the minimal reachable graph starting from an initial partition which separates the violating states from the rest. We could then check whether any block containing violating states were reachable. If any were, then the system would contain a violation.

The algorithm we implemented improves on this strategy by avoiding unnecessary refinement of the graph. The modifications are based on the simple observation that it is not necessary to generate the exact minimal reachable graph in order to determine whether the violating states are reachable. For instance, if a class A is unstable with respect to a class B, and it is known that no violation states are accessible from states in B, then it is unnecessary to split A with respect to B. Of course we do not know in advance exactly which states lie on violating paths, but we can ascertain for sure that some do not using the following reasoning. Given a set-graph G that has an edge between two nodes A and B whenever there are states $a \in A$ and $b \in B$ such that $N(a, b)$, then the states in $reach(G)$ contains $reach(\mathcal{S})$. Thus we may be able to determine that some blocks contain only states that definitely have no paths to violating states: although they may be reachable, we need not stabilize them, or stabilize other classes with respect to edges into them. Thus we advocate specializing the minimization algorithm by periodically removing from consideration all classes from which violation states are not accessible. We also developed refined methods for choosing which class to split next, an order for the transitions to be stabilized in, lookahead strategies for increasing the number of classes detected as reachable, and simple techniques to ensure refining of the graph occurred evenly across the state-space rather than potentially wasting effort in a localized area which does may not lie on any violating paths.

Comparison

Comparative results are displayed in figure 8.4. Not surprisingly, approximation outperforms set-reachability. It is also far more efficient, in time and space, than our implementation of the minimization-based verifier, despite the large number of heuristics added to the latter. The results suggest that an exact reachable state analysis of a real-time system is both expensive and unnecessary for many timing-based verification problems.

8.4.2 Symbolic model-checker KRONOS

The model-checker KRONOS [NSY92a] computes whether a given timed safety automaton satisfies a specification given as a formula in the branching-time temporal logic TCTL [ACD90] where modal operators are time-bounded. It implements the symbolic model-checking algorithm found in [HNSY92].

Ex.	TA locns	SET REACH. Time (s)	MINIM. Time (s)	APPROX. Time (s)
GTC	32	1	4	1
MX-3	344	5	2	8
MX-3-e	344	27	26	3
MX-4	1,704	28	15	23
MX-4-e	1,704	-m-	-m-	9
MX-7	120,863	-m-	-m-	417
MX-7-e	120,863	-m-	-m-	37
FMX-3	17,377	-m-	197	608
ETH-A	41,733	-m-	-m-	220
ETH-A-e	41,733	-m-	-m-	1501
ETH-B	27,045	-m-	-m-	108
ETH-B-e	27,045	-m-	-m-	1929

GTC	Gate-Train Controller
MX-i	Fischer mutual exclusion, i processes
FMX-i	Fast mutual exclusion, i processes
ETH-X	Ethernet examples, Specification X
-e	example contains error run
-m-	ran out of memory

Figure 8.4: Comparative results

Our process semantics exactly match that of KRONOS. However their specifications are more general than ours. They verify formulae written in TCTL, a branching-time temporal logic with time-bounded modal operators. Using this logic they are able to express every timed safety verification problem, since reachability is expressible in the logic. Furthermore, there are properties given as logical formulae which are not timed-safety properties, such as non-Zenoness, and the singularity constraint that an event is never enabled for an open interval of time. In any case, a very preliminary analysis shows our approximation algorithm completes in less time, and uses less memory. The results for the Fischer mutual exclusion protocol and the tick-tock protocol examples appear in figure 8.5. The parameter set E has values $\pi = 100$, $\tau_{min} = 75$, $\tau_{max} = 120$ and $\alpha = 50$, F has $\pi = 100$, $\tau_{min} = 50$, $\tau_{max} = 75$ and $\alpha = 150$, and G uses $\pi = 100$, $\tau_{min} = 75$, $\tau_{max} = 220$ and $\alpha = 50$. Results were obtained on a Sun Sparcstation 2 with 128 MB of memory, of which all the examples given were verified by our algorithm using less than 9 MB. Notice that we cannot verify the

	KRONOS		APPROX*		Factor
Ex.	# Itns	Time (s)	# Itns	Time (s)	Faster
MX-6	12	1174	4	74	16
MX-6-e	10	1323	2	30	44
MX-7	-m-	-m-	4	164	–
MX-7-e	-m-	-m-	2	78	–
MX-8	-m-	-m-	4	375	–
MX-8-e	-m-	-m-	2	220	–
MX-9	-m-	-m-	4	891	–
MX-9-e	-m-	-m-	2	596	–
TICK-TOCK					
E:iso-1	15	1016	8	112	9.0
E:iso-2	9	13	4	3	4.0
E:iso-3	1	1	N/A	N/A	—
E:transmd	17	1227	14	69	17.7
E:sp	7	26	4	4	6.0
E:imm	1	1	N/A	N/A	—
F:iso-1 -e	33	87	4	39	2.2
F:iso-2	7	5	4	3	1.7
F:iso-3	1	1	N/A	N/A	—
F:transmd -e	23	191	4	72	2.7
F:sp	8	33	4	5	6.9
F:imm	1	1	N/A	N/A	—
G:iso-1 -e	22	121	6	93	1.3
G:iso-2	9	7	4	3	2.0
G:iso-3	1	1	N/A	N/A	—
G:transmd	15	264	10	166	1.6
G:sp	7	20	4	5	4.2
G:imm	1	1	N/A	N/A	—

MX-i	Fischer mutual exclusion, i processes
E/F/G	indicates different timing parameters
-e	example contains error run
-m-	ran out of memory

(*) excludes 1 min compilation time

Figure 8.5: Comparative performance

singularity properties Iso-3 and Imm, since they are not expressible in our framework.

