EFFICIENT GRAPH-BASED ALGORITHMS FOR MODEL CHECKING IN THE MODAL MU-CALCULUS

A Dissertation Presented

by

Oleg Sokolsky

TO THE GRADUATE SCHOOL IN PARTIAL FULFILLMENT

OF THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

IN

COMPUTER SCIENCE

State University of New York at Stony Brook

May 1996

Abstract of the Dissertation

Efficient Graph-Based Algorithms for Model Checking in the Modal Mu-Calculus

by

Oleg Sokolsky

Doctor of Philosophy

in

Computer Science

State University of New York at Stony Brook

1996

Various model-checking algorithms for the alternation-free subset of the modal mu-calculus are considered. The algorithms are based on a data structure called the *product graph*, which represents a kind of Cartesian product of the labeled transition system serving as the specification of a distributed system and a graph representation of the logical formula.

We present several improvements over the basic model-checking algorithm, including a global incremental algorithm, a local algorithm, as well as a real-time extension of the local algorithm.

The work was done in the framework of the Concurrency Factory project, which is also briefly described.

Contents

List of Figures

List of Tables

Acknowledgments

First and foremost, my thanks go to my advisor, Scott Smolka. He possesses the two main qualities that make a professor be a good advisor, mixed in the perfectly balanced proportion: he gives you enough freedom so that you don't feel constrained and dependent when you're on the move, and offers enough guidance so that you don't feel lost and disillusioned when things look down. I am also indebted to Scott for his moral support during these years that were personally hard for me.

I also want to thank the permanent members of the Concurrency Club group, which, along with Scott, included Gene Stark, Andrew Uselton, Fred (Shi-pei) Zhang, Karen Bernstein, Sue-hwey Wu. The Club meetings contributed greatly to my understanding of how good research should be done.

This thesis would have been all but impossible without my friends, both in Stony Brook and dispersed all over the world. My friends are my world, I am nothing without them. Interaction with friends helped me retain at least some taste for life that a graduate student tends to forget *en route* to graduation. Thank you, Katya, Kostis, Simon, Matthias, Lenya, Asya, Lukas, Michael, Dima, Vanja, Sasha, Sanya and Shura.

And last but not least, I owe everything in this world to my family: my wife Valentine, my kids Sasha and Zhenya, my parents. It has been very hard to stay these long years away from you all, but you were my source of inspiration as well as a very compelling motivation to finish as fast as possible.

Chapter 1

Introduction

1.1 Model Checking as a verification technique

The subject of the thesis is efficient verification of distributed systems specified as finite-state *labeled transition systems* (LTS). Verification is performed with respect to properties specified as formulas in a highly expressive temporal logic. The work on the algorithms presented here was carried out as part of the Concurrency Factory project, a graphics-based interactive CASE environment for design and verification of distributed systems.

We start with a brief survey of verification techniques for systems represented as LTSs and then concentrate on model checking as one of the most powerful techniques.

Labeled transition systems [43] have been used for specification of hardware and software systems for a long time. Distributed systems are often specified as networks of LTSs representing separate modules of the system, and such a network can usually be converted into a single LTS (see, e.g. [56]). Moreover, LTSs provide operational semantics for various process calculi, e.g. CCS [54], ACP [11], CSP [39].

Several other formalisms, as well as extensions of the LTS model, are also used for specifications of distributed systems. We mention here only those that are closely related to our approach. A large body of work is based on ω-automata [74], which generalize finite automata on strings and trees to infinite objects. Recent advances in this field include [13, 78], which reduce several verification problem to the problem of deciding emptiness of ω-automata. An extension of the LTSs, called *modal transitions systems* [47], has been developed to facilitate step-by-step property-preserving refinement of specifications.

Verification techniques traditionally include *reachability analysis* [24], where the state space of the system is explored to see if states possessing a certain property can be reached by an execution within the system, and *equivalence checking* [42, 62, 31], which allows one to establish a relation between two systems (usually viewed as specification and implementation for the same problem) using different notions of behavioural equivalence and preorder relations.

In the past decade *model checking* ([18, 19]) has firmly established itself as a verification technique for both reactive systems (e.g. communication protocols) and hardware designs. Using this technique, desirable (or otherwise) properties of distributed systems are expressed as formulas in some logic, and then a model-checking algorithm is used to decide whether the system description satisfies a property (in other words, provides a model for the formula expressing this property).

A number of modal and temporal logics have been put forward for specification of properties of distributed systems. The two major categories of temporal logics are linear-time and branching-time logics. Linear-time logics historically appeared first, namely PTL [63], and are conceptually easier to understand. These logics treat the system as a set of linear execution paths through the state space, and thus every state along each execution path has one possible future. By contrast, branching-time temporal logics, such as

CTL [18], CTL∗ [19], and the propositional modal mu-calculus [44] concentrate on properties of states rather than execution paths. In this case, every state has many possible futures depending on how many execution paths go through this state. In general, branching-time temporal logics are strictly more expressive than their linear time counterparts, but have the same complexity for verification algorithms.

One of the most powerful logics is the *modal mu-calculus* [44]. The modal mu-calculus allows one to specify safety, liveness and fairness properties (see [72]). The logic makes use of explicit fixed-point operators for computing least and greatest fixed points. We focus here on the *alternation-free* fragment of the modal mu-calculus [29] which, intuitively, means that the "level" of mutually recursive greatest and least fixed-point operators is one. This fragment is still powerful enough to express most of the commonly used safety and liveness properties. In particular, it is strictly more expressive than the widely used logic CTL [18]. At the same time, the model-checking problem for this subset is solvable in linear time with respect to the size of the global state space of the verified system.

There are two main types of model-checking algorithms: *global* and *local* ones. Global algorithms assume that the global state space of the system we are trying to verify is already constructed, and then compute the values of all logical variables within the formula for every state of the system. This leads to the (in)famous state explosion problem. Indeed, for most systems of practical interest, the state space is too large to be constructed all at once. However, in order to establish the desired result, it not always necessary to explore the whole state space and compute the values of all variables. Algorithms that try to minimize the work by visiting as few states as possible and constructing fragments of the state space as needed are called local algorithms. The local approach often allows one to reduce the memory requirements of the algorithm. However, the price to be paid for that is at least a logarithmic increase in the worst-case run time.

A number of model-checking algorithms have been developed and implemented in verification tools. The most efficient global algorithms include [29, 23, 82] for the alternation-free mu-calculus. The algorithms in [21, 51] have the best time complexities for the full modal mu-calculus. Several local algorithms exist, e.g. [73, 20, 48, 9, 83, 14] to name just a few. Of these, the first two handle the full mu-calculus, but exhibit exponential worst-case behaviour, even for alternation-free formulas. [48] and [9] provide polynomial solutions for the alternation-free case, while [83] handles formulas of alternation depth two. The algorithm of [14] solves the model-checking problem for the fragments L_1 and L_2 (introduced in [28]).

1.2 Real-Time Verification

Specification of distributed systems as LTSs already contains a rudimentary notion of time in it: only after a transition from its source state to its target state occurs, can a transition of the target state be taken. That is, some time may elapse before the latter transition occurs. This idea is made more explicit in the fact that we use *temporal* logic to specify properties of systems.

However, this notion of time is very imprecise. It may be sufficient to express the idea that some event happened (or was observed) before another event, but the interval between the two can be arbitrarily big or arbitrarily small. Many systems, in practice, must include timing constraints in their specification. Examples of such requirements are "an alarm signal is sent within two seconds of discovering a problem" and "the resource is never idle for more than ten clock cycles". System whose correctness depends on properties with quantitative timing constraints are called *real-time systems*.

The need to specify such systems has led researchers to develop numerous real-time specification paradigms. Virtually every specification language of both a behavioural or logical nature has acquired a real-time extension. We mention only a few of the ones that are conceptually the closest to our work. In the field of process algebras, CCS [56] has several real-time reincarnations, most notably by Moller and Tofts [57, 58], Wang [85] and Čerāns [80]; ACP [11] has been extended to ATP [59] by Nicollin and Sifakis as well as to ACPρ [10] by Baeten and Bergstra; CSP has become Reed and Roscoe's Timed CSP [67]. Labeled transition systems have been extended by Henzinger [37] and modal transition systems by Larsen [47], becoming Timed Modal Specifications [81]. Timed Automata [5] have emerged from ω-automata as a result of the work by Alur and Dill. The I/O automaton model [52] has also acquired real-time versions [53, 76].

A large body of work exists in the field of real-time logics. There are extensions to both linear-time and branchin- time temporal logics. Extensions of liner-time PTL [63] include MTL [8], RTTL [61], TPTL [6] and XCTL [36]. The commonly used branching-time logic CTL [18] has been turned into RTCTL [30] and TCTL [3]. The modal mu-calculus has also been adapted for real-time specifications. T_μ of Henzinger *et al.* [38] is one such extension. Larsen *et al.* in [46] independently from us came up with a real-time mu-calculus that is very close to the one we will be using. A very different version of real-time mu-calculus that uses discrete time (see below) has been put forward by Emerson [27].

We now outline the main design decisions in constructing a real-time specification paradigm and motivate our choices. The first choice to make is discrete time vs. dense time. In the discrete time approach, events can happen only at fixed moments in time. Usually, a distinguished event is introduced, representing a tick of a global clock. All events occurring between two clock ticks are assumed to happen at the same time. This approach is quite natural for description of synchronous hardware systems. Discrete time models have the additional advantage that the resulting specifications are finite-state, and the usual finite-state verification methods are applicable. The discrete time approach has also been successfully applied to specification of many asynchronous systems. In this case, however, we have to approximate the occurrence of events by the nearest clock tick. As pointed out in [2], this can lead to subtle timing errors. Of course, by making clock ticks shorter, we can always make these approximations as close to the actual occurrence of events as is needed to detect the problem. Making the clock ticks happen more often, though, makes the specification considerably larger; in many cases, unnecessarily large. It is difficult in practice to tell in advance what time quantum will be sufficient for reliable verification.

Using a dense time domain, for example the real numbers, alleviates the problem of missing timing errors due to the imprecision of the specification. However, the state space of the specified system becomes uncountably infinite in all interesting cases. This raises new decidability questions and forces us to work with some finite representation of the state space. See, e.g. [5, 79] for some of the decidability issues in dense-time specifications.

Another design decision that we have to make concerns making references to specific moments in time in our specifications. Consider behavioural specifications first. We may either introduce distinguished *delay* events into specifications or constrain the moments when ordinary events may happen. The former choice is commonly used in real-time process algebras. It has an advantage that we do not have to refer to time instances directly. Instead, the current time implicit in the specification is incremented when a delay event happens. Additionally, this concept fits well with the notions of prefixing and sequential composition of events used in process algebras. In the other approach, used more often in automata-based real-time formalisms, timing constraints are used to restrict the moments in time when an event can occur. When we decide to employ the latter approach, we have yet another decision to make concerning the meaning of references to time instances in constraints. One approach, used, e.g. in [37], is to interpret constraints on a transition with respect to the moment when the system entered the source state of the transition. Thus lower bounds on a constraint naturally correspond to delays and upper bounds represent timeouts. This approach makes specifications simpler; however we have to sacrifice expressive power: we do not have the ability to accumulate delays over sequences of events. This deficiency is remedied by using explicit clocks or timers. Clocks are variables that take their values from the time domain and increase spontaneously as time progresses. All clocks are incremented at the same rate. Clocks can be reset to zero when transitions take place. This is, obviously, a generalization of a previous approach, which now can be modeled by using a single clock that is reset by every transition. The high expressiveness of this model has, of course, its price to pay: verification problems become exponential in the number of clocks used in the specification.

Similar issues exist in choosing temporal operators for real-time logics. One possibility that is often employed is using time-bounded versions of temporal operators, i.e. instead of "eventually" we would say "eventually within 5 seconds". Bounded temporal operators are similar to the use of an implicit clock in behavioural specifications, having the same advantages and disadvantages. If we want more expressive specifications, we have to use time variables in formulas. These time variables have to be made bound in some way. One possibility, as in [61, 36], is to use a first-order logic with quantifiers ranging over the values of time variables. As we want to stay within propositional logics, we prefer *reset* operators that do not seem to introduce as much complexity as first-order quantification does (see [7]), and also tend to make specifications more readable.

We may go one step further and allow clocks to increase at different rates. This would allow us to model such phenomena as clock drifts, which is an important issue in the design of real-time systems. Doing so brings us into the realm of *hybrid systems* [35]. Detailed consideration of hybrid systems is well beyond the scope of this thesis and I will only mention that, as one may expect, verification problems for hybrid systems are even harder than those for dense real-time systems. In fact, many seemingly minor extensions render verification problems undecidable.

To summarize, we use dense time represented by real numbers as our time domain. The behavioural model is timed automata with progress conditions (see Section 6.1), equipped with sets of clocks. Timing constraints over these clocks are associated with states and transitions of the automaton. We use an extension of modal mu-calculus for logical specifications (Section 6.2) that also employs clocks as time variables and uses timing constraints as propositions. The similarity in treatment of clocks in behavioural and logical specifications is exploited by the model-checking algorithm (Section 6.4).

1.3 Summary of Contributions

The algorithms presented in this paper solve the model-checking problem by constructing a data structure representing the "product" of the labeled transition system and the logical formula. We call this data structure a *product graph*, and the algorithms based on it *graph-based*. This approach is similar to the automata-theoretic approach of [77, 13]. Alternative approaches include tableau-based [73, 20] and symbolic [16] algorithms.

The work has been done in the context of the Concurrency Factory project, which is a CASE tool for specification and verification of distributed systems. The Concurrency Factory uses model checking as one of the main verification facilities. We have designed and implemented several model-checking algorithms to be used with the Concurrency Factory.

The four algorithms given in this paper are as follows. We start with reformulating the algorithm by Cleaveland and Steffen [23] as a graph-based algorithm. Henceforth, we refer to this algorithm as *CS*. The presentation of *CS* introduces the product graph and illustrates the basic ideas of fixed-point computation on product graphs used in all subsequent algorithms. We build upon this algorithm to develop an *incremental* global model-checking algorithm *MCI* (for Model Checking Incrementally). A variety of incremental algorithms for a wide range of graph problems have been presented in the literature. Among the ones related to ours are incremental dataflow algorithms, e.g. [64]. Incremental algorithms in general strive to minimize the amount of work needed in the repeated invocation of the same algorithms when small changes to the problem are being made. This is extremely important for an interactive environment such as the Concurrency Factory.

Next, a local model-checking algorithm *MCL* is given, which can be seen as an alternative implementation of the ideas of [9]. While not being overly important by itself, it serves as the basis for later developments. The algorithm demonstrates how to combine on-the-fly construction of the product graph with computation of fixed points.

Finally, we consider an application of this local approach to the verification of *real-time* systems. For this, we present a real-time extension of the modal mu-calculus, which can be used to reason quantitatively about timing properties of the system. We use a dialect of *timed automata* [5] to specify real-time systems. The extension of the modal mu-calculus that we are dealing with, although not the first one to appear in the literature (see, e.g. [38]), has several properties that make it interesting. First, it is very closely related to the commonly used version of the untimed mu-calculus. Second, there is a clear separation of real-time issues from time-independent ones in the semantics for the operators of the logic. Because of this, we can build upon our untimed local model-checking algorithm.

A local model-checking algorithm *MCT* is provided for verification of real-time systems. While not as abstract as some other real-time model-checking algorithms ([4, 38]), which are based on set-theoretic transformations, *MCT* is hopefully easier to understand and reasonably efficient. For example, it allows the timed automaton representing the real-time system itself to be constructed on-the-fly. To the best of our knowledge, this is the first local model-checking algorithm for real-time systems in the literature.

The rest of the dissertation is organized as follows. Chapter 2 introduces the Concurrency Factory project. Chapter 3 gives the necessary background for the labeled transition systems and their model checking in the modal mu-calculus. Chapter 4 presents the global algorithm for incremental model checking. The purpose of Chapter 5 is to introduce local model checking. Chapter 6 then builds upon it to exhibit a local algorithm for verification of real-time systems. Finally, Chapter 7 concludes and discusses possible directions of further work.

Parts of Chapter 2 were adapted from [22]. Results of Chapter 4 were first reported in [69] and those of Chapter 6, in [70].

Chapter 2

The Concurrency Factory Project

2.1 Main Features and Design Goals

The *Concurrency Factory* is an integrated toolset for specification, simulation, verification, and implementation of concurrent systems such as communication protocols and process control systems. The project is a joint effort between SUNY Stony Brook and North Carolina State University and officially started in Spring '92.

The main features of the Concurrency Factory are the following:

- A *graphical user interface* includes the graphical language GCCS [75], which allows one to design concurrent systems as hierarchically structured networks of finite-state processes. A graphical editor for GCCS specifications is provided. GCCS specifications have been given formal semantics by translation into CCS [56]. The formal semantics serves as the basis for the graphical simulator [41], as well as for all verification algorithms.

- A *textual user interface* for the language VPL, a simple language for concurrent processes that communicate values from a finite data domain (integers of bounded size). The usual high-level programming language constructs for expression evaluation, assignment of values to variables, conditional and iteration operators are provided. Composite data types such as arrays and records are also supported. A VPL compiler translates VPL programs into networks of finite-state processes.

- A *set of objects* for representation of system specifications. Both GCCS specifications and compiled VPL programs share the same internal representation. Operations defined on these objects include a set of "semantic routines" that capture the operational semantics of GCCS. These routines form a uniform interface for verification and simulation algorithms and allow the set of objects to be modified without changing existing verification and simulation routines.

- A *suite of design and analysis routines* that currently includes a linear-time global model checker for the alternation-free modal mu-calculus [23] (see Section 3.2), a local algorithm (Section 5) and a bisimulation checker based on the algorithm of [42].

- A *graphical compiler* that transforms GCCS and VPL specifications into executable code. Our current version produces Facile [33] code, a concurrent language that symmetrically integrates many of the features of Standard ML [55] and CCS [56].

While designing the Concurrency Factory, we made extensive use of the object-oriented approach. The core of the tool is formed by the set of objects used to represent the user designs. This set of objects is used by the user interface as well as all verification, simulation and compilation modules. The main goal of the object design was to provide an implementation-independent interface to the objects. It was expected, and turned out to be true as the project unfolded, that representation of systems will change, which should have as little impact on the algorithms working with these systems as possible. Another important goal was to achieve extensibility of the set of objects. The object design was to provide for the future extensions of the Concurrency Factory. One such extension, allowing one to use real-time constraints in the specification, is discussed in Section 2.4.

2.2 Internal Representation of Systems

The Concurrency Factory represents systems as sets of interacting objects. We will refer to them henceforth as CF objects. The interface to the objects is the same for all tools of the Concurrency Factory. This section briefly describes the objects of the Concurrency Factory.

The class hierarchy for the main type of CF objects in the Factory, *systems*, is depicted in Figure 1. Dashed arrows represent subclass relationships — e.g., `Process` is a subclass of `System` — and solid arrows represents containment relationships — e.g., a `Process` object contains one or more `State` objects. To enhance readability, we have omitted some non-essential objects and containment arrows.

Let us summarize the main ideas used in design of the CF objects. A `System` is either a `Network` or a `Process`. A `Network`, in turn, consists of one or more interconnected sub-`System`s. Thus `System`s belonging to the same specification are organized in a tree hierarchy, internal nodes of which are formed by `Network` objects and leaves of which are `Process`es. Sub-`System`s of the same `Network` are interconnected by embedding each of them in an `Interface` which defines the set of `Port`s a `System` can use to communicate with other `System`s. Two or more `System`s can communicate if they each possess a `Port` connected by a `Link` to a common `Bus`. A `Process` consists of

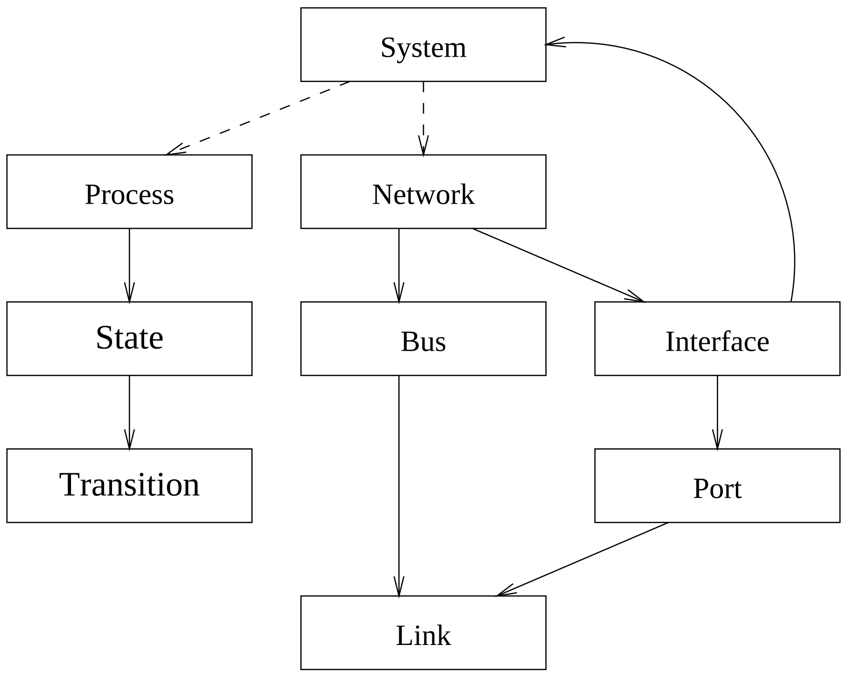


Figure 1: Class hierarchy for the Concurrency Factory objects.

one or more `State`s, each of which has some number of incoming and outgoing `Transition`s.

An example system is depicted in Figure 2, which we use to clarify the preceding discussion. The depicted system represents the arbiter of a mutual exclusion protocol. It consists of the control process that performs arbitration and three auxiliary processes that keep track of the state of the protocol. Consider `arb`, the arbitration process in Figure 2.a. It has a number of ports that specify the communication channels. Some of them are used in interaction with the auxiliary processes, as can be seen by the incident links. This interaction is meant to be local to the arbiter. Thus these ports are connected to unlabeled buses which makes communication on the ports invisible from outside the network. Ports that are not connected to buses, as in this example, or connected to named buses, can be used in communication with the environment of a network. In this case, two external processes competing for access to the critical section use `try, enter, and exit` actions to signal their intentions.

If we look now inside the control process of the arbiter (Figure 2.b), we see a collection of states and transitions that form a labeled transition system (LTS) capturing the behaviour of the component. The following notational conventions are used. A double circle represents a start state; in the transition labels, a leading apostrophe ("'") sign represents an output action. During communication, input actions synchronize with output actions. For an action to be allowed to happen, its name (without input/output distinctions) should appear as a port label on the enclosing interface.

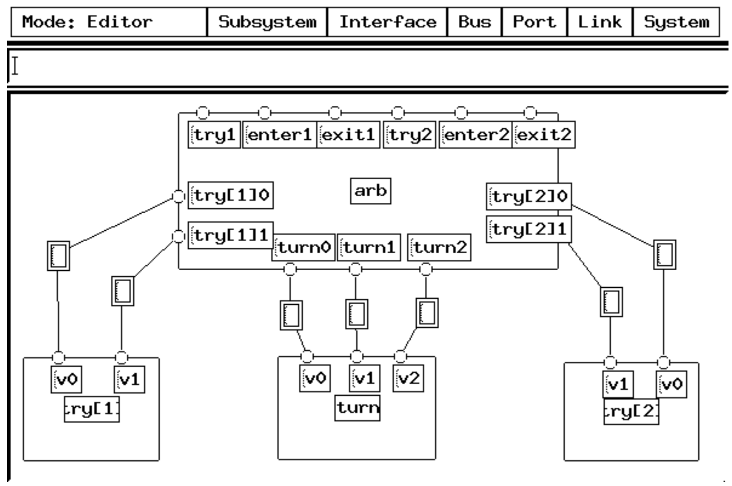
The hierarchy presented in Figure 1 depicts *logical objects*. This means that they only contain information that is essential to capture the structure of the systems we are describing and the most basic features of them. Additional features are added to CF objects using the inheritance mechanism, thus further extending the object hierarchy. The most prominent extension defined so far provides for the graphical capabilities of the tool. All visible CF objects are enhanced to contain information about the graphical representation of the objects and methods for displaying and manipulating them on the screen. The information is distributed over the set of CF objects in such a way that each object knows "how to display itself". We do not go into details of the graphical objects here.

We have paid particular attention in our design to the encapsulation of processes and networks as abstract data types. As an illustration of this, consider again the class `Process`. The prototypes for two of the methods we provide for this class are as follows:

```
Transition*  Process::AddTransition(String &act,
                                    State *src, State *tgt);
void         Process::DeleteTransition(Transition *t);
```

The first method creates a new `Transition` object from a source state `src` to a target state `tgt` labeled by string `act`. The newly created transition will automatically be added to the source state's set of outgoing transitions and to the target state's set of incoming transitions. Note that this is the only way a user can add transitions to these sets. The net result is a well-defined `Process` object with the new transition added, as desired. Similarly, the second method deletes a `Transition` object and all related objects so that the resulting `Process` is again in a well-defined state.

