Chapter 1

Three Case Studies on Verification of Infinite-State Systems

Javier Esparza and Jörg Kreiker
Technische Universität München,
Boltzmannstr. 3, 85748 Garching bei München, Germany
{*esparza,kreiker*}*@in.tum.de*

1.1. Introduction

The automatic verification of systems has made immense progress in the last two decades, in particular thanks to the efforts of the model-checking, program-analysis, and theorem-proving communities. Systems with finitely many states can be automatically verified is, at least in principle, by exhaustively exploring their state space. Symbolic search procedures which use special data structures to compactly store large sets of states have made this a technique with many important applications, especially in the hardware area. However, most software systems have an infinite number of reachable states due to unbounded data structures, timing information, or other factors (see below). While many properties can be decided by analyzing a suitable finite-state abstraction of the system, these abstractions are difficult to find, and may lead to finite state systems far too large for the existing tools. For this reason, since the early 90s much effort has been devoted to identifying classes of infinite-state systems with decidable verification problems.

Infinite state-spaces are often caused by the system exhibiting one or more of the following features:

- **Variables with unbounded data domains** such as integers or dynamic data types such as lists, trees, and DAGs. While the physical constraints of a machine impose a bound on the domain, the bound is too large to be of any practical use.
- **Control structures** like procedure calls and thread creation. Both can generate an unbounded number of processes whose execution has not terminated, and whose local states become part of the global state of the system.
- **Time**. In real-time systems, the current time – be it discrete or continuous – becomes part of the state.

- **Communication mechanisms** based on buffers or queues. Like variables, the physical implementation of a buffer or a queue always has a finite capacity, but the capacity may heavily depend of the details of the implementation, and the bounds are usually too large to be of any use for finite-state verifiers. For these reasons, mathematical models often assume unbounded capacity.
- **Parameters.** A system may contain parameters whose values are left undefined, like for instance the number of processes in a leader election protocol. When these parameters have an infinite domain of possible values, the system becomes in fact an infinite family of systems. For most purposes the family is equivalent to the infinite system consisting of all members of the family running independently of each other.

Each of these five "sources of infinity" has been studied in the literature using a variety of formal models: timed automata, channel systems, extensions of Petri nets, pushdown processes, broadcast protocols, counter automata, different process algebras, and rewrite systems to name but a few. This makes it difficult to perceive research on infinite-state systems as a field on its own. Moreover, due to the space constraints of conferences and to their intended audience, papers on infinite state systems usually follow a "horizontal" approach: a paper in a theoretical conference may present a decidability result, leaving an efficient algorithm for further research, or describe an efficient algorithm, leaving its implementation for further research; a paper in a more applied conference may describe an implementation and report on experimental results, without presenting the underlying theory, and without explaining the steps conducting from the system to a formal model.

This paper is a modest attempt at describing the underlying connections between many of the models and techniques used in the field. It does so by introducing the key concept of symbolic search in a very general framework (Section 1.2), along with the conditions for turning it into an effective algorithm. It then presents three case studies (Sections 1.3, 1.4, and 1.5) following a "vertical" approach: first, a (very small) verification challenge, consisting of a system and a desirable property, is informally described; then, a formal model of the system is presented; then, the theory of symbolic search for an adequate class of models is developed, leading to an effective algorithm for deciding the property; finally, the algorithm is executed on the formal model. The concrete case studies are: a small mutex protocol modeled as timed automaton (Section 1.3), a cache-coherence protocol modeled as a linear automaton (Section 1.4), and a skyline plotter modeled as a pushdown automaton (Section 1.5).

The paper does not contain any novel technical material, and in fact it unashamedly draws from several excellent papers by different colleagues; in particular we mention [1–3]. We hope that it may help researchers from other fields to understand some of the principles of automatic verification in the infinite case, and also provide useful material for undergraduate and graduate courses. In fact, the

paper is based on a course given by the first author at the Marktoberdorf Summer School in 2005.

1.2. Symbolic Search for Extended Automata

We introduce symbolic search for *extended automata*, automata whose transitions depend on and update a finite set of variables. In Sections 1.3, 1.4, and 1.5 we will consider three different classes of extended automata.

1.2.1. *Extended Automata*

Loosely speaking, an extended automaton is an automaton whose transitions are guarded by and operate on a set of variables. The variables may have arbitrary types: integers, reals, stacks, queues, etc. More formally, let $X = \{x_1, \ldots, x_n\}$ be a finite set of *variables* over sets $V_1, \ldots, V_n$ of *values*, and let $\mathcal{V} = V_1 \times \ldots \times V_n$ be the set of all possible *valuations* of X.

An *extended automaton* over X is a pair $E = (Q, R)$ where

- Q is a finite set of *control states*, and
- R is a set of *transition rules* or just *rules*. A rule is a tuple $r = (q, g, a, q')$, where
  - $q, q' \in Q$ are the *source* and the *target state* of r, respectively;
  - $g \subseteq \mathcal{V}$ is the *guard* of r; and
  - $a \subseteq \mathcal{V} \times \mathcal{V}$ is the *action* of r.

We often use predicates over the set X to describe guards, and predicates over X and a copy X' to describe actions. The variables in X and X' denote valuations before and after executing the action, respectively. For instance, if $X = \{x_1, x_2\}$ and $V_1 = V_2 = \mathbb{N}$ then we write $x_1 + x_2 > 5$ for the guard $\{(n_1, n_2) \in \mathbb{N} \times \mathbb{N} \mid n_1 + n_2 > 5\}$, and $x_1' = x_2 + 3 \wedge x_2' = 1$ for the action $\{((n_1, n_2), (n_1', n_2')) \mid n_1' = n_2 + 3 \wedge n_2' = 1\}$

An extended automaton is *finite* if $\mathcal{V}$ is finite; otherwise it is *infinite*.

The semantics of extended automata is what one would expect. A *configuration* is a pair $\langle q, \nu \rangle$, where q is a control state and ν is a valuation. If $\nu = (v_1, \ldots, v_n)$, we also write $\langle q, v_1, \ldots, v_n \rangle$. The set of all configurations of an extended automaton E is written $\mathcal{C}_E$ or $\mathcal{C}$, if E is clear from the context. The *transition system* $\mathcal{T}_E$ of an extended automaton E has: all elements of $\mathcal{C}_E$ as nodes, and an edge $\langle q, \nu \rangle \Longrightarrow \langle q', \nu' \rangle$ whenever E has a rule $r = (q, g, a, q')$ such that $\nu \in g$ (we say that ν *satisfies the guard* g, or that $\langle q, \nu \rangle$ *enables* r) and $(\nu, \nu') \in a$. So, loosely speaking, a rule is enabled if the extended automaton is in the source state, and the current valuation satisfies the guard; an enabled rule can occur, and its occurrence leads the automaton to the target state and to a new valuation determined by the execution of the action; note that the action can be nondeterministic.

If $\langle q, \nu \rangle \Longrightarrow \langle q', \nu' \rangle$ then $\langle q', \nu' \rangle$ is an *immediate r-successor* of $\langle q, \nu \rangle$, and $\langle q, \nu \rangle$ is an *immediate r-predecessor* of $\langle q', \nu' \rangle$. We say that a configuration is an *immediate*

successor (predecessor) of another if there exist a rule r in the extended automaton such that the configuration is an immediate r-successor (predecessor) of the other one.

1.2.2. *The Safety Problem*

Given an extended automaton E and a rule r of E, we define the immediate successor functions $post_r^E, post^E \colon 2^{\mathcal{C}_E} \to 2^{\mathcal{C}_E}$ as follows: c belongs to $post_r^E(C)$ if some immediate r-predecessor of c belongs to C, and c belongs to $post^E(C)$ if some immediate predecessor of c belongs to C. Usually E is clear from the context and we write $post_r(C)$ and $post(C)$. Clearly, if R is the set of rules of E we have $post(C) = \bigcup_{r \in R} post_r(C)$. The reflexive and transitive closure of $post$ is denoted by $post^*$, and so we have $post^*(C) = \{c' \in C \mid \exists c \in C.\ c \Rightarrow^* c'\}$. Similarly, we define $pre(C)$ as the set of immediate predecessors of elements in C and pre^* as the reflexive and transitive closure of pre. Given two configurations c, c' of an extended automaton E, we say that c' is *reachable* from c, or that c is *backward reachable* from c' if $c' \in post^*(\{c\})$ or, equivalently, if $c \in pre^*(\{c'\})$.

We are interested in the *safety problem* for extended automata, defined as follows: Given an extended automaton E over a set X of variables, a set I of initial configurations, and a set D of "dangerous' configurations, are there $c \in I$ and $c' \in D$ such that c' is reachable from c in E? Notice that we do not require the sets I and D to be finite.

The safety problem is undecidable even for very simple extended automata (notice that the sets I and D might be non-recursive), or for sets I and D containing only one element (Turing machines are a special class of extended automata). In this paper we consider *constrained safety problems* that impose restrictions on

(i) the type and/or the number of variables in the set X,
(ii) the possible combinations of guards and actions,
(iii) the possible sets of initial configurations,
(iv) the possible sets of dangerous configurations.

Some combinations of restrictions correspond to the reachability problem for well-known models of computation. Let us consider two examples.

Example 1.1. Counter machines are extended automata over nonnegative integer variables, i.e., $X = \{x_1, \ldots, x_n\}$ and $V_1 = \ldots = V_n = \mathbb{N}$. There are only two kinds of rules, with the following two combinations of guards and actions: $\mathbf{true}/x_i' = x_i + 1$ (where **true** is the empty guard, i.e., the guard indicating that the rule is enabled at any valuation) and $x_i > 0 / x_i' = x_i - 1$ (i.e., decrement x_i by 1, but only if $x_i > 0$).

Example 1.2. Pushdown systems are extended automata with a single variable over words, i.e., $X = \{x_1\}$ and $V_1 = \Gamma^*$ for some finite alphabet Γ. There is a single combination of guard and action, which can be represented as γ / w for $\gamma \in \Gamma$ and

$w \in \Gamma^*$: the guard γ indicates that the rule is only enabled when $x = \gamma w'$ for some word w'; the action consists of substituting γ by the word w, that is, x becomes ww' after the action.

1.2.3. *Symbolic Search*

When the set of valuations of an extended automaton is finite, the safety problem can be solved by a *forward* or a *backward* search. In forward search, we start at the set I of initial configurations and explore the space of reachable configurations moving "forward", i.e., moving from a configuration to its immediate successors. The search terminates when a dangerous configuration is reached, or when all the reachable configurations have been explored. Backward search starts at the set D, moves from configurations to their immediate predecessors, and terminates when an initial configuration is reached, or when all configurations that can be reached backwards have been explored. Notice that backward search may explore configurations that are not reachable from I.

In conventional search techniques a set of configurations is stored by explicitly storing each of its elements at a different memory address. For large configuration spaces this can be very costly, and for infinite configuration spaces impossible. The idea of *symbolic search* is to directly manipulate possibly infinite *sets of configurations* by means of adequate data structures. For instance, a linear inequation like $x > 5$ can be used as a data structure representing the infinite set of integers $\{6, 7, 8, ...\}$; a regular expression like $(ab)^*$ (or its corresponding finite automaton) can be used to represent the infinite set $\{\epsilon, ab, abab, \ldots\}$ of words. We say that a linear inequation is a *symbolic representation* of its set of solutions and that a finite automaton is a *symbolic representation* of the language it accepts.

The abstract structure of symbolic search is very simple. The forward case is shown in Figure 1.1.

Forward symbolic search

Initialize $C := I$

Iterate $C := C \cup post(C)$ until

$C \cap D \neq \emptyset$; return "reachable", or

a fixed point is reached; return "non-reachable"

Fig. 1.1. The abstract structure of symbolic forward search.

In order to make symbolic search effective for a constrained safety problem, it suffices to find a family $\mathcal{F}$ of sets of configurations satisfying **Conditions (1)-(6)** below. We call the elements of $\mathcal{F}$ *symbolic configurations*:

- **Condition (1).** Every symbolic configuration has a (not necessarily unique) finite symbolic representation.
- **Condition (2).** The set I of initial configurations is a symbolic configuration, i.e., $I \in \mathcal{F}$.
- **Condition (3).** If $C \in \mathcal{F}$, then $C \cup \mathit{post}(C) \in \mathcal{F}$. Moreover, the symbolic representation of $C \cup \mathit{post}(C)$ is effectively computable from the symbolic representation of C.
- **Condition (4).** For every $C \in \mathcal{F}$, the emptiness of $C \cap D$ is decidable (i.e., there is an algorithm that takes the symbolic representations of C and D as input, and decides whether the set $C \cap D$ is empty).
- **Condition (5).** Equality of symbolic configurations is decidable, i.e., there is an algorithm that takes the symbolic representations of two sets of $\mathcal{F}$ as input and decides whether they represent the same set. This is needed in order to check if a fixed point has already been reached: If after executing $C := C \cup \mathit{post}(C)$ the old and new values of C coincide, then the search can stop.
- **Condition (6).** If $\mathit{post}^*(I) \cap D = \emptyset$, then the chain $C_0 \subseteq C_1 \subseteq C_2 \ldots$ of symbolic configurations, where $C_0 = I$ and $C_{i+1} = C_i \cup \mathit{post}(C_i)$, reaches a fixed point after finitely many steps, i.e. there is an index i such that $C_i = C_{i+j}$ for every $j \geq 0$.