However, for all the properties which can be expressed by both methodologies, the approximation algorithm is more memory efficient and is able to complete verification for every example for which KRONOS completes. We are also able to verify systems with much larger control spaces. For example the Fischer protocol with 9 processes has 9 clocks and 3,259,136 control locations reachable in an untimed analysis, and verification completes in under 9 MB. Our implementation is up to 44 times faster over the 6 process example KRONOS can verify. The approximation algorithm is also consistently faster, up to a factor of 18, over examples published by the developers of KRONOS, even when the examples use tight timing constraints. The relative benefits of the two verifiers needs to be explored in more depth. Indeed, it appears the advantages of both verifiers could be exploited by using KRONOS to verify the TCTL properties not expressible as safety verification problems, and using our approximation scheme to verify more limited properties over large examples.

8.5 Lessons learnt

The approximation strategies discussed in this thesis, and some of the implementational choices, are the result of lessons we learnt in building verifiers and examining how they performed over the case studies described in the previous chapters. This section collects together some of our experiences, which are by no means unique, in the hope that it can guide future development of verification tools.

8.5.1 Complexity issues

The worst-case complexity is not always the most relevant feature of an algorithm: the adversarial problem inputs may occur rarely in practice. This fact suggests it may be useful to give a stronger characterization of problem inputs, to restrict analysis to certain useful subclasses of the problem domain, to perform an average-case analysis, or to provide an analysis which compares two algorithms over each individual input instance. However, it is usually difficult to define or even describe a "typical" problem, or give additional useful measures of the problem's complexity.

We note that in our experience some algorithms with poorer complexity outperform theoretically optimal ones. The regions construction of Alur and Dill [AD90] has worst-case complexity exponentially better than the set-reachability algorithm of Dill [Dil89], yet it is easy to see that in many instances of the train-gate example the size of the regions graph is far greater than the set-reachability graph. In addition, our implementation of the minimization algorithm of Lee and Yannakakis [LY92] does not perform better than that of Bouajjani et al [BFH90], despite its theoretical advantages of being polynomial in the size of the minimized graph. However, one of their key ideas in providing an upper bound on run-time, namely using points to mark classes, was very helpful in forcing the splitting of classes to occur throughout the state-space, rather than being localized. This marking of classes, together with giving each class a fair chance of being split, is used in their upper bound results. However, when we experimented with different orderings for splitting classes, we found Lee and Yannakakis's queuing strategy to have no practical effect on convergence, despite being required for their upper bound result. In fact, we implemented heuristics based on splitting classes whose successor classes were not marked, and these made a significant improvement.

8.5.2 Large control spaces

Realistic systems have not only complex timing constraints, but also large control spaces. While there may be ways to extract the timing properties of some systems, and analyze them separately, we believe that in general it is essential to be able to model both timing information and large control spaces in a single system description. It was this fact that lead us to consider algorithms which could share timing information over different control locations. Otherwise it is likely to be too expensive to associate timing constraints with every reachable location. A hash table can do this effectively when the exact same timing constraints apply at many different locations. However this is not always the case. Approximation can be used to associate numerous locations, having different exact timing constraints, with the same *approximate* timing constraints, thereby allowing even further reductions in storage. This motivates the use of *sets* of control locations in approximating sets. The results of this chapter demonstrate the success of this approach.

8.5.3 User-supplied information

A good heuristic algorithm should allow the user to provide some guiding information in a natural and simple way. There is a good chance that the system designer or verifier has some knowledge about why the system is correct (or why she suspects it is). It is potentially very useful for this information to be passed directly to the verifier. For example, enumeration-based techniques usually work with some fixed steps designed to exhaustively cover every combination of possibilities, without regard to how its search of the state-space could be optimized. User-intervention could be used to focus attention in particular areas, or supply invariant information about the state-space.

Our approximation algorithm has a straightforward means for the user to decide how roughly or accurately to begin approximating. While this information is very limited in form, we find it very effective in increasing the performance of our verifier. For more details, see subsection 8.3.1.

The user may also fix the maximum number of underapproximating sets within each class. This parameter can be matched with the size of the initial separating classes. In other words, if there are long paths within the separating classes, the underapproximations may increase very slowly, requiring numerous traversals before all reachable states within the class are detected. Thus having fewer separating classes requires more underapproximating sets per class for similar progress in the propagation of the underapproximations. Note that this strategy maintains a relatively constant total number of underapproximating sets.

8.5.4 Symbolic representations

While we have been suggesting that symbolic representations can lead to reductions in computation time and memory usage, it must be remembered that only a *good* symbolic representation of a problem will help. The representation must be small for most sets encountered, and admit efficient operations. Furthermore, the algorithm must consider only a small number of symbolic sets — otherwise it may use more memory to store sets of states than if it explicitly enumerated the individual states.

Time zones

Time zones and DBMs do work well for representing the reachable states of a system. As shown above, there are fast algorithms for finding successor states of any time zone. Their canonical form has an $O(n^2)$ representation and is $O(n^3)$ to compute. They are also closed under intersection, as required by the approximation algorithm. Their main disadvantage is that they are not closed under union, and in exact reachability algorithms this can result in a long list of time zones to represent the reachable time vectors for a given control location. Using approximation has the advantage of storing only a fixed number of time zones for a location, avoiding the blow-up due to lack of closure under union.

The overapproximation operator is an effective means of capturing the information in its operands. It returns the smallest possible zone which contains its operands, and is in effect the pairwise disjunction of all constraints needed in defining them. Quite often this zone encapsulates sufficient reachability information for an accurate approximation. For instance, it is common for the value of a particular clock to be irrelevant in determining the outgoing paths from a state s. Approximation over the values of such a clock at s does not directly lead to any false negatives. In other cases, outgoing traces from s depend only on whether a clock x lies above (or below) a certain threshold, l say. Storing information about the exact reachable values is no more useful that knowing whether any reachable values exceed the threshold l, and this information is retained by the overapproximation operator.