CF objects were given formal semantics in [41] by means of translation into CCS. However, this is not the only way to interpret these specifications. It is quite conceivable that future developments of the project will lead to new kinds of objects, or to different interpretation of existing ones (e.g. CSP-style multi-party synchronization instead of CCS rendez-vous). This means that we had to provide means to adjust the semantics of the objects so that different modules of the Factory (e.g. verification routines and the simulator) a)



b)

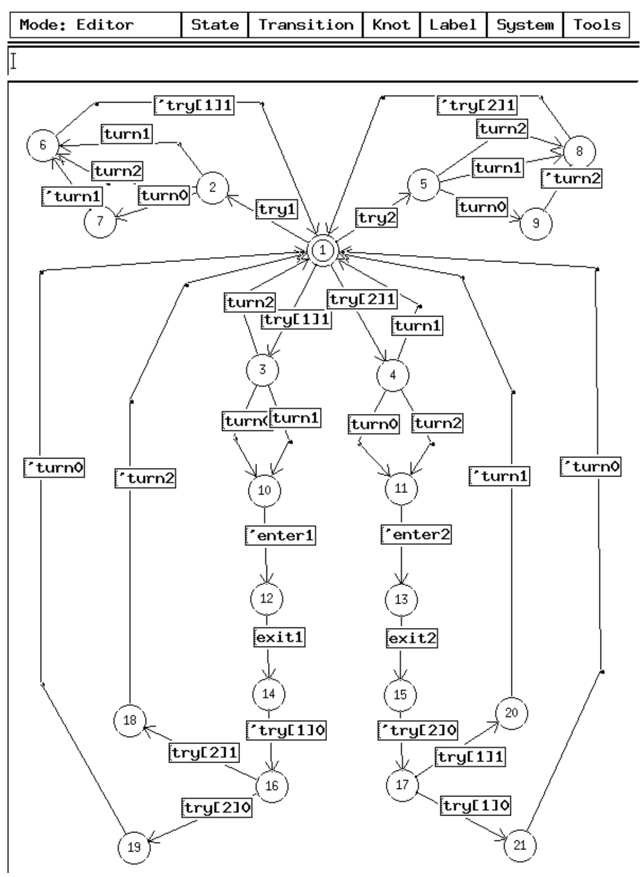


Figure 2: Representation of a mutual exclusion protocol.

can manipulate them in a uniform way. For that purpose we provided a set of methods, i.e. abstract operations on CF objects that reflect their semantics.

The key idea in this approach was that, regardless of what kind of objects we use in our specifications and what behaviour they exhibit, every system we are working with has a number of implicit states in it. During execution the system would evolve from state to state by executing transitions. Auxiliary objects `SystemState` and `SystemTransition` capture this idea. For a simple system (i.e. `Process`), these coincide with ordinary states and transitions. The state of a `Network` is a tuple of the states of processes within this `Network`. Similarly, a global transition of a `Network` is composed by moves in one or more of its sub-`Systems`. `SystemTransition` objects allow one to tell which of the sub-`Systems` of the `Network` participated in the move and what those moves were.

Every CF object derived from the `System` class is required to have the following methods, which provide the common interface to CF objects for all routines that are interested in their semantics:

```
SystemState*             System::CurrentState();
void                     System::SetState( SystemState* );
Set<SystemTransition>*   System::EnabledTransitions();
void                     System::Execute( SystemTransition* );
SystemState*             System::Target( SystemTransition* );
```

The first two of these methods allow one to obtain the current state of the `System` and set the current state as desired. `EnabledTransitions` is used to find the global transitions that are available from the current global state. Given a `SystemTransition` that is enabled in the current state of the `System`, `Execute` makes the system move to the target state of that transition, while `Target` tells what that state would be, without taking the step.

A set of internal methods is provided for computation of enabled transitions of a `Network`. Every CF object used in the system specification (`Bus, Port,` `Interface`) is supplied with methods that tell how this object affects enabled transitions. This allows one to change semantics of individual objects without affecting the rest (e.g. the above-mentioned switch from CCS-like synchronization to CSP-like one will only affect semantic methods of the `Bus` object). The following conventions were used in determining the set of enabled transitions. Every move of a network is either an individual one, taken by one of the subsystems on its own, or a synchronization move, which is taken by several subsystems in cooperation. It is assumed that each sub-`System` can decide on its individual moves and every synchronization transition is associated with a certain `Bus`. This suggests the implementation approach: the method thus will first query each of the subsystems to obtain the sets of transitions enabled in each of them, and then query buses for the synchronization transitions. The set of enabled transitions for the network, then, is the union of all transitions thus obtained.

This modular approach proved itself useful in the specification of the SPARC cache coherency protocol [17]. To model the protocol, it turned out to be necessary to extend the set of objects by adding a synchronous bus (multi-party synchronization), where all subsystems connected by links to the bus are required to participate in a global transition, as well as a more elaborate *priority* bus that allows a low-priority action to happen only if there is no high-priority action enabled.

2.3 VPL: a Value Passing Language

The graphical interface of the Concurrency Factory allows the user to design pictorial specifications of distributed systems in a hierarchical fashion. For smaller systems, it gives a manageable and understandable representation. When we want to use the Concurrency Factory to specify systems with involved control structure and non-trivial data structures, e.g. communication protocols, different means of specification are desirable. To provide for this, we propose a language that offers high-level constructs that would be useful in specification of protocols and the like. The language VPL serves as a textual interface to the Concurrency Factory objects and at the same time provides for higher-level specifications. VPL is very close in its nature to $\mathcal{M}_0$ of [56]; however there are several differences. In particular, complex data structures such as arrays and records are provided.

Control constructs include those commonly found in programming languages, e.g. constructs for conditional execution and iteration. Data structures supported by the language are integers of limited size and arrays and records composed of these.

The general structure of a system specification in VPL and even the terminology is made consistent with the CF objects. That is, a specification is a hierarchy of subsystems. Top levels of the hierarchy are formed by networks. Each network consists of several subsystems running in parallel that can communicate with each other through channels. The leaves of the hierarchy are formed by processes. The main conceptual difference between VPL processes and CF processes is the presence of typed data values. What appears to the user as a simple VPL process is a CF network consisting of a control CF process composed in parallel with CF processes representing data variables of the VPL process. Concurrent components in a VPL network can interact with each other using typed channels that allow them to communicate values of certain types. The correspondence between VPL typed channels and CF buses is discussed below.

The following sections give semantics to the constructs of VPL by presenting their translations into (the graphical representations of) CF objects. Since the CF objects themselves have process-algebraic semantics, this makes the VPL objects formal. To avoid confusion between similarly named objects, we will make a careful distinction between the VPL objects manipulated by VPL programs and the CF objects that serve as their internal representations.

2.3.1 Notes on Syntax

The following names are treated as keywords and cannot be used as identifiers: `array`, `begin`, `call`, `channel`, `do`, `else`, `end`, `if`, `network`, `of`, `procedure`, `process`, `record`, `select`, `skip`, `then`, `type`, `value`, `var`, `while`. VPL is case insensitive.

In addition, the following names are treated as predefined types and constants and cannot be used otherwise: `boolean, sync, false, true, tau`. The first four are discussed in Section 2.3.2. The name `tau` is used internally to represent τ, the silent transition of CCS. Using `tau` as a channel name would, in some cases, allow the user to introduce the silent transitions into the underlying LTS arbitrarily. We felt that every silent transition in the LTS should correspond to some internal computation, which is to be expressed by the regular means of the language.

The syntax of individual constructs is presented in the following sections together with their semantics.

Any text placed within matching braces "{ ... }" is treated as comments.

2.3.2 Types and constants

The basic data type in VPL is the unsigned integer of limited size. The following declaration defines `byte` as an integer less than 256:

```
type byte :  256
```

There are two predefined types in VPL: *boolean* and *sync*, which have the following implicit definitions:

```
type boolean :  2
```

```
type sync : 0
```

The latter is used for pure synchronization (without value passing) between two processes.

Complex data types can be constructed as records. A record is a collection of fields with specified types. The type of a field can be either a previously defined data type or a special type `channel` which will be discussed later. An example of a record definition is given below:

```
type rec : record
  a : byte ;
  b : boolean ;
end
```

Types defined in a program are used to declare variables (see Section 2.3.3). Arrays of variables of a certain type may be declared. It is sometimes useful to have arrays of fixed length appear as fields in records. However, in the current implementation, arrays do not form new data types. As each field in a record must have a type, arrays cannot be used to specify fields in record declarations.

VPL implements structural equivalence between types. That is, two objects have equivalent types if they either are integers of the same size or records containing fields of equivalent types in the same order. Names of fields in a record are irrelevant for determining type equivalence.

VPL allows integer constants. Constants are declared using the `value` construct: for example, `value TWO : 2` installs `TWO` as a symbolic name for number 2. There are two predefined constants, `true` and `false`, having values respectively 1 and 0.

2.3.3 Syntax and Semantics of Constructs

Networks and Processes

The main unit in VPL is a network, which contains definitions of its parts (*subsystems*) and specifies how they are connected to each other. Subsystems can be either other VPL networks or VPL processes. Recursion in VPL network specifications is not allowed. VPL networks and processes communicate through typed channels.

The definition of a VPL process or network consists of three parts: header, local declarations and body. The header, in both cases, specifies the name of the object and its interface to other subsystems. An interface specification is a sequence of channel names with their respective types. Channel names appearing in the header can be used in the subsystem body. We refer to the channels introduced in the header of a network as *formal channels* of the network, by analogy with formal parameters of procedures in conventional programming languages.

The following local declarations can appear in a VPL network:

- Type declarations.
- VPL processes and networks to be used as subsystems. Such declarations must appear together with their bodies.
- Local channels.

The body of a network is a parallel composition of its subsystems using the parallel composition operator |. Each subsystem declared within a network can be used several times. Each time, a new copy of the subsystem will be instantiated. This allows one to specify systems with multiple identical components without duplication of code. When a subsystem is used, a list of channels that the subsystem can use in communication is provided. Extending the analogy with procedure parameters, this list of channels can be called *actual channels*. The actual channels for a subsystem appearing in the body of a network must be valid channel names - declared either in the header of the network or as local channels in the local declaration section. The types of actual channels must match the types of the formal channels in the subsystem declaration. Effectively, the formal channels of the subsystem are renamed to those given as actual parameters.

A VPL network is translated into a CF network so that every component of the parallel composition instantiates a new copy of the specified subsystem according to its definition. Formal channels of a subsystem appear as ports on the interface for that subsystem. Local channels are translated into buses. When a channel is used as an actual parameter, a link from the bus representing the actual channel to the port corresponding to the formal channel is introduced.

Consider the following example. It may represent a system which performs some service for external users (represented as the `ext` action) and consists of a front-end that communicates with users and a back-end that carries out the computation. The two parts synchronize with each other through a local channel `act`. Such system is represented by the following VPL program:

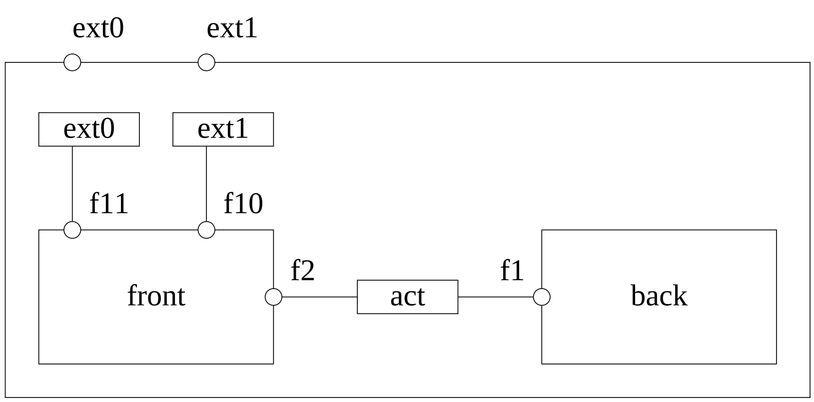
```
network server( ext:boolean )
begin
  process front( f1:boolean, f2:sync )
    { body of process front }

  process back( f1:sync )
    { body of process back }

  channel act:sync
```

```
  front( ext, act ) | back( act )
end
```

Channel `ext`, used for communication with the users, is represented as a formal parameter for the program. Channel `act` is declared in the local declarations section of the `server` network using `channel` construct. Since `act` is a local channel not mentioned in the interface of the network, communications along this channel are invisible outside `server`. The program is translated into the set of CF objects below. Each of the two constituent processes is represented as a subsystem. The ports on the interfaces correspond to the formal parameters of the processes. Note how a single boolean channel is translated into two ports, names of which represent values being sent.



Subsystems of a VPL network in form a tree hierarchy. As mentioned above, multiple occurrences of a subsystem are treated as disjoint copies of the subsystem bodies. The leaves of this hierarchy are formed by processes. While networks only define the modular structure of the system, the actual computation is carried out in processes.

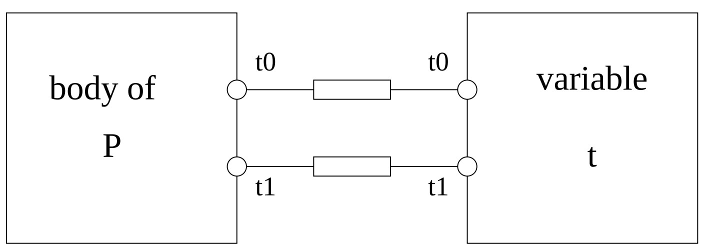
Local declarations of a VPL process may include the following:

- Type declarations.

- Variable declarations.
- Procedure declarations.

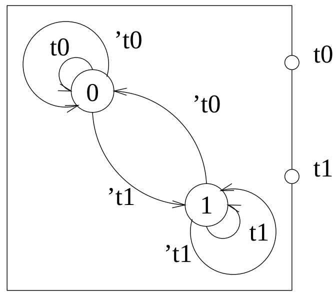
The body of a VPL process is a sequence of statements. Different kinds of statements and their translations are discussed below. A process declaration is translated into a CF *network*, which consists of a CF process representing the control part of the VPL process, and a CF process for each variable declared within a VPL process. The example below shows the structure of a VPL process `P` consisting of a body and a single variable.

```
process P ( ... ) begin
  var t: boolean
  { process body }
end ;
```



Variables

A variable of a simple type of size n is translated into a process having n states. Every state has an output transition to itself, representing the read operation, and incoming input transitions from every state, including itself, representing writes. The following example illustrates the representation of a variable declared as `var t: boolean`.



To help understand the picture above, each state is labeled with a number corresponding to the respective value of the variable it represents. Recall that actions whose labels begin with "'" represent input events.

Arrays are collections of variables of the same type. An array of 10 boolean values is declared by `var ba: array [10] of boolean`. Like all other variables, arrays are declared statically and thus must have constant expressions as their sizes.

Arrays of variables and variables of record types are represented as sets of simple variables with an appropriate adjustment of names. Thus, the declaration of `d` as a variable of type `rec`:

```
type rec :  record
  a :  2 ;
  b :  3 ;
end
var d :  rec
```

is translated into the following two declarations of integer type:

```
var d_a :  2
var d_b :  3
```

In fact, `d_a` and `d_b` are the names generated internally for `d.a` and `d.b` respectively in the current implementation[1].

[1]The reason for this transformation is that all internal names have to be valid C

It may appear that these two kinds of declarations are equivalent and the record type is redundant, but this is not the case. The difference between these two kinds of declarations is that records can be transmitted through a channel of a matching type using a single VPL statement, while a sequence of output or input statements over different channels would be required to transfer the corresponding set of individual variables. It should be noted, though, that the operation of transmitting a message through a channel of complex type is not atomic and transmission of individual fields of the message will be interleaved with activities in other subsystems.

When a reference to a value of a simple variable is made, a transition is introduced into the LTS of the VPL control process, labeled by an input action if the value is read and an output action if the value is written. The name of the action is formed by concatenating the internal name for the variable with the integer value. For example, reading 0 from variable `a` results in action name `'a0`, while for `d.a` above the corresponding action name would be `'d_a0`. Internal names for values of array variables are of the form `c_i_j`, which corresponds to writing the value j into variable `c[i]`.

Currently, the VPL compiler makes one pass through the program. Thus declaration of every variable has to appear before the first use of the variable. This happens naturally in most cases, since variables are used in the body of a process, which comes after the declaration section. The only exception is procedure bodies (see below).

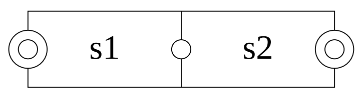
Statements

Every VPL statement is translated into a labeled transition system with single start and end states. Henceforth, when representing statements, we will denote start and end states as double circles. A generic statement will be represented

identifiers.

as a box with the start state on the left of the box and the end state on the right. In every isolated statement, its end state is terminal, i.e. has no outgoing transitions and the start state is initial, i.e. has no incoming transitions. This can be easily verified by looking at the translation schemes for different kinds of statements represented in the following subsections. This fact is important for establishing correctness of conditional and nondeterministic constructs.

Statements can be composed sequentially. The statement separator ";" is used as the sequential composition operator. `s1 ; s2` results in merging the start state of `s2` with the end state of `s1`:



A sequence of statements can be treated as a single statement by means of the composite statement `begin ... end`. The following kinds of statements are defined.

Skip statement

`skip` is a statement corresponding to *inaction* in process algebras. It is rather like `SKIP` of CSP [39] than *nil* of CCS: since we are dealing with sequential composition rather than prefixing, it is possible to compose `skip` with another process to get a non-deadlocked system. For every statement `s`, `skip; s` is equivalent to `s`.

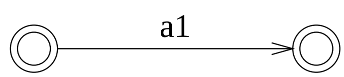
`skip` is represented as a single state, i.e. its start and end states coincide, no new transitions are introduced.

Assignment statement

The assignment operator `:=` allows one to assign the value of an expression to a variable of simple type, or to copy a variable of complex type into another variable of an equivalent type.

First, the right-hand side expression is processed to obtain a set of values that may result from its evaluation (details on handling of expressions are given below). Then, for each possible outcome resulting in a distinct value, a transition assigning that value to the variable on the left-hand side is introduced. The values are taken modulo the size of the variable they are assigned to. The simplest case of the assignment – when the right-hand side is a constant – is illustrated below:

```
a := 1
```



If the range of values of the right-hand side expression exceeds the size of the variable, the assignment is carried out modulo the variable size. A more involved example of assignment is given in the paragraph describing expression evaluation.

Input/output statements

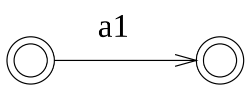
Input/output statements specify communication between subsystems through channels. Historically, the syntax for input from a channel involves operator "?" and operator "!" for output into a channel. Here we follow the same convention. `c1 ! expr` allows one to send the value of expression `expr` to channel `c1`. Similarly, `c2 ? var` reads a value from channel `c2` into the variable `var`. Types of the `expr` and `c1`, as well as types of `c2` and `var`, must match.

In the same way as assignments provide for copying of complex variables within a process, a channel can have a complex type, in which case an arbitrarily complex record may be communicated in one statement.

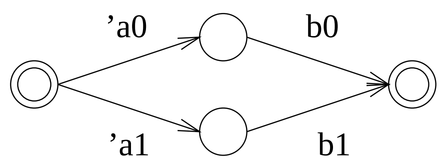
These statements are very similar to the assignment statement, except one of the sides must be a channel name. Here are simple translations:

```
channel  a :  boolean
var      b :  boolean
```

`a!1`



`a?b`



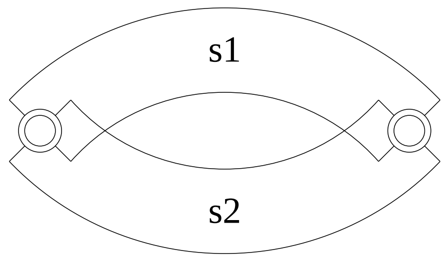
If the channel is used for pure synchronization, i.e. no values are passed, it is declared to be of type `sync`. A special value "*" is used instead of the source expression or target variable. If a channel `c` is of type `sync`, `c?*` translates into a transition labeled `'c` and `c!*` gives rise to `c`.

Select statement

The select statement provides means of nondeterministic execution in the program and is useful to specify environments for a system, e.g. media in protocol specifications. Syntactically, the statement combines an arbitrary number of choices `s1, s2, ...`, each of which is a single statement. Composite statements enclosed in `begin ... end` brackets may be used as choices. The choices are separated by "%" operator.

During the translation into an LTS, the statement joins the start states of the component statements together, as well as their end states. For example, the picture below translates the following portion of code:

```
select s1 % s2 end
```



Expressions

Before introducing the last two kinds of statements, namely conditional and iterative constructs, we discuss how arithmetic and boolean expressions are represented.

Expressions in VPL statements have to be "evaluated" at compile time in order to generate correct transition labels that handle results. Effectively, a decision tree is constructed, having a branch for every value that a variable appearing in the expression may have at run time. An expression containing n distinct variables is translated into a tree LTS with n levels (see example below). Each level corresponds to reading one of the variables. Since expressions do not have side effects, each variable is "read" only once.

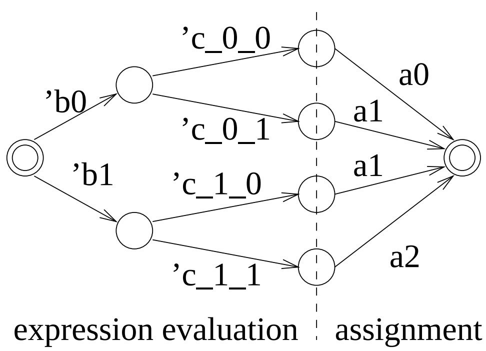
One can think of the states of this tree LTS as being labeled by values of variables read along the path leading from the start state to the current one. At each leaf of the tree, evaluation of the expression is assumed to have occured, so that subsequent transitions may use the resulting value.

Expressions in array indices are evaluated as parts of the expression containing the array variable, i.e. a variable appearing in the index and elsewhere in the expression will be "read" once.

Boolean expressions are intentionally kept separate from all other expressions. This is to guarantee that no values other than boolean are possible when we evaluate conditions of `if` and `while` statements, and also to avoid unnatural expressions for array indices.

The following operators can be used to construct expressions: arithmetical operators: $+, -, *, /$, boolean connectives: & (and), | (or), ~ (not). Relations $=, >, <$ can be used as elementary boolean expressions. The list of operators is likely to be expanded in the future versions.

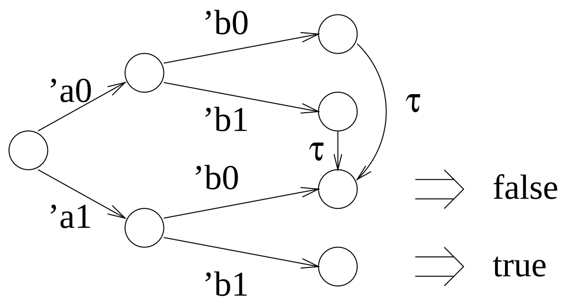
The following example shows the translation of `a := b + c[b]`. Assume for simplicity that `b` is a boolean variable, `c` is an array of boolean variables, and `a` is large enough to hold the result, i.e. at least of size 3.



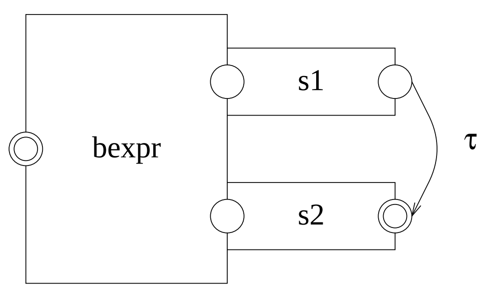
The left side of the picture represents expression `b + c[b]`. Note that when the value of `b` happens to be 0 (corresponds to `'b0` transition), variable `c_0_` is accessed, which corresponds to `c[0]`. The right side of the picture represents assignment to variable `a`. One can check that the value being assigned is indeed the sum of the values read from the two variables in the expression.

Conditional statement

The conditional statement has the form `if bexpr then s1 else s2`. Here `bexpr` is a boolean expression that translates into a tree LTS as described above. The syntax of VPL guarantees that leaves of the resulting LTS correspond only to boolean values. The tree LTS is transformed into an LTS with only two terminal states by connecting the states where the expression is true (resp. false) by introducing τ transitions. As an example of this transformation, consider expression `a & b`, where `a` and `b` are boolean variables. Of the four leaves of the resulting tree LTS, only the one corresponding to true values of `a` and `b` yields the value true. The other three correspond to value false and are grouped together.

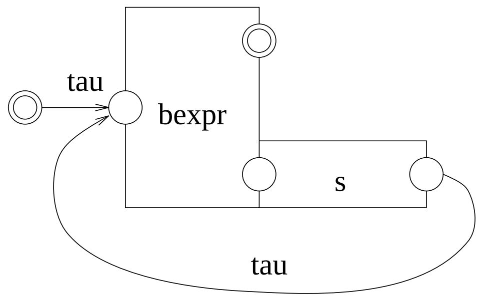


After that, statements corresponding to the true and false branches are attached to the respective end states. Finally, end states of `s1` and `s2` are connected by a τ transition to make a well-formed LTS (i.e. with only one end state).



While statement

The intended meaning of the `while` statement is the usual one: it represents a loop whose body is repeated as long as the guarding boolean expression remains true. The body of a loop is a sequence of statements. The following picture represents the translation of `while bexpr do s end`:



A leading τ transition is introduced to prevent incorrect behaviour in the case when the `while` construct is put into the context of `select`. This transition can be a source of inefficiency since it is not needed in other contexts. Future versions of the VPL compiler should detect this case and only introduce this transition when it is needed. As with the conditional statement, the boolean expression is transformed into one with two terminal states. The state corresponding to false becomes the end state of the `while` statement as this means that the guarding condition is no longer valid. The true branch is used to close the loop.

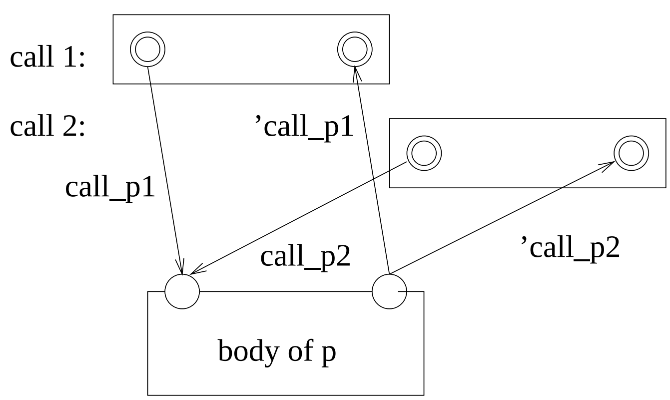
Procedures

Procedures in VPL are parameterless sequences of statements. Each VPL procedure is declared within a VPL process and is local to that process. The statements of the procedure body are allowed to use variables defined in the enclosing process. At the moment, procedures cannot have local variables. It is possible, however, that in the future this restriction will be relaxed to allow variables local to a procedure.