Conditions (1)-(5) are needed so that the algorithm can run. Condition (6) guarantees termination. Notice that Condition (5) can be relaxed. It suffices to have an algorithm satisfying the following properties for every $i \geq 0$: if $C_i \neq C_{i+1}$ then the algorithm answers "not equal"; if $C_i = C_{i+1}$ (and so $C_i = C_{i+j}$ for every $j \geq 0$), then there is a $k \geq i$ such that for input C_k, C_{k+1} the algorithm answers "equal". In other words, if two symbolic configurations are not equal, then the algorithm immediately answers "not equal". If they are equal, then the algorithm may not immediately recognize it, but it eventually answers "equal".

The algorithm and the effectivity conditions for symbolic backward search are obtained from those for forward search by exchanging I and D and *pre* and *post*.

Forward and backward search look deceptively symmetric, but in practice they are not. In Section 1.4 we will find a case study in which the set D of dangerous configurations has a certain property not enjoyed by the set I of initial configurations. This property will make backward search from D effective, while forward search from I may not terminate.

1.2.4. *The Powerset Case*

Often the sets belonging to $\mathcal{F}$ are finite unions of sets belonging to another family $\mathcal{B}$. That is, for every $C \in \mathcal{F}$ there are sets $B_1, \ldots, B_m \in \mathcal{B}$ such that $C = B_1 \cup \ldots \cup B_m$. In this case one can symbolically represent C by the set of the symbolic representations of the B_i's (we assume that they have one), and conditions (3) and (4)

can be simplified a bit. Since $post(C) = \bigcup_{r \in R} post_r(C) = \bigcup_{r \in R} \bigcup_{1 \leq i \leq m} post_r(B_i)$, Condition (3) can be replaced by: if $B \in \mathcal{B}$, then $post_r(B) \in \mathcal{F}$ for every rule r (this holds in particular if $post_r(B) \in \mathcal{B}$). Condition (4) can be replaced by: for every $B \in \mathcal{B}$, it is decidable whether $B \cap D = \emptyset$.

1.2.5. *Making Symbolic Search Terminate*

In the next sections we present three concrete verification challenges that can be solved using symbolic search. They are examples of three different, and increasingly sophisticated, ways of making Condition (6) hold:

- **Finite $\mathcal{F}$.** If we manage to find a family $\mathcal{F}$ containing only finitely many sets, say k, then Condition (6) is guaranteed to hold, since necessarily $C_k = C_{k+i}$ for every $i \geq 0$. This method will be applied to timed automata in Section 1.3.
- **Finite ascending chains.** If the family $\mathcal{F}$ is infinite but only contains finite ascending chains with respect to set inclusion, then the chain $C_0 \subseteq C_1 \subseteq C_2 \ldots$ of Condition (6) contains a set C_j such that $C_j = C_{j+k}$ for every $k \geq 0$, and so the condition holds. We will use this reasoning in the case of linear automata in Section 1.4.
- **Accelerations.** This is a special case of the well-known widening technique of abstract interpretation [4]. Assume that $\mathcal{F}$ has infinite ascending chains, but we manage to find an operator $\nabla \colon \mathcal{F} \times \mathcal{F} \to \mathcal{F}$ satisfying the following properties:

  (a) $C \nabla post(C) \supseteq C \cup post(C)$ for every $C \in \mathcal{F}$, and
  (b) the chain $(\widehat{C}_i)_{i \geq 0}$, where $\widehat{C}_0 = C_0$ and $\widehat{C}_{i+1} = \widehat{C}_i \nabla post(C_i)$, reaches a fixed point after finitely many steps.

  We call such an operator ∇ a *widening* operator. Consider the modified forward search algorithm in which the line $C := C \cup post(C)$ is replaced by $C := C \nabla post(C)$. The modified algorithm clearly terminates. However, after termination only $C \supseteq post^*(I)$ is guaranteed, and not $C = post^*(I)$, because at some step of the algorithm we can have $C \subseteq post^*(I)$ but $C_i \nabla post(C_i) \not\subseteq post^*(I)$, i.e., the application of ∇ may produce non-reachable states. But assume that ∇ further satisfies

  (c) for every $C \in \mathcal{F}$: $C \nabla post(C) \subseteq post^*(C)$

  In this case we know that Condition (6) holds for the modified algorithm. We call a modified forward search algorithm with a widening operator satisfying (a), (b) and (c), an *acceleration* of forward search. An instance of this acceleration will be applied to pushdown automata in Section 1.5.

1.3. Timed Automata

As outlined in the Introduction, this section and the next two follow a vertical approach. First we introduce a small but natural verification challenge in an informal way. Then we introduce a class of extended automata and use it to produce a formal model. After that, we define a family of symbolic configurations and show that it satisfies Conditions (1)-(6) of symbolic search. The crucial part is mostly Condition (6), and we shall use increasingly sophisticated techniques to establish it, as described in Section 1.2.5.

1.3.1. *The Case Study: A Small Mutex Algorithm*

Consider the following simple version of the well-known *mutual exclusion* problem. Two processes wish to access a shared resource, and access is granted by entering a critical section. It must be ensured that exactly one process is granted access (we say that this process "wins" and the other "loses"). There is no central process guarding access to the resource, and the processes can only communicate through *one* single shared boolean variable v.

If the variable can be tested and set to a new value *in a single atomic action* then there is a simple solution:

> The initial value of v is set to 1. Both processes execute the following algorithm: if $v = 0$, give up (the process loses); otherwise, *and in the same atomic action*, set v to 0; then enter the critical section (the process wins).

Clearly, the first process to access the variable wins, and the other loses.

If an atomic action can *either* test the value of v *or* set it to a new value, *but not both*, then it is well-known that there is no solution within the limits of a conventional programming language: more variables are needed. However, Michael Fischer observed that there is a very simple solution if one assumes that processes also have access to a clock:

> The initial value of v is arbitrary. The i-th process, with $i \in \{1, 2\}$, executes the following algorithm: at any time before a time unit passes (*strictly* before), the process sets v to i; then it waits for any amount of time exceeding one time unit (*str ictly* exceeding), and then it tests whether $v = i$ holds: if $v = i$, then the process enters the critical section (and wins), otherwise it gives up (and loses).

Notice that time is not assumed to be discrete; how much less or more than one time unit the process waits should be irrelevant for the correctness of the algorithm.

We give a formal model of Fischer's protocol as a *timed automaton*, and automatically check that exactly one process accesses the critical section. Timed automata (TA) were introduced in Alur and Dill's seminal paper [5]. Most of the ideas and illustrating examples (except for the worked out case study) are based on

the excellent survey paper by Bengtsson and Yi [1], with a number of changes and additions to make the description fit the framework presented in Section 1.2.3.

1.3.2. *Timed Automata*

Timed automata are a special class of extended automata over variables called *clocks*, and with two kinds of rules, called *time-delay* and *state-switch* rules. The formal definition requires some preliminaries.

We call a variable ranging over the non-negative reals a *clock*. Throughout the section we assume a set $\mathcal{X} = \{x_1, \ldots, x_k\}$ of clocks. Consequently, a valuation is a vector of k nonnegative real numbers.

A clock constraint is defined by the following BNF, where $\sim \in \{<, \leq, =, \geq, >\}$, $x, y \in \mathcal{X}$, and n is a non-negative integer:

$$g \ ::= \ \mathbf{true} \mid x \sim n \mid x - y \sim n \mid g \wedge g$$

A constraint is *pure* if it does not contain any atomic formulas of the form $x - y \sim n$. A clock valuation ν may *satisfy* a clock constraint g, written $\nu \models g$. The satisfaction relation is defined as follows:

- $\nu \models \mathbf{true}$ for every valuation ν
- $\nu \models x \sim n$ iff $\nu(x) \sim n$
- $\nu \models x - y \sim n$ iff $\nu(x) - \nu(y) \sim n$
- $\nu \models g_1 \wedge g_2$ iff $\nu \models g_1$ and $\nu \models g_2$

Given a valuation ν of a set $\{x_1, \ldots, x_n\}$ of clocks and a real number δ, we denote by $\nu + \delta$ the valuation given by $(\nu + \delta)(i) = \nu(i) + \delta$ for every $i \in \{1, \ldots, k\}$. The *time-delay action* is the relation $TD = \{(\nu, \nu + \delta) \mid \nu \in \mathcal{V}, \delta > 0\}$. Given a subset $Y \subseteq \mathcal{X}$ of clocks and a valuation ν, we denote by ν_Y the valuation given by $\nu_Y(i) = 0$ for every $x_i \in Y$ and $\nu_Y(i) = \nu(i)$ for every $x_i \notin Y$. The *Y-reset action* is the relation $R_Y = \{(\nu, \nu_Y) \mid \nu \in \mathcal{V}\}$.

A *timed automaton* is an extended automaton over $\mathcal{X}$ whose rules satisfy the following conditions:

- For each control state q there is a rule $(q, 2^{\mathcal{V}}, TD, q)$, i.e., a rule that is enabled at every clock valuation, does not change the state, and whose action advances all clocks by a nondeterministically chosen lapse of time δ.
- All other rules are of the form (q, g, R_Y, q') for some *pure* clock constraint g and some $Y \subseteq \mathcal{X}$; i.e., the action consists of resetting the clocks in Y, while keeping the values of those in $\mathcal{X} \setminus Y$.

Rules of the first and second kind are called *time-delay* and *state-switch* rules, respectively. Notice that since the time-delay rules are completely determined by the set of states one does not need to mention them explicitly. For this reason, they are omitted in the usual syntax for timed automata and in the graphical representation.

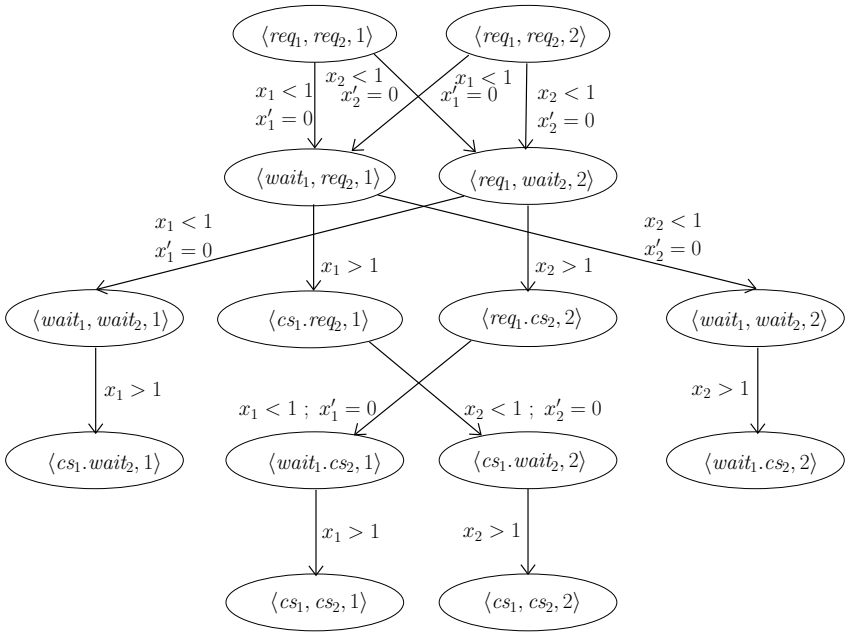

Fig. 1.2. A small mutex algorithm modeled as a timed automaton. Control locations are triples representing the local state of the first process, the second process, and the value of a local variable, which can be either 1 or 2.

We can now model our simplified version of Fischer's mutex algorithm as the timed automaton shown in Figure 1.2. A more conventional way of depicting the protocol in terms of a *network* of two automata is given in Figure 1.3.[a] The automaton has two clocks, x_1 and x_2, one for each process; we denote the action of resetting clock x_i by $x_i' = 0$. We assume that the i-th process ($i = 1, 2$) can be in three states: req_i, previous to setting v to i; $wait_i$, where the process waits before testing if $v = i$ holds, and cs_i, in which the process has reached the critical section. A state of the automaton is a triple (q_1, q_2, v) where q_1 and q_2 are the current states of processes 1 and 2, and v is the current value of the shared variable. A configuration of the automaton consists thus of a state of the automaton and the values of the two clocks. We write it as a five-tuple; for example, $(req_1, req_2, 1, 0.3, 0.3)$ denotes the configuration where process 1 is in state req_1, process 2 is in state req_2, the shared variable has value 1 and both clocks, x_1 and x_2, currently show 0.3 time units. Here are two sample runs of the automaton, where a number on top of a transition indicates the time elapsed.

[a]Loosely speaking, the network consists of two timed automata working in parallel with common clocks. We refrain from giving a formal semantics here.

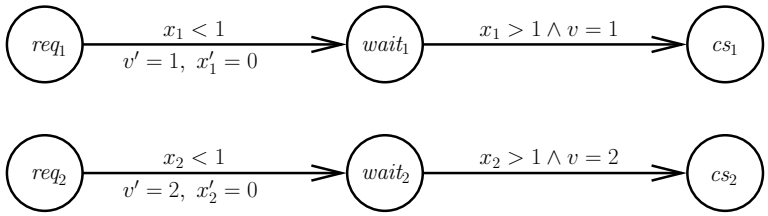

Fig. 1.3. The case study specified more compactly as a network of two automata. The full version of Figure 1.2 results from this one by a product construction.