OBDDs for control information

We find that using OBDDs for the control component of the state-space is also effective. Firstly, there are potentially many control locations with the same timing constraints on reachable states. In systems with large control spaces, there may be many events which are essentially asynchronous, and only a small part of the system which is really timing dependent. Many events may have no timing constraints associated with them. The constraints associated with state s are the same as for its successor state s' if the only event into s' originates at s and is independent of the clock values. Thus the time zones for these adjacent states are identical. Furthermore sets of adjacent states often have small OBDDs since they may be obtained via untimed events occurring individually in different components. This observation also allows effective initial partitioning which clusters together locations which are separated only by untimed events.

The benefit of using OBDDs for control information is even greater when approximations are used. The arguments in the previous subsection for why approximate timing information is often good enough still holds over sets of locations. Thus we have the potential to pool together states across different locations with slightly varying timing constraints into single approximating sets, without much loss of information. We find this space saving to be necessary for analyzing systems with control spaces too large for storing individual DBMs per location.

Finally we note that the form of initial partitioning we use, dividing the control space via a crossproduct of partitions per component, leads to small OBDDs for each initial separating class, and therefore helps to keep the size of OBDDs in subsequent separating classes small.

OBDDs for timing information

It is possible to use OBDDs for encoding timing information, *i.e.* they can encode the detailed regions of the Alur-Dill equivalence relation, and then arbitrary sets of timed states can be represented within a single framework. However, this approach does not look promising. The problem is that there are too many dependencies across clocks in different components, leading to large OBDDs. For example, computing the time successors of a set of regions involves checking that the values of all clocks increase at the same rate. In fact, our own experiments with OBDD-encoded regions graphs resulted in worse performance than explicit analysis.

8.5.5 Simplify the problem

Hard problems should be simplified wherever possible until the work of the verifier is computationally feasible. In other words, it is extremely helpful if some human reasoning can be used to reduce a verification problem into a simpler form before handing it to the automatic verifier.

Unnecessary computation may be avoided via restrictions on the problem domain. In our case, we first choose to concentrate only on reachability properties. This simplifies our algorithms and enables us to focus on the key issue of representing state information. Furthermore, if we cannot tackle the simpler problems, there is little hope for the harder ones. However, we do of course sacrifice expressiveness.

Secondly, we rely on syntactic conditions to guarantee our systems are non-Zeno. This choice saves the verifier from checking this property. One approach we took in earlier work [ACD$^+$92] was to have the verifier iteratively create graphs whose paths were guaranteed to be divergent. The algorithms first generate graphs which represented all timed runs. If these graphs are empty, the system is verified correct. If not, they are successively refined until the only remaining paths corresponded to divergent traces. This extra computation is time-consuming and causes the graphs to grow rapidly.

Finally, we note that minimization-based techniques use the wrong criterion for splitting classes, if the problem to be solved is reachability. The splitting is too exacting for plain reachability analysis, since it is really bisimulation-based. This is no poor reflection on the minimization algorithms themselves, but rather a comment to tailor techniques to match the problem at hand.

8.5.6 Indications of progress

When attempting to verify large systems, a verification attempt will often run for a long time, seemingly indefinitely, or simply fail reporting a lack of memory. In such cases, it is useful to have an idea of how close the verifier is to solving the problem. This information can be helpful in deciding whether a particular encoding of a system is effective for a given verification algorithm. It can also be a useful measure of whether one algorithm is better than another, and is thus extremely useful in designing heuristics.

As an example of a progress indicator, for explicit enumeration methods, the ratio of new states encountered (or the size of the search stack) may give some indication of how much of the reachable state-space has been found. In symbolic reachability, the sizes of the OBDDs often slowly increase to a peak and then decline, so their sizes can help predict how far the algorithm is from terminating.

For our approximation scheme, where there is a choice of parameters for the verifier's execution, it is even more important to have an indication of how close the verifier is to deciding correctness. Indicators can be used to guide how to choose effective parameters. Both kinds of convergence patterns mentioned above have been observed in the execution of our approximation algorithm. Firstly, for any given approximation the size of the search stack gives some indication of progress. Secondly, and perhaps of more concern is how close the successive approximations are to deciding correctness. Interestingly enough, for the examples we have looked at which require more than just a few traversals, there is a clear convergence pattern in the size of the approximations. In the first two traversals the size of the approximations usually decreases, since large parts of the state-space can be eliminated as being not *both* forwards reachable from the initial states and backwards reachable from violating states. Then the sizes usually increase, close to monotonically, and then decrease. We offer an intuitive explanation of this rise and fall in the size of the approximations. Changes in the size of the approximations are due to two competing factors. Separating classes which are too large need to be split, leading to larger approximations. On the other hand, as approximations become more accurate, some previously included states can be eliminated, including some entire classes, leading to smaller approximations. Initially the approximations are too crude and the overly large separating classes need to be refined. Each successive traversal splits more classes, and enables the underapproximations to increase accordingly. Rough approximations are not good at eliminating states falsely believed to be reachable, so the number of classes eliminated is initially small. Thus the approximations start increasing in size. When the separating classes give more accurate approximations, more classes will be eliminated, and fewer classes need to be split further. This phase is detected as the decline in the size of the approximations.

Having this simple guide to convergence can be helpful in deciding how to configure the approximation algorithm. Recall that the two primary parameters to the verifier's execution are the initial partition, and the number of underapproximating sets allowed per separating class. Adjusting either or both of these parameters and watching the convergence pattern of the approximations gives an idea of how effective changes are. Furthermore the relative sizes of the underapproximations to the overapproximations indicates how effectively the underapproximations are propagating through the state-space. If the underapproximations progress too slowly, the parameters can be adjusted accordingly.

8.5.7 Debugging information

It is crucial for a verification tool to provide useful debugging information for systems which are found to be incorrect. The first few attempts to describe a system inevitably contain syntax errors, or modeling errors, and a stark certification of "not correct" from the verifier does nothing to help the designer model the system more accurately.