Procedures are declared in the local declarations section of a process. Procedure bodies have to be specified together with their declarations. As mentioned in Section 2.3.3, every reference to a variable must occur after the variable is declared. A procedure declaration thus has to appear after the declaration of any variable it mentions, but before the body of the process starts.

The body of a procedure forms an initially isolated fragment of the state space of the process it belongs to. Since the body of a procedure is a sequential composition of statements, it has unique start and end states. The labels of these states are recorded in the symbol table entry for the procedure name.

The procedure body is shared by all the calls to that procedure. To make this sharing work correctly, we have to simulate the call/return mechanism of programming languages. To make it simple, we assign numbers to syntactic occurrences of calls of every procedure in the program and introduce an internal variable that keeps the number of call that is being made. Whenever a call to the procedure is made, a transition that sets the variable to the number assigned to that call is created, connecting the last state of the previous statement to the start state of the procedure body, and another transition that reads the same value from that variable connects the end state of the procedure body to a new state which becomes the end state of the call statement. The following picture describes the LTS produced by the compiler corresponding to two calls to procedure `p`, which were the first and second calls encountered by the compiler.



It is clear that the described scheme does not provide for recursive calls, since no stack of procedure calls is kept. In general, recursive calls can make the state space of a process infinite. Since we construct the whole state space of every VPL process, this makes full implementation of recursive calls infeasible. However, recursive calls of bounded depth can be provided by introducing a more elaborate call mechanism. This is one of possible future extensions to VPL.

2.3.4 Example: Medium Specification

The following example illustrates the use of the less obvious constructs of VPL. It describes the faulty medium that may be used in verification of a communication protocol. The medium transfers packets from a sender to a receiver, which are not shown here. Each packet has a packet type, a sequence number, a data part and a parity check field. The following simplifications are made: the data part, the contents of which is assumed to be unimportant to the operation of the protocol, is omitted. It is also assumed that the sender always sends the correct data and the parity check is always true at that moment. Thus the parity field is generated inside the medium, to relieve the sender from its manipulation[2].

The following declarations are in effect:

```
type  packet_type      :  PACKETS
type  sequence_number  :  WINDOW_SIZE
```

Here, constants PACKETS and WINDOW_SIZE are supposed to have been given values previously and denote the number of valid packet types and the range of sequence numbers. Packets are represented as records; separate types are used for reduced and full packets.

```
{ a reduced version of packet, as sent to the medium }
type pak_in:  record
```

[2]Our experience with protocol verification shows that this simple optimization reduces the size of the specification considerably.

```
   pty: packet_type; { type of the packet }
   pno: sequence_number; { sequence number of the packet }
end

{ a full packet, as received from the medium }
type pak_out: record
   bod: pak_in;
   pck: boolean;
end
```

In the following fragment of code, the medium receives a packet from the sender and then can nondeterministically choose to either lose it or pass it along to the receiver. In the latter case, the medium also has the choice of whether to corrupt the packet or send it intact. The parity check field is set accordingly.

```
process medium( in: pak_in, out: pak_out )
   begin
      type pak: record
         in: channel;
         pck: boolean;
      end
      var buf: pak

   while true=true do
      select
         skip { drop the packet altogether }
         %
         begin
            select { choose checksum value }
```

```
              buf.pck := true
              %
              buf.pck := false
            end;
            out!buf { output the resulting packet }
          end
        end
      end
end; { process medium }
```

The above code exhibits one more optimization that substantially reduces the number of states in the LTS describing the medium. The straightforward implementation would have an input statement, assigning the incoming packet from the `in` channel to a local variable and then output the value of the variable onto the `out` channel. However, the medium does not alter the value of the packet fields and thus does not have to store these fields, which is quite expensive in terms of the size of the resulting specification. Instead, a special kind of record is used to specify the local variable `buf`: one of the fields is of type `channel`, and its name is indeed the name of the input channel. When, after the checksum value is produced, the variable is output to the channel `out`, it results in the contents of the input channel being transferred directly to the output channel.

2.3.5 Possible Extensions to VPL

VPL may be extended in several ways to provide more flexibility and expressive power to users. One useful extension already mentioned above is to allow procedures to have local variables. Two different kinds of procedures may turn out to be useful. The first is the existing one, where procedures are essentially abbreviations for sequences of statements belonging to the body of a single process. The other kind of procedures are independent entities that may be shared by several independent processes. Conceptually, procedures of this kind are more like servers in a network of workstations accepting "remote procedure calls" from client processes.

Another extension to VPL which we are now considering is a different specification paradigm. Currently VPL allows one to describe message-passing distributed systems where individual processes can only exchange messages. Support for shared-memory specifications may include variables declared within a VPL network and shared by all processes that are parts of the network.

A correct implementation of shared variables should feature "multiport" memory cells. That is, different subsystems of the network much use different internal names to refer to the values of the variable. Otherwise, when one subsystem wants to write a value into the variable and another subsystem is trying to read from the same variable, the rendez-vouz semantics of channels will allow the two subsystems to synchronize with each other, leaving the variable unchanged.

Finally, it seems to be useful to provide parameterization of processes, i.e. a way to specify a set of identical processes, differing only in the value of some parameter. Effectively, this will allow one to specify arrays of processes, distinguishable by their indices. This will make it easier to describe systems with high degree of regularity seen, for example, in many VLSI applications.

2.4 Real-Time Extension to the Concurrency Factory Objects

This section contains a proposal for extending CF objects with the ability to deal quantitatively with real-time constraints in specifications. Our real-time model-checking algorithm (Chapter 6) uses timed automata [5] to represent real-time systems. Thus the most straightforward extension would be to extend LTSs that are currently used as `Process` objects to timed automata. However, giving the full power of timed automata to the user can be more of a confusion than help. In [38] the authors point out similarities between timed automata specifications and assembly language programming and conclude that higher-level languages are needed to make specification of real-time systems practical.

Driven by the same concerns, we want to extend the Concurrency Factory graphical specification language in such a way that would, on the one hand, provide intuitive and easy-to-use constructs to the user and, on the other hand, will give rise to specifications that can be easily translated into timed automata for the purposes of verification. This section discusses a preliminary proposal for such extension.

We retain the basic hierarchical structure of the Concurrency Factory specifications. The lower lever of the hierarchy are processes, in this case endowed with additional timing information, and other other levels are formed by networks. Network specifications do not carry any timing information and thus do not need to be extended. However, we have to specify here what is the semantics of parallel composition of subsystems in a network. We intend to have the maximal progress property for the objects, i.e. communication between components of a network must occur as soon as both parties are ready. One of the consequences of this property is that internal actions are urgent.

We propose to specify real-time CF processes as LTSs with additional labels on transitions that serve as enabling conditions for the transitions. Each timed process is provided with a default clock, invisible directly to the user. The clock is reset every time a transition in the process happens. The transition constraints are interpreted with respect to this implicit clock.

Timing constraints on transitions usually take the form of upper and lower bounds on times when the transition can occur. In this case, a drawback of the maximal progress paradigm, as discussed in [80], is that specifications become too rigid. For example, if we have only fixed lower bounds, then the choice between two transitions leaving the same state is truly nondeterministic in any environment only if their delays are the same.

To add flexibility to our specifications, we introduce more complex timing constraints. Constraints labeling transitions take the form of triples $([c,d],e)$. The meaning of the three bounds is as follows. c and d ($c \leq d$) specify the *nondeterministic delay* for the transition. That is, the transition becomes enabled at a random moment between c and d. The upper bound e ($d \leq e$) specifies when the transition becomes disabled again. This is a way to represent time-outs, which has the advantage that no additional states are introduced. It also allows one to time out on a single transition while leaving other outgoing transition from the same state enabled.

A few special cases of these triple constraints are interesting: $([x,x],z)$ specifies a deterministic transition, which becomes enabled after a fixed delay. $([x,y],\infty)$ denotes a *persistent* transition [85], the one that does not get disabled by mere passage of time. With $([x,x],\infty)$ we get persistent transitions with fixed delays – the ones commonly found in real-time process algebras ([85, 59]).[3]

Maximal progress offers another way to specify timeouts. Since the internal moves are urgent, when an internal action becomes enabled in a state, it is immediately taken, effectively disabling all other transitions from the state. This is a coarser approach, as it works on the per-state rather than per-transition basis. It also may require more states than the first one. It, however, is more natural if a specific action has to be performed when a timeout occurs: the internal action will take us to the state where this timeout action is guaranteed to happen.

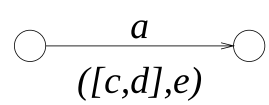
[3]Note that in some process algebras nondeterministic enabling can be achieved using infinite summation over fixed delays.

Note that both kinds of timeout mentioned above are local timeouts. That is, once a transition occurs, the default clock is reset and all information about prior waiting is lost. It is often necessary to specify global timeouts, i.e. ones where delays are accumulated over a sequence of transitions. This can be achieved by adding clocks to specifications that would be reset to 0 at desired moments rather that with every transition.

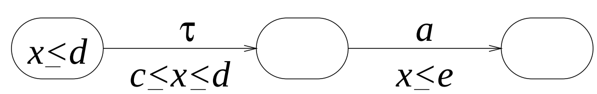
There are several possible ways to introduce additional clocks into specifications. The easiest way is to allow users to define new named clocks. These user-defined clocks may be reset by selected transitions, and there may be constraints parameterized by the names of the clocks. In this case, such constraints are interpreted with respect to the specified clock. The drawback of this approach is that it is too low-level and essentially brings us back to timed automata. Another possibility is to employ some version of a timeout operator [60] that would introduce additional clocks implicitly and make sure that the resulting specification is well-behaved (e.g. non-Zeno).

We now discuss translation of the proposed objects into timed automata. A formal definition of a timed automaton is delayed until Section 6.1. Informally, a timed automaton is a finite-state LTS endowed with a set of real-valued clocks. Both states and transitions are labeled with constraints. Constraints are sets of inequalities over clocks. Each inequality can compare a clock value or the difference between two clock values with an integer constant. Constraints on transitions are enabling conditions specifying when a transition can take place. Constraints on states specify invariants that have to hold while the automaton is in a given state. This is used to enforce progress, as the automaton has to leave the state by taking one of the outgoing transitions before the invariant of the state becomes false. However, this kind of progress is not necessarily maximal progress.

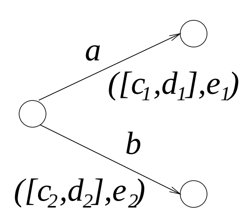
Consider the translation of the following object into a timed automaton:



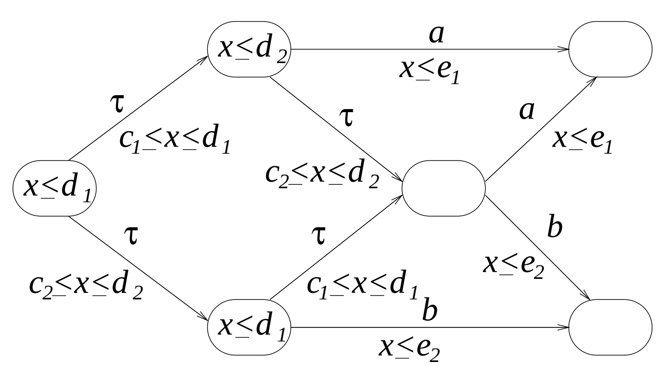
The expected behaviour of this object is that at some point between c and d it internally comes to a state where a is enabled. This is captured by the automaton below. x is the implicit or explicit clock with respect to which the constraint is interpreted. Since timed automata do not have maximal progress, the internal τ action is not urgent. However, the state invariant $x \leq d$ guarantees that after time d the automaton will leave the state and thus a will be enabled.



If we have several transitions leaving the same state, the translation becomes more involved. For example, for two outgoing transitions:



the corresponding timed automaton becomes:



Here we assumed, without loss of generality, that $d_1 \leq d_2$. It is important for the first state of the automaton, which should be labeled with two constraints, $x \leq d_1$, $x \leq d_2$. To make the picture more readable, we show only the essential constraint, as one of them would necessarily imply the other.

It is clear that this construction, in general, is exponential in the number of outgoing transitions of a state. Indeed, for every subset of the outgoing transitions there should be a state present in the automaton, from which only this subset is enabled. Some of the internal transitions, though, may have trivially false enabling conditions, thus making some of these states unreachable.

Chapter 3

Model Checking Basics

3.1 Definitions

Model checking is a verification technique that uses formulas of a modal or temporal logic to express properties of a system expressed in some other kind of specification language, and then match them against each other to decide whether the property holds for the system in question (in logical terms, does the system provide a *model* for the formula — hence the name). Thus there are two languages involved in the verification process: the system specification language and the property specification language. We are using finite-state labeled transitions systems for to specify systems. It has been, probably, the most common specification paradigm in recent years. We choose the alternation-free subset of the modal mu-calculus as our property specification language. It is a reasonable compromise between expressive power and complexity of model checking.

A *Labeled Transition System* (LTS) is a 4-tuple $\langle \mathcal{S}, Act, \rightarrow, s_0 \rangle$ where $\mathcal{S}$ is the set of *states*, Act is the set of *actions*, $\rightarrow \subseteq \mathcal{S} \times Act \times \mathcal{S}$ is the *transition relation*, and s_0 is the *start state*. An LTS is finite state when $\mathcal{S}$ is finite. We do not require *Act* to be finite, but assume that every state has finitely many successors.

We next give the syntax and semantics of a version of the alternation-free modal mu-calculus defined in [23], which we refer to as *CS-logic*. Formulas in CS-logic are of two types: basic formulas and equational blocks. The syntax of *basic formulas* is given by the following grammar:

$$\Phi ::= A \mid X \mid \Phi \vee \Phi \mid \Phi \wedge \Phi \mid [a]\Phi \mid \langle a \rangle \Phi$$

where $A \in \mathcal{AP}$, a fixed set of atomic propositions, and $X \in \mathit{Var}$, a countably infinite set of variables.

Basic formulas are interpreted with respect to an LTS $\mathcal{L} = \langle \mathcal{S}, \mathit{Act}, \rightarrow, s_0 \rangle$, a *valuation mapping* $\mathcal{V} : \mathcal{AP} \rightarrow \mathcal{P}(\mathcal{S})$ relating every atomic proposition A to the set of states in which A holds, and an *environment* $e : \mathit{Var} \rightarrow \mathcal{P}(\mathcal{S})$ mapping each variable X to the set of states that satisfy X. For a fixed environment e, the meaning of basic formulas is given by the semantical function $[\![\cdot]\!]e : \Phi \rightarrow \mathcal{P}(\mathcal{S})$ defined in Figure 3.

An *equational block* B is formed by applying operator *min* or *max* to a set E of mutually recursive equations of the form

$$\begin{array}{rcl} X_1 & = & \Phi_1 \\ & \vdots & \end{array}$$

$$\begin{array}{l} [\![A]\!]e = \mathcal{V}(A) \\ [\![X]\!]e = e(X) \\ [\![\Phi_1 \wedge \Phi_2]\!]e = [\![\Phi_1]\!]e \cap [\![\Phi_2]\!]e \\ [\![\Phi_1 \vee \Phi_2]\!]e = [\![\Phi_1]\!]e \cup [\![\Phi_2]\!]e \\ [\![[a]\Phi]\!]e = \{s \mid \forall s'.s \xrightarrow{a} s' \Rightarrow s' \in [\![\Phi]\!]e\} \\ [\![\langle a \rangle \Phi]\!]e = \{s \mid \exists s'.s \xrightarrow{a} s' \wedge s' \in [\![\Phi]\!]e\} \end{array}$$

Figure 3: Semantics of basic formulas.

$$X_n = \Phi_n,$$

where each Φ_i is a basic formula and the X_i are pairwise distinct. Operators *min* and *max* are understood respectively as the least and greatest fixed points of E. Following [23], we assume that the Φ_i are *simple*, i.e., either an atomic proposition, or constructed by the application of exactly one operator to variables. Every formula can be made simple with at most linear blow-up in size. Note that we presented operators $\vee$ and $\wedge$ as binary operators. This does not have to be the case. In fact, in the implementation of the algorithms, presented in the next chapters, we allow $\vee$ and $\wedge$ to be applied to a set of disjuncts (resp. conjuncts). The size of this set is supposed to be bounded by a constant. Such formulas are also considered to be simple.

Semantically blocks are understood as functions from environments to environments. Let a block B contain a set of equations E with variables $X_1, \ldots, X_n$ defined as left-hand sides. Let $\overline{S} = \langle S_1, \ldots, S_n \rangle \in (2^{\mathcal{S}})^n$ and let $e_{\overline{S}} = e[X_1 \mapsto S_1, \ldots, X_n \mapsto S_n]$. Then the function

$$f_E^e(\overline{S}) = \langle [\![\Phi_1]\!] e_{\overline{S}}, \ldots, [\![\Phi_n]\!] e_{\overline{S}} \rangle,$$

defined on the lattice of tuples of sets of states ordered by point-wise set inclusion is monotonic. By the Tarski-Knaster fixed-point theorem, f_E^e has both least and greatest fixed points given by:

$$\nu f_E^e = \bigcup \{\overline{S} \mid \overline{S} \subseteq f_E^e(\overline{S})\}$$
$$\mu f_E^e = \bigcap \{\overline{S} \mid f_E^e(\overline{S}) \subseteq \overline{S}\}$$

Blocks can now be interpreted in the following fashion:

$$[\![maxE]\!] e = e_{\nu f_E^e}$$
$$[\![minE]\!] e = e_{\mu f_E^e}.$$

Finally, a *formula* $\mathcal{B} = \{B_1, \ldots, B_m\}$ is a set of blocks, with the following syntactic restrictions: all variables appearing on the left-hand sides in the set of blocks are distinct, and the formula's block graph is acyclic. The *block graph* of $\mathcal{B}$ is the directed graph with nodes $B_1, \ldots, B_m$ and edges $\langle B_i, B_j \rangle$ whenever a variable appearing as a left-hand side of an equation in B_i is used in B_j (we say that B_j *depends* on B_i in this case). Restricting the block graph to be acyclic ensures that no alternating fixed points [29] can occur.

The meaning $[\![\mathcal{B}]\!]e$ of the formula $\mathcal{B}$ containing blocks $B_1, \ldots, B_m$, topologically sorted by the dependency relation, can be computed through a sequence of environments

$$\begin{array}{rcl} e_1 & = & [\![B_1]\!]e \\ & \vdots & \\ e_m & = & [\![B_m]\!]e_{m-1} \end{array}$$

with $[\![\mathcal{B}]\!]e = e_m$. Due to the acyclicity restriction on block graphs, we are ensured that $[\![\mathcal{B}]\!]e_m = e_m$.

If $\mathcal{B}$ is a closed formula, i.e., every variable mentioned in the right-hand side of some equation appears on the left-hand side of an equation in one of the blocks, then for every two environments e and e', we have $[\![\mathcal{B}]\!]e = [\![\mathcal{B}]\!]e'$. We omit the reference to a specific environment in this case. Now, for every variable X defined in the formula we can compute the set of states in which X holds as $[\![X]\!][\![\mathcal{B}]\!]$.

When the LTS is finite-state, f_E^e is continuous and the fixed points also have iterative characterizations: $\nu f_E^e = \bigcap_{i=0}^{\infty} f_i$ and $\mu f_E^e = \bigcup_{i=0}^{\infty} \hat{f}_i$, where

$$\begin{array}{rcl} f_0 & = & \langle \mathcal{S}, \ldots, \mathcal{S} \rangle \\ \hat{f}_0 & = & \langle \emptyset, \ldots, \emptyset \rangle \\ f_{i+1} & = & f_E^e(f_i), i \geq 0 \\ \hat{f}_{i+1} & = & f_E^e(\hat{f}_i), i \geq 0 \end{array}$$

Both the *CS* and *MCI* algorithms make use of these iterative characterizations to compute fixed points.

The problem of *model checking in CS-logic* can now be defined as follows: given an LTS $\mathcal{L} = \langle \mathcal{S}, A, \rightarrow, s_0 \rangle$ and a CS-logic formula $\mathcal{B}$ with a designated variable X defined within it, determine whether $s_0 \in [\![X]\!][\![\mathcal{B}]\!]$.

We can view a given formula as a graph of dependencies between its variables. Namely, the vertices of the graph are variables appearing as left-hand sides of equations. For each variable X, the corresponding vertex is labeled by the operator of the right-hand side of the equation defining X (we may abuse the terminology sometimes by saying that X *is* that operator). For every variable X' appearing on the right-hand side of the equation for X, there is an edge $X \rightarrow X'$. Sometimes, when we need to specify the operator labeling X, we put it above the $\rightarrow$ sign. As in [23], we will call this representation of a formula its *expression graph.* Note that direction of edges in the expression graph is reversed compared to [23], as it is more natural for taking the product with an LTS.

The *product graph* of an LTS $\mathcal{L} = \langle \mathcal{S}, Act, \rightarrow, s_0 \rangle$ and a mu-calculus formula $\mathcal{B}$ is a directed graph with set of vertices $\{\langle X_i, s\rangle \mid X_i \in Var, s \in \mathcal{S}\}$ and set of edges given by the following rules:

- if $X_i \xrightarrow{\vee} X_j$ or $X_i \xrightarrow{\wedge} X_j$, then for every $s \in \mathcal{S}$, $\langle X_i, s\rangle \rightarrow \langle X_j, s\rangle$
- if $X_i \xrightarrow{\langle a\rangle} X_j$ or $X_i \xrightarrow{[a]} X_j$, $s \xrightarrow{a} s'$, then $\langle X_i, s\rangle \rightarrow \langle X_j, s'\rangle$.

The following terminology and notations will be used. Since product graphs will be the center of our attention in the forthcoming chapters, we will simply say "node" to mean "product graph node". If operator $\vee$ or $\langle a\rangle$ is used to define X, the node $\langle X, s\rangle$ is called an $\vee$-node of the product graph; otherwise, it is called an $\wedge$-node. If B is a block in a formula, the *subgraph* of B is defined to be the subgraph of the product graph induced by the set of nodes $\{\langle X, s\rangle \mid X \text{ is defined in } B\}$. We say that node $\langle X, s\rangle$ is defined by some operator *op* and belongs to block B when X is defined by *op* in B.

The following straightforward lemma is important in proofs of correctness of the algorithms presented in the following chapters.

Lemma 3.1.1 *All nodes of each strongly connected component (SCC) of a product graph belong to the same block.*

Proof: The proof is by contradiction. Suppose there was an SCC in the product graph that contained nodes from two different blocks, say B_1 and B_2. Pick two nodes in that SCC, t and w belonging to B_1 and B_2, respectively. By the definition of an SCC, there are paths $t \to^* w$ and $w \to^* t$ in the product graph. Since t and w belong to different blocks, there are nodes t_1 in B_1 and w_1 in B_2 such that $t_1 \to w_1$. This means that there is an edge $\langle B_1, B_2 \rangle$ in the block graph. Repeating the same argument for the path $w \to^* t$, we see that there is an edge $\langle B_2, B_1 \rangle$ in the block graph. This constitutes a cycle in the block graph and thus violates the syntactic restriction on the formula that we imposed earlier. $\Box$

An important notion, used in heavily in subsequent chapters, is that of an *interference-free cycle*. Let C be a cycle in the product graph, and let t be a node lying in C. C is *interference-free* in a given environment if every successor of t not lying in C is false, if t is an $\vee$-node, and every successor of t not lying in C is true if t is an $\wedge$-node. If the environment is a fixed point, setting all nodes in the cycle uniformly to either true or false yields a fixed point. It is easy to see that in the greatest fixed point all interference-free cycles have their nodes set to true, while in the least fixed point all of them are set to false.

3.2 The Cleaveland-Steffen Model-Checking Algorithm

We now briefly present a model-checking algorithm due to Cleaveland and Steffen [23], to which we refer as the *CS* algorithm. The algorithm is *global*, meaning that it computes values of all variables of the formula in all states of the LTS. In other words, it constructs the whole product graph. It should be noted that the algorithm does not construct the product graph explicitly. However, its presentation becomes more concise using the notion of product graph.

The *CS* algorithm serves as a prototype for the incremental algorithm presented in Section 4. That is, the two algorithms use the same data structures and have follow similar sequences of steps. Also, we can identify redundancies in the algorithm, and avoiding these will lead us to the local model-checking algorithm.

The *CS* algorithm has been implemented for the first time in the Concurrency Factory.

Data Structures:

For each node $\langle X_i, s\rangle$ of the product graph, the *CS* algorithm maintains the following variables:

- A boolean variable indicating whether or not variable X_i is true of state s in the current stage of the analysis. We simply use the name of the product graph node, i.e, $\langle X_i, s\rangle$, as the name of this variable, and sometimes refer to it as the *value* of the node. $\langle X_i, s\rangle$ is initialized to true if X_i is defined in a max-block, and, dually, is initialized to false if X_i is defined in a min block (we refer to these initializations as $trivial$), with the following exceptions:
  - The right-hand side of the equation for X_i is an atomic proposition

A. Then $\langle X_i, s\rangle$ = true if $s \in \mathcal{V}(A)$, and false otherwise.

– If state s has no a-derivatives and $X_i \xrightarrow{\langle a\rangle} X_j$, then $\langle X_i, s\rangle$ = false, and if $X_i \xrightarrow{[a]} X_j$, then $\langle X_i, s\rangle$ = true.