Sample Run 1:

$$\begin{aligned}(req_1, req_2, 1, 0, 0) &\Longrightarrow (wait_1, req_2, 1, 0, 0)\\ &\overset{2.1}{\Longrightarrow} (wait_1, req_2, 1, 2.1, 2.1)\\ &\Longrightarrow (cs_1, req_2, 1, 2.1, 2.1)\end{aligned}$$

Sample Run 2:

$$\begin{aligned}(req_1, req_2, 1, 0, 0) &\overset{0.7}{\Longrightarrow} (req_1, req_2, 1, 0.7, 0.7)\\ &\Longrightarrow (wait_1, req_2, q, 0, 0.7)\\ &\overset{0.2}{\Longrightarrow} (wait_1, req_2, 1, 0.2, 0.9)\\ &\Longrightarrow (wait_1, wait_2, 2, 0.2, 0)\\ &\overset{1.8}{\Longrightarrow} (wait_1, wait_2, 2, 2, 1.8)\\ &\Longrightarrow (wait_1, cs_2, 2, 2, 1.8)\end{aligned}$$

1.3.3. *Zones and Symbolic Search*

In this section we introduce a family $\mathcal{F}$ of symbolic configurations for timed automata, and show that both forward and backward search satisfy Conditions (1) to (6) (actually, we only present the case of forward search and leave backward search, which is analogous, to the reader).

For the rest of the section we fix a timed automaton $\mathcal{A}$ with a set Q of states over a set $\mathcal{X}$ of clocks, where $k = |\mathcal{X}|$.

Let max be the maximum integer appearing in the guards of the state-switch rules of $\mathcal{A}$. A clock constraint has max as *ceiling* if all the integers appearing in it are smaller than or equal to max. A set of valuations is a *zone* of $\mathcal{A}$ if it is the set of *all* valuations satisfying a clock constraint (pure or not) with ceiling max. We denote the set of all zones by $\mathcal{Z}$. Notice that, while a clock constraint uniquely determines a zone, the same zone may correspond to many clock constraints. Observe further that $\mathcal{Z}$ is finite: while there are infinitely many clock constraints with ceiling max, only finitely many of them are non-equivalent. We call the elements of the set $Q \times \mathcal{Z}$ *indexed zones*. We identify an indexed zone (q, Z) and the set of configurations $\{\langle q, z\rangle \mid z \in Z\}$,

We choose the family $\mathcal{F}$ of symbolic configurations as follows: a set of configurations C is an element of $\mathcal{F}$ if there are indexed zones $(q_1, Z_1), \ldots, (q_m, Z_m)$ such that $C = \bigcup_{i=1}^{m} (q_i, Z_i)$. We say that C is a union of indexed zones, and write $\mathcal{F} = \mathcal{P}(Q \times \mathcal{Z})$, i.e., we choose $\mathcal{F}$ as the powerset of the set of indexed zones.

In the remainder of this section we check that this family satisfies Conditions (1) to (6) for forward search. Since Condition (1) requires the elements of $\mathcal{F}$ to have a finite symbolic representation, we first introduce *Difference Bound Matrices* [6] to represent zones, and describe some of their properties.

Difference Bound Matrices (DBMs) In order to define DBMs, we add an additional reference variable $\mathbf{0}$ with constant value 0 to the set of clocks, and write $\mathcal{X}_0 = \mathcal{X} \cup \{\mathbf{0}\}$. Any clock constraint g, can then be rewritten as a conjunction of constraints of the form $x - y \preceq n$ for $x, y \in \mathcal{X}_0$, $\preceq \in \{<, \leq\}$, and $n \in \mathbb{Z}$. First, every constraint involving only one variable, say $x < 20$, can be rewritten as $x - \mathbf{0} < 20$. Second, teh conjunction of two constraints on the same pair of variables (e.g. $x_1 - x_2 \leq 3$ and $x_1 - x_2 \leq 4$), can be replaced by the intersection of both ($x_1 - x_2 \leq 3$ in our case). Therefore, every constraint is equivalent to another one with one atomic constraint for each pair of clocks from $\mathcal{X}_0$, and so g can be represented as a $(k+1) \times (k+1)$ matrix, where each element corresponds to an atomic constraint.

Example 1.3. Consider the zone

$$g_0 = x < 20 \ \wedge \ y \leq 20 \ \wedge \ y - x \leq 10 \ \wedge \ y - x \geq 5 \ \wedge \ z > 5$$

over clocks $\{x, y, z\}$. It can be rewritten using $\mathbf{0}$ and $<, \leq$ only:

$$g_0' = x - \mathbf{0} < 20 \ \wedge \ y - \mathbf{0} \leq 20 \ \wedge \ y - x \leq 10 \ \wedge \ x - y \leq -5 \ \wedge \ \mathbf{0} - z < -5$$

The DBM of g_0' is a 4×4 matrix, whose entries are elements of $(\mathbb{Z} \times \{<, \leq\}) \cup \{\infty\}$. Assuming $\mathbf{0}$ has index 0 and x, y, z have indices 1, 2, and 3, respectively, entry $(-5, \leq)$ in row 2, column 3 denotes $x - y \leq -5$, and entry ∞ in row 4, column 1, means "$z - \mathbf{0} \leq \infty$", that is, z has no upper bound. Entries on the diagonal will always be $(0, \leq)$ to include trivial constraints like $x - x \leq 0$. The matrix, $M(g_0')$, corresponding to g_0' is then:

$$\begin{pmatrix} (0,\leq) & (0,\leq) & (0,\leq) & (-5,<) \\ (20,<) & (0,\leq) & (-5,\leq) & \infty \\ (20,\leq) & (10,\leq) & (0,\leq) & \infty \\ \infty & \infty & \infty & (0,\leq) \end{pmatrix} = M(g_0')$$

Closing a DBM. Different DBMs can represent the same zone. Consider for instance the DBM above, and the one in which the constraint $x - \mathbf{0} < 20$ is replaced by $x - \mathbf{0} \leq 15$. Clearly, the zone of the new DBM is included in the zone of the DBM above. But the contrary also holds: every clock assignment of the DBM above

satisfies $x - y \leq -5$ and $y - \mathbf{0} \leq 20$, and so it also satisfies $x - \mathbf{0} \leq 15$; in other words, the constraint $x - \mathbf{0} < 20$ of the original DBM can be tightened without changing the zone.

Fortunately, each nonempty zone has a *unique* DBM that cannot be further tightened. The process of finding it is called *closing* a DBM and it yields a canonical DBM.

For the purpose of this presentation, it shall suffice to say that the canonical form of a DBM can be computed by a graph interpretation of a DBM, where each clock is a node, and a constraint between two clocks is a edge labeled by the constraint. Then closing a DBM amounts to computing all pairs-shortest paths, for example by using an algorithm like Floyd-Warshall [7].

For instance, in the graph representation of $M(g'_0)$ there is an edge labeled $(20, <)$ from x to $\mathbf{0}$. However, there is also an edge from x to y labeled $(-5, \leq)$ and an edge1 from y to $\mathbf{0}$ labeled $(20, \leq)$. Going from x to $\mathbf{0}$ via y adds up to $(15, \leq)$ and is in fact a shorter path than the direct one.

We will mostly write down zones as clock constraints or graphically. In particular, for two zones, we will draw some illustrating diagrams as the ones in Figure 1.4.

In the rest of the section we sketch the proof of the following theorem:

Theorem 1.1. *Let $\mathcal{A}$ be a timed automaton, and let I and D be elements of $\mathcal{P}(Q \times \mathcal{Z})$. Forward search satisfies Conditions (1)-(6).*

Condition (1). Since a zone is finitely represented by its associated clock constraint (or DBM), each element of $\mathcal{P}(Q \times \mathcal{Z})$ has a finite representation as a set of pairs (q, g), where q is a state, and g is a clock constraint.

Condition (2). The set I of initial configurations is an element of $\mathcal{P}(Q \times \mathcal{Z})$ by hypothesis. In the case of Fischer's mutex algorithm the singleton set of clock assignments $\{(0, 0)\}$ is a zone, because it is the set of solutions of the clock constraint $x_1 = 0 \wedge x_2 = 0$. Since $I = \{(req_1, req_2, 1, 0, 0), (req_1, req_2, 2, 0, 0)\}$, we have $I \in \mathcal{P}(Q \times \mathcal{Z})$.

Condition (3). We have to show that if C is an element of $\mathcal{P}(Q \times \mathcal{Z})$, then so is $C \cup post(C)$. Since the set of symbolic configurations is the powerset of another set, we can apply the observation to Condition (3) in Section 1.2.4; so it suffices to show that for every indexed zone (q, g) and for every rule r of the timed automaton the set $post_r(\{(q, g)\})$ is the union of indexed zones. In fact, we can show a stronger result: either $post_r(\{(q, g)\})$ is empty (and so the union of the empty set of zones), or a zone (q', g') (and so the union of a set of zones containing only one element). We consider separately the case in which r is a time-delay and a state-switch rule.

- **Time-delay rules**: Consider an indexed zone (q, g). Notice that if g implies a constraint $x - y \leq n$, then so do all clock valuations reachable from g by a time delay (because a delay increases the values of all clocks by the same amount). Furthermore, the valuations reachable through time delays can reach arbitrarily large values for each clock. From this observation it is

easy to see that if g is given as a DBM $M(g)$, then the valuations reachable from those satisfying g by means of time delays are represented by the DBM obtained by setting all entries of $M(g)$ in column 0 to ∞. Recall that column 0 contains entries representing $x - \mathbf{0} \leq n$, that is, just upper bounds on individual clocks. From this DBM we can easily get a constraint g' for the set of reachable valuations, and we have $post_r(\{(q,g)\}) = (q,g')$.

- **State-switch rules**: Consider an indexed zone (q,g) and a state-switch rule $r = (q, g'', R_Y, q')$. If $g \wedge g''$ is unsatisfiable, then $post_r(\{(q,g)\})$ is empty. If $g \wedge g''$ is satisfiable, let $M(g \wedge g'')$ be a closed DBM for $g \wedge g''$ (which can be computed because clock constraints are closed under conjunction). We compute another closed DBM corresponding to resetting the clocks of Y in all clock valuations satisfying $g \wedge g''$. This DBM is obtained from $M(g \wedge g'')$ as follows: for every clock $x_i \in Y$, replace the constraints in row i, column 0, and row 0, column i of $M(g)$ by $(0, \leq)$, and remove all other bounds on x_i (i.e., replace all other constants in row i or column i by ∞ or $-\infty$); finally, close the result. From this DBM we extract a clock constraint g', and we have $post_r(\{(q,g)\}) = (q', g')$.

Notice that the operations on DBMs do not introduce any constant larger than the ceiling of the automaton under consideration, and so the result is indeed a zone as we defined it.

Condition (4). We have to show that if $C, D \in \mathcal{P}(Q \times \mathcal{Z})$, then it is decidable whether $C \cap D = \emptyset$. Clearly, it suffices to show that if g and g' are zones, then we can decide whether $g \wedge g'$ is empty. For this, notice that since clock constraints are closed under conjunction we can get a DBM for $g \wedge g'$. For a zone to be empty there must be a pair of clocks such that the upper bound on their difference is smaller than the lower bound. This corresponds to a negative cycle in the graph interpretation, of the DBM, which can be detected using well-known techniques.

Condition (5). The strong version of the condition states that given two symbolic configurations C_1, C_2, it is decidable whether $C_1 = C_2$ holds. While equality of symbolic configurations is in fact decidable, the weaker condition explained in Section 1.2.3 is much easier to prove (and leads to a more efficient search algorithm). Recall the condition: given the chain $C_0 \subseteq C_1 \subseteq C_2 \ldots$ of symbolic configurations, where $C_0 = I$ and $C_{i+1} = C_i \cup post(C_i)$, there is an algorithm satisfying the following properties: if two sets are not equal, then the algorithm answers "not equal"; if there is an index i such that $C_i = C_{i+j}$ for every $j \geq 0$, then there is also $k \geq i$ such that for input C_k, C_{k+1} the algorithm answers "equal". We provide such an algorithm. Let γ_i denote the symbolic representation of C_i as a set of pairs (q, M), where q is a state and M is a closed DBM. Since $post((q,M))$ is a set of zones, we have $\gamma_0 \subseteq \gamma_1 \subseteq \gamma_2 \ldots$. Since the set of indexed zones is finite, there is a γ_k such that $\gamma_k = \gamma_{k+1}$, i.e., γ_k and γ_{k+1} contain exactly the same pairs (q, M). So the algorithm just compares the symbolic representations, and answers 'equal' if they

are *syntactically* equal.

Condition (6). The chain $C_0 \subseteq C_1 \subseteq C_2 \ldots$ necessarily reaches a fixed point after finitely many steps, because the set $\mathcal{P}(Q \times \mathcal{Z})$ of symbolic configurations is finite. This is an instance of the finite case of Section 1.2.5.

1.3.4. *Zone Graph of Fischer's Mutex Algorithm*

We have shown in Condition (3) above that if C is an indexed zone, then for every rule r the set $post_r(C)$ is also an indexed zone. One can then define the *zone graph* of a timed automaton: the nodes of the graph are the indexed zones, and there is an edge from (q, g) to (q', g') if $\{(q', g')\} = post_r(\{(q, g)\})$ for some rule r. The reachable part of the zone graph of the timed automaton modelling Fischer's algorithm is shown in Figure 1.5. More precisely, the figure only shows one half of the graph, the other half is completely symmetric with the roles of the processes swapped. We use clock constraints to represent zones. All zones occurring in the zone graph are also given as a two-dimensional point set in Figure 1.4, and it is advisable to digest both figures in parallel.