Our current implementation provides traces whenever errors are found. However these are only violating pseudo-traces (see section 2.3.5). The algorithm could be adapted to produce true violating traces, but this feature is not supported in the current prototype. Furthermore, timing information is output via DBMs which are not easy to interpret — there is no explicit distinction between defining constraints and inferred constraints. The DBMs could be output via their defining constraints only, and a path of timed-states could be extracted from a path of regions, but again the necessary routines are not currently implemented. The control information is output in a more user-friendly fashion, not as OBDDs, but as a listing of the control locations they represent, in disjunctive normal form over each component's locations. While admittedly limited, this debugging information has generally proven sufficient for understanding errors.

8.6 Summary

Despite the fact that the verification of hard real-time systems is a difficult computational problem (PSPACE-complete), many examples are solvable in practice using our heuristic approximation methodology. Our implementation has been able to automatically verify systems with reasonably large control spaces and complex timing information — largely due to our ability to combine timing information across different locations of the state-space into single approximating sets.

The method does have its shortcomings. Sometimes even approximate analysis is expensive to compute. Furthermore there may be many iterations required before convergence. It is not always easy to choose a good initial partitioning: too fine a partition means that there is little advantage gained from approximating, and the size of the approximation can be large, whereas too coarse a partition may cause the approximations to be too crude, and require numerous traversals of the state-space. If all timing constraints in the system are tight, then approximation will have little benefit since correctness will not be detected until the approximations converge to being close to exact.

The algorithm performs well in detecting bugs in systems. More often than not, an attempted verification contains a description error which leads to false violations. It is therefore desirable for a verification algorithm to report errors efficiently. Our implementation has proven effective in catching such errors and in providing useful debugging information, albeit encoded in a symbolic form (DBMs) where the defining timing constraints are not clear. This could be improved in future implementations.

The structure of the approximation algorithm is sufficiently flexible to enable numerous enhancements and heuristics beyond the basic algorithmic description of chapter 5. Because each traversal need not compute an exact set of reachable states, there is a great deal of freedom in how an approximation algorithm can be designed. While some heuristics have been outlined here, the wealth of possible extensions is enormous.

Additional care and optimization could be applied to the code independently of the approximation strategies. For example, there are specialized algorithms that could be used for computing successor regions faster, minimizing the number of canonicalizations necessary, performing faster canonicalizations when only a few constraints are changed, and coalescing adjacent zones into single zones when possible [Rok93]. While we have concentrated on the approximation aspects of the algorithm during its implementation, there is no reason why these other optimizations could not also be incorporated.

Finally, it should be noted that no computationally efficient algorithm can counter the shortcomings of describing real-time systems in the low-level language of timed automata. In our experience, we encountered many description errors resulting from incorrect modeling in the timed automaton framework. It would be extremely helpful to have high-level description and specification languages. These could then be compiled into timed automata for the purpose of verification. Indeed, Nicollin et al [NSY92a] have developed a compiler from the process algebra ATP into timed safety automata, and Daws et al [DOY94] give translations from ET-LOTOS to timed safety automata.

Chapter 9

Conclusions

This thesis proposes a flexible approximation scheme for efficient safety verification. It has been specialized for the verification of real-time systems. An implementation of this algorithm shows very promising results. We now make suggestions for the future and offer concluding remarks.

9.1 Further work

9.1.1 Extensions

The approximation framework we have described is very general. There is plenty of scope for defining additional heuristics to either split classes further, not split them at all, or even recombine them. Also, in the current set-up, one approximation is computed at a time, either forwards or backwards, either overapproximating or underapproximating. It would be interesting to see how well approximations could be generated simultaneously. Another interesting direction to investigate is user-directed refinement, rather than having the algorithm run fully automatically.

9.1.2 Real-time verifier

We have found that our verifier works very well on the reasonably large real-time examples we have tested. We are still investigating further heuristics to increase the algorithm's effectiveness. One such example is the use of a "widening" operator [Hal93b] to accelerate the convergence of iterations within each individual traversal. A straightforward widening operator has been implemented for simple timed automata, resulting in mixed success only. Fairness could also be introduced into the semantics of processes.

The verifier we have built is definitely a prototype. It was developed to test and explore the ideas in this thesis. No work has been put into designing a friendly user-interface. There are also many inefficiencies in our implementation, such as memory handling and the storage of DBMs, which could be removed to improve efficiency.

It would be interesting to see how well the approximation technique of Alur et al [AIKY93] and Balarin et al [BSV93] could be combined with our state-based approximations. In principle, it is not difficult to iteratively add timing constraints into our approximation algorithm, as a special case of overapproximating next-state relations. An off-line examination of potentially false negatives can drive the convergence of the approximating relations.

The implementation needs to be tested on a wider variety of examples. This would lead to a better understanding of the verification problems that occur in practice and point to improved heuristics. A more detailed performance comparison with the other verifiers would be valuable, especially the timing approximation methods of Alur et al and Balarin et al.

9.1.3 Other problem domains

It would be interesting to see how well the approximation algorithm works when applied to systems other than real-time systems. The success of the algorithm for timed systems is due to the fact that timing information can sometimes be clustered together into a single zone without including timer vectors which exhibit different behavior.

Given a different problem domain, we need an efficient symbolic representation and approximation operators which are meaningful, and in some sense likely to cluster together only bisimilar states. We suggest possible generic operators for overapproximation and underapproximation. Given a domain of approximating sets for a problem, the overapproximating operator could return the smallest enclosing approximating set, and the underapproximating operator its right operand if it includes the left operand, and the left operand otherwise.

For untimed systems, OBDDs are an obvious candidate for a symbolic representation, since they can easily represent sets of states and next-state relations. One potential set-up for the algorithm is to use *hypercubes* as approximating sets, *i.e.* sets which can be defined by a single conjunction of literals. The operators suggested above result in (1) overapproximating by taking the conjunction of all positive or negative literals which appear in one operand, and whose negation does not appear in the other, and (2) underapproximating by taking the right operand iff all its conjuncts appear in the left operand, and the left operand otherwise. Both these operations are obviously efficient to compute, and over some untimed domains they may be sufficiently accurate.