- A counter $C_{\langle X_i,s\rangle}$ that keeps track of the immediate successors of $\langle X_i, s\rangle$ in the product graph: if X_i is defined in a *max*-block and $\langle X_i, s\rangle$ is an $\vee$-node, then $C_{\langle X_i,s\rangle}$ records how many immediate successors of $\langle X_i, s\rangle$ are currently true. $C_{\langle X_i,s\rangle}$ is used dually in the case that X_i is defined a *min*-block and $\langle X_i, s\rangle$ is an $\wedge$-node; i.e., it records how many immediate successors are currently false. In either case, $C_{\langle X_i,s\rangle}$ is initialized to the number of immediate successors of $\langle X_i, s\rangle$ in the product graph. These are the only cases in which counters are used by the *CS* algorithm.

Also, for every block B_j of the formula, a list M_j of nodes of the product graph is maintained such that $\langle X_i, s\rangle$ is in M_j if X_i is defined in B_j, $\langle X_i, s\rangle$ recently changed its value, and the effect of this change on other nodes has yet to be determined. Initially, M_j contains all nodes $\langle X_i, s\rangle$ that were initialized non-trivially.

The *CS* algorithm is captured by the following procedure, where $\mathcal{L}$ is a finite-state LTS and $\mathcal{B}$ is a CS-logic formula.

procedure $CS(\mathcal{L}, \mathcal{B})$
  topologically sort the blocks of $\mathcal{B}$
  initialize the $\langle X_i, s\rangle$, $C_{\langle X_i,s\rangle}$, and M_j as described above
  for each $B_j \in \mathcal{B}$ in topological order **do**
    if B_j is of type *max* **then** $MAX(B_j)$
    else $MIN(B_j)$

Procedure MAX, invoked on a block B_j, proceeds as follows:

procedure $MAX(B_j)$

while M_j not empty **do**
  delete some $\langle X_i, s\rangle$ from M_j
  $DOWN(\langle X_i, s\rangle)$
for each $\langle X_i, s\rangle$ such that X_i is defined in B_j and $\langle X_i, s\rangle$ is true **do**
  $UP(\langle X_i, s\rangle)$

Procedure *MIN* is dual to it, with all occurrences of *UP* and *DOWN*, as well as true and false, interchanged. *MAX* first propagates the changes to the variable assignment for block B_j recorded in M_j by repeatedly calling the procedure *DOWN* (given below). When M_j is finally empty, the greatest fixed point for B_j will have been computed. *MAX* then invokes *UP* on each variable $\langle X_i, s\rangle$ such that X_i is defined in B_j and $\langle X_i, s\rangle$ was *not* falsified during the preceding while loop. This is necessary because there may exist variables in min-blocks, trivially initialized to false and dependent on $\langle X_i, s\rangle$, whose values should now be true. The calls to *UP* will produce the desired effect.

Procedure *DOWN* takes as a parameter a product graph node that has recently changed its value from true to false, and checks whether any of its predecessors are affected by the change.

procedure $DOWN(\ \langle X_i, s\rangle\ )$:
for each $\vee$-node $\langle X_j, s'\rangle$ such that $\langle X_j, s'\rangle \rightarrow \langle X_i, s\rangle$, $\langle X_j, s'\rangle$ is true and
    X_j defined in B_k **do**
    decrement $C_{\langle X_j, s'\rangle}$ by 1
    if $C_{\langle X_j, s'\rangle} = 0$, **then**
      $\langle X_j, s'\rangle$:= false
      add $\langle X_j, s'\rangle$ to M_k
for each $\wedge$-node $\langle X_j, s'\rangle$ such that $\langle X_j, s'\rangle \rightarrow \langle X_i, s\rangle$, $\langle X_j, s'\rangle$ is true and
    X_j defined in B_k **do**
    $\langle X_j, s'\rangle$:= false
    add $\langle X_j, s'\rangle$ to M_k

The name of the procedure stresses the fact that the function δ on the environment (variable assignment) computed by *DOWN* satisfies $\delta(e) \leq e$; i.e., the resulting assignment moves *down* in the lattice of tuples of sets of states described in Section 3.1. Procedure *UP* is dual to *DOWN* and can be obtained by syntactically interchanging all occurrences of $\vee$ and $\wedge$, and true and false.

Correctness of the *CS* algorithm is proved by induction on the position of a block B a fixed topological ordering of the block graph of $\mathcal{B}$. Assuming that all preceding blocks were processed correctly, it is shown that, in the case of a *max* block, the initial variable assignment is above the greatest fixed point, and calls to *DOWN* monotonically lower it until the fixed point is reached.

We illustrate the work of *CS* with a small verification example that will be used in subsequent chapters to stress the difference between different approaches to model checking. Let the property of interest be "always eventually p", where p is an arbitrary atomic proposition. This property is expressible in CTL [18] as AGEFp. The corresponding formula in the CS-logic (for an LTS containing only action a) is

$$\begin{array}{ll} B_1 : \quad max & \left\{ \begin{array}{lcl} X_0 & = & X_1 \wedge X_2 \\ X_1 & = & [a]X_0 \end{array} \right. \\ B_2 : \quad min & \left\{ \begin{array}{lcl} X_2 & = & X_3 \vee X_4 \\ X_3 & = & p \\ X_4 & = & [a]X_2 \end{array} \right. \end{array}$$

The LTS for our example is shown in Figure 4, and p is chosen to be false in every state. The product graph is shown in Figure 5. Initialization assigns true to every node in B_1 (the top row of nodes in Figure 5) and false to every node in B_2. Since all nodes were initialized "trivially", i.e. consistently with the default values for their respective blocks, both lists M_1 and M_2 are initially empty. Fixpoint computation starts with block B_2. The first part of procedure

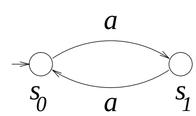


Figure 4: An LTS for the verification example

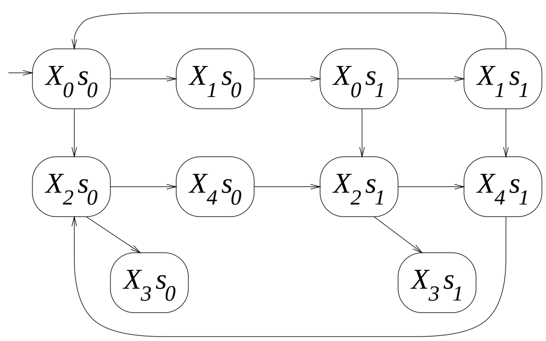


Figure 5: The product graph

UP does not do anything as the variable assignment already corresponds to the least fixed point. Now the second part of *UP* has to notify nodes in B_1 that the nodes remained false. As a result, the false values of $\langle X_2, s_0\rangle$ and $\langle X_2, s_1\rangle$ are propagated to $\langle X_0, s_0\rangle$ and $\langle X_0, s_1\rangle$, respectively. These are $\wedge$-nodes, so they acquire false values as well and are both added to M_1. Now, when *DOWN* is called for B_1, $\langle X_0, s_0\rangle$ resets $\langle X_1, s_1\rangle$ to false and $\langle X_0, s_1\rangle$ resets $\langle X_1, s_0\rangle$ at which point the computation completes. When the algorithm terminates, every variable is false in every state.

An important observation that will lead us in Chapter 5 to the idea of the local algorithm is that most of the work has been done during the initialization stage, while the fixed-point computation itself took very little effort. Another point to notice is that node $\langle X_0, s_0\rangle$, the value of which is the answer to the model checking problem, has had its final value assigned before processing of B_1 started, and thus processing nodes in M_1 was redundant in this respect.

Chapter 4

Incremental Model Checking

4.1 Algorithm *MCI*

In this section, we modify the *CS* algorithm to obtain the incremental global algorithm *MCI*. *MCI* works incrementally in the following sense: Let $\mathcal{L}$ and $\mathcal{B}$ constitute a given instance of the model-checking problem, and assume that the desired variable assignment has been previously computed (say, by an application of *CS*). Given a set Δ of *changes* to the LTS, where a change may correspond to either an inserted or deleted transition, *MCI* computes the new variable assignment by using the previously computed one as the "starting point" of the computation.

In this presentation, we assume that changes to the LTS are in the form of added and deleted transitions. This restriction, however, is not essential. We later discuss how, with minimal modifications to the algorithm, we can handle added and deleted states as well as a number of changes to CS-logic formulas.

The top-level structure of *MCI* is basically the same as in *CS*: an initialization phase, in which the immediate effects of Δ on the previously computed variable assignment are ascertained, is followed by a for-loop in which blocks are processed in topological order of the block graph by calling modified *MAX* and *MIN* procedures. In the incremental case, however, it is necessary to start off the fixed-point computation of a block by making certain adjustments to the current variable assignment to ensure that the proper fixed point is computed.

As discussed in greater detail below, the adjustments will raise the variable assignment in the case of a max-block (by calling a modified *UP* procedure), and lower it in the case of a min-block (by calling a modified *DOWN* procedure). Like before, iterations of the fixed-point computation will then lower the assignment in the case of a max-block, and raise it in the case of a min-block. As discussed in the previous chapter, "raise" and "lower" refer to the position of the assignment in the lattice of variable assignments.

Since, for either type of block, we may need to shift the variable assignment up or down the lattice, it is no longer sufficient to provide procedures *MAX* and *MIN*, invoked on a block B_j, a single "work list" M_j, as in the non-incremental case. Rather, two such lists are now required, for both types of blocks: $down_j$, which records variables that change their values from true to false, and up_j, containing variables that change their values from false to true. Moreover, every product graph $\wedge$- and $\vee$-node now has an associated counter, regardless of the type of the block they are defined in. Counters, as before, reflect the number of true successors for an $\wedge$-node and the number of false successors for an $\vee$-node.

The initialization phase uses Δ to update both the product graph and the variable assignment of the previous computation. Changes to the product graph reflect the semantics of basic formulas; in particular, the modal operators (the insertion and deletion of LTS transitions has no immediate effect on basic formulas constructed out of logical operators). When a transition $s \xrightarrow{a} s'$ is inserted into the LTS, for every pair of variables X, X' such that $X \xrightarrow{[a]} X'$ or

$X \overset{\langle a \rangle}{\rightarrow} X'$, the edge $\langle X, s \rangle \rightarrow \langle X', s' \rangle$ is inserted into the product graph. Conversely, when a transition is deleted from the LTS, the corresponding set of edges is deleted from the product graph.

In response to changes in the product graph, counters are updated as one would expect. If t is an $\vee$-node and w is true, then C_t is incremented by 1 if the edge $t \rightarrow w$ is inserted into the product graph, and is decremented by 1 if this edge is deleted. The situation is dual for $\wedge$-nodes.

The following cases require changing the value of a variable as an immediate result of inserting or deleting a product graph edge, independent of the block type. In each case, assume that t is defined in block B_k.

- An edge $t \rightarrow w$ such that t is an $\wedge$-node is added to the product graph. If t is true and w is false, then t is changed to false and added to $down_k$.
- An edge $t \rightarrow w$ such that t is an $\vee$-node is added to the product graph. If t is false and w is true, then t is changed to true and added to up_k.
- An edge $t \rightarrow w$ such that t is an $\wedge$-node is deleted from the product graph. If w is false and t had only one false successor (this number is recorded in C_t), then t is changed to true and added to up_k.
- An edge $t \rightarrow w$ such that t is an $\vee$-node is deleted from the product graph. If w is true and t had only one true successor, then t is changed to false and added to $down_k$.

The initializations described above depend only on the semantics of basic formulas and are therefore independent of the type of fixed point being computed. Simply restarting the fixed-point iteration (e.g., calls to *DOWN* in the case of a *max*-block) at this point would bring us to a fixed point, but not necessarily the required fixed point! Rather, we must conclude the initialization phase by making certain *assumptions* about the existence of strongly connected components (SCCs) in the product graph. Assumptions made during initialization, and their subsequent propagation through the product graph, will serve to adjust the variable assignment to a level where fixed-point iteration can be safely restarted.

To motivate our use of assumptions, consider the following scenario. The insertion of a transition in the LTS has resulted in the formation of a new SCC, call it C, in the product graph. Assume that C is contained in the subgraph of a max-block B_j, and according to the results of the previous fixed-point computation, all nodes in C are false. Further assume that C is interference-free (see Section 3.1). Then both variable assignments in which all nodes in C are uniformly set to true or false are fixed points. The point is, however, that the required fixed point for C is the greatest one, i.e., the one in which all nodes are assigned the value true.

We will therefore, in general, assume that when an edge $t \rightarrow w$ is added to the product graph such that t and w are defined in the same max-block B_k and both are false, that a new SCC, free of external interference, has been created. We record this assumption by setting t to true and adding it to up_k and a new list called $assumptions_k$. Note that the counter C_t is not updated to reflect t's new value and thus an inconsistency is introduced. This is intentional and will be used later to determine whether the assumption was a valid one.

A number of small optimizations can be made here to prevent obviously false assumptions, e.g. do not make the assumption if either of the nodes has no incoming edges or no outgoing edges, and thus cannot be on a cycle. We could make better assumptions if we knew more about the structure of strongly connected components in the product graph. However, updating the information about SCCs of a graph when edges can be added and deleted is about as expensive as computing the variable assignment itself. We can derive additional information from the formula graph. A node $\langle X, s\rangle$ in a product graph can belong to a non-trivial SCC only if X belongs to a non-trivial SCC in a formula graph. Since we assume here that changes do not affect the formula graph, we can compute its SCCs once and use this information in all subsequent runs of the algorithm.

The case of a *min*-block is dual: when an edge is added between two true nodes corresponding to variables defined in the same *min*-block, we change one of them to false and update the block's *assumptions* list. The changed value is reflected in the *down* list.

Deleted edges can also cause us to make assumptions during initialization. Suppose that a *max*-block SCC had only one source of external interference, which was eliminated when an edge was deleted from the product graph. The desired variable assignment in this case has all nodes in the SCC uniformly set to true. We therefore assume that whenever an edge $t \rightarrow w$ is deleted from the product graph such that t is defined in a *max*-block B_k and both nodes are false, w constituted the only source of external interference in the SCC containing t. We record this assumption by setting t to true and adding it to both up_k and $assumptions_k$.

Consider now the propagation of the changes made to the variable assignment during initialization. These are recorded in up_j and $down_j$, for each block B_j. As before, blocks are processed in topological order of the block graph, by calling a modified *MAX* or *MIN* procedure depending on the type of the block. *MAX* and *MIN* now include an *adjustment phase*, during which the variable assignment is shifted up (in the case of a *max* block) or down (in the case of a *min* block) in the lattice of assignments to ensure that the fixed-point computation can proceed normally. The new *MAX* procedure is given by:

procedure $MAX(B_j)$
  while up_j not empty **do** /* Adjustment Phase */
    delete some t from up_j
    $UP(t)$

```
while assumptions_j not empty do        /* Check validity of assumptions */
  delete some t from assumptions_j
  if t is an ∧-node then
    if C_t ≠ 0 then
      t := false
      add t to down_j
  if t is an ∨-node then
    if C_t = 0 then
      t := false
      add t to down_j
while down_j not empty do               /* Iteration Phase */
  delete some t from down_j
  DOWN(t)
```

The adjustment phase makes as many variables true as possible by iteratively invoking *UP*. As shown in Section 4.2, the resulting variable assignment will be high enough in the lattice to contain every fixed point of the semantic function. At this point, the validity of any previously made assumptions is determined by checking whether the value of t is consistent with C_t, for each node t on the list of assumptions. If an inconsistency is detected, i.e., according to C_t, t should be false, we reset the variable and let *DOWN* undo the effects of the assumption. When finished, the fixed-point iteration (applications of *DOWN*) can be safely restarted. Procedure *DOWN* is modified as follows:

```
procedure DOWN( w ):
  for each ∨-node t defined in B_k such that t → w, do
    decrement C_t by 1
    if t = true and C_t = 0 then
      t := false
      add t to down_k
```

else if $C_t \neq 0$, $t =$ true and B_k is a min-block **then**
$t :=$ false
add t to $down_k$
add t to $assumptions_k$
for each $\wedge$-node t defined in B_k such that $t \rightarrow w$ **do**
increment C_t by 1
if $t =$ true **then**
$t :=$ false
add t to $down_k$

The overall structure of *DOWN* is retained, except that counters are updated both when t is an $\vee$-node or an $\wedge$-node, and an assumption is made when *DOWN* encounters a true variable in a min-block. The intuition for the assumption is somewhat different from the one for assumptions made at initialization. Here we are assuming that t is a part of an SCC and the change to w eliminated a source of external interference for the SCC. As before, procedures *MIN* and *UP* are dual to those given above and are obtained by interchanging all occurrences of *UP* and *DOWN*, *up* and *down*, and $\wedge$ and $\vee$.

It remains to mention several implementation details that are not conceptually important for understanding the algorithm but are crucial if we want to retain the linear complexity of the global model-checking procedure. Because *MCI* accepts as input a *set* of changes to the LTS, updates to the variable assignment made during initialization can potentially "overlap." For example, assume an $\vee$-node had one true successor before the changes were applied. Suppose the change removed the existing edge to the true successor and introduced a new edge to another true node. If processed naively, the changes may cause the node to appear in both *up* and *down* lists. It is clear however, that the value of the node should not change in this case. Thus we make sure to look at all changes involving the node before we make the decision to include it in any of the lists. Moreover, a true node in a *max* block could acquire a false value during initialization and be placed in the corresponding *down* list. However, before we get to process that *down* list, it is possible that the value of the node has been made true again by calls to *UP* during the adjustment phase. We could modify *DOWN* to check that the node it takes as a parameter is still false. Instead, we chose to construct lists in such a way that it is possible to see, looking at the node, which list it is in, if any. Thus, in constant time, we can make sure that a node cannot be in both *up* and *down* lists.

So far we have assumed that the changes to the LTS only concern transitions. States, however, can be added and deleted with almost no additional effort. The basic idea is to assume that the variable assignment for an isolated state, i.e. one devoid of incident transitions, is known — it can be computed during the first, non-incremental run of the algorithm. During incremental runs, state additions are processed before any other changes by setting variables of the form $\langle X_i, s\rangle$, where s is a new state, in accordance to the variable assignment of an isolated state. The processing of inserted and deleted transitions can now proceed as before. For state deletions, we assume that any incident transitions are deleted as well.

Extending the algorithm to handle certain changes to the CS-formula requires even less work. The following kinds of changes can be accomodated. Logical connectives and modal operators can be changed to their duals. This will result in changing the types of the product graph nodes that contain the variable corresponding to the changed operator. For every such node, we have to look at its counter and revert it — that is, if c is the current value of the counter and n is the total number of successors of the node, the new value of the counter should be $n - c$. Then we can decide on the new value for the node based on the new value of the counter and add the node to the corresponding *up* or *down* list if the value has changed. An assumption may have to be made, as with the change of the type of the node, its successors may no longer be the source of external interference.

Another change that is easy to handle is changing the value of an atomic proposition in some of the states of the LTS. Once again, we have to add the nodes to the corresponding list. No assumptions can be required in this case as all nodes corresponding to atomic propositions are guaranteed to be terminal nodes.

We can anticipate, though, that potential savings from incremental recomputation of fixed points after a change to the formula should be less than in the case of changes to the LTS. When a transition is added or deleted from the LTS, the number of immediately affected product graph nodes is equal to the number of variables in the CS-formula. Conversely, when a logical operator is changed, there is one affected node for every state in the LTS. Typically, the LTS is much larger than the formula graph, thus we have to do more work in the latter case. Because of this, we decided against implementing the case of changing the CS-formulas.

4.2 Correctness and Complexity

We begin with an informal discussion of the difficulties encountered while constructing the proof of *MCI*. The main device, which is commonly used in the proofs of this kind, namely the Tarski-Knaster theorem, cannot be immediately applied in this case. The proof of the *CS* algorithm, outlined in the end of Section 3.2, is very typical in the sense that the algorithm starts in a state where the variable assignment is definitely above any fixed point in the lattice of variable assignments and subsequently is monotonically lowered by the algorithm until a fixed point is reached. Proofs of this kind are common in many areas. However, in the case of *MCI*, this approach does not apply because the computation it performs is not necessarily monotonic. Moreover, because the initial variable assignment may lie between fixed points in the lattice, even in the case of monotonic modification a suitable restatement of the Tarski-Knaster theorem can only guarantee that fixed-point iterations will terminate at a fixed point, but not necessarily of the right kind. An excellent, even if informal, discussion of these issues is offered in [68].

We thus have to come up with a different approach that would allow us to reason about computation "inside" a lattice, i.e. between fixed points. To the best of our knowledge, such device does not exist in an abstract form, in terms of monotonic functions on complete lattices, as the Tarski-Knaster theorem is. We had to resort to concrete reasoning about the specific nature of the function we are computing.

The correctness of the *MCI* algorithm is given by the following theorem.

Theorem 4.2.1 *Let $\mathcal{L}$ be an LTS, $\mathcal{B}$ a CS-logic formula, Δ a set of changes to $\mathcal{L}$ in the form of inserted and deleted transitions, and $\mathcal{L}'$ the LTS obtained by applying Δ to $\mathcal{L}$. Furthermore, let e be the variable assignment obtained by algorithm CS on input $\mathcal{L}$ and $\mathcal{B}$, and, similarly, let e' be the variable assignment obtained by CS on input $\mathcal{L}'$ and $\mathcal{B}$. Then MCI, using e as the initial variable assignment, terminates on input $\mathcal{L}$, $\mathcal{B}$, and Δ with variable assignment e'.*

Proof: Consider a block B in $\mathcal{B}$. Without loss of generality, assume that B is a max-block; the case where B is a min-block is completely dual and therefore omitted. For O an arbitrary topological order of $\mathcal{B}$'s block graph, the proof is by induction on the position of B in O. The inductive hypothesis is that all blocks preceding B in O have their variable assignments correctly computed. To facilitate the proof of the base case, we assume a fictitious empty block which is placed first in every O. Since this block doesn't have any variables in it, obviously its presence has no bearing on the course of the computation. Also, it is trivially processed correctly by the algorithm; thus the inductive assumption immediately holds for the next block in O.

The proof proceeds in two main steps. We show that (1) when B's *up* list is processed, the current variable assignment is not lower in the lattice of variable assignments than any fixed point of B's recursive equations. In particular, it contains B's greatest fixed point. Then we prove that (2) when B's *down* list is processed, the variable assignment corresponds to the greatest fixed point.

The proof of (1) is by induction on the topological order of the strongly connected components of B's subgraph. The ordering is well defined since the acyclicity of the block graph guarantees that every SCC is entirely contained within the subgraph of one block. Indeed, assume the contrary. Let there be a cycle in the product graph of the form $\langle X_i, s_i \rangle \rightarrow \langle X_j, s_j \rangle \rightarrow \ldots \rightarrow \langle X_k, s_k \rangle \rightarrow \langle X_l, s_l \rangle \rightarrow \ldots \rightarrow \langle X_i, s_i \rangle$, such that variables $X_j, \ldots, X_k$ belong so the same block, say B_1 and variables $X_l, \ldots$ and X_i belong to the same block B_2. Since $\langle X_i, s_i \rangle \rightarrow \langle X_j, s_j \rangle$, it must be the case that $X_i \rightarrow X_j$ in the expression graph and thus $B_2 \rightarrow B_1$ in the block graph. Similarly we have that $X_k \rightarrow X_l$ and thus $B_1 \rightarrow B_2$. This constitutes a cycle in the block graph, which is a contradiction with the initial restriction on the formulas. This argument easily generalizes to the case when the cycle passes through several blocks.

Now fix an SCC C and assume the result for any SCC having edges leading into C. If C is a trivial SCC, i.e. it contains only one node, then its value is completely determined by the value of its successors. Appealing to the two inductive hypotheses and the fact that e was a fixed point before Δ was applied, we immediately have the following three cases: (1) if the set of successors of C was changed by applying Δ, the necessary changes to the value of C were made during initialization stage; (2) if none of C's successors' values were changed, then the value of C is correct; (3) if some successor t has changed its value, then the value of C was appropriately updated when *UP* or *DOWN* was invoked on t.

Now, let C be a non-trivial SCC. In this case, the value of a node in C depends not only on C's successors, but also on the other nodes in C. We consider nodes within C that are *false* and show that these nodes must be false in e'. The following cases are considered for a false node t:

- The value of t is uniquely determined by a successor in a preceding block or a preceding SCC in the same block. A true successor uniquely determines the value of an $\vee$-node, and the false successor determines the values of an $\wedge$-node. In this case, the value of t is correct by the same argument as for the trivial SCC.

- Otherwise, there is a sequence of false nodes in C which forms a path starting at t. Two sub-cases are possible:

  - Every such path leads to a node whose value is uniquely determined by a successor outside C. That the value of t is correct can be seen by an inductive argument on the number of nodes in the longest such path: if the length of every path is 0, then we have the previous case. For n, note that every successor of t is now dependent on false paths of length at most $n-1$. If t is an $\wedge$-node, it suffices to know that at least one of its false successors has the correct value. For an $\vee$-node, note additionally, that there can be no true successors because whenever a node is set to true, it is added to the *up* list and thus would cause t to be true.

  - There is a cycle of false nodes within C, none of which depend uniquely on an external node. To establish the desired result, it suffices to prove that this sub-case cannot occur. This can be easily seen, for this cycle could not have been present in e, or it would not have been the greatest fixed point. This cycle also could not have been formed by an inserted edge, since whenever the new edge connects two false nodes, an assumption is made and one of them is made true. Finally, a cycle of false nodes could have been present

> in e if some of the nodes on the cycle were fixed by external interference. But if these sources of external interference were eliminated, either by changing values of interfering nodes or by disconnecting them through edge deletions, the assumptions made during the computation would set at least one node on the cycle to true.

Before we proceed to prove (2), we note that after we check the validity of assumptions, every node in B that had its value changed from true to false at any moment after Δ was applied is recorded in the *down* list of B.