From the zone graph one can easily see how the symbolic forward search procedure works. The set I is the indexed zone at the top of the figure. After k iterations the set C contains all the indexed zones that can be reached from i in k steps. Since the zone graph does not contain any node in which both processes are in the critical section, mutual exclusion holds.

We shall now *symbolically* follow Sample Run 2 presented at the end of Section 1.3.2. The initial symbolic configuration is a pair of state $\langle req_1, req_2, 1\rangle$ and zone $x_1 = 0 \wedge x_2 = 0$, where both processes are requesting access to the shared resource. The point set representation of this zone consists of a single point in the two-dimensional space and is depicted in Figure 1.4 (b).

- From this configuration, we can either let time elapse, as indicated by the dashed arrow, or we could let process 1 move to state $wait_1$, as indicated by the solid arrow. The latter transition does not change the zone part of the symbolic configuration, while the first one results in the zone shown in (c). Graphically, a time-delay action amounts to constructing the shadow of a light source at $(-\infty, -\infty)$. Note that the origin is excluded since at least an arbitrarily small amount of time must elapse. Point exclusion is depicted as a circle.
- While x_1 is still smaller than 1, process 1 may enter the $wait_1$ state, while process 2 remains unaffected. The effect to the zone part of the symbolic configuration, resets x_1 to zero. Note that we apply the reset to the *conjunction* of zone (c) and the guard $x_1 < 1$. So the resulting zone is (d), where x_2 is known to be strictly between 0 and 1.
- From here we let time elapse again and, according to the shadow intuition, end up in a zone that resembles (a) and (g), but is not shown in Figure 1.4.

It is obtained by extending (g) to the x_2 axis.
- Now process 2 moves to $wait_2$ thereby setting the value of v to 2. Together with the guard $x_2 < 1$, the previous zone has x_2 reset yielding zone (e).
- Faithful to our behavior so far, we let time elapse again, that is, apply the shadow operation to (e), which results in zone (a).
- As the final transition of this run we let process 2 move to the critical section. The guard $x_2 > 1$ ensures that the shadow only starts at values greater than 1.

1.3.5. *Conclusion and Further Reading*

We have introduced timed automata, and shown that by taking zones as symbolic configurations forward search becomes effective. A slight modification of the arguments shows that backward search is effective as well.

Much of the content of this section is taken from [1], where Bengtsson and Yi describe the efficient implementation of forward and backward search used in UPPAAL, the leading tool for the analysis of timed automata. A survey by Alur and Madhusudan on the decidability and complexity of decision problems for timed automata can be found in [8].

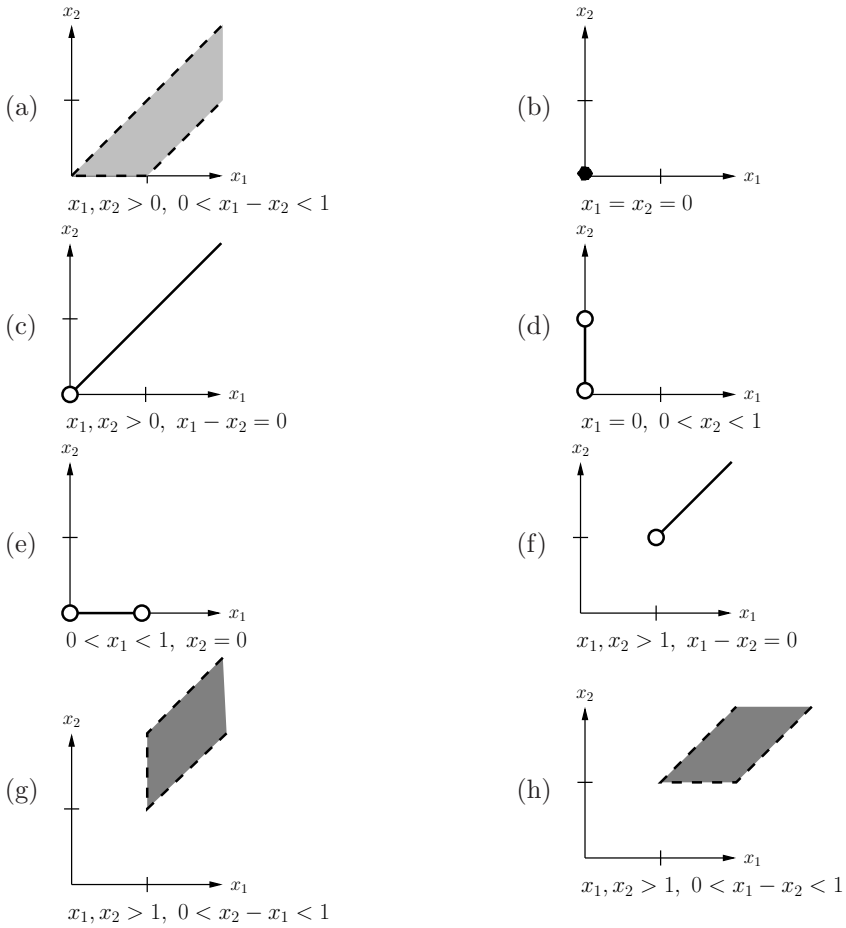

Fig. 1.4. Some zones of the zone graph shown as point sets.

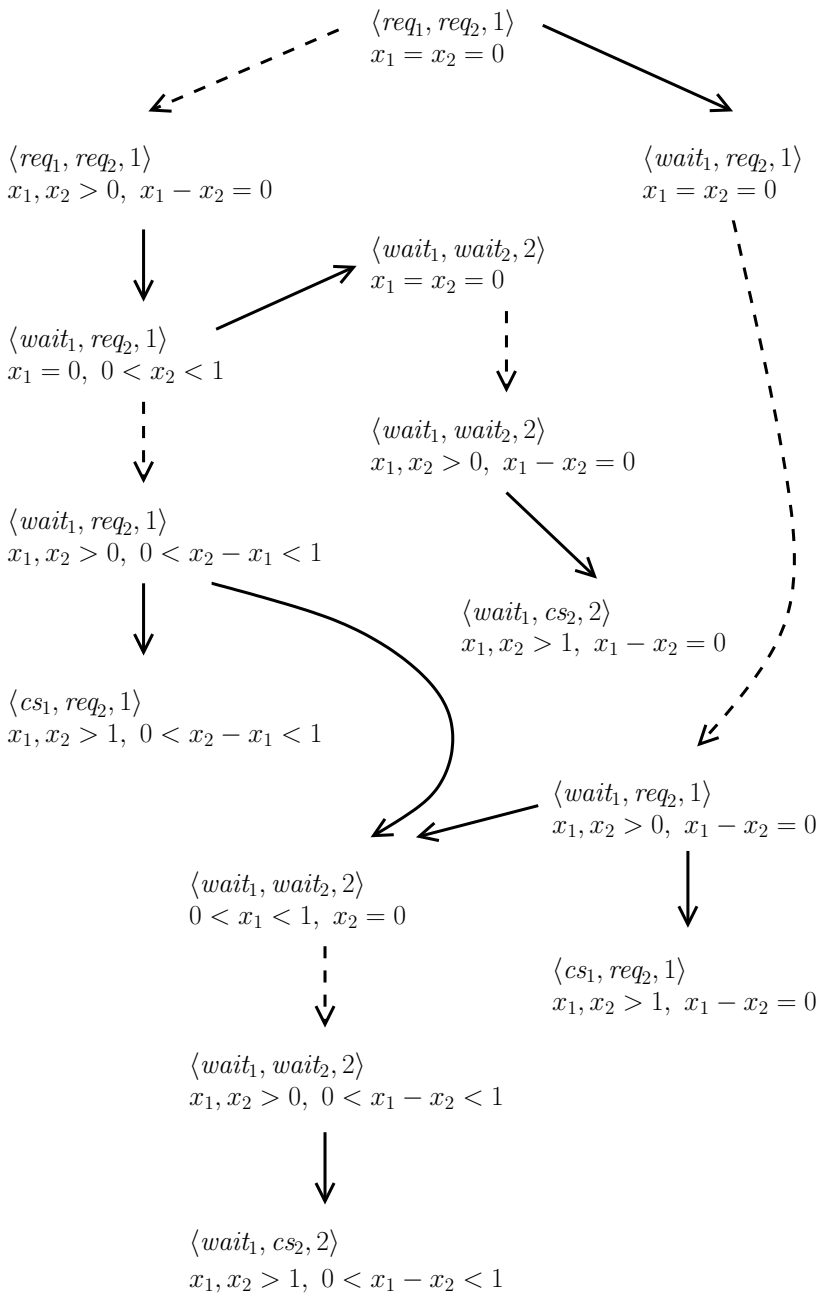

Fig. 1.5. One half of the zone graph of our case study.

1.4. Linear Automata

In this section we model a cache coherence protocol as a linear automaton, an extended automaton over non-negative integer variables whose guards and actions are linear expressions, and verify that it satisfies a property by means of a symbolic backward search. We prove that this search always terminates by showing the absence of infinite ascending chains in our family of symbolic configurations.

1.4.1. *The Case Study: A Cache Coherence Protocol*

Our case study is a simple cache coherence protocol for a multiprocessor system in which each processor has a local cache connected to main memory by a bus.

Recall that a cache memory provides a processor with a copy of a part of the current main memory for fast access during program execution. Local caches on multiprocessor machines improve performance, but introduce the *cache coherence problem*: multiple cached copies of the same block of memory must be kept consistent. The following scenario must be avoided: processor A writes a memory block for which processor B has a local copy in cache; before the cache is updated, processor B reads it; processor B believes it is getting the updated value of the block, while in fact it only gets the old value. We say that a memory block (in main memory or in a cache) is *valid* if it has been updated with the value of the last write access by any processor, and *invalid* otherwise. A cache management protocol is called *coherent* if a read access always provides a valid block.

Coherence is usually guaranteed by means of *write-invalidate* protocols: whenever a processor modifies a cache memory block (a *cache line*), the block address and a bus invalidation signal is sent to all other caches. Receiving this signal invalidates the corresponding cache line and an updated copy of the block must be fetched from main memory. (*Write* protocols send a copy of the new data to all caches sharing the old data). *Snoopy caches* continuously "listen" to the block addresses sent over the bus by other processors, and react when the addresses match their own cache lines (*bus snooping*).

We describe the MESI protocol, a write-invalidate, snoopy protocol [9]. For simplicity, we assume that each processor has one single cache-line, i.e., only one memory block is copied in the cache, and so we speak of a cache instead of a cache line. We model each processor as logically divided into a CPU that sends read and write requests to a *cache-management* unit (CMU), which is also connected to the bus.

In the MESI protocol a CMU is always in one out of four possible states: *modified* (M), *exclusive* (E), *shared* (S), and *invalid* (I), with the following *intended* meaning: in I, the cache is invalid; in S, the cache is valid, and there maybe other valid caches; in E, the last write access was performed by the CMU's own CPU, and so it is valid; moreover, the value has already been transferred to main memory, i.e., the value in main memory is also valid, but no other cache is valid; finally, in state M the last

n most recent writes, with $n > 1$, have been performed by the cache's own CPU; however, the last write has not been transferred to main memory yet; so the cache is valid, but main memory is not, and no other cache is valid.

The CMU can receive read or write requests. A read/write request to a valid cache is a *read/write hit*, otherwise a *read/write miss*. The reactions of the CMU are as follows:

- **Read Hit**. Only possible from states other than *invalid*. The CMU returns the local value of the cache, no coherence action is required.
- **Read Miss**. Only possible in state *invalid*. The CMU goes from *invalid* to *shared*, all caches in states *exclusive* or *modified* go to *shared*.
- **Write Hit**. Only possible in states other than invalid. If the CMU is in state *shared*, it goes to *exclusive* after invalidating the contents of all other caches; if it is in *exclusive* or *modified* it goes to *modified*, no bus transition is required.
- **Write Miss**. Only possible in state *invalid*. The CMU goes from *invalid* to *exclusive* after invalidating all other caches.

The actual property we are interested in is cache coherence. However, proving cache coherence for a protocol is too complex to achieve within the limits of this chapter. So we shall establish and formally prove an important necessary condition for cache coherence: A cache in state M (*modified*) should be the only valid cache.

1.4.2. *Linear Automata*

Linear automata are extended automata over non-negative integer variables, with guards and actions given by linear expressions. The formal definition requires some preliminaries.

Let $\mathbf{x} = (x_1, \ldots, x_n)$ be a tuple of *non-negative* integer variables. A valuation of $\mathbf{x}$ is therefore an element of $\mathbb{N}^n$. A *linear constraint* over $\mathbf{x}$ is inductively defined as follows:

- a linear inequation over $x_1, \ldots, x_n$ with rational coefficients is a linear constraint;
- a boolean combination of linear constraints is a linear constraint.

A *solution* of a linear constraint is a tuple $\mathbf{k} = (k_1, \ldots, k_n) \in \mathbb{N}^n$ such that simultaneously substituting $k_1, \ldots, k_n$ for $x_1, \ldots, x_n$ yields a true expression. The set of solutions of a linear constraint ϕ is denoted by $[\![\phi]\!]$. An example of a linear constraint over the variables x, y, z is $\phi = (x \geq y + 1 \wedge y = 3z)$, and we have $[\![\phi]\!] = \{(3k + 1 + l, 3k, k) \mid k, l \in \mathbb{N}\}$. We often identify a constraint and its set of solutions.