Hybrid systems [AHH93] model continuously changing variables that operate under a finite number of modes. Variables are usually modeled as satisfying restricted forms of differential equations. They are more general than real-time systems, where the clocks are a special case of variables all increasing at a fixed rate. Most problems studied in this domain are undecidable, so the need for heuristic algorithms is even greater than for real-time systems. Already, one of the ideas proposed in this thesis has been applied to verifying hybrid systems. Henzinger and Ho [HH94] use the iteratively refined overapproximations of figure 2.4. They also incorporate useful widening operators.

9.1.4 Solving other problems

We believe combining overapproximation and underapproximation information to refine approximations is both sufficiently powerful and flexible to be applied successfully to a variety of problems other than state-reachability. We are investigating the practicality of verification of more general real-time processes and specifications (such as including fairness), not just real-time safety properties. The ideas behind the algorithms could also prove fruitful for model-checking logical specifications. Dams et al [DGG94] show how various abstractions can be combined for model-checking in an abstract interpretation framework. However they provide no means of dynamically refining their abstractions if they prove too weak. It may be possible to extract parameterized information about when a system will operate correctly, as Halbwachs [Hal93a] does for his single overapproximations. We are also interested in applying iterative approximation to real-time controller synthesis algorithms [WTH91, HWT92a].

9.1.5 Analytic analysis

The approximation algorithm proposed is clearly a heuristic. It would be of tremendous value to have analytical arguments for when it would perform well, and when it would not. It would also be helpful to have metrics for how close the approximations are to convergence.

9.2 Discussion

We believe there is a good semantic basis for the permissible-join heuristic used to refine approximations. The performance results of our prototype implementation for real-time systems show extremely promising results. However it should always be remembered that the algorithm still has poor worst-case complexity, exponentially worse than exact explicit analysis. What works well on some examples could do extremely poorly on others. Nevertheless our verifier has so far been proven consistently efficient.

As systems grow larger, performing an exact analysis becomes harder and harder. While exact enumerative methods have the theoretical advantage of guaranteed termination over finite-state systems, they are restricted in a very practical sense by the sizes of their state-spaces. It will become more important to have methods that do not exhaustively enumerate possibilities which may not be necessary. A form of clever decision-making is required. Our policy is to use quick and simple decisions designed to keep the size of the approximations small. It would be interesting to see whether a more careful analysis using greater lookahead would pay off in the long run. Another desirable property of a verifier is that progress is always being made towards a solution. The approximation method here is an attempt to combine the ideas of approximate analysis with the ability to converge towards a solution.

Approximate analysis has been used for years in more traditional fields of engineering. Typically systems are described using continuous variables, and differential equations are solved to determine system behavior. Algorithms are designed with a step-size parameter that dictates how accurately the system is tracked. In areas of instability, where the system behavior is more unpredictable, a finer step-size is used. These methods have been tremendously successful. What then is the difficulty in using similar ideas for verification? Continuous systems have compact representations, often so do discrete systems given in modular format. However states in a continuous system can be said to have similar behavior when they are close together, whereas the very nature of discrete systems means that there is no reliable way to easily detect when two discrete states have similar outgoing behaviors. The approximation method of this thesis is an attempt to decide exactly when "neighboring" states share similar behavior, and to approximate more finely when they do not.

Bibliography

[ACD90] Rajeev Alur, Costas Courcoubetis, and David Dill. Model-checking for real-time systems. In *Proceedings of the Fifth IEEE Symposium on Logic in Computer Science*, pages 414–425, 1990.

[ACD+92] R. Alur, C. Courcoubetis, D. Dill, N. Halbwachs, and H. Wong-Toi. An implementation of three algorithms for timing verification based on automata emptiness. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 157–166, Phoenix, AZ, December 1992.

[ACH+92] R. Alur, C. Courcoubetis, N. Halbwachs, D. Dill, and H. Wong-Toi. Minimization of timed transition systems (extended abstract). In *Proceedings of CONCUR '92, Third International Conference on Concurrency Theory*, pages 340–354, Stony Brook, NY, August 1992. Springer-Verlag. Lecture Notes in Computer Science, Volume 630.

[ACHH93] Rajeev Alur, Costas Courcoubetis, Thomas A. Henzinger, and Pei-Hsin Ho. Hybrid automata: an algorithmic approach to the specification and verification of hybrid systems. In *Hybrid Systems*, pages 340–354. Springer-Verlag, 1993. R.L. Grossman and A. Nerode and A.P. Ravn and H. Rischel, editors. Lecture Notes in Computer Science 736.

[AD90] Rajeev Alur and David L. Dill. Automata for modeling real-time systems. In *Proceedings of 17th International Colloquium on Automata, Languages and Programming*, pages 322–335. Springer Verlag, LNCS 443, 1990.

[AFH91] Rajeev Alur, Tomas Feder, and Thomas A. Henzinger. The benefits of relaxing punctuality. In *Proceedings of 10th Annual Symposium on Principles of Distributed Systems*, pages 139–152. ACM Press, 1991.

[AH89] Rajeev Alur and Thomas A. Henzinger. A really temporal logic. In *Proceedings of 30th Symposium on Foundations of Computer Science*, pages 164–169, 1989.

[AH92] Rajeev Alur and Thomas A. Henzinger. Logics and models of real time: a survey. In *Proceedings of the 1991 REX workshop "Real Time: Theory in Practice"*, pages 74–106. Springer-Verlag, 1992. Lecture Notes in Computer Science 600.

[AH94] Rajeev Alur and Thomas A. Henzinger. Real-time system = discrete system + clock variables. In *Theories and Experiences for Real-Time System Development (Proceedings First AMAST Workshop on Real-Time System Development)*, chapter 1. World Scientific Publishing, 1994. ed. Teo Rus and Charles Rattray.