Now that we have established that when the *up* list has been emptied the current variable assignment contains B's greatest fixed point, the proof of (2) basically coincides with the proof of correctness of the *CS* algorithm. That is, we show that the processing of entries in B's *down* list monotonically lowers the variable assignment, and, when the list is empty, the greatest fixed point will have been reached (see [23] for details). □

Consider now the computational complexity of the *MCI* algorithm. In the worst case, its complexity is the same as that of the *CS* algorithm: linear in the product of the size of the LTS and the size of the formula (plus the number of changes to the LTS, which are assumed to be small compared to the LTS size), where the size of the LTS is taken to be the total number of states and transitions, and the size of the formula is the total number of equations over all blocks. The proof is similar to that of [23], and is based on the fact that node t defined in block B_j can appear at most once in each list up_j and $down_j$. We ensure this property by checking (in constant time) if a node is already present in a list before attempting to add it to the list. Thus *UP* and *DOWN* can only be invoked on a node at most once each, and each such invocation traverses each outgoing edge once.

We have shown that in the worst case our algorithm possesses the same asymptotic running time as the original non-incremental algorithm. We would like to know whether it is possible to gain an asymptotic speedup in the incremental case. The results put forward in [65] suggest that there are limits to what can be achieved. There, a classification of incremental problems with respect to the size of input change is presented. We show that our problem falls into the category of *unbounded* problems, i.e. the running time of an incremental update cannot be expressed solely in terms of the size of input change.

Since we do not have values associated with edges, we slightly simplify the original definition from [65] to depend only on the vertices of the graph. The size of the input change is defined in terms of the neighbors of vertices, immediately affected by an update. If K is a subset of vertices of a graph G, then the set of neighbors of order i is defined inductively as $N^i_G(K) = N_G(N^{i-1}_G(K))$, where $N_G(K) = K \cup Adj(K)$ and $N^0_G(K) = K$ ($Adj(K)$ is the set of vertices adjacent to K). We say that a vertex is immediately affected by an incremental update if it was either added during the update, or one of its outgoing edges was added or deleted. We then define the size of incremental update δ, $|\delta|$, to be the number of affected vertices.

An algorithm operating on a graph G is called *bounded* if its running time can be specified in terms of $N^i_G(\delta)$ for some fixed i. Following [65], we would call problem an unbounded one if no bounded algorithm for solving this problem exists within a chosen model of computation.

Our incremental algorithm falls into the class of *locally persistent* algorithms, as defined by [1] for graph problems. Two main characteristics of this model of computation are:

- No global information is maintained between updates. Each vertex of the graph has an associated storage block which contains pointers to successors and predecessors of the vertex, and possibly an arbitrary amount of additional information, but no pointers to non-adjacent nodes.

- The algorithm starts the incremental update with the vertices that have been modified by the update and proceeds by following the pointers in the storage blocks of the vertices. The algorithm may keep global information only about vertices it has visited during the current update.

Ramalingam and Reps [66] showed that there is no bounded incremental algorithm in the locally persistent model for the single-source reachability (SS-REACHABILITY) problem, which is defined as follows: given a directed graph (V, E) and a fixed vertex $s \in V$, find for every vertex $v \in V$ whether v is reachable from s. We use this result to show that the model-checking problem for L_{μ_1} doesn't have a bounded locally persistent algorithm, using the same idea of reduction that is exhibited in [65] for the dataflow analysis problem.

Given an instance of the SS-REACHABILITY problem, $\langle (V, E), s\rangle$, we reduce it to the model-checking problem as follows. Let the LTS be $L = \langle V \cup t, \{a\}, E^a \cup \{t \xrightarrow{a} s, s \in V\}, t\rangle$, where $t \notin V$ and $E^a = \{v \xrightarrow{a} v' \mid v' \rightarrow v \in E\}$ denotes the set of reversed edges of the graph, each labeled by action a. This LTS is checked against the formula $B = min\{X = R \vee \langle a\rangle X\}$. Here R is the atomic proposition which is true for s and false for every other state in L. Intuitively, B means "either R is true in the current state, or is reachable eventually by doing an a-action". Since transition $t \xrightarrow{a} s$ is present in L, the model-checking problem for the whole LTS is trivial, but since our algorithm performs global computation, the value of X will be computed in every state, and will be true in a state v *iff* v is reachable from s. From the unboundedness of SS-REACHABILITY it follows that the model-checking problem is also unbounded.

4.3 Implementation and Examples

The *MCI* algorithm has been implemented as part of the Concurrency Factory project. We started with the implementation of *CS*, which we later modified to make use of incremental computation. Although the non-incremental version is still needed for the initial computation of fixed points, we were able to avoid unnecessary duplication of code. In particular, with only minor changes, the incremental versions of *UP* and *DOWN* produce correct results in the initial computation.

We illustrate *MCI* by extending the example at the end of Section 3.2, treating that example as the first, non-incremental run of the algorithm. We now have the product graph shown in Figure 5, in which every node is assigned false. To make the presentation simpler, we decided not to manipulate edges of the product graph, but instead change the value of atomic proposition p in state s_0 to true. Thus the initialization phase reduces to setting the value of $\langle X_3, s_0\rangle$ to true and adding it to list up_2. Repeated calls to *UP* propagate the value along the cycle in B_2, making all nodes of block B_2 true, except $\langle X_3, s_1\rangle$. Additionally, nodes $\langle X_0, s_0\rangle$ and $\langle X_0, s_1\rangle$ in B_1 each acquired one true successor. This is not enough to make them true, however, as both of them are $\wedge$-nodes. Indeed, now the cycle in B_1 is an interference-free cycle. *MCI* makes an assumption, justified in this case, sets $\langle X_0, s_0\rangle$ and $\langle X_0, s_1\rangle$ to true, and adds them to list up_1. The counters are left at their present values, which is 1 for both nodes. The true value of $\langle X_0, s_1\rangle$ makes $\langle X_1, s_0\rangle$ true as well. The next application of *UP* visits $\langle X_0, s_0\rangle$, which is true already. The counter $C_{\langle X_0, s_0\rangle}$ is decremented and becomes 0, which consistent with the true value of $\langle X_0, s_0\rangle$. This shows the validity of the assumption made for $\langle X_0, s_0\rangle$. The same happens with $\langle X_0, s_1\rangle$. This completes the run of *MCI*.

Note that in this case we did not gain much from the incremental run, but

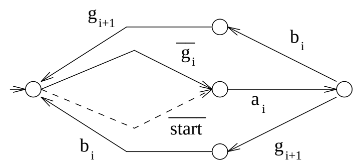


Figure 6: The LTS for cycler C_i

neither did we lose anything since all our assumptions were valid. The algorithm visited all nodes of the product graph except one. We next consider an example of *MCI* run that is intended to demonstrate the best-case behavior of the algorithm. The assumptions concerning the design process, however, seem to be realistic. The system in question is Milner's scheduler [54], consisting of a circular chain of simple "cycler" processes $C_0, \ldots, C_{n-1}$. Milner's scheduler is often used as a benchmark for verification tools, partly because its state space grows exponentially with the size of the scheduler (number of cyclers).

Each C_i is initiated by the previous one in the chain by means of action g_i, after which it carries out the sequence of observable actions a_i, b_i "in parallel" with initiating C_{i+1}. The LTS for C_i is depicted in Figure 6. C_0 must be furnished with a transition labeled by action $\overline{start}$ (the dashed line in Figure 6) that allows it to be initiated by a separate starter process.

Imagine that the designer has completed the scheduler and checks it for the absence of deadlocks. The property "there is a reachable deadlocked state" is expressed by X_1 in the following min-block:

$$\begin{array}{rrcl} min\{ & X_1 & = & X_2 \vee X_3 \\ & X_2 & = & \langle - \rangle X_1 \\ & X_3 & = & [-] X_4 \\ & X_4 & = & ff \qquad \}. \end{array}$$

Modalities with '$-$' instead of an action name are meant to denote that any action action taken by the process is applicable. $[-]\phi$ can be seen as an abbreviation for $\bigwedge_{a_i \in Act}[a_i]\phi$, with $\langle - \rangle\phi$ being the obvious dual.

The scheduler is correct and thus the formula is not satisfied. Imagine now that the designer, in an attempt to simplify the implementation, decides to omit the $\overline{start}$ transition from C_0 and then checks the scheduler again. The scheduler is now deadlocked in the start state, which disconnects the start state from the rest of the scheduler. In a real system, this can easily go unnoticed, and if we do not reconstruct the global state space from scratch, a global algorithm will perform all the unnecessary work with the unreachable part of the state space.

The second, incremental run of *MCI* finds the deadlock. Intuitively, the effects of the update to C_0 should not propagate very far, and Table 1 reveals this to indeed be the case. There, three rows of results are presented: (1) execution times of our implementation of *MCI* on the original, deadlock-free scheduler for increasing numbers of cyclers; (2) execution times for incremental runs of *MCI* on the updated, deadlocked scheduler; and (3) execution times of our implementation of *CS* on the original scheduler. The second row shows that the verification of the updated scheduler can be performed incrementally in *constant time*, independent of the number of cyclers. The third row allows us to compare how the *MCI* and *CS* algorithms perform on the first, necessarily non-incremental verification of the scheduler. As can be seen, the difference in execution times is negligible and, thus, the extra information we maintain in the incremental case does not significantly affect the constant factors of the algorithm.

We have also considered an example of the worst-case behavior of the incremental computation, involving a linear chain of transitions which we again test for deadlock. The update to the LTS is to extend the chain with yet another transition. Obviously, the LTS still has a deadlock, but the assumption made during the initialization phase of the incremental computation results in a wave of changes to the values of the variables that reaches the start state. At this point, it is determined that the assumption was wrong, so the wave of

No. of cyclers		2	3	4	5	6	7	8	9
incr.	1st pass	0.01	0.02	0.05	0.19	0.48	1.36	3.06	7.52
	2nd pass	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01
non-incr.		0.01	0.02	0.07	0.19	0.47	1.22	3.07	7.50

Table 1: Execution times, in seconds, for the scheduler example

changes reverses direction and traverses the whole length of the chain again. The second application of *MCI* ends up taking about 75% more time than the first application.

Chapter 5

Local Model Checking

5.1 The Algorithm *MCL*

In this section we present *MCL*, a local model-checking algorithm that is very similar to the graph-based algorithm by Andersen [9]. Despite their similarity, the algorithms were developed independently. The main difference is that *MCL* was designed to handle arbitrary alternation-free formulas, while Andersen's algorithm works with single-block formulas and is later extended to handle multiple blocks in a rather awkward manner. A more careful comparison with [9] is given below.

The main idea behind local model checking is that, since we are only interested in the value of the formula in the start state of the system, it may not be necessary to look at all states. In our case, we attempt to avoid constructing unnecessary product graph nodes.

The difference between local and global algorithms may be likened to the difference between top-down and bottom up computation in logic programming. *MCL* constructs the product graph in a top-down manner. That is, it starts with a single node, namely $\langle X_0, s_0 \rangle$, and explores adjacent nodes only if necessary to determine the current node's value. As a result, the product graph is explored using depth-first search.

Every node $\langle X, s \rangle$ is assumed to have a default value, which depends on the type of the block X is defined in: true for *max* blocks, false for *min* blocks. Unlike [9], the value of $\langle X, s \rangle$ is not accessible by other nodes until the predicate *done* becomes true for the node, meaning that the fixed-point value of the node has been computed and will not change again.

The notion of a product graph interference-free cycle has been introduced in Chapter 3.1 and plays an important role in the algorithm. Let C be a cycle in the product graph, and let t be a node lying in C. Then, recall, C is *interference-free* in a given environment (i.e. an assignment of values to product graph nodes) if every successor of t not lying in C is false, if t is an $\vee$-node, and every successor of t not lying in C is true if t is an $\wedge$-node. In this case, all nodes in the cycle can be uniformly set to either true or false without violating the semantics of basic formulas.

Unfortunately, depth-first exploration of the product graph by itself cannot determine the values of nodes on an interference-free cycle. Indeed, a path $t \rightarrow^* t$ along an interference-free cycle will lead to a node (t itself) that was previously visited, but whose value is yet undetermined. Furthermore, exploring any successor of any node on this path will not allow us to uniquely determine the value of any node on the path. We must therefore modify the depth-first search procedure to identify interference-free cycles (in general, SCCs) and process them at the appropriate time.

Before presenting the algorithm itself, we concentrate on the data structures employed by *MCL*. A counter, as before, is associated with every product graph node. The counters are used to record the number of potentially true successors of $\vee$-nodes and false successors of $\wedge$-nodes. The counter for a node t is initialized with the number of successors of t. For $\vee$-nodes, counters are decremented as false successors are discovered. When a counter reaches 0, there are no true successors and the value of the node is false. If a true successor is found, the value is finalized immediately. The case of an $\wedge$-node is dual.

The following sets of product graph nodes are used: V is the set of visited nodes (the constructed part of the node space); L is the stack of nodes used in the depth-first search; D is the set of nodes whose value is determined, i.e. *done* is true, but whose predecessors are not yet notified (this is analogous to the M lists in the global algorithm); U is the stack of nodes that have all successors visited but whose value is yet undetermined. U is used in detecting interference-free SCCs in the product graph. V is implemented as a search structure that allows us to determine efficiently whether a given node was visited previously.

In the following pseudo-code, B_t denotes the block to which t belongs. Assume that the successors of a node $\langle X, s\rangle$ are ordered in some way and can be selected in that order (i.e. the operation of choosing the next successor is meaningful).

```
procedure MCL(𝓛, 𝓑)
add ⟨X_0, s_0⟩ to V and L
while L ≠ ∅
    t := first(L)
    if t has no successors then          # t is a terminal node
        decide on t's value
        done(t) := true
        move t from L to D
        processD
    else if ¬done(t) then
        if t has unexplored successors then
            choose the next successor w of t
            introduce the edge t → w
            if w ∉ V                     # w was not visited before
                add w to V and L
            else if done(w)
                report the value of w to t
                if done(t) then
                    add t to D
                    processD
        else                             # all successors explored but ¬done(t)
            move t from L to U
    else remove t from L                 # t is done
    if L is empty or B_t ≠ B_first(L) then
        processU(t)
```

Procedure *processU* removes nodes from U belonging to B_t, where t is passed to *processU* as a parameter. Nodes that are still not *done* at this point are given their default values as their final values.

26 **while** $B_{\mathit{first}(U)} = B_t$
27 remove w from U
28 **if** $\neg done(w)$ **then**
29 $done(w) :=$ true # value of w is its default
30 add w to D
31 *processD*

Procedure *processD* propagates the values of recently done nodes to their predecessors.

32 **while** $D \neq \emptyset$
33 remove some t from D
34 **for each** w such that $w \rightarrow t$ and $\neg done(w)$
35 report the value of t to w
36 **if** $done(w)$ **then**
37 add w to D

Some comments about the pseudo-code are in order. In lines 17 and 35, we refer to a procedure for reporting a value of a node to a predecessor, but do not describe it in detail. The procedure is similar to procedures *UP* and *DOWN* in the global algorithm: based on the type of the predecessor node and the reported value, either the predecessor node's value is finalized immediately or its counter is decremented and the value is finalized if the counter has reached 0. Unlike *UP* and *DOWN*, this is done for one predecessor node at a time instead of all of them.

In line 16, we have encountered a *done* node w, whose final value had previously been propagated to its constructed predecessors by *processD*. At that time, however, t had not yet been identified as a predecessor of w. Simply adding w to D and calling *processD* again would erroneously report w's value to its other predecessors for a second time, which can lead to wrong results. Therefore, line 17 reports w's value exclusively to t.

Finally, we explain the significance of the condition of line 24. As will be shown by Lemma 5.2.2, L can be viewed as a sequence of blocks. The condition becomes true when we remove the last node of B_t from L. Crossing a block boundary is important for the following reason. We have to identify interference-free SCCs of the block graph. We know that an SCC is free of interference only after all nodes from that SCC have been visited and their successors analyzed. However, maintaining dynamic information about the SCCs of the product graph is expensive and contradicts the local approach. Thus, the only indication we have that the exploration of some SCC is complete is when we cross a block boundary in L. This is because all nodes of a product graph SCC graph must belong to the same block (see Lemma 3.1.1).

5.2 Correctness Proof

We now prove correctness of *MCL*. The main result will be Theorem 5.2.3, which shows partial correctness of the algorithm. Additionally, we begin with a straightforward lemma that proves termination of *MCL*.

Lemma 5.2.1 *MCL eventually terminates with* $done(\langle X_0, s_0 \rangle) = true$.

Proof: Let N be the number of nodes in the actual product graph and M be the number of edges. Every node placed in L will eventually be removed from L, and a node is never placed in L more than once. The argument for this is similar to that of depth-first search. Indeed, a node is put in L if it is not already in V; a node is put in L and V simultaneously, and nodes are never deleted from V. Whenever a node appears at the top of L, the next of its successors is explored. Thus every outgoing edge from each node in L is followed at most once. Thus, after iterating the main **while** loop at most $N + M$ times L will become empty and *MCL* terminates.

We argue that every node that is placed in L eventually becomes *done*. Since $\langle X_0, s_0\rangle$ is placed in L initially, this will prove the desired result. When a node is removed from L, it either has *done* true, or it is placed in U. All nodes in U are made *done* when L becomes empty, at the latest.

Finally, *processD* and *processU* obviously terminate since *processU* removes a node from U at every iteration (and never adds a node to U). *processD* also removes a node from D at every iteration of the outer loop. Since *processD* only reports values to unfinished nodes, every node is added to D once, when it is made *done*. $\square$

Let $\langle X_0, s_0\rangle \ldots \langle X_n, s_n\rangle$ be the sequence of nodes kept in L, in top-to-bottom order, at some point during the execution of *MCL*. With any such sequence we can associate a sequence of blocks of the formula as follows: substitute block B_l for every subsequence of nodes $\langle X_i, s_i\rangle \ldots \langle X_{i+k}, s_{i+k}\rangle$ such that all $X_{i+j}, 0 \leq j \leq k$ are defined in B_l. This operation takes a quotient of L w.r.t. the set of blocks in the formula, so we denote the resulting sequence as $L\backslash\mathcal{B}$. The following lemma will be useful in the proof of partial correctness:

Lemma 5.2.2 *Every block B appears in $L\backslash\mathcal{B}$ at most once, and the order of blocks in $L\backslash\mathcal{B}$ respects any topological ordering O of the block graph of $\mathcal{B}$.*

Proof: Since L is the depth-first search stack, the sequence of nodes in L is a path through the product graph. Therefore a block appearing in $L\backslash\mathcal{B}$ more than once represents a cycle in a block graph, which is ruled out by the syntactic restriction we impose on formulas. Also, for any two adjacent nodes $\langle X_{i+1}, s_{i+1}\rangle$, $\langle X_i, s_i\rangle$ in L we have $\langle X_{i+1}, s_{i+1}\rangle \rightarrow \langle X_i, s_i\rangle$ in the product graph and $X_{i+1} \rightarrow X_i$ in the formula graph. Thus either X_i and X_{i+1} belong to the same block, or there is a corresponding dependency between the two respective blocks in the block graph. $\square$

Similarly, we can associate a sequence of blocks $U\backslash\mathcal{B}$ with the nodes in U. It is easy to see that $L\backslash\mathcal{B}$, restricted to the blocks of nodes appearing in U, yields $U\backslash\mathcal{B}$. Indeed, nodes are moved to U from L, and when the last node corresponding to some block B is removed from L, all nodes on top of U belonging to B are removed from U by *processU*.

Theorem 5.2.3 *Let $\mathcal{L}$ be an LTS with start state s_0 and $\mathcal{B}$ a CS-logic formula with main variable X_0. When MCL terminates, the value of $\langle X_0, s_0\rangle$ is true iff $s_0 \in [\![X_0]\!][\![\mathcal{B}]\!]$.*

Proof: We prove that $\forall t = \langle X, s\rangle\ done(t) \Rightarrow (value(t) = \text{true} \Leftrightarrow s \in [\![X]\!][\![\mathcal{B}]\!])$, i.e. after a node is fully processed, its value corresponds to the desired fixed point.

We fix a topological ordering O of the blocks in the block graph. The proof is by induction on the position of a block B in O. As in Theorem 4.2.1, the basis for the induction is provided by an empty block placed first in O. For the induction step we prove the following: after the call to *processU* at line 25,

(1) all constructed nodes of B are *done*;

(2) all constructed nodes of B have their final values correctly computed;

(3) no nodes of B are in U.

(3) is the easiest to prove. Since we encountered a block boundary, by Lemma 5.2.2 and the fact that $U\backslash\mathcal{B}$ is a restriction of $L\backslash\mathcal{B}$, *first*$(U\backslash\mathcal{B}) = B$. The result follows as *processU* removes all nodes from B from the top of U.

For (1), note that every newly constructed node is placed in L. When a node is removed from L, it is either *done* or placed in U. By Lemma 5.2.2, when the condition of line 24 is satisfied, there are no more nodes from B in L. Thus every constructed node of B that has not been placed in U is already done. Removal of nodes from U by *processU* makes them done, if they were not done already. Thus when *processU* finishes, all nodes of B will be done.

It remains to show (2), namely that all nodes of B will acquire correct values. Every node of B that has been made *done* before *processU* is called acquired its value by one of the two following cases:

i. $t = \langle X, s\rangle$ is a terminal node (line 6). We can easily see that the value of t is decided correctly in this case: either X is defined by an atomic proposition A and thus t's value is determined by that of A in s; or X is a modal operator and s doesn't have any matching outgoing transitions, in which case the value of t is determined by the type of modal operator.

ii. As a result of receiving a value from a successor (line 17 or 35). Correctness in this case is proved by induction on the total number of calls to the procedure that reports a value of a *done* node to one of its predecessors. The basis for this induction is provided by the nodes added to D outside of *processD* (lines 8 and 30). The values of these nodes are correct by *i.* and the induction hypothesis. Details of the induction step include reasoning about the reporting of values: depending on the type of the node and the value being reported, the node either becomes *done* or its counter is decremented. It is easy to see that an $\vee$-node never acquires a false value if it has a true successor, and an $\wedge$-node never becomes true if it has a false successor.

To complete the induction step, consider the set U_B of nodes of B in U that are not done at the time *processU* is invoked on B. We show now that when the condition of line 24 is satisfied and t is defined in B, every node in U_B must have its value equal to its default value. Let G be the subgraph of the product graph induced by the nodes in U_B, and consider the supergraph $\overline{G}$ of G induced by the SCCs of G. We argue that the terminal nodes of $\overline{G}$ are interference-free SCCs, while all other nodes are SCCs that contain product graph nodes whose values critically depend on a node in another SCC.

We use a second, nested induction on a fixed reverse topological ordering of $\overline{G}$ to show that the default values are the correct final values for all nodes in G. This induction is well-founded since $\overline{G}$ is obviously acyclic.

All terminal nodes of the supergraph are interference-free SCCs. Were this not the case, the value of some node on such an SCC would have been determined by one of its successors outside of the SCC and it would be done at this moment. Clearly, default values are the correct ones in this case. This gives us the base case for the induction.

For the induction step of the inner induction, assume that B is a max block (as always, the case of a min block is dual). Let w be a successor of some node t in the current SCC C. If w is not in U_B, then it is *done* and its value has been processed correctly by $i.$ or $ii.$ above or the inductive hypothesis of the outermost induction. Alternatively, w can be in some preceding SCC in the ordering on $\overline{G}$. In the latter case, w was set to *true* (which is the default value in a max block) correctly by the induction hypothesis of the inner induction, and *processD* was invoked to propagate the true value of w to t. If t is an $\vee$-node, this would finalize its value at *true*. If t is an $\wedge$-node, C would remain interference-free and should have all its nodes set to *true*. This completes our induction on $\overline{G}$, proving that all nodes from U_B are correctly assigned default values. This, in turn, completes the induction step of the outermost induction, thus proving that all constructed nodes acquire correct values. $\Box$

Note that Lemma 5.2.1 allows us to establish the complexity of *MCL*: the number of iterations of the main **while** loop is linear in the size of the product graph. By the same token, *processD* traverses each edge in the product graph once, and thus cumulative time used by all invocations of *processD* is linear in the size of the product graph. All other operations take constant time, except adding newly constructed nodes to V and checking $w \in V$, which is done once per iteration of the main loop. V is implemented as a balanced tree search structure, which provides for insertion and containment operations to be accomplished in logarithmic time. Putting it all together, we find the running time of *MCL* to be $O((N + | \rightarrow |) \times \log N)$, which matches the worst-case complexity of the best previously known local model-checking algorithm of [9].

Let us now more carefully compare *MCL* to the algorithm of [9]. Both algorithms construct the product graph (called the *boolean graph* in [9]) on-the-fly and explore it in a depth-first manner. After the value of a node is determined, it is propagated to the node's predecessors. However, the two algorithms employ different notions of when the value of a node is determined. *MCL* makes sure that the node's value is never reported until the correct fixed point value is known. The other algorithm reports the value of a node as soon as all its successors have been visited. In the presence of cycles in the product graph, this can lead to an incorrect value being propagated through the graph. Later, when this is discovered, values of some nodes may have to be changed. When the formula contains a single block, in the worst case the value of every node may have to be changed twice. With multiple-block formulas, additional care has to be taken to avoid propagating erroneous values too far through the graph, otherwise the worst case may be exponential in the number of blocks in the formula. The presentation in [9] does not give enough details on handling multiple-block formulas to decide whether it is done efficiently. Since *MCL* propagates only correct values through the graph, the algorithm can be optimized by terminating as soon as the start node becomes *done*.