A *linear transformation* is a system of linear equations over two tuples $\mathbf{x} =$

$(x_1, \ldots, x_n)$ and $\mathbf{x}' = (x'_1, \ldots, x'_n)$ of variables of the form

$$\begin{aligned} x'_1 &= f_1(x_1, \ldots, x_n) \\ &\ldots \\ x'_n &= f_n(x_1, \ldots, x_n) \end{aligned}$$

We sometimes denote a linear transformation by $\mathbf{x}' = \tau(\mathbf{x}$, and write $\tau\tau(\mathbf{x} = (f_1(\mathbf{x}, \ldots, f_n(\mathbf{x}))$ As for linear constraints, a solution of a linear transformation is a pair $((k_1, \ldots, k_n), (k'_1, \ldots, k'_n)) \in \mathbb{N}^n \times \mathbb{N}^n$ satisfying all equations.

We are now ready to formally define linear automata. A *linear automaton* is an extended automaton over a tuple $\mathbf{x}$ of non-negative integer variables that satisfies the following two conditions: the guards of the rules are linear constraints over $\mathbf{x}$, and the actions of the rules are linear transformations over $\mathbf{x}$ and $\mathbf{x}'$.

Let us now see how the MESI protocol can be formalized as a linear automaton. The automaton has only one state q, and acts on a four-tuple (m, e, s, i) of variables. A configuration $\langle q, (k_m, k_e, k_s, k_i)\rangle$ indicates that the number of CMUs in states M, E, S, and I, is k_m, k_e, k_s and k_i, respectively. Notice that $k_m + k_e + k_s + k_i$ is the total number of CMUs, and does not change during the execution of the protocol. Guards are linear constraints over the tuple (m, e, s, i) of variables, and actions are linear transformations over the tuples $(m, e, s, i), (m', e', s', i')$. The dynamics of the protocol is modelled by the rules of Table 1.1, which correspond to the intended behavior of the protocol after a Read Hit, Read Miss, Write Hit, and Write Miss. For instance, a Write Hit can take place if some cache is in state S, i.e., if $s \geq 1$, and it changes the state of this cache to E, and the state of all other caches to I, i.e., $e' = 1$ and $i' = m + e + s + i - 1$. Notice that, since the automaton only has one state, it is not necessary to indicate the source and the target state of a rule.

Table 1.1. Rules of the MESI protocol.

Event	Guard	Action	Rule Name
Read Hit	$m + e + s \geq 1$	$m' = m \quad e' = e$ $s' = s \quad i' = i$	r_1
Read Miss	$i \geq 1$	$m' = 0 \quad e' = 0$ $s' = m + e + s + 1 \quad i' = i - 1$	r_2
Write Hit	$m \geq 1$	$m' = m \quad e' = e$ $s' = s \quad i' = i$	r_3
	$e \geq 1$	$m' = m + 1 \quad e' = e - 1$ $s' = s \quad i' = i$	r_4
	$s \geq 1$	$m' = 0 \quad e' = 1$ $s' = 0 \quad i' = m + e + s + i - 1$	r_5
Write Miss	$i \geq 1$	$m' = 0 \quad e' = 1$ $s' = 0 \quad i' = m + e + s + i - 1$	r_6

The initial configurations of the protocol are those satisfying the constraint $(m = 0 \wedge e = 0 \wedge s = 0)$, i.e., initially all CMUs are in the state *invalid*.

An n-configuration is a configuration whose valuation satisfies the constraint $m + e + s + i = n$. For instance, $\langle q, (2, 0, 2, 0)\rangle$ is a 4-configuration. Since the

number of CMUs does not change during the execution of the protocol a transition leaving a n-configuration always leads to another n-configuration. Therefore, for each fixed value of n the set of reachable states of the protocol is finite and can be exhaustively explored. However, this only allows to verify properties of the protocol for one particular instance of the protocol, the one with n participating CMUs. The challenge is to automatically verify that a property holds *for any number of CMUs*.

We can formally state our example property – a cache in state *modified* should be the only valid cache – in terms of linear constraints as follows: The MESI protocol should never reach any configuration satisfying the linear constraint $(m \geq 1 \wedge e+s \geq 1) \vee m \geq 2$. We automatically check that this is the case.

1.4.3. *Lc-sets and Backward Search*

In this section we introduce a family $\mathcal{F}$ such that backward search satisfying conditions (1) to (5), but not condition (6). This means that, while we can apply this version of backward search to linear automata, there are instances for which the algorithm does not terminate. In Section 1.4.4 we will first apply the algorithm to the MESI protocol, and observe that in this case it does terminate. In Section 1.4.5 we will show that this is not a coincidence: we prove that the algorithm always terminates for the subclass of *monotonic linear automata*, of which the MESI protocol is an instance, when the set D of dangerous configurations is *upward closed*, which will also be the case.

All linear automata modeling cache-coherence protocols have only one control state. In what follows, for the sake of simplicity, we only consider this special case, but the reader will have no difficulty in extending the results to the case of several states.

When the automaton has one state, a configuration is completely determined by its valuation. So we write ν instead of $\langle q, \nu \rangle$ and speak of "the configuration ν". Also, instead of $\langle q, \nu \rangle \Longrightarrow \langle q, \nu \rangle$ we just write $\nu \Longrightarrow \nu'$.

A set of configurations C is *linearly constrained*, or an *lc-set* for short, if there is a linear constraint ϕ such that $C = [\![\phi]\!]$. We choose the lc-sets as symbolic configurations, i.e., $\mathcal{F}$ contains all lc-sets. In the rest of this section we prove the following result:

Theorem 1.2. *Let $\mathcal{A}$ be a linear automaton, and let I and D be lc-sets. Symbolic backward search with lc-sets satisfies conditions (1)-(5), but not condition (6).*

Condition (1). An lc-set C is finitely represented by the linear constraint ϕ such that $C = [\![\phi]\!]$.

Condition (2). The set D is an lc-set by hypothesis. Notice that in the case of the MESI protocol we have $D = (m \geq 1 \wedge e + s \geq 1) \vee m \geq 2$, and so D is an lc-set.

Condition (3). We have to show that if C is an lc-set, then $C \cup \mathit{pre}(C)$ is also an lc-set. Since linear constraints are closed under disjunction, the union of two

lc-sets is also an lc-set. So it suffices to show that $pre_r(C)$ is an lc-set for every rule r. We need a bit of notation.

Let ϕ be a linear constraint over a tuple $\mathbf{x} = (x_1, \ldots, x_n)$ of variables, and let $\mathbf{x}' = \tau(\mathbf{x})$ be a linear transformation . We write $\phi[\mathbf{x}/\tau(\mathbf{x})]$ to denote the constraint obtained by syntactically replacing every occurrence of x_i in ϕ by the expression $f_i(x_1, \ldots, x_n)$ of τ (i.e., the right-hand-side of the equation for x_i' in τ).

For example, let $\phi_0 = (e \leq m + 1 \wedge s = 3m + 2i)$, and let $\tau(\mathbf{x})$ be the transformation of rule r_4 of Table 1.1. Then

$$\begin{aligned}\phi_0[\mathbf{x}/\tau(\mathbf{x})] &= (e \leq m + 1 \wedge s = 3m + 2i)[e/e - 1, m/m + 1] \\ &= (e - 1 \leq m + 1 + 1 \wedge s = 3(m + 1) + 2i)\end{aligned}$$

which can be simplified to $(e \leq m + 3 \wedge s = 3m + 2i + 3)$.

We can now prove the following lemma, which shows how to compute a linear constraint for $pre_r(C)$ given a linear constraint for C.

Lemma 1.1. *Let ϕ and τ be the linear constraint and the linear transformation corresponding to the guard and the action of a rule r. Let ψ be a constraint representing an lc-set C (i.e., $C = [\![\psi]\!]$). We have:*

$$pre_r(C) = [\![\phi \wedge \psi[\mathbf{x}/\tau(\mathbf{x})]]\!] \ .$$

Proof. Observe first that $[\![\psi[\mathbf{x}/\tau(\mathbf{x}[\![$ is the weakest precondition of $[\![\psi]\!]$ under the simultaneous assignment $\mathbf{x} := \tau(\mathbf{x})$. This follows immediately from the well-known weakest-precondition rule for simultaneous assignment. So $[\![\psi[\mathbf{x}/\tau(\mathbf{x})]]\!]$ contains all states that are transformed into states of C by τ. However, for such a state to belong to $pre_r(C)$ it must also satisfy the guard ϕ, and the lemma follows. □

For example, consider rule r_4 of the MESI-protocol of Table 1.1, and let $C = [\![\phi_0]\!]$ with ϕ_0 as above. We have

$$\begin{aligned}pre_{r_4}(C) &= pre_{r_3}(e \leq m + 1 \wedge s = 3m + 1/2i) \\ &= e \geq 1 \ \wedge \ e \leq m + 3 \wedge s = 3m + 1/2i + 3\end{aligned}$$

With the help of this lemma we can easily establish **Condition (3)**: if C is an lc-set of configurations, then so is $C \cup pre(C)$. Since linear constraints are closed under disjunction, it suffices to show that $pre_r(C)$ is an lc-set for every rule r. By Lemma 1.1, it suffices to show that $\phi \wedge \phi[\mathbf{x}/\tau(\mathbf{x})]$ is a linear constraint, where ϕ and τ are the guard and the transformation of the rule r, respectively. Since ϕ is linear, we only have to show that $\phi[\mathbf{x}/\tau(\mathbf{x})]$ is linear. This follows from the fact that τ is a linear transformation. Let $x_i' = f_i(\mathbf{x})$ the the i-th inequation of τ. For every $1 \leq i \leq n$, we have to replace every occurrence of x_i in ϕ by $f_i(\mathbf{x})$. Since $f_i(\mathbf{x})$ is linear, the result of the substitution is again a linear constraint.

Condition (4) For every lc-set C the emptiness of $C \cap I$ is decidable. Since we assume that both C and I are lc-sets represented by linear constraints, say ϕ_C and ϕ_I, we have $C \cap I = [\![\phi_C \wedge \phi_I]\!]$, and so it suffices to prove the decidability of the

satisfiability problem for linear constraints. This follows easily from the following observations:

- Every linear constraint is equivalent (has the same solutions) to a linear constraint without negations.
  To prove this, observe first that negations can be pushed inwards through conjunctions and disjunctions. Then, observe that the negation of a linear (in)equation is equivalent to a disjunction of linear (in)equations. For instance $\neg(x \leq y)$ is equivalent to $(x > y')$, and $\neg(x = y)$ is equivalent to $(x < y) \vee (y < x)$.
- Every constraint is equivalent to a constraint in disjunctive normal form without negations.
- Every disjunct of a constraint in disjunctive normal form without negations is a system of Diophantine equations and inequations.

So the satisfiability problem reduces to deciding if a system of linear Diophantine equations and inequations has a solution. In turn, this problem can be reduced to Integer Linear Programming, which is known to be solvable in nondeterministic polynomial time (see for instance [10]). Specific algorithms for Diophantine equations and inequations also exist, see for instance [11]. The algorithm corresponding to this sketch of a decidability proof is not efficient, but the issue of efficiency is beyond the scope of this paper, and we refer the reader to the literature.

Condition (5) The equality of lc-sets is decidable. Given two constraints ϕ_1, ϕ_2, we have $[\![\phi_1]\!] = [\![\phi_2]\!]$ if and only if $[\![(\phi_1 \wedge \neg\phi_2) \vee (\neg\phi_2 \wedge \phi_1)]\!] = \emptyset$, which we have shown to be decidable when proving that Condition (4) holds.

It remains to show that **Condition (6)** fails. Consider the linear automaton acting on two variables x, y, and having one single rule with $x \leq 1$ as guard and $x' = x - 1, y' = y + 1$ as action. Let $D = (x = 0)$. Then $pre^*(D) = \mathbf{true}$, but $pre^i(D) = (x \leq i)$, and so backward search does not terminate.

1.4.4. *Backward Search for the MESI Protocol*

We apply backward search with lc-sets to the MESI protocol. We start with:

$$D = (m \geq 1 \wedge e + s \geq 1) \vee m \geq 2\,.$$

We compute $pre(D)$ using the procedure sketched in the proof of **Condition (3)**. Notice that for our reachability problem it is not necessary to consider rules r_1 and r_3 of Table 1.1, because their actions are the identity transformation.

$$
\begin{array}{ll}
 & \mathit{pre}(D) \\
= & \mathit{pre}_{r_2}(D) \cup \mathit{pre}_{r_4}(D) \cup \mathit{pre}_{r_5}(D) \cup \mathit{pre}_{r_6}(D) \\
= & \quad (i \geq 1 \wedge D[m/0, e/0, s/m+e+s+1, i/i-1]) \\
 & \vee\ (e \geq 1 \wedge D[m/m+1, e/e-1]) \\
 & \vee\ (s \geq 1 \wedge D[m/0, e/1, s/0, i/m+e+s+i-1]) \\
 & \vee\ (i \geq 1 \wedge D[m/0, e/1, s/0, i/m+e+s+i-1]) \\
= & \quad i \geq 1 \wedge ((0 \geq 1) \wedge 0+m+e+s+1 \geq 1) \vee 0 \geq 2) \\
 & \vee\ e \geq 1 \wedge ((m+1 \geq 1 \wedge e-1+s \geq 1) \vee m+1 \geq 2) \\
 & \vee\ s \geq 1 \wedge ((0 \geq 1) \wedge 1+0 \geq 1) \vee 0 \geq 2) \\
 & \vee\ i \geq 1 \wedge ((0 \geq 1) \wedge 1+0 \geq 1) \vee 0 \geq 2) \\
= & \quad e \geq 1 \wedge (e+s \geq 2 \vee m \geq 1) \\
= & \quad (e \geq 1 \wedge e+s \geq 2) \vee (e \geq 1 \wedge m \geq 1)
\end{array}
$$

(Note that, since we are reasoning about non-negative integers only, it is always possible to drop constraints of the form $x \geq 0$.)