[AHH93] Rajeev Alur, Thomas A. Henzinger, and Pei-Hsin Ho. Automatic symbolic verification of embedded systems. In *Proceedings of IEEE Real-Time Systems Symposium*, Raleigh-Durham, NC, December 1993.

[AIKY93] Rajeev Alur, Alon Itai, Robert Kurshan, and Mihalis Yannakakis. Timing verification by successive approximation. In *Proceedings of Fourth International Workshop on Computer Aided Verification (CAV '92)*, pages 137–50, Montreal, Canada, 1993. Springer-Verlag. Lecture Notes in Computer Science 663.

[AKH88] A. Ayyagari, S. Kumara, and I. Ham. Robot path planning in 2-d, using modified quad-tree approach. In *Recent Developments in Production Research*, pages 707–713. Elsevier Science Publishers B.V., 1988.

[AL92] M. Abadi and L. Lamport. An old-fashioned recipe for real time. In *Proceedings of the 1991 REX workshop "Real Time: Theory in Practice"*, pages 1–27. Springer-Verlag, 1992. Lecture Notes in Computer Science 600.

[Alu91] Rajeev Alur. *Techniques for Automatic Verification of Real-Time Systems*. PhD thesis, Department of Computer Science, Stanford University, August 1991.

[AT92] Rajeev Alur and Gadi Taubenfeld. Results about fast mutual exclusion. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 12–21, Phoenix, AZ, December 1992.

[BCM+90] J.R. Burch, M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of Logic in Computer Science*, 1990.

[BFH90] A. Bouajjani, J. Fernandez, and N. Halbwachs. Minimal model generation. In *Proceedings of Second Workshop on Computer-Aided Verification, Rutgers University*, 1990.

[BFH+92] A. Bouajjani, J. Fernandez, N. Halbwachs, P. Raymond, and C. Ratel. Minimal state graph generation. *Science of Computer Programming*, 18(3):247–269, June 1992.

[BM83] Bernard Berthomieu and Miguel Menasche. An enumerative approach for analyzing time petri nets. In *Proceedings of IFIP Congress*, pages 41–46, Paris, September 1983. Elsevier Science Publishers BV (North-Holland).

[BPV94] Doeko Bosscher, Indra Polak, and Frits Vaandrager. Verification of an audio control protocol. In *Proceedings of Formal Techniques in Real Time and Fault Tolerant Systems Symposium*, 1994.

[Bry86] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[Bry92] Randal E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *Computing Surveys*, 24(3):293–318, September 1992.

[BSV93] F. Balarin and A.L. Sangiovanni-Vincentelli. A verification strategy for timing constrained systems. In *Proceedings of Fourth International Workshop on Computer Aided Verification (CAV '92)*, pages 151–63. Springer-Verlag, 1993. Lecture Notes in Computer Science 663.

[Cad92] Marco Cadoli. Two methods for tractable reasoning in artificial intelligence: Language restriction and theory approximation, December 1992. Extended abstract of PhD thesis.

[CB89] K. Cho and R.E. Bryant. Test pattern generation for sequential MOS circuits by symbolic fault simulation. In *Proceedings of the 26th ACM/IEEE Design Automation Conference*, pages 418–423, Las Vegas, June 1989.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the 4th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977.

[CC92] Patrick Cousot and Radhia Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2-3):103–179, July 1992.

[Cer93] K. Cerans. Decidability of bisimulation equivalence for parallel timer processes. In *Proceedings of Fourth Workshop on Computer-Aided Verification (CAV '92)*, Montreal, Canada, 1993. Springer-Verlag. Lecture Notes in Computer Science 663.

[CES83] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications: A practical approach. In *Proceedings of the 10th Annual ACM Symposium on Principles of Programming Languages*, pages 117–126, 1983.

[CHR91] Zhou Chaochen, C.A.R. Hoare, and A.P. Ravn. A calculus of durations. *Information Processing Letters*, 40(5):269–276, December 1991.

[CHS93] Zhou Chaochen, M.R. Hansen, and P. Sestoft. Decidability and undecidability results for duration calculus. In *Proceedings of Tenth Symposium on Theoretical Aspects of Computer Science, STACS-93*, pages 58–68, 1993. Lecture Notes in Computer Science, Volume 665.

[CK91] E.M. Clarke and R.P. Kurshan, editors. *Computer-Aided Verification '90*, volume 3 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, Association for Computing Machinery, 1991.

[Cou78] Patrick Cousot. *Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique de programmes*. PhD thesis, Université scientifique et médicale de Grenoble, Grenoble, France, March 1978.

[Cou90] Patrick Cousot. *Methods and logics for proving programs*, chapter 15, pages 843–993. Elsevier Science Publishers B.V. (North-Holland), Amsterdam, 1990.

[CY92] C. Courcoubetis and M. Yannakakis. Minimum and maximum delay problems in real-time systems. In *Proceedings of the Third Workshop on Computer-Aided Verification (CAV '91)*, Aalborg, Denmark, 1992. Springer-Verlag. Lecture Notes in Computer Science 575.

[DGG94] Dennis Dams, Orna Grumberg, and Rob Gerth. Abstract interpretation of reactive systems: abstractions preserving $\forall$CTL*, $\exists$CTL* and CTL*. Technical Report 94/24, Eindhoven University of Technology, Department of Mathematics and Computing Science, May 1994.

[Dil89] David L. Dill. Timing assumptions and verification of finite-state concurrent systems. In *Automatic Verification Methods for Finite State Systems, International Workshop, Grenoble, Lecture Notes in Computer Science 407*, pages 197–212. Springer-Verlag, 1989.

[DOY94] C. Daws, A. Olivero, and S. Yovine. Verifying ET-LOTOS programs with KRONOS. In *Proceedings of FORTE '94*, 1994.

[EMSS89] E.A. Emerson, A.K. Mok, A.P. Sistla, and J. Srinivasan. Quantitative temporal reasoning. In *Proceedings of International Workshop on Automatic Verification Methods for Finite State Systems*, Grenoble, France, 1989. Springer-Verlag. Lecture Notes in Computer Science 407.