Once again, we illustrate the works of *MCL* by the same running example that appeared in the two previous chapters. The portion of the product graph that is constructed by the algorithm is shown in Figure 7 in solid lines. The dashed nodes and edges are not visited. Note that the cycle containing nodes

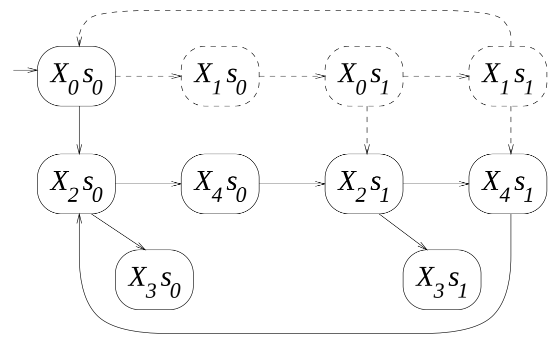


Figure 7: Partially constructed product graph

$\langle X_2, s_0\rangle$, $\langle X_4, s_0\rangle$, $\langle X_2, s_1\rangle$, $\langle X_4, s_1\rangle$ is interference-free. By the time this cycle is closed by the algorithm, L contains the following sequence of nodes:

$$\langle X_0, s_0\rangle \to \langle X_2, s_0\rangle \to \langle X_4, s_0\rangle \to \langle X_2, s_1\rangle \to \langle X_4, s_1\rangle$$

Node $\langle X_4, s_1\rangle$ does not have any other successors except the one already in L. It is thus taken off L and placed in U. The same then happens to its predecessors on the cycle until L contains just node $\langle X_0, s_0\rangle$. Since X_0 is defined in a different block, all nodes from B_2 (in fact, there are no other nodes in U at this time) are to be taken from U and assigned their default values, which are false. However, $\langle X_2, s_0\rangle$ has been the last to be placed in U, and since U is a stack, it will be the first to be taken off. At that point *processD* will propagate the false value to $\langle X_0, s_0\rangle$ and the algorithm will terminate.

Chapter 6

Local Model Checking for Real Time

In this chapter, we present *MCT*, a model-checking algorithm for a timed extension of the modal mu-calculus. *MCT* inherits its overall structure from the algorithm of the previous chapter; as such, most of the arguments for its correctness carry over from the untimed case. In particular, if neither the system specification nor the logical formula contain any timing information, the algorithm performs the same steps as *MCL*.

6.1 Timed Automata

To model real-time systems, we use timed safety automata [38], which are derived from timed automata [5]. Before defining these automata, some auxiliary definitions are in order. A *clock* is simply a real-valued variable drawn from the countably infinite set $\mathcal{C}$. We will use η and ξ, possibly primed and subscripted, to range over subsets of $\mathcal{C}$. A *clock constraint* $\mathbf{c}$ is an expression of the form $x\,R\,c$ or $x + c\,R\,y + d$, where x, y are clocks, c, d are integer constants, and R is taken from $\{\leq, \geq, <, >\}$. A *clock assignment* π to a set of n clocks is a point in $\mathbf{R}^n$, i.e. an assignment of a real value to every clock. If π is a clock assignment, $\pi + d$ for some $d \in \mathbf{R}$ stands for the clock assignment obtained by adding d to the value of every clock in π. Finally, let $\eta \subseteq \mathcal{C}$ be a set of clocks. Then $\pi[\eta := 0]$ is the clock assignment obtained from π by setting every clock in η to 0.

A *region* is a subset of $\mathbf{R}^n$ formed by a set of clock constraints. The meaning of a set of constraints is the conjunction of the member constraints. We will use ρ, possibly primed or subscripted, to range over regions, and Σ to denote the set of all regions. A region ρ is *past-closed* if whenever $\pi \in \rho$, for every $d \in \mathbf{R}^+$, $\pi - d \in \rho$.

A *timed safety automaton* (TSA) is a 6-tuple $\langle \mathcal{L}, Act, \xi, \longrightarrow, l_0, L \rangle$, where

- $\mathcal{L}$ is a finite set of *locations*.
- Act is the set of *actions*.
- ξ is a finite set of real-valued *clocks*.
- $\longrightarrow \subseteq \mathcal{L} \times Act \times \Sigma \times 2^{\xi} \times \mathcal{L}$ is the *transition relation*, where every transition is labeled by an action, a region, and a set of clocks.
- $l_0 \in \mathcal{L}$ is the *start location*.
- $L : \mathcal{L} \to \Sigma$ is the function assigning a past-closed region, called a *location invariant*, to every location.

Intuitively, a TSA operates by taking transitions from location to location. Executing a transition takes no time. If no transitions are taken, time progresses by incrementing every clock by the same arbitrary real number. The intended meaning for the regions labeling locations and transitions is the following. $L(l)$ serves as a *location environment*. The automaton can stay in a location l as long as $L(l)$ is satisfied. Regions labeling the transitions are *enabling conditions*, i.e. a transition can only be taken if the constraints defining its enabling condition are satisfied. Also, when a transition occurs, all clocks in the set of clocks labeling it are reset to 0.

If there is always a way for the automaton to leave the location before its invariant is violated, the automaton is called *non-Zeno*. We restrict our attention to non-Zeno automata, as it was shown in [38] that every automaton can be turned into an equivalent non-Zeno one by strengthening its invariants.

The semantics for TSAs are given in terms of labeled transition systems, which are often called *dense* due to the uncountability of their state sets. Formally, every TSA induces a dense transition system $\mathcal{D} = \langle \mathcal{S}, Act, \longrightarrow, \langle l_0, \pi_0 \rangle \rangle$, where $\mathcal{S}$ is a set of *timed states*, i.e. pairs of the form $\langle l, \pi \rangle$, $l \in \mathcal{L}$, $\pi \in \mathbf{R}^n$; $\langle l_0, \pi_0 \rangle$ is the timed start state with π_0 assigning 0 to every clock; $\longrightarrow \subseteq \mathcal{S} \times (Act \cup \{\epsilon\}) \times \mathcal{S}$ with $\epsilon \notin Act$ being a distinguished action name, is the transition relation defined by the following rules:

- $\langle l, \pi \rangle \xrightarrow{a} \langle l', \pi[\eta := 0] \rangle$ if $l \xrightarrow{a,\rho,\eta} l'$, $\pi \in \rho$, $\pi \in L(l)$, and $\pi[\eta := 0] \in L(l')$.
- $\langle l, \pi \rangle \xrightarrow{\epsilon} \langle l, \pi + d \rangle$ for every $d \in \mathbf{R}$ such that $\pi, \pi + d \in L(l)$.

In the first rule, the automaton moves from location l to l' via action a. Time does not progress, but all clocks in η, $\eta \subseteq \xi$, are reset. Moreover, it must be the case that $L(l)$ and ρ are satisfied by π and $L(l')$ is satisfied by $\pi[\eta := 0]$. Timed state $\langle l', \pi[\eta := 0] \rangle$ is said to be a *transition successor* of $\langle l, \pi \rangle$. In the second rule, time progresses (by d) and the automaton remains in location l, provided that that the invariant for the current location remains satisfied. Timed state $\langle l, \pi + d \rangle$ is said to be a *time successor* of $\langle l, \pi \rangle$.

6.2 Timed Modal Mu-Calculus L_μ^T

We will specify properties of timed automata using the logic L_μ^T, a real-time extension of the modal mu-calculus. Our logic is inspired by both Timed

HML [85, 40] and the timed mu-calculus T_μ of Henzinger *et al.* [38]. As in CS-logic, the logic defined in Chapter 3 for untimed systems, a formula of L^T_μ is a set of *blocks* of mutually recursive equations of the form $X = \phi$, with operator *min* or *max* applied to each block. These operators are understood respectively as the least and greatest fixed points of the set of equations. A variable X is said to be defined in a block B if it appears on the left-hand side of an equation in B. We will restrict our attention to *closed* formulas, where all variables are defined. Also, we only consider alternation-free formulas in which the logical dependency relation between blocks is acyclic (recall that this restriction was also present in CS-logic). We refer to this fragment of the logic as $L^T_{\mu_1}$.

To enhance CS-logic with the ability to specify real-time properties, we augment the syntax and semantics of basic formulas (see Figure 3 in Section 3.1) with additional operators. The additional syntax for basic formulas is

$$\phi ::= A_t \mid x.\phi \mid \exists\phi \mid \forall\phi.$$

Here, A_t is a clock constraint as defined in the preceding section.

The first two of these operators come from T_μ [38]: A_t asserts a relationship between clock values in a state and $x.\phi$ is a *reset* operator. The other operators, $\exists\phi$ and $\forall\phi$, which we refer to as *time modalities*, allow one to reason about the time successors of a given state. They appeared in [40].

Time modalities complement the modal operators of the modal mu-calculus in the following sense. Modal operators $\langle a\rangle$ and $[a]$ are used to reason about transition successors of a given timed state: $\langle a\rangle$ provides existential quantification over the state's transition successors. That is, $\langle a\rangle\phi$ is true in timed state s if from s the automaton can *immediately* (i.e. without letting time progress) take an a-transition and reach a timed state where ϕ holds. Dually, $[a]$ serves as universal quantification. In a similar fashion, $\exists$ and $\forall$ provide quantification over time successors of a timed state: $\exists\phi$ is true in timed state s if there exists a time increment, i.e. a real value, that when added to each clock in s, yields a timed state in which ϕ holds. The dual is true for $\forall\phi$.

The clock reset and clock constraint constructs allow one to introduce clocks into $L^T_{\mu_1}$ formulas and to reason about the values of these clocks. We denote the set of clocks appearing in a formula ϕ either in clock constraints or in reset operators as ξ_ϕ. For every TSA A and $L^T_{\mu_1}$ formula ϕ we assume that their sets of clocks ξ_A and ξ_ϕ are disjoint.

An example $L^T_{\mu_1}$ formula is as follows.

$$\begin{array}{lcll} max & \{ & Y = \forall[-]Y \wedge \forall[a]\, z.X & \} \\ min & \{ & X = \exists\langle b\rangle z \leq 5 \vee (\forall[-]X \wedge \exists\langle -\rangle tt) & \} \end{array}$$

The dash '$-$' in modalities stands for "any action" and is a syntactic abbreviation for a boolean expression over ordinary modalities. The formula states that universally, after performing action a, it is always possible to engage in action b within 5 units of time. The formula is constructed using the following reasoning. Y, which corresponds to the desired property, should hold at every point in time and after every transition. However, if at any moment an a-transition occurs, clock z is reset and the property "b can happen within 5 time units", expressed by X, should hold. The first disjunct of X states that, at some future point, b is possible from the current state and clock z will not exceed the specified value when that move is taken. The purpose of the second disjunct is to allow other transitions to happen before b.

Given a TSA $T = \langle \mathcal{L}, Act, \xi, \rightarrow, l_0, L\rangle$, the semantics for a formula ϕ of $L^T_{\mu_1}$ is given with respect to the dense transition system $\mathcal{L}_T = \langle \mathcal{S}, Act, \rightarrow, \langle l_0, \pi_0\rangle\rangle$ induced by the TSA $T' = \langle \mathcal{L}, Act, \xi', \rightarrow, l_0, L\rangle$, where $\xi' = \xi \cup \xi_\phi$. That is, the set of clocks of T' is extended by all clocks mentioned in ϕ.

As with the untimed logic, we assume a *valuation* mapping $\mathcal{V} : \mathcal{A} \rightarrow \mathcal{P}(\mathcal{S})$ and an *environment* $e : Var \rightarrow \mathcal{P}(\mathcal{S})$. Intuitively, a valuation mapping (environment) maps each atomic proposition (variable) to the set of timed states in which it holds. The valuation mapping is assumed to be untimed in the sense that for all timed states $\langle l, \pi \rangle$ corresponding to the same location l, each proposition A has the same value.

The semantics of the new operators is as follows:

$$
\begin{aligned}
&[\![A_t]\!]e = \{\langle l, \pi \rangle \mid \pi \in A_t \wedge L(l)\} \\
&[\![\exists \phi]\!]e = \{\langle l, \pi \rangle \mid \exists \pi'.\ \langle l, \pi \rangle \xrightarrow{\epsilon} \langle l, \pi' \rangle \wedge \langle l, \pi' \rangle \in [\![\phi]\!]e\} \\
&[\![\forall \phi]\!]e = \{\langle l, \pi \rangle \mid \forall \pi'.\ \langle l, \pi \rangle \xrightarrow{\epsilon} \langle l, \pi' \rangle \Rightarrow \langle l, \pi' \rangle \in [\![\phi]\!]e\} \\
&[\![x.\phi]\!]e = \{\langle l, \pi \rangle \mid \langle l, \pi[x := 0] \rangle \in [\![\phi]\!]\}
\end{aligned}
$$

The semantics for the modal and propositional operators of $L^T_{\mu_1}$ remain as in Section 3.1, when interpreted over the dense transition system $\mathcal{L}_T$. The semantics of equational blocks also remain unchanged (see Section 3.1). The new operators are monotonic on the lattice of tuples of sets of timed states, and thus the desired least and greatest fixed points are guaranteed to exist.

A graph representation similar to the one in the untimed case is used for formulas of $L^T_{\mu_1}$. The only difference is in the way reset operators are treated. The nodes of the expression graph are the variables of the formula, not defined by a reset operator. As before, the node corresponding to variable X is labeled by the operator defining X. The transition relation is given by the following rules:

- If X is the variable on the left hand side of some equation, X' appears on the right-hand side of the same equation and the operator of that equation is op, then $X \xrightarrow{\emptyset} X'$, except when op is the clock reset operator.
- If $X \xrightarrow{\eta} X'$ for some set of clocks η and $X' = x.X''$, then $X \xrightarrow{\eta \cup \{x\}} X''$.

To give an example of how this definition works, the expression graph for $X = Y \vee x.Z$ (this corresponds to two equations with simple right-hand sides) will have nodes for X, Y, Z and edges $X \xrightarrow{\emptyset} Y$ and $X \xrightarrow{\{x\}} Z$. Note that this approach disallows the reset operator to define the main variable of the formula (as there may be no edge leading to the corresponding node in the expression graph to label with the clock of that operator). This is not a restriction, however, since all clocks are initially reset to 0.

6.3 Region Product Graph

As in the untimed case, we would like to employ the idea of a product graph (in this case it should be called a *timed product graph*) as the basis of our model-checking algorithm for $L^T_{\mu_1}$. Unfortunately, the natural extension of the product graph to the timed setting yields a data structure that is infinite and therefore unsuitable for algorithmic purposes. Nonetheless, it is convenient to define the timed product graph as it facilitates reasoning about correctness of our model-checking algorithm.

Let $\mathcal{D} = \langle \mathcal{S}, Act, \longrightarrow, \langle l_0, \pi_0 \rangle \rangle$ be a dense transition system and $\mathcal{B}$ an $L^T_{\mu_1}$ formula. Then the *timed product graph* (TPG) of $\mathcal{D}$ and $\mathcal{B}$ is the directed graph with set of vertices $\{\langle X, l, \pi \rangle \mid X \in Var, \langle l, \pi \rangle \in \mathcal{S}\}$ and set of edges given by:

- if $X(\vee) \xrightarrow{\eta} X'$ or $X(\wedge) \xrightarrow{\eta} X'$, then for every $\langle l, \pi \rangle \in \mathcal{S}$, $\langle X, l, \pi \rangle \longrightarrow \langle X', l, \pi[\eta := 0] \rangle$.
- if $X(\langle a \rangle) \xrightarrow{\eta} X'$ or $X([a]) \xrightarrow{\eta} X'$, $\langle l, \pi \rangle \xrightarrow{a} \langle l', \pi \rangle$, then $\langle X, l, \pi \rangle \longrightarrow \langle X', l', \pi[\eta := 0] \rangle$.
- if $X(\exists) \xrightarrow{\eta} X'$ or $X(\forall) \xrightarrow{\eta} X'$, $\langle l, \pi \rangle \xrightarrow{\epsilon} \langle l, \pi' \rangle$, then $\langle X, l, \pi \rangle \longrightarrow \langle X', l, \pi'[\eta := 0] \rangle$.

Thus there is an edge from $\langle X, l, \pi \rangle$ to $\langle X', l', \pi' \rangle$ if the value of X in timed state $\langle l, \pi \rangle$ depends on the value of X' in timed state $\langle l', \pi' \rangle$. If $\vee$, $\langle a \rangle$, or $\exists$

are used to define X, the vertex $\langle X, l, \pi\rangle$ is called an $\vee$-*node*; otherwise, it is called an $\wedge$-*node*.

A *region product graph (RPG)* is a finite quotient of a timed product graph whose nodes are *sets* of TPG nodes with the same variable and location components and whose time components form a region. For variable X, location l, and region ρ, we denote such an RPG node as $\langle X, l, \rho\rangle$. There is an edge in an RPG from $\langle X, l, \rho\rangle$ to $\langle X', l', \rho'\rangle$ if there exists a node $\langle X, l, \pi\rangle \in \langle X, l, \rho\rangle$ having a successor $\langle X', l', \pi'\rangle \in \langle X', l', \rho'\rangle$ (in terms of the infinite TPG).

This definition actually describes a family of RPGs corresponding to different choices of regions. An RPG is called *stable* if whenever there is an edge $\langle X, l, \rho\rangle \longrightarrow \langle X', l', \rho'\rangle$, every node $\langle X, l, \pi\rangle \in \langle X, l, \rho\rangle$ has a successor in $\langle X', l', \rho'\rangle$. Our local model-checking algorithm will strive to construct the *smallest stable RPG* by making regions as coarse as possible.

Edges of a region product graph are labeled with sets of clocks in the following way:

- if $X(\vee)\overset{\eta}{\to}X'$, $X(\wedge)\overset{\eta}{\to}X'$, $X(\exists)\overset{\eta}{\to}X'$ or $X(\forall)\overset{\eta}{\to}X'$, then edge $\langle X, l, \rho\rangle \to \langle X', l, \rho'\rangle$ is labeled by η.

- if $X([a])\overset{\eta_1}{\to}X'$ or $X(\langle a\rangle)\overset{\eta_1}{\to}X'$ and $l\overset{a,\eta_2}{\to}l'$ then $\langle X, l, \rho\rangle \to \langle X', l', \rho'\rangle$ is labeled by $\eta_1 \cup \eta_2$.

Given an RPG node $\langle X, l, \rho\rangle$ the corresponding *untimed node* is simply $\langle X, l\rangle$. Further, the *untimed successors* of $\langle X, l, \rho\rangle$, are given by the following rules:

- if $X(\vee)\overset{\eta}{\to}X'$, $X(\wedge)\overset{\eta}{\to}X'$, $X(\exists)\overset{\eta}{\to}X'$ or $X(\forall)\overset{\eta}{\to}X'$, then $\langle X', l\rangle$ is an untimed successor of t.

- if $X([a])\overset{\eta_1}{\to}X'$ or $X(\langle a\rangle)\overset{\eta_1}{\to}X'$ and $l\overset{a,\eta_2}{\to}l'$ then $\langle X', l'\rangle$ is an untimed successor of t.

Note that this relation is identical to the one used in the construction of the untimed product graph. As explained in the next section, our local model-checking algorithm uses the untimed successors of RPG nodes to guide the exploration of the RPG.

6.4 The Model-Checking Algorithm

6.4.1 On-the-Fly RPG Construction

Like *MCL*, our local model-checking algorithm for untimed systems (Chapter 5), *MCT* constructs the RPG on-the-fly: the RPG start node is constructed first and new successors are explored one node at a time. For efficiency purposes, the fixed-point computation is performed in tandem with the RPG construction by propagating the value of a newly explored node back to its predecessors. To ensure the correctness of the computation, *MCT* maintains the following two invariants:

I_e: whenever an edge $\langle X, l, \rho\rangle \longrightarrow \langle X', l', \rho'\rangle$ is present in the RPG, every node $\langle X, l, \pi\rangle \in \langle X, l, \rho\rangle$ has a successor in $\langle X', l', \rho'\rangle$.

I_v: the value of X in all timed product graph nodes in an RPG node $\langle X, l, \rho\rangle$ is the same. This value is recorded in a boolean variable associated with the RPG node, which is referred to as the *value* of the node.

I_e states that the constructed portion of the RPG is stable, while I_v allows one to treat an RPG node as an atomic entity rather than an infinite set of TPG nodes. Combined together, the two invariants let the values of RPG nodes be propagated from successors to predecessors during the fixed-point computation in the same way it was done in *MCL*. Indeed, when an RPG node w satisfies I_v and a predecessor t of w satisfies I_v and I_e, if some TPG node in t has a successor in w with a certain value, then every other TPG node in t will have a successor with the same value. Thus after the value of w is propagated to t, I_v will continue to hold for for t. Although a step of the algorithm may temporarily invalidate either invariant, they are restored by splitting RPG nodes, which is discussed below.

MCT explores the RPG in a depth-first manner. A stack of nodes that have unvisited successors is maintained. For each node t appearing on the top of the stack, the algorithm iterates through its successors until the value of the node is determined or all successors are visited. However, in general we cannot tell by looking at t alone what its actual successors are, as this will depend on what part of the RPG has been constructed so far. Consider an example: let $X = \exists X'$, and z be the only clock under consideration, and l be a TSA location with $L(l) = \emptyset$. Then the node $t = \langle X, l, z = 0 \rangle$ will have $w = \langle X', l, z > 0 \rangle$ as a successor if w has not been visited previously by the algorithm. However, if w has been visited and split into, say, $z < 5$ and $z \geq 5$, then t will have *two* successors instead of one.

To address this issue, the algorithm iterates through the untimed successors rather than actual successors of each RPG node. If the next untimed successor of the current RPG node has not been previously visited, the largest possible region is taken to form the new RPG node; assuming the new node's location component is l, then this region is $L(l)$. Otherwise, the untimed node has been visited already and there is a set of corresponding RPG nodes. In this case, edges to appropriate RPG successors are introduced and the current node is split if needed to preserve I_e.

6.4.2 Data Structures

The data structures employed by *MCT* are similar to the ones used in the untimed algorithm, except that we have to account for splitting of RPG nodes.

This necessitates a change to the main search data structure used for storing the RPG nodes that have been generated by the algorithm.

When the algorithm encounters an untimed node $\langle X, l\rangle$ for the first time, we assume that all timed product graph nodes corresponding to this untimed node have the same value and group them together as one RPG node $\langle X, l, L(l)\rangle$. However, as we take timing constraints into consideration, this assumption may turn out to be false, forcing us to split the node. We split an RPG node with one constraint at a time. As more constraints are discovered by the algorithm, the regions are split to become even finer. In order to make splitting efficient, we keep track of the history of splits that were made previously. In particular, RPG nodes having the same untimed node are kept in a data structure that we call an *SH-tree* for "splitting history."

An SH-tree for an untimed node $\langle X, l\rangle$ is a binary tree organized as follows. Each node of the tree represents a region with the root node corresponding to $L(l)$. Each node is labeled by a clock constraint $\mathbf{c}$ (unless it is a leaf) which is used to split the node, forming its children. The left child of the node is the node satisfying $\neg\mathbf{c}$ and the right one satisfies $\mathbf{c}$. Thus the region associated with a node t in an SH-tree is the one defined by the union of all the constraints labeling the path from the root to t (excluding t itself) with $L(l)$.

Figure 8 illustrates the SH-tree data structure; the tree on the left corresponds to the regions on the right. $L(l)$ in this case is the set $\{x < 5, y < 4\}$.

The collection of SH-trees for all untimed nodes encountered so far forms a forest which is similar to the data structure used in [84].[1] This forest, with its root nodes suitably organized, e.g. as a balanced tree, serves as the search structure that is used to determine which untimed nodes were already visited.

We use sets D and U and stack L of RPG nodes, which serve the same purpose as in algorithm *MCL*. The search structure V now becomes the forest

[1] SH-trees are also similar to the data structure used in the partition refinement algorithm for bisimulation checking of Paige and Tarjan [62].

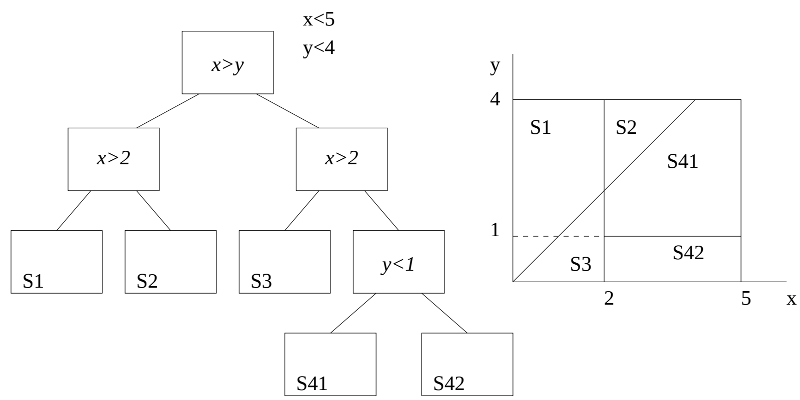


Figure 8: An example of an SH-tree

of SH-trees described above.

We also have a set Sp of RPG nodes that is used to split other RPG nodes. Members of this set are internal nodes of some SH-tree, i.e. they were already split by the algorithm. Procedure *processS* will propagate the effect of all such splits to the predecessors of these nodes.

As in the untimed case, for each RPG node $t = \langle X, l, \rho \rangle$, the value of t is initialized to true if the variable X is defined within a max block and false otherwise. A counter, which is initialized to the number of untimed successors of t, is used to keep track of t's successors. Note that for a time modality node the number of successor RPG nodes may increase due to the splitting of a successor node, affecting the counter value. Otherwise, the number of successors stays constant.

6.4.3 Splitting RPG Nodes

As RPG construction proceeds, more and more timing information, in the form of clock constraints contained in the TSA or $L^T_{\mu_1}$ formula, are encountered. Newly encountered clock constraints may invalidate the I_v and I_e invariants. To re-establish these invariants, certain RPG node regions must be partitioned into finer ones, thus effectively splitting the nodes. In most partition-refinement algorithms, e.g. [42, 62, 84, 15], the process of splitting is tied to the notion of stability of a set of nodes, which involves reasoning about successors of nodes within the set. Here we employ a similar notion. However, we reason about successors indirectly through the clock constraints found in the specification (and those derived from them as described below). That is, TPG nodes satisfying the same constraints will have the same successors. TPG nodes of an RPG node satisfying a given new constraint are considered to be different from the ones that do not satisfy it (as such nodes may acquire different values in the future) and thus have to be kept in separate RPG nodes.