We can simplify $D \cup \mathit{pre}(D)$ by getting rid of $e \geq 1 \wedge m \geq 1$, which is implied by the first disjunct of D, to obtain

$$D \cup \mathit{pre}(D) = (m \geq 1 \wedge e+s \geq 1) \vee m \geq 2 \vee (e \geq 1 \wedge e+s \geq 2)\,.$$

It is easy to see that $D \subset D \cup \mathit{pre}(D)$ holds – for instance because the configuration $(0,1,1,0) \in \mathit{pre}(D) \setminus D$. So we proceed with the search by computing $\mathit{pre}^2(D)$, this time without so much detail as in the previous case:

$$
\begin{array}{l}
\quad \mathit{pre}^2(D) \\
= \mathit{pre}_{r_4}(\mathit{pre}(D)) \\
= e \geq 1 \wedge (e-1 \geq 1 \wedge (m+1 \geq 1 \vee e-1+s \geq 2)) \\
= e \geq 2 \wedge (m \geq 0 \vee e+s \geq 3) \\
= e \geq 2
\end{array}
$$

Since $[\![e \geq 2]\!] \subset [\![e \geq 1 \wedge e+s \geq 2]\!]$, we obtain $\mathit{pre}^2(D) \subset \mathit{pre}(D)$, which implies $D \cup \mathit{pre}(D) = D \cup \mathit{pre}(D) \cup \mathit{pre}^2(D)$. So backward search has reached a fixed point, and we can conclude

$$\mathit{pre}^*(D) = D \cup \mathit{pre}(D) = (m \geq 1 \wedge e+s \geq 1) \vee m \geq 2 \vee (e \geq 1 \wedge e+s \geq 2)\,.$$

Recall that the set I of initial configurations of the MESI protocol is given by $I = (m = 0 \wedge e = 0 \wedge s = 0)$. So $I \cap \mathit{pre}^*(D) = \emptyset$, and so the MESI protocol satisfies that a cache in state *modified* is the only valid cache.

1.4.5. *Monotonic Linear Automata and Upward-Closed Sets*

Termination of the algorithm for the MESI protocol is not just luck. We show that backward search always terminates for *monotonic linear automata*, of which the MESI protocol is an instance, when the set of dangerous configurations is *upward closed*, which is also the case.

A linear automaton is *monotonic* if the following condition holds: for all configurations $\nu_1, \nu_1', \nu_2 \in \mathbb{N}^n$, if $\nu_2 \geq \nu_1$ and $\nu_1 \Longrightarrow \nu_1'$ (i.e., ν_1' is reachable from ν_1 in one step), then there exists $\nu_2' \geq \nu_1'$ such that $\nu_2 \Longrightarrow \nu_2'$.

The automaton of the MESI protocol is monotonic: Assume $\nu_2 \geq \nu_1$ and $\nu_1 \Longrightarrow \nu_1'$. Then there exists a rule r of the automaton with a guard ϕ and a linear transformation τ such that $\nu_1 \in [\![\phi]\!]$ and $\nu_1' = \tau(\nu_1)$. Since $\nu_2 \geq \nu_1$ and ϕ is of the form $q \geq 1$ for some $q \in \{m, e, s, i\}$ (see Table 1.1), we have $\nu_2 \in [\![\phi]\!]$. So $\nu_2 \Longrightarrow \nu_2'$ for $\nu_2' = \tau(\nu_2)$. Since $\nu_2 = \nu_1 + \delta$ for some non-negative vector δ and τ is linear, we have $\nu_2' = \tau(\nu_1 + \delta) = \tau(\nu_1) + \tau(\delta)$. For the linear transformations of Table 1.1 the vector $\tau(\delta)$ is also non-negative, and so $\nu_2' \geq \nu_1'$.

We study symbolic backward on monotonic linear automata with a new class of symbolic configurations, the upward-closed sets.

Given $\nu, \nu' \in \mathbb{N}^n$, we say $\nu \leq \nu'$ if $\nu(i) \leq \nu'(i)$, for every $i \in [1..n]$, where $\nu(i)$ and $\nu'(i)$ denote the i-th component of ν and ν', respectively; we say $\nu \lneq \nu'$ if $\nu \leq \nu'$ and $\nu(i) < \nu'(i)$ for some $i \in [1..n]$. A set of configurations C of a linear automaton is *upward-closed* if $\nu \in C$ and $\nu' \geq \nu$ implies $\nu' \in C$. A linear constraint ϕ is *upward closed* if $[\![\phi]\!]$ is upward closed.

Observe that the linear constraint $\phi = (m \geq 1 \wedge s + e \geq 1) \vee m \geq 2$ is upward closed, and so the set $D = [\![\phi]\!]$ of dangerous configurations of the MESI protocol is upward closed.

At first glance, the relation between the upward-closed and the lc-sets is not clear. We show that upward-closed sets are a special class of lc-sets. For this we need the following well-known result, a variant of Dickson's lemma. For the sake of completeness, we sketch a proof.

Lemma 1.2. *Any set $C \subseteq \mathbb{N}^n$ has finitely many minimal elements with respect to the partial order $\leq$.*

Proof. Assume there exists $C \subseteq \mathbb{N}^n$ such that the set $M \subseteq C$ of minimal elements of C is infinite. We prove by induction on n that M contains two elements ν, ν' such that $\nu \lneq \nu'$, contradicting the assumption that ν' is a minimal element of C. In fact, we prove a stronger statement: M contains an infinite chain $\nu_1 \lneq \nu_2 \lneq \nu_3 \ldots$. For the base case $n = 1$, let ν_1 be a minimal element of M, ν_2 a minimal element of $M \setminus \{\nu_1\}$, ν_3 a minimal element of $M \setminus \{\nu_1, \nu_2\}$ etc. Since for $n = 1$ the order $\leq$ is total, we have $\nu_1 \lneq \nu_2 \lneq \nu_3 \ldots$. For the induction step, assume $n \geq 1$. Given a configuration $\nu \in \mathbb{N}^n$, let $\nu' \in \mathbb{N}^{(n-1)}$ denote the projection of ν onto the first $(n-1)$ components, and let $l \in \mathbb{N}$ denote the value of ν's last component, i.e., ν is obtained by adding l to ν' as last component. By induction hypothesis, M contains configurations $\nu_1, \nu_2, \nu_3, \ldots$ whose projections satisfy $\nu_1' \lneq \nu_2' \lneq \nu_3' \ldots$. Choose an index i_1 such that $l_{i_1} \leq l_j$ for every $j \in \mathbb{N}$; then an index i_2 such that $l_{i_2} \leq l_j$ for every $j \in \mathbb{N} \setminus \{1, 2, \ldots, i_1\}$; then an index i_3 such that $l_{i_3} \leq l_j$ for every $j \in \mathbb{N} \setminus \{1, 2, \ldots, i_2\}$etc. This produces an infinite sequence of indices $i_1 < i_2 < i_3 \ldots$ such that $l_{i_1} \leq l_{i_2} \leq l_{i_3} \ldots$. Since $\nu_{i_1}' \lneq \nu_{i_2}' \lneq \nu_{i_3}' \ldots$ also holds, we

get $\nu_{i_1} \lneq \nu_{i_2} \lneq \nu_{i_3} \ldots$. □

Now, let C be an arbitrary upward-closed set. By Lemma 1.2, C has finitely many minimal elements $\nu_1, \ldots, \nu_k$. Let $\nu_i = (k_i^1, \ldots, k_i^n)$; then $C = [\![\phi_1 \vee \ldots \vee \phi_k]\!]$, where $\phi_i = x_1 \geq k_i^1 \wedge \ldots \wedge x_n \geq k_i^n$. So C is an lc-set.

In the rest of this section we sketch the proof of the following result:

Theorem 1.3. *Backward search satisfies Conditions (1)-(6) for each monotonic linear automaton $\mathcal{A}$, upward closed set D, and lc-set I.*

Condition (1). Since upward-closed sets are linearly constrained, they have a finite symbolic representation as the set of solutions of a constraint. Moreover, since an upward-closed set is completely determined by its set of minimal elements, and by Lemma 1.2 this set is finite, an upward-closed set can also be finitely represented by its set of minimal elements.

Condition (2) holds by hypothesis. In the case of the MESI protocol we have $D = (m \geq 1 \wedge e + s \geq 1) \vee m \geq 2$, which is equal to the upward-closed set with $\{(1,1,0,0),(1,0,1,0),(2,0,0,0)\}$ as set of minimal elements.

For **Condition (3)** we have to show that, if C is an upward-closed set of configurations of a monotonic linear automaton, then $pre(C)$ is also upward closed. This amounts to proving that $\nu_1 \in pre(C)$ and $\nu_2 \geq \nu_1$ imply $\nu_2 \in pre(C)$. Since $\nu_1 \in pre(C)$, there exists $\nu_1' \in C$ such that $\nu_1 \Longrightarrow \nu_1'$ and $\nu_1' \in C$. Since the automaton is monotonic and $\nu_2 \geq \nu_1$, there exists $\nu_2' \geq \nu_1'$ such that $\nu_2 \Longrightarrow \nu_2'$. Since C is upward-closed, $\nu_1' \in C$, and $\nu_2' \geq \nu_1'$, we have $\nu_2' \in C$, and so $\nu_2 \in pre(C)$. Note that the union of two upwards closed sets is upwards closed, and so $C \cup pre(C)$ is upwards closed if C is.

We have not yet shown that a symbolic representation of $pre(C)$ can be effectively computed from the symbolic representation of C, but this holds for the representation as linear constraint because of Lemma 1.1. It is not difficult to see that it also holds for the representation by the set of minimal elements. We leave the design of an algorithm that, given the minimal elements of C as input, yields the minimal elements of $pre(C)$ as output, as an exercise for the reader.

Conditions (4) and (5) follow immediately from Lemma 1.2 and the fact that the same conditions hold for lc-sets.

Finally, we prove that **Condition (6)** also holds, namely that every infinite chain $U_1 \subseteq U_2 \subseteq U_3 \ldots$ of upward-closed sets contains an element U_k such that $U_k = \bigcup_{i \geq 1} U_i$. By Lemma 1.2, the set M of minimal elements of $\bigcup_{i \geq 1} U_i$ is finite, and so there exists an index k such that $M \subseteq U_k$. Let M_k denote the set of minimal elements of U_k. Since U_k is upward-closed and $M \subseteq U_k$, we necessarily have $M = M_k$. It follows $U_k = \bigcup_{i \geq i} U_i$, which concludes the proof of Theorem 1.3.

Therefore, the fact that our backward search computation terminated for the MESI protocol was not a stroke of luck: for monotonic protocols, and if the set D is upward-closed, the computation is guaranteed to terminate.

1.4.6. *Conclusion and Further Reading*

Delzanno has used backward search to automatically prove properties of other cache-coherence protocols [2]. Some of them can be modelled by monotonic automata, others cannot. Remarkably, backward search terminates in all cases

The termination of backward search has been shown by proving that the domain of symbolic configurations satisfies the ascending chain condition. In turn, this result is based on the existence of an order $\leq$ on the set $\mathcal{V}$ of valuations ($\mathbb{N}^n$ in our case) satisfying the following two fundamental properties:

- the extended automaton is monotonic w.r.t. $\leq$, and
- every subset of $\mathcal{V}$ has finitely many minimal elements w.r.t. $\leq$; an order satisfying this property is called a *well-order*.

These properties turn out to hold not only for monotonic linear automata, but for other classes of extended automata working on other data structures, like lossy channel systems, a class of automata working on tuples of words, or timed Petri nets. Historically, the approach was first applied by Abdulla and Jonsson to lossy channel systems [12]. In a series of papers, with different co-authors, Abdulla and Jonsson have vastly generalized and extended the original idea (see for instance [13]); Finkel and Schnoebelen have also contributed very substantially [14].

1.5. Pushdown Automata

In this section we model a small recursive sequential program as a pushdown automaton. Pushdown automata were already shown to be an instance of extended automata in Example 1.2. We prove a simple property of the program by means of an *accelerated* forward symbolic search.

1.5.1. *The Case Study: Skylines*

Consider the C-program of Figure 1.6, taken from [3]. The question marks in the conditionals mean nondeterministic choice, which is not a primitive in C but could be easily simulated. The program may be understood as a controller for a plotter or for printing on the screen. The procedures `up`, `right`, and `down` are supposed to draw a line of unit length, starting at the current position of the pen or the cursor, in the direction indicated by the name of the procedure; the code for this is omitted, for our purposes we can just assume that they return immediately.