[FKM91] M. Fujita, T. Kakuda, and Y. Matsunga. Redesign and automatic error correction of combinational circuits. In *Logic and Architecture Synthesis: Proceedings of the IFIP TC10/WG10.5 Workshop on Logic and Architecture Synthesis*, pages 253–262. Elsevier, 1991. P. Michel and G. Saucier, editors.

[Hal93a] Nicolas Halbwachs, 1993. Private communication.

[Hal93b] Nicolas Halbwachs. Delay analysis in synchronous programs. In *Proceedings of Conference on Computer-Aided Verification*, Heraklion, Crete, Greece, June 1993.

[Hen91] Thomas A. Henzinger. *The Temporal Specification and Verification of Real-Time Systems*. PhD thesis, Department of Computer Science, Stanford University, August 1991.

[HF90] Pei-Yung Hsiao and Wu-Shing Feng. Using a multiple storage quad tree on a hierarchical VLSI compaction scheme. *IEEE Transactions on Computer-Aided Design*, 9(5):522–536, 1990.

[HH94] Thomas A. Henzinger and Pei-Hsin Ho. Model checking strategies for hybrid systems. In *Proceedings of the International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems*, 1994.

[HLP90] E. Harel, O. Lichtenstein, and A. Pnueli. Explicit-clock temporal logic. In *Proceedings of the Fifth Annual Symposium on Logic in Computer Science*, pages 402–413. IEEE Computer Society Press, 1990.

[HNSY92] Thomas A. Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic model checking for real-time systems. In *Proceedings of the Seventh IEEE Symposium on Logic in Computer Science*, pages 394–406, 1992.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[HPV94] Thomas A. Henzinger, Anuj Puri, and Pravin Varaiya. Clock transformation of hybrid systems with rectangular differential inclusions. Presented at the Workshop on Hybrid Systems and Autonomous Control (Ithaca, NY), 1994.

[HWT92a] Gérard Hoffmann and Howard Wong-Toi. The input-output control of real-time discrete event systems. In *Proceedings of the 1992 IEEE Real-Time Systems Symposium*, pages 256–265, Phoenix, AZ, December 1992.

[HWT92b] Gérard Hoffmann and Howard Wong-Toi. Symbolic synthesis of supervisory controllers. In *Proceedings of the 1992 American Control Conference*, pages 2789–2793, Chicago, IL, June 1992.

[KL94] Inhye Kang and Insup Lee. An efficient generation of the timed reachability graph for the analysis of real-time systems. Technical Report MS-CIS-94-36, University of Pennsylvania, 1994.

[KPSY93] Y. Kesten, A. Pnueli, J. Sifakis, and S. Yovine. Integration graphs: a class of decidable hybrid systems. In *Hybrid Systems*. Springer-Verlag, 1993. R.L. Grossman and A. Nerode and A.P. Ravn and H. Rischel, editors. Lecture Notes in Computer Science 736.

[KU80] M. Kaplan and J.D. Ullman. A general scheme for the automatic inference of variable types. *Journal of the Association for Computing Machinery*, 27(1):128–145, 1980.

[LA90] N.A. Lynch and H. Attiya. Using mappings to prove timing properties. In *Proceedings of the Ninth ACM Symposium on Principles of Distributed Computing*, pages 265–280, 1990.

[LB93] William K.C. Lam and Robert K. Brayton. Alternating RQ timed automata. In *Proceedings of Fifth International Conference on Computer Aided Verification, CAV-93*, pages 237–252. Springer-Verlag, 1993. Lecture Notes in Computer Science 697.

[Lev84] Hector J. Levesque. A logic of implicit and explicit belief. In *Proceedings of the Fourth National Conference on Artificial Intelligence, AAAI-84*, pages 198–202, 1984.

[Lev89] Hector J. Levesque. A knowledge-level account of abduction. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence (IJCAI-89)*, pages 1061–1067, 1989.

[Lew90] Harry Lewis. A logic of concrete time intervals. In *Proceedings of the Fifth IEEE Symposium on Logic in Computer Science*, pages 380–389, 1990.

[LLD94] G. Leduc, L. Léonard, and A. Danthine. The Tick-Tock case study for the assessment of timed FDTs. In *The OSI95 transport service with multimedia support on HSLAN's and B-ISDN*, 1994.

[LS85] N. Leveson and J. Stolzy. Analyzing safety and fault tolerance using timed petri nets. In *Proceedings of International Joint Conference on Theory and Practice of Software Development*, pages 339–355, 1985. Lecture Notes in Computer Science 186.

[LV92] N.A. Lynch and F. Vaandrager. Forward and backward simulations for timing-based systems. In *Proceedings of the 1991 REX workshop "Real-Time: Theory in Practice"*, pages 397–446. Springer-Verlag, 1992. Lecture Notes in Computer Science 600.

[LY92] David Lee and Mihalis Yannakakis. Online minimization of transition systems (Extended Abstract). In *Proceedings of ACM Symposium on Theory of Computing 1992*, Vancouver, B.C., 1992.

[MC91] J.-C. Madre and O. Coudert. A logically complete reasoning system based on a logical constraint solver. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence*, pages 294–299, Sydney, August 1991.

[McM92] Kenneth McMillan. *Symbolic model checking: an approach to the state explosion problem.* PhD thesis, School of Computer Science, Carnegie Mellon University, 1992.

[MF76] P. Merlin and D.J. Faber. Recoverability of communication protocols. *IEEE Transactions on Communications, COM-24*, (9), September 1976.

[Mil80] Robin Milner. *A Calculus of Communicating Systems.* Springer-Verlag, 1980. Lecture Notes in Computer Science 92.

[MM88] R.W. McColl and G.R. Martin. Quad-tree modelling of colour image regions. In *Proceedings of SPIE, Vol 1001 Visual Communications and Image Processing*, pages 231–238, 1988.