Since regions are defined as sets of constraints, it is easy to split a region ρ by a clock constraint $\mathbf{c}$: simply partition ρ into $\rho \cup \{\mathbf{c}\}$ and $\rho \cup \{\neg\mathbf{c}\}$. Note that one of the new regions may be empty, which means that no splitting has occurred. There are effective procedures, similar to those in [26], to check for emptiness of a region and to "tighten" region boundaries by removing irrelevant constraints.

As indicated above, splits are propagated from successors to predecessors in the RPG. Splits originate with violations of I_v, which may fail to hold for a newly constructed node $t = \langle X, l, L(l)\rangle$ when X is a clock constraint or a transition modality. When X is a clock constraint $\mathbf{c}$, t is split into two nodes, one satisfying $\mathbf{c}$ and the other satisfying its negation. When X is a transition modality, and $\mathbf{c}$ is a constraint on a matching transition from l, the subregion of $L(l)$ violating $\mathbf{c}$ should be split off as it does not have the transition and thus the value of the corresponding RPG node may be different from the subnode of t whose region satisfies $\mathbf{c}$ (see also Section 6.4.4). Formally these two cases are handled as follows:

- $X = A_t$. $L(l)$ is split into two subregions using the constraint A_t. The subnode with region $L(l) \cup A_t$ is assigned true and the subnode with region $L(l) \cup \neg A_t$ is assigned false.

- $X \stackrel{[a]}{\to} X'$ or $X \stackrel{\langle a \rangle}{\to} X'$. $L(l)$ is split using all clock constraints on all a-transitions out of l.

A violation of I_e may arise every time a node $w = \langle X_w, l_w, \rho_w \rangle$ is split, necessitating the splitting of a predecessor $t = \langle X_t, l_t, \rho_t \rangle$ of w. Assume first that X_t is not a time successor operator ($\forall$ or $\exists$), t has not been previously split (i.e. it is a leaf in an SH tree) and $\mathbf{c}$ is the constraint splitting w. The splitting procedure computes the constraint to split t from $\mathbf{c}$ in the following way. First, the set η of clocks that were reset in moving from t to w is detected. This is the set of clocks labeling the edge $X_t \longrightarrow X_w$ in the formula graph and, if X_t is a transition modality ($\langle a \rangle$ or $[a]$), those labeling the corresponding transition in the TSA. Then $\mathbf{c}$ is *projected* onto $\eta = 0$, i.e.

- $\mathbf{c}$ is discarded if it is of the form $x \, R \, c, x \in \eta$ or $x + c \, R \, y + d, x, y \in \eta$. t is not split.

- $\mathbf{c}$ becomes $c - d \, R \, y$ if it is of the form $x + c \, R \, y + d, x \in \eta, y \notin \eta$.

- $\mathbf{c}$ is unchanged otherwise.

We henceforth denote by $\mathbf{c}_\eta$ the result of projecting $\mathbf{c}$ onto $\eta = 0$.

$\mathbf{c}_\eta$ is used to split t, and an edge $t' \to w'$ is introduced between every pair of matching children w' of w and t' of t. The nodes match if, whenever w' satisfies $\mathbf{c}$, t' satisfies $\mathbf{c}_\eta$. For the case when $\mathbf{c}$is discarded, t matches the subnode of w that has a non-empty intersection with the region $\{\eta = 0\}$.

The example in Figure 9 illustrates the three cases of projection and the associated splitting. We are given RPG nodes w and t with $t \longrightarrow w$ (only their regions are depicted for the purpose of illustration), and clocks x and y. Assume that y is reset in going from t to w. Note that in this case, all TPG nodes in w that are successors of some TPG node in t lie on the line

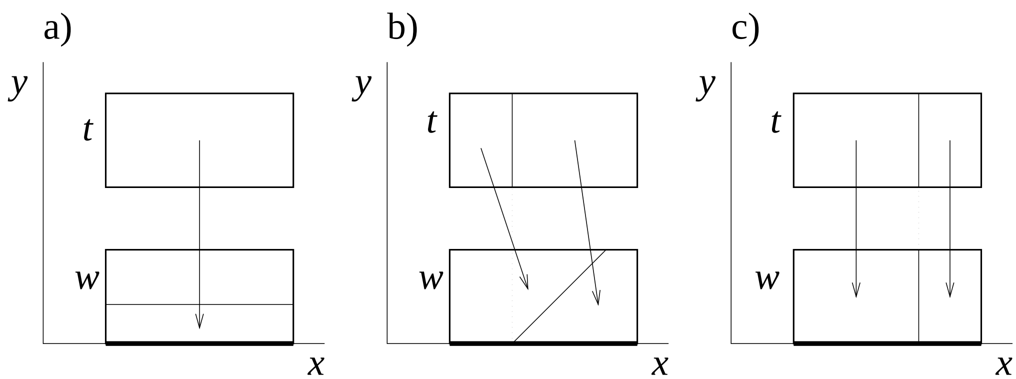


Figure 9: Splitting of t

$y = 0$ (shown in bold in Figure 9). Solid lines in Figure 9 denote constraints, while dashed lines are extensions of $\mathbf{c}_\eta$ beyond t. Arrows indicate the new successor relation. On the left, a constraint of the form $y < c$ is turned into true, and thus t is not split. The bottom subnode of w becomes the successor of t. The middle case depicts a constraint involving two clocks, one of which has been reset. The right case presents a constraint that is not affected by the projection.

When X_t is defined by a time successor operator, additional clock constraints may have to be taken into account. For example, consider Figure 10.b, where w has been split by the one-clock constraint $\mathbf{c} = x < d$, and t is a predecessor of w. All the successors of a TPG node in t lie along the line extended in the direction of time progress. Depending on where this line intersects with $\mathbf{c}$, the TPG node may have successors in the left, the right, or both subnodes of w. Consideration of all TPG nodes in t may thus result in the partitioning of t into (at most) three subnodes. Note that if w is split with a two-clock constraint, a TPG node in t cannot have successors in more than one subnode of w since such a constraint never intersects the time line (Figure 10.a).

Again, let η be the set of clocks reset in moving from t to w. To split t, we first project the constraints splitting w onto η as described above. Projected

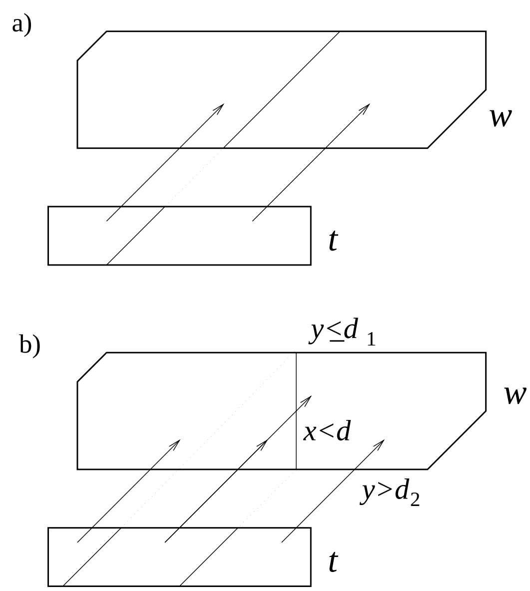


Figure 10: Splitting of a node by time successors

constraints of the form $x + c\, R\, y + d$ are used to split t, as before. However, projected constraints of the form $\mathbf{c}_\eta = x\ R\ d$ are used to split t only if the region of t is contained in the region of w. Otherwise, constraints derived from $\mathbf{c}_\eta$ are used. In particular, for every single-clock constraint $\mathbf{c}' = y\, R'\, d'$ from the set of constraints that form w's region ($y \neq x$), we use the derived constraint $y - d'\ R''\ x - d$ to split t. For each clock y, there are at most two constraints $\mathbf{c}'$ of this kind, and hence at most two derived constraints. The counter of the subnode of t that satisfies both derived constraints is incremented, as it has both subregions of w induced by $\mathbf{c}$ as time successors[2]. The rules to calculate R'' based on R and R' are given in Table 2.

The derived constraints and the rules to compute them are illustrated in Figure 10. Assume, for simplicity, that no clocks are reset. Figure 10a shows the simple case when the same multi-clock constraint that was used to split w

[2]Note that this subnode of t may be empty or coincide with t if neither constraint splits t.

is used to split t as well. In this case, the constraint is parallel to the direction in which time progresses, and thus all successors of every TPG node in t lie on a line parallel to the constraint either in one subnode of w or the other. In Figure 10.b, w is split by the one-clock constraint $c = x < d$, and two derived constraints are used to separate TPG nodes in t that have successors in both subnodes of w from those that have successors only in one of them. Consider first the constraint derived from $\mathbf{c}$ and $\mathbf{c}' = y \leq d_1$ forming the top boundary of w. Call this constraint $\mathbf{c}_1$; from Table 2 we see that $\mathbf{c}_1 = x - d \leq y - d_1$. The argument for this choice of a comparison operator is as follows. The point $(x = d, y = d_1)$ where $\mathbf{c}$ and $\mathbf{c}_1$ intersect belongs to w. Moreover, it belongs to the right subnode of w, while the rest of c_1 lies in the left subnode. This means that points in t lying on the line $x - d = y - d_1$ have successors in both subnodes and should be a part of the subnode with two successors. We thus make the constraint inclusive. On the other hand, the point $(x = d, y = d_2)$ is not a part of w, so TPG nodes of t along the line $x - d = y - d_2$ do not get a successor in the left subnode of w. Thus they are excluded from the middle subnode of t. Note also that although the constraint splitting w also intersects t, it is not used to split t as this split would be irrelevant.

Finally, Figure 11 illustrates the special case where the region of t is contained in the region of w. In this case, one-clock constraints of the form $x\,R\,d$ are also used to split t. Indeed, all TPG nodes in t that satisfy $x > d$ (or $x \geq d$, depending on R) have successors only in the subnode of w that satisfies the same constraint, while those that satisfy its inverse necessarily have successors in the other subnode of w (and maybe in both). Thus the additional constraints are applied only to the subnode of t corresponding to $x < d$ or $x \leq d$ as appropriate. In fact, it is easy to see that only one of the two additional constraints for each clock, the one derived from the top boundary, is used in splitting, as the other one does not intersect with this subnode.

We have assumed until now that neither t nor the subnodes of w induced by

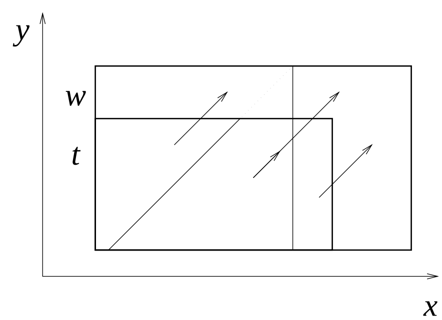


Figure 11: Splitting of an enclosed node

$\mathbf{c}'$ $\mathbf{c}_\eta$	$<$	$\leq$	$>$	$\geq$
$<$	$<$	$<$	$<$	$<$
$\leq$	$\leq$	$<$	$<$	$\leq$
$>$	$>$	$>$	$>$	$>$
$\geq$	$\geq$	$>$	$>$	$\geq$

Table 2: Operator of the additional splitting constraint

c have been previously split. This, however, does not have to be the case. For example, when w corresponds to a transition modality, multiple constraints may have to be applied to w when it is constructed. Thus, before the effect of splitting w is propagated to its predecessors, w's SH-tree may already have multiple levels. To handle this case, whenever a new edge $t' \rightarrow w'$ is introduced as a result of splitting t, the algorithm checks whether w' is split and performs the split of t' if necessary.

Also, when a node t has multiple successors, say w_1 and w_2, by the time w_2 is split and the effect of this splitting is propagated to t, t may have already been split into two or more subnodes through w_1. In this case, to preserve the structure of t's SH-subtree, the constraint splitting w_2 should be applied not to t, but to the leaf subnodes in its SH-subtree.

Both initial (for I_v) and further (for I_e) splitting is implemented in *MCT* using the same procedure *Split*(t, w). In the former case, where no successor node is involved in the splitting, the procedure is invoked as *Split*(t, nil).

6.4.4 The Algorithm

Let $T = \langle \mathcal{L}, Act, \xi, \rightarrow, l_0, L \rangle$ be a TSA and $\mathcal{B}$ an $L^T_{\mu_1}$ formula with X_0 its main variable. By t_0 denote an RPG node that contains $\langle X_0, l_0, \pi_0 \rangle$, the main variable of the formula in the start state of the timed system. Initially, $t_0 = \langle X_0, l_0, L(l_0) \rangle$. Note that t_0's region can change (become smaller) as the algorithm proceeds.

The pseudo-code of *MCT*, our local model-checking algorithm, now follows.

procedure $MCT(T, \mathcal{B})$

add t_0 to V and L

while $L \neq \emptyset$

  $t := first(L)$

  if t has no successors **then**

decide on t's value
$done(t)$:= true
move t from L to D
processD
else if $\neg done(t)$ **then**
if t has unexplored successors **then**
choose the next untimed successor $\langle X, l \rangle$ of t
if $\langle X, l \rangle$ does not have a tree in V # $\langle X, l \rangle$ not visited yet
add $w = \langle X, l, L(l) \rangle$ to V
add edge $t \rightarrow w$
push the elements of $Split(w, nil)$ onto L
processS
else # $\langle X, l \rangle$ visited and possibly split
replace t with $Split(\ t, \langle X, l, L(l) \rangle\ )$ in L
processS
else # all successors visited, move to U
move t from L to U
else remove t from L # t is done
if L is empty or (X_t and $X_{first(L)}$ are in different blocks) **then**
processU(t)

Procedure *processD* propagates the value of a node w whose processing is finished to its predecessors. The difference between *processD* here and in *MCL* is that any such predecessor node t may have been split by another successor and the value of w should be applied to every leaf in t's SH-tree.

while $D \neq \emptyset$
remove some w from D
for each t such that $t \rightarrow w$
for each t' such that t' is a leaf in the SH-subtree rooted in t and $\neg done(t')$

    report the value of w to t'
    if $done(t')$ **then**
        add t' to D

Procedure *processU*, like *processD*, differs from procedure *processU* of *MCL* in that any node removed from U may have been already split and the procedure has to get to the leaves of the SH-tree before assigning values to nodes.

Procedure *processS* gets nodes from the set Sp, which contains nodes that were recently split, and for each predecessor of such a node, calls *Split*. Nodes are added to Sp by *Split*.

while $Sp \neq \emptyset$
  remove some w from Sp
  for each t such that $t \rightarrow w$ and $\neg done(t)$
    $Split(t, w)$

To avoid redundant work, the algorithm never splits a node for which *done* is true. Indeed, such splitting is not necessary as the value of all TPG nodes in this RPG node has been already finalized and the split will not have any bearing on the subsequent fixed-point computation.

Due to insufficient information about the unconstructed portion of the RPG, the algorithm may make splits that later turn out to be irrelevant. For example, consider an $L^T_{\mu_1}$ formula with $X_1 = \langle a \rangle X_2$ and a TSA with transition $l_1 \xrightarrow{a, y>2} l_2$ ($L(l_1) = \emptyset$). When we first encounter the untimed node $\langle X_1, l_1 \rangle$, node $t = \langle X_1, l_1, \emptyset \rangle$ is immediately split into nodes $t_1 = \langle X_1, l_1, y \leq 2 \rangle$ and $t_2 = \langle X_1, l_1, y > 2 \rangle$. t_1 is a terminal $\vee$-node and thus is *done* with its value false. t_2 has an outgoing edge and its successors should be explored. If the result of this exploration makes t_2 false, all subnodes of t are false and the splitting of t into t_1 and t_2 turned out to be unnecessary.

To avoid propagating this unnecessary split of $\langle X_1, l_1 \rangle$, we modify the definition of when a node is *done*. Now, a node is *done* if it is a leaf of an SH-tree and has its value determined by its successors (this corresponds to the definition of *done* in the untimed case), or both children of the node in the SH-tree are *done* and have the same value.

Note that *MCT* could have alternatively explored t's successors (which are the same as t_2's successors) and decided on t's value as false without having ever split t. However, should t_2's value turn out to be true, the split will have to be performed as the values of t_1 and t_2 will be different. Giving preference to forward exploration over splitting in this way would complicate the book-keeping in *MCT*; specifically, reasoning about cycles in the product graph becomes problematic: when the regions are not fully refined, what appears to be a cycle in the RPG may not correspond to cycles of TPG nodes. Therefore, when there is a choice between propagation of splits to predecessors of a node and forward exploration of its successors, we perform splitting to ensure that the fixed-point computation is always performed on a stable RPG.

6.5 Correctness of *MCT*

The proof of correctness for *MCT* consists of two quite independent parts: the correctness of operation *Split*, and the correctness of the computation of the values of nodes in the RPG constructed by the algorithm. Indeed, the main requirement we have for *Split* is that it maintains stability of the RPG (i.e., invariant I_e). This property is proved by purely geometrical reasoning and is not a part of the iterative fixed point computation *per se* (i.e. propagation of values of product graph nodes from successors to predecessors). We therefore defer the proof of correctness of *Split* until the end of this section. The other requirement for *Split* is to make sure that initially constructed RPG nodes have the same value assigned to all constituent TPG nodes without violating the semantics of $L^T_{\mu_1}$, which is trivial to observe.

Given a TSA and an $L^T_{\mu_1}$ formula, let k be the largest constant appearing in the clock constraints within either of them. If we only consider regions formed by constraints in which parameters do not exceed k, we have finitely many regions to deal with.

We use the following strategy to construct the proof. An assignment of values to RPG nodes gives rise to an environment e (see Section 6.2) having the property that timed states $\langle l, \pi \rangle$ belonging to the same RPG node $\langle X, l, \rho \rangle$ either all belong to $e(X)$ or none of them do (i.e. invariant I_v is satisfied). It is clear that the meaning of variable X in such an environment will be a "finite union of regions". In this case, we say that e is *composed of regions* and show that it is possible to perform the fixed-point computation using only environments composed of regions. Then we go one step further and show that the regions composing the environments arising in the course of fixed point computation belong to the same RPG. We say that environment e *respects* a given RPG $\mathcal{R}$ if e can result from assigning values to the nodes of $\mathcal{R}$. We show that the fixed-point computation can be performed using only environments of this form as long as the RPG is stable (i.e. invariant I_e is satisfied). This means that RPG nodes can be treated as atomic entities and the fixed-point computation on an RPG can be performed in the same way as it was on a product graph in the untimed case.

Formally, we say that environment e is *composed of regions* if for every X we have $e(X) = \bigcup_{l \in \mathcal{L}} \{\langle l, \pi \rangle \mid \pi \in \cup_{i \in I_x} \rho_i\}$. Here, I_x is a finite index set associated with variable X.

Let B be a block in an $L^T_{\mu_1}$ formula, with $X_1, \ldots, X_n$ defined in B via set of equations E. Recall (Section 3.1) that the semantics of B, in environment e, is defined in terms of the function f^e_E which essentially transforms e into e' by assigning every X_i defined in B the meaning of the basic formula defining X_i in E. Let $\overline{S}$ be an n-tuple of sets of timed states such that the ith component of $\overline{S}$ is $e(X_i)$. Then:

Lemma 6.5.1 *If environment e is composed of regions, then $f_E^e(\overline{S})$ is also composed of regions.*

Proof: We recollect the definition of *state predicates* from [38] (without atomic propositions). State predicates are given by the grammar $\phi ::= x \leq c \mid d \leq y \mid x + c \leq y + d \mid \neg\phi \mid \phi \vee \phi$. Semantics for constraints are the same as we employ here. It is shown in [38] that state predicates are closed over all boolean operations, and, for any region ρ, the set $\rho_{\downarrow} = \{\pi \mid \exists\delta \geq 0.\, \pi + \delta \in \rho\}$ (the past closure of ρ) can be expressed as a state predicate. It is straightforward to see that a finite union of regions is equivalent to a state predicate in disjunctive normal form.

We need another auxiliary operation on regions. If ρ is a region and η is a set of clock, let $\rho_{\uparrow\eta}$ denote the set $\{\pi \mid \pi[\eta := 0] \in \rho\}$. $\rho_{\uparrow\eta}$ is the region obtained by uniformly substituting 0 for each clock in η, in every constraint of ρ.

We now consider the following cases:

- $X = X_1 \vee X_2$. $[\![X]\!] = [\![X_1]\!] \cup [\![X_2]\!] = \bigcup_{l \in \mathcal{L}}(\{\langle l, \pi\rangle \mid \pi \in \cup_{i \in I_{X_1}} \rho_i\} \cup \{\langle l, \pi\rangle \mid \pi \in \cup_{j \in I_{X_2}} \rho_j\}) = \bigcup_{l \in \mathcal{L}}\{\langle l, \pi\rangle \mid \pi \in \cup_{i \in I_{X_1} \cup I_{X_2}} \rho_i\}$.
- $X = \langle a\rangle X_1$. $[\![X]\!] = \bigcup_{l \in \mathcal{L}} \cup_{l' \xrightarrow{a,\rho,\eta} l} \{\langle l', \pi\rangle \mid \pi \in \cup_{i \in I_{X_1}}(\rho_{i\uparrow\eta} \cap L(l') \cap \rho)\}$.
- $X = \exists X_1$. $[\![X]\!] = \bigcup_{l \in \mathcal{L}}\{\langle l, \pi\rangle \mid \pi \in \cup_{i \in I_{X_1}} \rho_{i\downarrow}\}$.
- $X = z.X_1$. $[\![X]\!] = \bigcup_{l \in \mathcal{L}}\{\langle l, \pi\rangle \mid \pi \in \cup_{i \in I_{X_1}} \rho_{i\uparrow\{z\}}\}$.

Clearly, in each of the above cases $[\![X]\!]$ is finite union of regions since I_X is finite by the condition of the lemma. In the second case, $\rho_{i\uparrow\eta} \cap L(l') \cap \rho$ is a region since the intersection of two regions is a region.

For the remaining operators, we use the facts that $X_1 \wedge X_2 = \neg(\neg X_1 \vee \neg X_2)$, $[a]X = \neg\langle a\rangle\neg X$ and $\forall X = \neg\exists\neg X$, and that finite unions of regions are closed under negation. $\Box$

Now, since there are only finitely many regions, the fixed point of f_E^e can be computed in an iterative manner, by setting, in a *max* block, $\overline{S}_0 = \langle\{\langle l, L(l)\rangle, l \in \mathcal{L}\}^n\rangle$, which is composed of regions, and computing $\overline{S}_{i+1} = f_E^e(\overline{S}_i)$ until $\overline{S}_{i+1} = \overline{S}_i$.

Lemma 6.5.1, however, does not tell us anything about the shape of regions that we will be dealing with during fixed-point computation. In particular, we must ensure that we remain within the same RPG during the computation. This is captured by the following lemma, for which we need the following definition. For an environment e composed of regions, we say that *e respects an RPG $\mathcal{R}$* if for every X, there exists an $M \subseteq \{\langle l, \rho_i\rangle \mid \langle X, l, \rho_i\rangle$ is a node of $\mathcal{R}\}$ such that $e(X) = \bigcup M$. Also, if X is defined by A_t, then $e(X) = A_t$. Thus, e respects $\mathcal{R}$ if it is composed of regions of $\mathcal{R}$.

Lemma 6.5.2 *Let $\mathcal{R}$ be a stable RPG. If e respects $\mathcal{R}$, then $f_E^e(\overline{S})$ also respects $\mathcal{R}$.*

Proof: Consider an RPG node $\langle X, l, \rho\rangle$ of $\mathcal{R}$. Without loss of generality, assume that it is an $\vee$-node (the case of $\wedge$-nodes is dual). Consider any two timed states, $\langle l, \pi_1\rangle$ and $\langle l, \pi_2\rangle$, in $\langle l, \rho\rangle$. Suppose that the TPG node $\langle X, l, \pi_1\rangle$ has a successor $\langle X', l', \pi_1'\rangle$ such that $\langle l', \pi_1'\rangle \in e(X')$ (i.e. X' is true in that timed state). Thus X will be true of $\langle l, \pi_1\rangle$ in the updated environment. Let $\langle X', l', \pi_1'\rangle$ belong to an RPG node $\langle X', l', \rho'\rangle$. Since $\mathcal{R}$ is stable, TPG node $\langle X, l, \pi_2\rangle$ will also have a successor in $\langle X', l', \rho'\rangle$. Since e respects $\mathcal{R}$, the timed state of that successor also will belong to $e(X')$. A dual argument deals with the case when X, in the updated environment, is false of $\langle l, \pi_1\rangle$. Since the argument does not depend on the choice of π_1 and π_2, it proves that all

TPG nodes forming $\langle X, l, \rho \rangle$ will have the same value of X in the environment updated by f_E^e; thus the new environment also respects $\mathcal{R}$. $\Box$

This shows that all timed product graph nodes within one region change their values simultaneously during fixed point computation. Note that the value we assign to RPG node $\langle X, l, \rho \rangle$ of $\mathcal{R}$ can be seen as a uniform value of X in all timed states in ρ in an environment that respects $\mathcal{R}$. We can thus apply the same arguments that were used in the proof of the untimed local model checking algorithm to prove the final result.