The program is supposed to draw *skylines* like the one shown on the left of Figure 1.7. In the course of this section we will prove that the program never draws *degenerate skylines* like the one on the right of Figure 1.7. A skyline is degenerate if at some point the program draws a line going up, and immediately after a line going down. In a non-degenerate skyline, the program always moves right at least once between any pair of up and down moves.

```
   main() {                    void m() {
0:    s();                  0:    if (?) {
1: }                        1:       s();
                            2:       right();
                            3:       if (?)
   void s() {               4:          m();
0:    if (?) {                       } else {
1:       up();              5:       up();
2:       m();               6:       m();
3:       down();            7:       down();
      }                             }
4: }                        8: }
```

Fig. 1.6. A C program drawing skylines. Procedures for actual drawing – `up`, `right`, and `down` – are left unspecified.

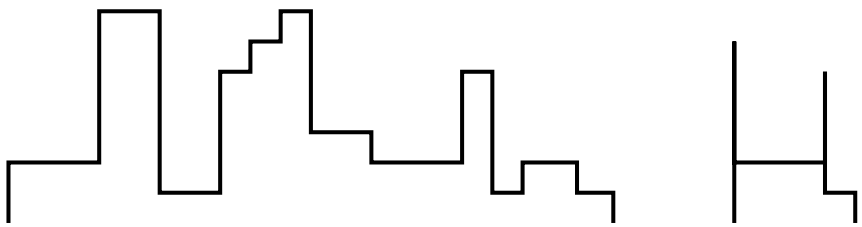

Fig. 1.7. A skyline and a degenerate skyline

1.5.2. *Pushdown Automata*

We model sequential programs with possibly recursive procedures as pushdown automata. Recall that, as explained in Example 1.2, we define a pushdown automaton as an extended automaton over a single variable, the *stack*, whose values are words over an alphabet Γ of *stack symbols*. The words are also called *stack contents*. A configuration of a PDA is therefore a pair $\langle q, v \rangle$, where q is a state and $v \in \Gamma^*$. The guards of the rules check if the topmost symbol of the current stack content is equal to a fixed symbol; the actions replace the topmost symbol by a fixed word. So a rule is determined by its source and target states, the fixed symbol that the guard compares with the topmost stack symbol, and the word that the action replaces the top symbol with.

We formally define a PDA using a notation that slightly differs from the one we have used for extended automata. The reason is that we wish the notation to resemble the classical one for pushdown automata. A PDA is a triple (Q, Γ, Δ), where Q is the set of states, often called *control states*, Γ is the stack alphabet,

and $\Delta \subseteq (Q \times \Gamma) \times (Q \times \Gamma^*)$ is the set of rules. We write $\langle q, \gamma \rangle \longrightarrow \langle q', v \rangle$, if $(q, \gamma, q', v) \in \Delta$. Without loss of generality we assume $|v| \leq 2$ for all $\langle q, \gamma \rangle \longrightarrow \langle q', v \rangle$ in Δ.

A configuration of a PDA is a pair of a control state and the current stack content (the current value of the stack variable in terms of extended automata). The PDA can move from configuration $\langle q, \gamma w \rangle$ to configuration $\langle q', vw \rangle$ by means of a rule $\langle q, \gamma \rangle \longrightarrow \langle q', v \rangle$ that replaces the topmost stack symbol γ with v. Formally, $\langle q, \gamma w \rangle \Longrightarrow \langle q', vw \rangle$ if there exists a rule $\langle q, \gamma \rangle \longrightarrow \langle q', v \rangle$ such that $w = \gamma v'$ and $w' = vv'$.

Sequential Programs as PDA We shall now discuss how a sequential, procedural program may be encoded as a PDA. Such programs are mainly determined by

- control flow: assignments, conditionals, loops, and procedure calls, possibly with parameters and return values;
- local variables of each procedure; and
- global variables.

Consequently, any given runtime state encountered during a program execution is determined by

- the program pointer indicating, where the flow of control has currently arrived;
- the values of global variables; and
- the values of local variables of the each procedure that has not yet returned.

Note that there is a tight correspondence between the *stack* of activation records and the *stack* of a PDA. Indeed, we shall interpret a configuration $\langle q, \gamma v \rangle$ such that γ encodes the current activation record and v encodes the pending records. Following these lines, γ represents the program pointer within the active procedure as well as the content of the local variables of this procedure. Pending procedures, pending return address and pending local variable contents are saved in v. Control state q has a more global flavor and will be used to hold global variables. Since we are dealing with finite sets of stack symbols, we must restrict ourselves to values of finite domain data types when saving variable contents.

The correspondence between program statements and PDA rules is as follows:

- A simple statement, like an assignment, only influences the topmost activation record and, potentially, the value of global variables. So it is encoded by rules of the form $\langle q, \gamma \rangle \longrightarrow \langle q', \gamma' \rangle$.
- Procedure calls push a new record on top of the call stack. The global variables and the current activation record do not change. So procedure calls can be modeled by rules of the form $\langle q, \gamma \rangle \longrightarrow \langle q, \gamma' \gamma \rangle$.

- When a procedure returns, it simply pops the topmost record from the stack. This can be modeled by a rule of the form $\langle q, \gamma\rangle \longrightarrow \langle q, \epsilon\rangle$.

Skylines – Formally. We model the skyline program as a PDA. Since the program does not have any global variables, the PDA will only needs a single control state, which we call q_s. Also since there are not any local variables either, the stack symbols correspond to the possible program points. We denote the program points belonging to procedure p by $p_0, p_1, \ldots$, with p_0 as the initial point. Note that the actual drawing procedures, which are not explicitly modeled, only have a single program point each: up_0, $down_0$, and $right_0$, respectively. So we can model the skyline program by a PDA

$$A_s = (\{q_s\},\ \Gamma_s,\ \Delta_s)$$

over the set Γ_s of stack symbols given by

$$\Gamma_s = \{main_0, main_1, s_0, \ldots, s_5, m_0, \ldots, m_8, up_0, down_0, right_0\}$$

Notice how there is exactly one stack symbol for each control point of the skyline program in Figure 1.6. The set Δ_s of rules is shown in Figure 1.8. For instance, the rule $\langle q_s, m_2\rangle \longrightarrow \langle q_s, right_0 m_3\rangle$ models the call to procedure `right()` at program point `2` of procedure `main()`. The rules marked with `if (true)` and `if (false)` model the conditionals. The marks `(*)` and `(**)` are used later.

Non-degenerate skylines. Recall that we are interested in proving the absence of a call to `up` immediately followed by a call to `down`. In terms of our encoding we like to prove the absence of transitions like

$$\langle q_s, v\rangle \Longrightarrow \langle q_s, up_0 v'\rangle \Longrightarrow \langle q_s, v'\rangle \Longrightarrow \langle q_s, down_0 v''\rangle$$

Our property is thus a property of a *sequence* of configurations instead of a property of a *single* configuration. For this reason symbolic reachability of configurations cannot directly prove this property. However, there is a generally useful trick to circumvent this problem: monitors. A monitor is a finite automaton running in parallel with the pushdown system. Our monitor has two states, q_{up} and $\overline{q_{up}}$. The monitor will be in state q_{up} if the most recent drawing action (`up`, `down`, or `right`) encountered has been `up`. So the monitor moves to state q_{up} whenever up_0 is pushed onto the stack, stays there during all subsequent `up` calls, and moves to $\overline{q_{up}}$ when it sees a call to `down` or `right`.

Running the monitor in parallel with the skyline PDA is achieved by constructing the product of the monitor and the PDA, which is simply the product of a finite automaton and a PDA. So we consider the "monitored" skyline PDA

$$A'_s = (\{q_{up}, \overline{q_{up}}\},\ \Gamma_s,\ \Delta'_s)$$

where Δ'_s is obtained from Δ_s by

Procedure `main`

$\langle q_s, main_0\rangle \longrightarrow \langle q_s, s_0 main_1\rangle$
$\langle q_s, main_1\rangle \longrightarrow \langle q_s, \epsilon\rangle$

Procedure `up`

$\langle q_s, up_0\rangle \longrightarrow \langle q_s, \epsilon\rangle$

Procedure `down`

$\langle q_s, down_0\rangle \longrightarrow \langle q_s, \epsilon\rangle$

Procedure `right`

$\langle q_s, right_0\rangle \longrightarrow \langle q_s, \epsilon\rangle$

Procedure `s`

$\langle q_s, s_0\rangle \longrightarrow \langle q_s, s_1\rangle$ `if (true)`
$\langle q_s, s_0\rangle \longrightarrow \langle q_s, s_4\rangle$ `if (false)`
$\langle q_s, s_1\rangle \longrightarrow \langle q_s, up_0 s_2\rangle$ $(\star)$
$\langle q_s, s_2\rangle \longrightarrow \langle q_s, m_0 s_3\rangle$
$\langle q_s, s_3\rangle \longrightarrow \langle q_s, down_0 s_4\rangle$ $(\star\star)$
$\langle q_s, s_4\rangle \longrightarrow \langle q_s, \epsilon\rangle$

Procedure `m`

$\langle q_s, m_0\rangle \longrightarrow \langle q_s, m_1\rangle$ `if (true)`
$\langle q_s, m_0\rangle \longrightarrow \langle q_s, m_5\rangle$ `if (false)`
$\langle q_s, m_1\rangle \longrightarrow \langle q_s, s_0 m_2\rangle$
$\langle q_s, m_2\rangle \longrightarrow \langle q_s, right_0 m_3\rangle$ $(\star\star)$
$\langle q_s, m_3\rangle \longrightarrow \langle q_s, m_4\rangle$ `if (true)`
$\langle q_s, m_3\rangle \longrightarrow \langle q_s, m_8\rangle$ `if (false)`
$\langle q_s, m_4\rangle \longrightarrow \langle q_s, m_0 m_8\rangle$
$\langle q_s, m_5\rangle \longrightarrow \langle q_s, up_0 m_6\rangle$ $(\star)$
$\langle q_s, m_6\rangle \longrightarrow \langle q_s, m_0 m_7\rangle$
$\langle q_s, m_7\rangle \longrightarrow \langle q_s, down_0 m_8\rangle$ $(\star\star)$
$\langle q_s, m_8\rangle \longrightarrow \langle q_s, \epsilon\rangle$

Fig. 1.8. The encoding of the skyline program as a PDA. Program points belonging to procedure p are written p_i for naturals i. They correspond to the program labels of the C program of Figure 1.6.

- replacing each rule $\langle q_s, \gamma\rangle \longrightarrow \langle q_s, up_0\gamma'\rangle$ marked by $(\star)$ in Figure 1.8 by two rules $\langle q_{up}, \gamma\rangle \longrightarrow \langle q_{up}, up_0\gamma'\rangle$ and $\langle \overline{q_{up}}, \gamma\rangle \longrightarrow \langle q_{up}, up_0\gamma'\rangle$;
- replacing each rule $\langle q_s, \gamma\rangle \longrightarrow \langle q_s, \gamma'\gamma''\rangle$ marked $(\star\star)$ in Figure 1.8 by two rules $\langle q_{up}, \gamma\rangle \longrightarrow \langle \overline{q_{up}}, \gamma'\gamma''\rangle$ and $\langle \overline{q_{up}}, \gamma\rangle \longrightarrow \langle \overline{q_{up}}, \gamma'\gamma''\rangle$; and
- replacing each unmarked rule $\langle q_s, \gamma\rangle \longrightarrow \langle q_s, w\rangle$ in Figure 1.8 by two rules $\langle q_{up}, \gamma\rangle \longrightarrow \langle q_{up}, w\rangle$ and $\langle \overline{q_{up}}, \gamma\rangle \longrightarrow \langle \overline{q_{up}}, w\rangle$.

We assume that the skyline program starts with a call to `main` in control location $\overline{q_{up}}$; that is, no drawing action, and in particular no upwards drawing action, has yet been observed. Our set I_s of initial configurations can thus be written as:

$$I_s = \{\langle \overline{q_{up}}, main_0\rangle\}$$

Now, the skyline program produces degenerate configurations if A'_s can reach any of the configurations of the set

$$D_s = \{\langle q_{up}, down_0 w\rangle \mid w \in \Gamma_s^*\}$$

a configuration in D_s is reached if an `up` drawing action is immediately followed by a `down` drawing action, regardless of the history represented by w in the definition of D_s. We can automatically check whether the program produces degenerate

configurations by answering the question

$$post^*_{A'_s}(I_s) \ \cap \ D_S \stackrel{?}{=} \emptyset$$

In Section 1.5.3, we introduce *regular configurations* as our set of symbolic configurations, and show that they satisfy **Conditions (1)-(6)**. The symbolic representation of the set $post^*_{A'_s}(I_s)$ is computed and discussed in Section 1.5.4.

1.5.3. *Regular Configurations*

The number of reachable configurations of a PDA may be infinite, due to the possibility of unbounded stacks. In particular, the skyline program may go `up` forever, stacking more and more unfinished calls to procedure `s`. So we must find a finite symbolic representation of possibly infinite sets of configurations.

We fix a PDA $A = (Q, \Gamma, \Delta)$. Since stack contents are words, sets of stack contents are languages over the alphabet Γ. Since *regular* languages can be finitely represented by finite automata, we choose as our family $\mathcal{F}$ of symbolic configurations the *regular sets of configurations*. A set C of configurations of A is *regular* if the set $\{w \in \Gamma^* \mid \langle q, w\rangle \in C\}$ is regular for all $q \in Q$.