[MP93] Z. Manna and A. Pnueli. Verifying hybrid systems. In *Hybrid Systems.* Springer-Verlag, 1993. R.L. Grossman and A. Nerode and A.P. Ravn and H. Rischel, editors. Lecture Notes in Computer Science 736.

[MV94] Jennifer Mcmanis and Pravin Varaiya. Suspension automata: a decidable class of hybrid automata. In *Proceedings of Sixth International Conference on Computer-Aided Verification (CAV-94)*, pages 105–117, 1994. Lecture Notes in Computer Science 818.

[NOSY93] X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. An approach to the description and analysis of hybrid systems. In *Hybrid Systems.* Springer-Verlag, 1993. R.L. Grossman and A. Nerode and A.P. Ravn and H. Rischel, editors. Lecture Notes in Computer Science 736.

[NSV90] Xavier Nicollin, Joseph Sifakis, and J. Voiron. ATP: an algebra for timed processes. In *Programming Concepts and Methods. Proceedings of the*

IFIP Working Group 2.2/2.3 Working Conference. M. Broy and C.B. Jones (editors), pages 415–442, Sea of Galilee, Israel, 1990.

[NSY92a] Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Compiling real-time specifications into extended automata. *IEEE TSE Special Issue on Real-Time Systems*, 18(9):794–804, September 1992.

[NSY92b] Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. From ATP to timed graphs and hybrid systems. In *Proceedings of the 1991 REX workshop "Real-time: theory in practice"*. Springer-Verlag, 1992. Lecture Notes in Computer Science 600.

[Ost92] J.S. Ostroff. Verification of safety critical systems using TTM/RTTL. In *Proceedings of the 1991 REX Workshop in "Real-Time: Theory in Practice"*. Springer-Verlag, 1992. ed. J.W. de Bakker and C. Huizing and W.P. de Roever and G. Rozenberg. Lecture Notes in Computer Science 600.

[OSY94] A. Olivero, J. Sifakis, and S. Yovine. Using abstractions for the verification of linear hybrid systems. In *Proceedings of Sixth International Conference on Computer-Aided Verification (CAV-94)*, pages 81–94. Springer-Verlag, 1994. Lecture Notes in Computer Science 818.

[PD94] Fong Pong and Michel Dubois. A new approach for the verification of cache coherence protocols. *IEEE Transactions on Parallel and Distributed Systems*, 1994. To appear.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, pages 46–77, 1977.

[Pnu86] Amir Pnueli. Applications of temporal logic to the specification and verification of reactive systems: a survey of current trends. In *Current Trends in Concurrency*, pages 510–584. Springer-Verlag, 1986. Lecture Notes in Computer Science 224.

[PV94] Anuj Puri and Pravin Varaiya. Decidability of hybrid systems with rectangular differential inclusions. In *Proceedings of Sixth International Conference on Computer-Aided Verification (CAV-94)*, pages 95–104. Springer-Verlag, 1994. Lecture Notes in Computer Science 818.

[Ram74] C. Ramchandani. Analysis of asynchronous concurrent systems by timed Petri nets. Technical Report 120, Massachusetts Institute of Technology, February 1974. Project MAC.

[Rok93] Tomas Rokicki. *Representing and Modeling Circuits*. PhD thesis, Department of Computer Science, Stanford University, 1993.

[RR88] G.M. Reed and A.W. Roscoe. A timed model for communicating sequential processes. *Theoretical Computer Science*, 58:249–261, 1988.

[SBM92] Fred B. Schneider, Bard Bloom, and Keith Marzullo. Putting time into proof outlines. In *Proceedings of the 1991 REX Workshop "Real-time: Theory in Practice"*, pages 618–39. Springer-Verlag, 1992. Lecture Notes in Computer Science 600.

[SK91] Bart Selman and H. Kautz. Knowledge compilation using horn approximations. In *Proceedings Ninth National Conference on Artificial Intelligence*, pages 904–909, 1991.

[SS93] Jens Ulrik Skakkebaek and Peter Sestoft. Checking validity of duration calculus formulas, 1993. unpublished manuscript.

[Van93] Peter Vanbeckbergen. *Synthesis of Asynchronous Controllers from Graph-theoretic specifications*. PhD thesis, Katholieke Universiteit Leuven, September 1993.

[VW86] M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of IEEE Symposium on Logic in Computer Science*, Cambridge, 1986.

[WTD94] Howard Wong-Toi and David L. Dill. Approximations for verifying timing properties. In Teo Rus and Charles Rattray, editors, *Theories and Experiences for Real-Time System Development (Proceedings First AMAST Workshop on Real-Time System Development)*, chapter 7. World Scientific Publishing, 1994.

[WTH91] Howard Wong-Toi and Gérard Hoffmann. The control of dense real-time discrete event systems (extended abstract). In *Proceedings of 30th IEEE Conference on Decision and Control*, pages 1527–1528, Brighton, England, December 1991.

[WZ92] Henri B. Weinberg and Lenore D. Zuck. Timed ethernet: Real-time formal specification of ethernet. In *Proceedings of Third International Conference on Concurrency Theory, CONCUR '92*, pages 370–385, Stony Brook, NY, August 1992. Springer-Verlag. Lecture Notes in Computer Science 630.

[Yi90] Wang Yi. Real time behavior of asynchronous agents. In *CONCUR 90: Theories of Concurrency*, pages 502–520. Springer-Verlag, 1990. Lecture Notes in Computer Science 458.

[YL93] Mihalis Yannakakis and David Lee. An efficient algorithm for minimizing real-time transition systems. In *Proceedings of Fifth International Conference on Computer-Aided Verification*, Elounda, Greece, June 1993. Springer-Verlag. Lecture Notes in Computer Science 697.

[YTK91] T. Yoneda, Y. Tohma, and Y. Kondo. Acceleration of timing verification method based on time petri nets. *Systems and Computers in Japan*, 22(12):37–52, 1991.