Theorem 6.5.3 *Let T be a TSA with start location l_0 and $\mathcal{B}$ an $L_{\mu_1}^T$ formula with main variable X_0. When MCT terminates, the value of t_0 is true* iff *$\langle l_0, \pi_0 \rangle \in [\![X_0]\!][\![\mathcal{B}]\!]$.*

Now we come back and fulfill our promise to show the correctness of procedure *Split*. We claim that if t satisfies I_e and w, a successor of t, is split with a constraint $\mathbf{c}$, then after splitting is performed, the resulting subnodes of t satisfy I_e. Again (as in Section 6.4.3), we start with the case when X_t is not defined by a time successor operator. For this case, *Split* transforms $\mathbf{c}$ into $\mathbf{c}_\eta$. Since before w was split I_e had been true for the RPG edge $t \to w$ we know that every TPG node $\langle X_t, l_t, \pi \rangle \in t$ has the successor $\langle X_w, l_w, \pi' \rangle \in w$ such that $\pi' = \pi[\eta := 0]$. We show by case analysis that π' satisfies $\mathbf{c}$ *iff* π satisfies $\mathbf{c}_\eta$. Since the argument does not depend on the choice of π, this proves that I_e is restored. We consider the following cases:

- All clocks of $\mathbf{c}$ are reset. If $\mathbf{c}$ is of the form $x \le c$ or $x < c$, then, for every successor TPG node, π' satisfies $\mathbf{c}$ since $x = 0$. For $x \ge c$ or $x > c$, π' satisfies $\neg\mathbf{c}$ for the same reason. If $\mathbf{c}$ is $x + c\, R\, y + d$, then, depending on R and the values of c and d, either every π' satisfies $\mathbf{c}$ or none of them does. Thus the successors of every TPG node in t belong to one of the regions resulting from splitting w by $\mathbf{c}$.

- None of the clocks of $\mathbf{c}$ are reset. Thus, for every clock x of $\mathbf{c}$, $\pi(x) = \pi'(x)$, and π satisfies $\mathbf{c}$ *iff* π' satisfies $\mathbf{c}$. Thus, I_e holds again.

- Finally, let $\mathbf{c}$ be of the form $x + c\,R\,y + d$ such that $y \in \eta$, $x \notin \eta$. Then, a TPG node in t has a successor in w such that π' satisfies $\mathbf{c}$ *iff* π satisfies $x\ R\ d - c$ as $\pi(x) = \pi'(x)$ and $\pi'(y) = 0$.

One may easily notice that the three cases above directly correspond to the cases of the transformation of $\mathbf{c}$ into $\mathbf{c}_\eta$ described in Section 6.4.3.

When X_t is a time successor operator, the relation between the clock assignments of a TPG node in t and its successor in w is $\pi' = (\pi + \delta)[\eta := 0]$ for some $\delta \geq 0$. The argument above shows that $\pi + \delta$ satisfies $\mathbf{c}_\eta$ *iff* π' satisfies $\mathbf{c}$. It remains to analyze the relation between π and $\pi + \delta$.

When $\mathbf{c}$ is $x + c\ R\ y + d$, clearly $\pi + \delta$ satisfies $\mathbf{c}$ *iff* π satisfies $\mathbf{c}$, so no further transformation of constraints is needed to split t. This leaves us with the most involved case when $\mathbf{c} = x\ R\ c$. Without loss of generality, let $\mathbf{c}$ be $x \leq c$. In this case, there exist δ_1, δ_2 such that $\pi + \delta_1$ satisfies $\mathbf{c}$, $\pi + \delta_2$ satisfies $\neg\mathbf{c}$. Thus, a TPG node in t potentially has successors in both subnodes of w. Let $y \leq d_1$ and $y \geq d_2$, for some $y \notin \eta$, be among the constraints forming ρ_w. Then if π satisfies $x - c < y - d_1$, for al $\delta > 0$ we have $\pi + \delta$ satisfies $\neg\mathbf{c}$ implies $\Rightarrow \pi + \delta$ satisfies $y > d_1$. Indeed, in this case $0 < x - c < y - d_1$ or $y > d_1$. Thus no TPG node of t can have a successor in the "right" subnode of w. Similarly, if π satisfies $x - c > y - d_2$, for all δ, $\pi + \delta$ satisfies $\mathbf{c}$ implies $\pi + \delta$ satisfies $y < d_2$. The same analysis is performed for other combinations of strict and non-strict inequalities.

6.6 Examples

To test our local model-checking algorithm, we applied it to perform partial verification of the railroad crossing gate controller. This problem has become a

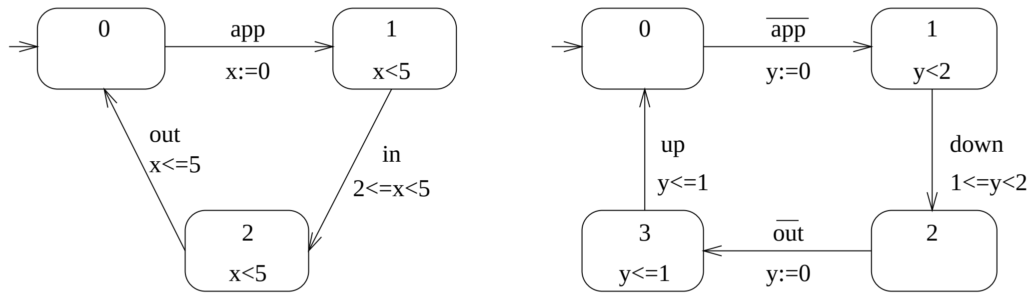


Figure 12: Railroad crossing specification

well-known benchmark for real-time verification algorithms. The specification of the controller as a composition of timed automata is shown in Figure 12. It was taken from [38] without any changes to make comparison easier. The first automaton represents an oncoming train. The train approaching the intersection first sends a warning signal *app* and then, within 2 units of time but no later than 5 units, enters the intersection, sending the signal *in*. It is guaranteed to leave the intersection no later than 5 units after *app* signal was sent. This event is modeled by the signal *out*. The controller reacts to the approach of the train by lowering the gate with the signal *down*, and after the train leaves the intersection, lifts the gate again using the signal *up*.

We also verified the same property as in [38], namely that the gate is never closed for more than five units of time. The $L^T_{\mu_1}$ formula expressing this property, represented using fixed-point operators μ and ν rather than the block format for clarity reasons, is as follows:

$$\nu Y.\forall[-]Y \wedge \forall[down]z.\mu X.\exists\langle up\rangle(z \leq 5) \vee (\forall[-up]X \wedge \exists\langle -up\rangle tt)$$

The formula, intuitively, implements two nested temporal operators. The outermost fixed-point operator is a real-time version of invariance property. The innermost fixed-point implements bounded eventuality operator: "within 5 units of time". Clock z is used to measure time elapsed between actions *down*

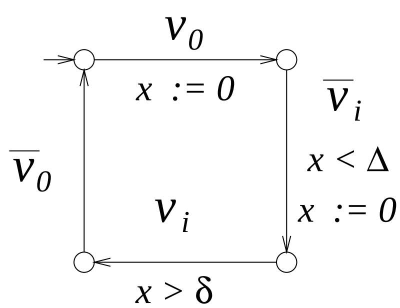


Figure 13: Fischer's mutual exclusion protocol

and *up*. That is, it is reset immediately after *down* occurs and performing *up* should take us to a state where $z \leq 5$.

Verification showed that the property is indeed true for the gate controller. The constructed portion of the RPG contained 203 RPG nodes and verification took 0.36 seconds on a Sun Sparc1000.

Another common benchmark for real-time verification algorithms is Fischer's mutual exclusion protocol [45]. The protocol ensures mutual exclusion among an arbitrary number of identical processes. Figure 13 depicts the ith such process. Mutual exclusion is achieved by using an additional variable v that, for n processes, may assume values between 0 and n. When the critical section is not occupied, the value of the variable is 0. A process i that wants to use the critical section sets the value of v to i (denoted by the $\overline{v_i}$ action) and, before proceeding into the critical section, makes sure that the value of v has not been changed by another process. Although the untimed version of this algorithm does not guarantee mutual exclusion, the real-time version is correct. The idea behind the protocol is that if we know the maximum delay Δ between the moments a process i checks whether the critical section is available ($v = 0$) and writes its value to v, we can make the process wait long enough before it checks the value of v for the second time (for $v = i$) to make sure that any other process that has read $v = 0$ concurrently with i

already finished updating v. By this argument, the algorithm is correct whenever $\delta > \Delta$. Each process has a separate clock x_i that is used for measuring delays.

The protocol provides an interesting benchmark for verification algorithms because it is conceptually quite simple and easy to specify. On the other hand, it exhibits highly exponential behaviour in the number of processes. First, the global timed automaton corresponding to the composition of the constituent automata grows exponentially as processes are added. Additionally, it is known that verification problems for timed automata are exponential in the number of clocks in the specification, and each new process introduces one more clock. Using MCT, we were able to prove the mutual exclusion property for the Fischer's protocol with 2, 3 and 4 processes. This compares well with the results obtained for several other tools in [12]. It should be noted, however, that proving mutual exclusion, like other global properties, presents the worst case for local algorithms when the system is correct — as it is in this case. That is, regardless of how computation proceeds, every reachable state in the system must be checked before a global property can be proved correct. In fact, when our specification of the protocol contained a bug due to an erroneously written constraint, the algorithm performed considerably better in reporting the problem.

Chapter 7

Conclusions

Summary of Results

In this dissertation, we have considered several approaches to model checking formulas in a version of propositional modal mu-calculus with respect to LTSs. The three different approaches investigated were global model checking, incremental global model checking and local model checking. All three approaches used the common framework of product graphs. This provided for more adequate comparison of the approaches and made design of the algorithms and proofs of their correctness easier.

Lessons Learned

Each of the three kinds of algorithms presented here - global, local and incremental - seems to have its own area of application where it is the most efficient. Global algorithms are suitable for systems with relatively small state spaces, since all of the state space has to fit in memory before the computation can proceed. However, global algorithms can offer the best performance, both asymptotically and in the constant factors involved.

Local algorithms are suitable for "local" properties. These are the properties that are concerned with a specific part of the system. For example, if we are checking for the absence of deadlocks, which is a global property, the algorithm has to visit every state in the global state space in order to prove that there are no deadlocked states. In this case, a local algorithm will almost necessarily lose to a global algorithm since it has to spend additional time during the on-the-fly construction of the state space.

On the other hand, negation of a global property is a local property since it concerns only the states violating the property. Thus a local algorithm can be very efficient, for example, in finding deadlocks. We have found, however, that the outcome of the algorithm depends very much on the order in which it explores successors of a product graph node. In many cases, it appears helpful to be able to guide the algorithm by changing this order of exploration.

Our incremental algorithm can be viewed as occupying a position between the global and local algorithms. The algorithm is global in the sense that the whole product graph is constructed before the application of the algorithm. The first, non-incremental run of the algorithm is not different from a run of the global algorithm, except for keeping some additional information for future incremental runs. On the other hand, incremental updates can be expected to affect only a minor part of the product graph and in that sense have a "local" flavour. They can be seen as runs of an "inverted" local algorithm in the following sense. A local algorithm explores successors of a given (typically, initial) product graph node to find those that affect its value. In the same manner, an incremental algorithm explores predecessors of a node affected by the change to see which ones are affected by it.

Directions for Further Research

The similarities between local and incremental algorithms discussed above expressed themselves further in the correctness proofs for the two algorithms. In both cases, we had to reason about strongly connected components of the product graph. This suggests that a more general theory of "locality analysis" on graphs is worth exploring. A uniform treatment of these related problems in terms of abstract mathematical structures will contribute to better understanding of the nature of the problem and may lead to further applications.

The algorithms presented here all use an explicit representation of product graphs. This representation is often wasteful as nodes corresponding to equivalent (in some sense) states will be kept separately. The following two approaches are commonly used to overcome inefficiencies in explicit representations.

One approach, suitable for global algorithms, is minimization with respect to a chosen notion of equivalence – for model checking, such equivalence can be induced by the logical formula in question. There is a wide variety of algorithms for minimization of LTSs (see, for example, [49, 34]). It should be possible to adapt these algorithms to minimization of product graphs. However, incorporating minimization into local algorithms is more difficult since nodes in a partially constructed graph may appear equivalent while in the full graph they are not. The algorithm of Bouajjani *et al.* [15] performs minimization with respect to bisimulation without constructing the whole state space using a symbolic approach (see below). Alur *et al.* [4] extends their algorithm to real-time systems, at the same time providing formula-dependent minimization.

The other successfully used approach (which may be considered as a generalization of the first one) is the application of symbolic techniques. Symbolic algorithms operate not on single entities (be it states, system variables or graph nodes), but directly on sets of these entities. This approach produces substantial reductions in many practical cases [16]. However, the symbolic approach is usually seen as orthogonal to the local one. Combining the two seems to be an interesting and challenging problem.

The symbolic approach is even more important when the verified system has an infinite state space. In this case, symbolic approaches help to come up with finite quotients of the state space. In particular, our real-time model checking algorithm treats sets of TPG nodes as RPG nodes. However, *MCT* considers locations of the TSA individually and can benefit from symbolic treatment of TSA locations. A similar approach has been taken in [4]. However, our algorithm may be potentially more efficient due to the history of splits kept by *MCT*, so a symbolic extension to *MCT* is worth considering.

An interesting application for the incremental algorithm has come up in a recent discussion with Bernhard Steffen [71]. When performing model checking for mu-calculus formulas of alternation depth 2, recomputations of the inner fixed point due to the influence of the outer one can be expected to be small and can potentially be processed efficiently using an incremental approach.

There is another open problem concerning the incremental algorithm. In general, an incremental update of the fixed point may take *longer* than recomputation from the very beginning. It would be useful to have a way to estimate for a given update, whether one should try incremental recomputation of fixed points or compute fixed points from scratch.

Bibliography

[1] B. Alpern, R. Hoover, B. K. Rosen, P. F. Sweeney, and F. K. Zadeck. Incremental evaluation of computational circuits. In *Proc. of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms*, 1990.

[2] R. Alur. *Techniques for Authomatic Verification of Real-Time Systems.* PhD thesis, Stanford University, 1991.

[3] R. Alur, C. Courcoubetis, and D. Dill. Model-checking for real-time systems. *Information and Computation*, 104(1):2–34, 1993.

[4] R. Alur, C. Courcoubetis, N. Halbwachs, D. Dill, and H. Wong-Toi. Minimization of timed transition systems. In *Proceedings of CONCUR'92*. LNCS 630, 1992.

[5] R. Alur and D. L. Dill. The theory of timed automata. *Theoretical Comput. Sci.*, 126(2), 1994.

[6] R. Alur and T. Henzinger. A really temporal logic. In *Proceedings of FOCS '89*, pages 164–169. IEEE Computer Society Press, 1989.

[7] R. Alur and T. A. Henzinger. Logics and models of real time: A survey. In *Real-Time: Theory in Practice*. LNCS 600, 1991.

[8] R. Alur and T. A. Henzinger. Real-time logics: Complexity and expressiveness. *Information and Computation*, 104(1):35–77, 1993.

[9] H. R. Andersen. Model checking and boolean graphs. *Theoretical Comput. Sci.*, 126(1), 1994.

[10] J. C. M. Baeten and J. A. Bergstra. Real time process algebra. *Formal Aspects of Computing*, 3(2), 1991.

[11] J. C. M. Baeten and W. P. Weijland. *Process Algebra.* Cambridge University Press, 1990.

[12] J. Bengtsson, K. G. Larsen, F. Larsson, P. Petersson, and W. Yi. Uppaal - a tool suite for automatic verification of real-time systems. In *Proceedings of the 4th DIMACS Workshop on Verification and Control of Hybrid Systems*, 1995.

[13] O. Bernholz, M. Y. Vardi, and P. Wolper. An automata-theoretic approach to branching-time model checking. In *Proceedings of CAV'94.* LNCS 818, 1994.

[14] G. Bhat and R. Cleaveland. Efficient local model-checking for fragments of the modal μ-calculus. In *Proceedings of TACAS'96*, 1996.

[15] A. Bouajjani, J.-C. Fernandez, N. Halbwachs, P. Raymond, and C. Ratel. Minimal state graph generation. *Sci. Comput. Programming*, 18(3):247–269, 1992.

[16] J. R. Burch, E. M. Clarke, K.L.McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of LICS '90.* IEEE Computer Society Press, 1990.

[17] B. Catanzaro. *Multiprocessor system architectures : a technical survey of multiprocessor/multithreaded systems using SPARC, multilevel bus architectures and Solaris (SunOS).* Prentice Hall, 1994.

[18] E. M. Clarke and E. A. Emerson. *Design and Synthesis of Synchronization Skeletons Using Branching Time Temporal Logic.* LNCS 131, 1981.

[19] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications. *ACM Trans. Prog. Lang. Syst.*, 8(2), 1986.

[20] R. Cleaveland. Tableau-based model checking in the propositional mu-calculus. *Acta Inf.*, 27, 1990.

[21] R. Cleaveland, M. Dreimüller, and B. Steffen. Faster model checking for the modal mu-calculus. In *Proceedings of CAV '92*, 1992.

[22] R. Cleaveland, J. N. Gada, P. M. Lewis, S. A. Smolka, O. V. Sokolsky, and S. Zhang. The concurrency factory – practical tools for specification, simulation, verification and implementation of concurrent systems. In *Proceedings of the DIMACS Workshop on Specification Techniques for Concurrent Systems, Princeton, NJ.*, 1994.

[23] R. Cleaveland and B. Steffen. A linear-time model checking algorithm for the alternation-free modal mu-calculus. *Formal Methods in System Design*, 2, 1993.

[24] O. Coudert, C. Berthet, and J. Madre. Verification of synchronous sequential machines based on symbolic execution. In *Proceedings of CAV '91.* LNCS 407, 1989.

[25] J. de Bakker *et al.*, editor. *Real-Time: Theory in Practice*, Proceedings of REX Workshop. LNCS 600, 1991.

[26] D. Dill. Timing assumptions and verification of finite-state concurrent systems. In *Proceedings of CAV'89.* LNCS 407, 1990.

[27] E. A. Emerson. Real-time and the mu-calculus. In de Bakker *et al.* [25].

[28] E. A. Emerson, C. S. Jutla, and A. P. Sistla. On model checking for fragments of μ-calculus. In *Proceedings of CAV'93*. LNCS 697, 1993.

[29] E. A. Emerson and C.-L. Lei. Efficient model checking in fragments of the propositional mu-calculus. In *Proceedings LICS '86*. IEEE Computer Society Press, 1986.

[30] E. A. Emerson, A. K. Mok, A. P. Sistla, and J. Srinivasan. Quantitative temporal reasoning. In *Proceedings of CAV'89*, 1989.

[31] J.-C. Fernandez and L. Mounier. "on the fly" verification of behavioural equivalences and preoders. In *Proceedings of CAV '91*. LNCS 575, 1991.

[32] M. J. Fischer and R. Ladner. Propositional dynamic logic of regular programs. *Journal of Computer and System Sciences*, 18:194–211, 1979.

[33] A. Giacalone, P. Mishra, and S. Prasad. Facile: A symmertric integration of concurrent and functional programming. *International Journal of Parallel Programming*, 18(2), 1989.

[34] S. Graf and B. Steffen. Compositional minimization of finite state systems. In *Computer-Aided Verification '90*. LNCS 531, 1990.

[35] R. Grossman, A. Nerode, A. Ravn, and H. Rischel, editors. *Hybrid Systems*. LNCS 736, 1993.

[36] E. Harel, O. Lichtenstein, and A. Pnueli. Explicit-clock temporal logic. In *Proceedings of LICS '90*, pages 402–413. IEEE Computer Society Press, 1990.

[37] T. A. Henzinger. *The Temporal Specification and Verification of Real-Time Systems*. PhD thesis, Stanford University, 1991.

[38] T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111(2), 1994.

[39] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall Intl., 1985.

[40] U. Holmer, K. G. Larsen, and W. Yi. Deciding properties of regular real timed processes. In *Proceedings of CAV'91.* LNCS 575, 1991.

[41] S. Jain. VTSIM: A graphical simulator for finite-state networks. Master's thesis, Department of Computer Science, North Carolina State University, 1994.

[42] P. C. Kanellakis and S. A. Smolka. CCS expressions, finite state processes, and three problems of equivalence. *Information and Computation*, 86(1):43–68, May 1990.

[43] R. M. Keller. Formal verification of parallel programs. *Communications of the ACM*, 19(7):371–384, 1976.

[44] D. Kozen. Results on the propositional mu-calculus. *Theoretical Comput. Sci.*, 27:333–354, 1983.

[45] L. Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 5(1):1–11, 1987.

[46] F. Laroussinie, K. G. Larsen, and C. Weise. From timed automata to logic - and back. In *Proceedings of MFCS'95*, 1995.

[47] K. G. Larsen. Modal specifications. In *Proceedings of CAV'89.* LNCS 407, 1990.

[48] K. G. Larsen. Efficient local correctness checking. In *Proceedings of CAV'92*. LNCS 663, 1992.

[49] D. Lee and M. Yannakakis. Online minimization of transition systems. In *Proceedings of ACM Symposium on Theory of Computing*, 1992.

[50] I. Lee and S. A. Smolka, editors. *Proceedings of CONCUR '95*, LNCS 962, 1995.

[51] D. E. Long, A. Browne, E. M. Clarke, S. Jha, and W. M. Marrero. An improved algorithm for the evaluation of fixpoint expressions. In D. L. Dill, editor, *Proceedings of CAV '94*, LNCS 818, pages 338–350, 1994.

[52] N. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of PODC'87*, 1987.

[53] M. Merritt, F. Modugno, and M. Tuttle. Time constrained automata. In *Proceedings of CONCUR'91*. LNCS 527, 1991.

[54] R. Milner. *A Calculus of Communicating Systems*. LNCS 92, 1980.

[55] R. Milner. A proposal for standard ML. Technical Report CSR-157-83, Department of Computer Science, University of Edinburgh, 1984.

[56] R. Milner. *Communication and Concurrency*. Prentice Hall Intl., 1989.

[57] F. Moller and C. Tofts. A temporal calculus of communicating systems. In *Proceedings of CONCUR'90*. LNCS 458, 1990.

[58] F. Moller and C. Tofts. Behavioural abstraction in tccs. In *Proceedings of 19th ICALP*. LNCS 623, 1992.

[59] X. Nicollin, J.-L. Richier, J. Sifakis, and J. Voiron. Atp: an algebra for timed processes. In de Bakker *et al.* [25].

[60] X. Nicollin and J. Sifakis. An overview and synthesis on timed process algebras. In *Proceedings of CAV'91*. LNCS 575, 1991.

[61] J. Ostroff. *Temporal Logic of Real-Time Systems*. Research Studies Press, 1990.

[62] R. Paige and R. E. Tarjan. Three partition refinement algorithms. *SIAM Journal of Computing*, 16(6):973–989, 1987.

[63] A. Pnueli. The temporal logic of programs. In *Proceedings of FOCS '77*, pages 46–57. IEEE Computer Society Press, 1977.

[64] L. L. Pollock and M. L. Soffa. An incremental version of iterative data flow analysis. *IEEE Trans. Software Engineering*, 15(12), 1989.

[65] G. Ramalingam. *Bounded Incremental Computation*. PhD thesis, Computer Sciences Dept., University of Wisconsin-Madison, 1993.

[66] G. Ramalingam and T. Reps. On the computational complexity of incremental algorithms. Technical Report TR-1033, Computer Sciences Dept., University of Wisconsin-Madison, 1991.

[67] G. M. Reed and A. W. Roscoe. A timed model for communicating sequential processes. *Theoretical Comput. Sci.*, 58, 1988.

[68] B. G. Ryder, T. J. Marlowe, and M. C. Paull. Conditions for incremental iteration: Examples and counterexamples. *Sci. Program.*, 11(1), 1988.

[69] O. Sokolsky and S. A. Smolka. Incremental model checking in the modal mu-calculus. In *Proceedings of CAV'94*. LNCS 818, 1994.

[70] O. Sokolsky and S. A. Smolka. Local model checking for real-time systems. In P. Wolper, editor, *Proceedings of CAV '95*, LNCS 939, pages 211–224, 1995.

[71] B. Steffen. Private communication. August 1995.

[72] C. Stirling. An introduction to modal and temporal logics for ccs. In *Concurrency: Theory, Language and Architecture*. LNCS 491, 1989.

[73] C. Stirling and D. Walker. Local model checking in the modal mu-calculus. *Theoretical Comput. Sci.*, 89(1), 1991.

[74] W. Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B. Formal models and semantics, pages 133–191. Elsevier, 1990.

[75] V. Trehan. VTVIEW: A graphical editor for hierarchical networks of finite-state processes. Master's thesis, Department of Computer Science, North Carolina State University, Dec. 1992.

[76] F. Vaandrager and N. Lynch. Action transducers and timed automata. In *Proceedings of CONCUR'92*. LNCS 630, 1992.

[77] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of LICS'86*, pages 322–331. IEEE Computer Society Press, 1986.

[78] M. Y. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Control*, 115(1):1–37, 1994.

[79] K. Čerāns. Decidability of bisimulation equivalences for parallel timer processes. In *Proceedings of CAV '92*. LNCS 663, 1992.

[80] K. Čerāns. Ctr: A calculus of timed refinement. In Lee and Smolka [50], pages 516–530.

[81] K. Čerāns, J. C. Godskesen, and K. G. Larsen. Timed modal specification — theory and tools. In *Proceedings of CAV'93*. LNCS 697, 1993.

[82] B. Vergauwen and J. Lewi. A linear algorithm for solving fixed-point equations on transition systems. In *Proceedings of CAAP '92*. LNCS 581, 1992.

[83] B. Vergauwen, J. Lewi, I. Avau, and A. Pote. Efficient computation of nested fix-points, with applications to model checking. In *Proceedings of ICTL'94, 1st Intl. Conference on Temporal Logic*. LNCS 827, 1994.

[84] M. Yannakakis and D. Lee. An efficient algorithm for minimizing real-time transition systems. In *Proceedings of CAV '93*. LNCS 697, 1993.

[85] W. Yi. *A Calculus of Real Time Systems*. PhD thesis, Chalmers University of Technology, 1991.