We show that regular configurations satisfy **Conditions (1)-(5)** for forward search, but not **Condition (6)**. We then show that it is possible to accelerate the search so that it terminates.

Theorem 1.4. *Let I and D be regular sets of configurations. Symbolic forward search with regular sets satisfies conditions (1)-(5), but not condition (6).*

Condition (1). We need to show that every regular set of configurations has a finite symbolic representation. As the possible stack contents for each control location are known to be regular languages, one can compactly represent regular sets of configurations as a non-deterministic finite automaton with ϵ-transitions (NFAs). Accepting a configuration is achieved by making all control locations of a given PDA, A, initial states of the NFA. A configuration $\langle q, w\rangle$ is accepted by the NFA if it accepts w starting in state q. It is straightforward that every regular set of configurations of A has such a finite symbolic representation as an NFA. Formally, we shall call an NFA for accepting regular sets of configurations of A an A-automaton. It is defined as follows:

Let $A = (Q, \Gamma, \Delta)$ be a PDA. An *A-automaton* is a finite automaton $B = (P, \Gamma, Q, F, \delta)$, where P is a set of states containing Q, i.e., $P \supseteq Q$ holds, Γ is an alphabet, Q is a set of initial states, $F \subseteq P$ is a set of final states, and $\delta \subseteq P \times (\Gamma \cup \{\epsilon\}) \times (P \setminus Q)$ is a transition relation. B *accepts* a configuration $\langle q, w\rangle$ of A if some path of B leading from q to some final state q' is labeled by w; in this case we write $q \stackrel{w}{\rightsquigarrow}{}^* q'$. Notice that δ denotes the transition relation of an A-automaton, while Δ denotes the set of rules of the PDA.

Our definition of δ forbids transitions in B leading into some initial state. This is important for the correctness of the forward search algorithm presented below.

Observe that every regular language is accepted by some NFA satisfying this condition.

Example 1.4. Figure 1.9 shows two A'_s-automata. The one on the left accepts the set I_s, and the one on the right the set D_s.

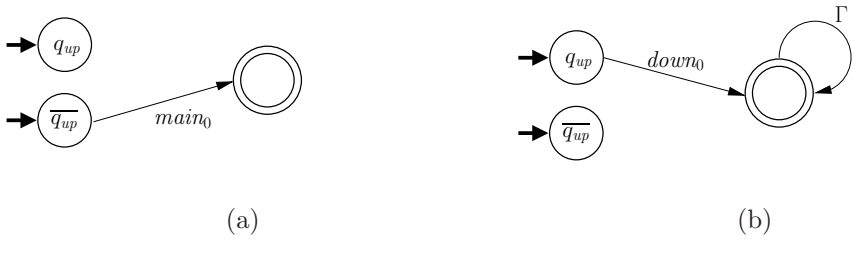

Fig. 1.9. Automata accepting the initial and dangerous configurations of the skyline PDA. Initial states are marked by big arrows, final states by double circles. If a set of stack symbols is attached to an arc, it denotes a set of transitions, one for each element of the set.

Condition (2). The set I_s is regular by hypothesis. The set $I_s = \{\langle \overline{q_{up}}, main_0 \rangle\}$ of our case study is accepted by the automaton on the left of Figure 1.9.

Condition (3) We have to show that if C is a regular set of configurations of PDA A, then also $C \cup post_A(C)$ is regular and efficiently computable from C's symbolic representation.

Let B be an A-automaton accepting a regular set C of configurations of A. We construct an A-automaton accepting $C \cup post_A(C)$. First, as we are interested in computing the *union* $C \cup post_A(C)$, we can safely keep all states and transitions of B. We add more states and transitions to also accept the configurations of $post_A(C)$.

The configurations of $post_A(C)$ are obtained by applying a rule to the configurations of C. The PDA A can have three types of rules: rules removing the topmost stack symbol, rules replacing it with one other symbol, and rules replacing it with a word of length 2. Figure 1.10 illustrates how to add new states and transitions to deal with each of these three cases. The second row of the Figure shows the three kinds of rules, and the third the transitions and states added. If in B we can go from q to a state p by reading the symbol γ (possibly together with an arbitrary number of ϵ's), and we have a rule $\langle q, \gamma \rangle \longrightarrow \langle q', \epsilon \rangle$ in Δ, then $post_A(C)$ must accept any configuration $\langle q', w \rangle$ such that $\langle p, \gamma w \rangle$ is accepted by B. Therefore, we add an ϵ-transition (q', ϵ, p) to the transition relation δ of B. This will exactly achieve this goal. The other two kinds of actions are treated similarly .

We obtain the whole of $C \cup post_A(C)$ by applying this construction to all rules in Δ and to all matchings of $q \overset{\gamma}{\rightsquigarrow}{}^* p$ in B. Note that we must indeed require initial states in B not to have incoming transitions. The reason is essentially the same as the reason for introducing a new initial state in the union construction of two

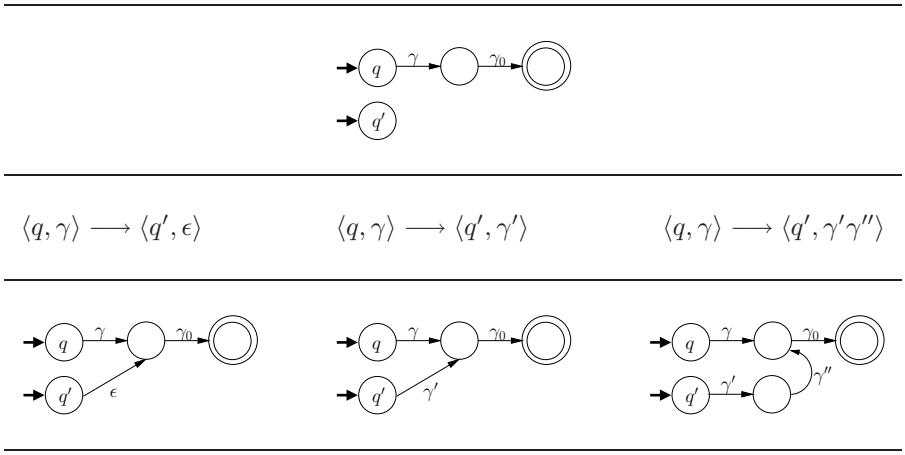

Fig. 1.10. Construction of $C \cup post_A(C)$. The first row shows part of an automaton accepting C. The second row shows the three possible type of PDA rules. The third row shows an automaton accepting $C \cup post_r(C)$ for each rule r.

NFAs. If, for instance, q' in Figure 1.10 had a self-loop, it would be easy to see that this may add non-reachable configurations to $C \cup post(C)$.

Having defined the notion of A-automata makes the proof of **Condition (4)**. The emptiness of $C \cap D$ can be easily checked using standard automata-theoretic techniques.

Condition (5). Checking $C_1 = C_2$ is decidable, because equality of languages accepted by NFA is decidable. Checking equality is known to be computationally expensive, but once we present the acceleration in the next section it will become clear that in our special case the procedure can be extremely simplified.

To check that **Condition (6)** does not hold, consider the PDAwith only one rule shown on the left of Figure 1.11 and the A-automaton shown on the right (ignore the black states for the moment), accepting only the configurations $\langle q, \gamma \rangle$. Each application of the procedure described in the proof of Condition (3) adds a new state to the automaton, and so forward search computes an infinite sequence of A-automata, each of them accepting exactly one word more than the previous one. So we have $C_0 \subset C_1 \subset C_2 \ldots$, and forward search does not terminate. This concludes the proof of Theorem 1.4.

If we look at the example of Figure 1.11 in more detail, we observe that $post^*(\langle q, \gamma \rangle) = \langle p, \gamma\gamma'^* \rangle$ is a regular set. This set is recognized by an automaton where we keep only the first black state in Figure 1.11, and add a self-loop to it, labeled by γ'. So, while the fixed point is a member of the family $\mathcal{F}$, forward search never reaches it, it keeps constructing better and better approximations to it, but never getting there. In the next section we show how to deal with this problem.

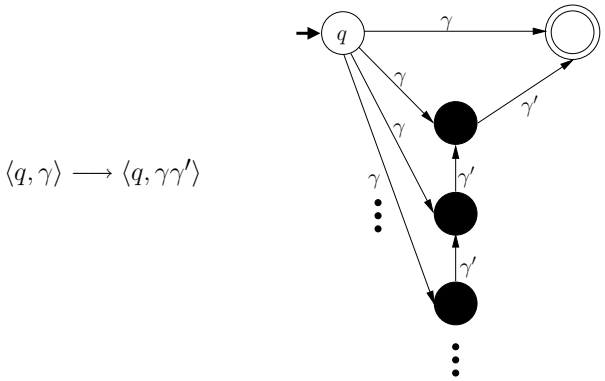

Fig. 1.11. An infinite ascending chain in the family of regular sets of configurations. Application of *post* for the rule to the left keeps on generating a new state (a black one). However, $post^* = \langle p, \gamma\gamma'^* \rangle$ is regular.

Termination by Acceleration. Figure 1.11 shows that forward search can compute an infinite ascending chain of regular sets of configurations. We resort to an *acceleration* ∇, as defined in Section 1.2.5.

Instead of computing $C \cup post_A(C)$ we will compute $C \nabla post_A(C)$ such that $post_A^*(C) \supseteq C \nabla post_A(C) \supseteq C \cup post_A(C)$. The key to defining ∇ is to *re-use states*. As hinted above, we only introduce *one* of the black states in Figure 1.11. More generally, we only introduce one state for each pair $\langle q', \gamma' \rangle$ such that Δ contains at least one rule of the form $\langle q, \gamma \rangle \longrightarrow \langle q', \gamma'\gamma'' \rangle$ (we also call the state $\langle q', \gamma' \rangle$). These states will then be "re-used".

Below we present the algorithm for computing an A-automaton accepting $C \nabla post_A(C)$ from an A-automaton B accepting C. Again, we assume that B has no initial states with incoming transitions. The algorithm adds states and transitions to B according to the following saturation rules:

(1) If A has a rule $\langle q, \gamma \rangle \longrightarrow \langle q', \epsilon \rangle$ and B has a transition (q, γ, p), then add a transition (q', ϵ, p) to B.
(2) If A has a rule $\langle q, \gamma \rangle \longrightarrow \langle q', \gamma' \rangle$ and B has a transition (q, γ, p), then add a transition (q', γ', p) to B.
(3) If A has a rule $\langle q, \gamma \rangle \longrightarrow \langle q', \gamma'\gamma'' \rangle$ and B has a transition (q, γ, p), then
    - if B does not yet contain a state $\langle q', \gamma' \rangle$, then add such a state to B; and
    - add transitions $(q', \gamma', \langle q', \gamma' \rangle)$ and $(\langle q', \gamma' \rangle, \gamma'', p)$ to B.

The accelerated forward search replaces the line $C := C \cup post_A(C)$ by $C := C \nabla post_A(C)$, where the automaton for $C \nabla post_A(C)$ is computing using the algorithm we have just sketched. Let us apply the accelerated search to the example of Figure 1.11. The first iteration adds a new state $\langle q, \gamma \rangle$, which corresponds to the first black state in the Figure, and a transition $(q, \gamma, \langle q, \gamma \rangle)$. In the second

iteration, because of this transition and the third saturation rule, the algorithm adds no new states (the state $\langle q, \gamma \rangle$ has already been added), but a new transition $(\langle q', \gamma' \rangle, \gamma', \langle q', \gamma' \rangle)$: this is the self-loop mentioned above. In the third iteration, the algorithm does not add any new state or transition, and so the accelerated forward search terminates. The search has "jumped" to the limit $post^*(\langle q, \gamma \rangle)$ in only two steps.

The accelerated search *always terminates*. The number of states that can be added is bounded, for instance by the number of rules of the PDA, and so the number of transitions is also bounded. So the search eventually reaches a point at which the saturation rules cannot add any new state or transition.

The accelerated search must also satisfy $post^*_A(C) \supseteq C \nabla post_A(C)$ and $C \nabla post_A(C) \supseteq C \cup post_A(C)$. The second of this conditions is easy to prove: each word accepted by the A-automaton for $C \cup post_A(C)$ is also accepted by the A-automaton for $C \nabla post_A(C)$. The other condition is non-trivial, but not difficult. We refer the reader to the correctness proof in [3].[b] In the same reference, a detailed complexity analysis of the accelerated search is carried out. The search needs polynomial time in both the size of PDA and the size of the A-automaton accepting the set I of initial configurations.

1.5.4. *Forward Search for the Skyline Program*

Initial and Dangerous Configurations. The symbolic representations of I_s and D_s as A'_s-automata are shown in Figure 1.9, which proves both sets regular. Actually, initial configurations are typically quite simple and regular, and the same holds for many interesting sets of dangerous configurations, applying the monitor trick if necessary. However, we must mention that there are interesting properties that cannot be tested by a monitor, if the monitor has to be a finite automaton. For instance, the skyline program satisfies that in any execution the number of up and down moves is equal. However, no finite automaton can monitor this property, because, loosely speaking, it would then be an automaton accepting the language $\{\texttt{up}^n\texttt{down}^n \mid n \geq 0\}$. If we allow general PDAs as monitors, then the approach cannot be applied, because the intersection of two PDAs may not be equivalent to a PDA.

Let us now illustrate the first few iterations of the accelerated search, starting from the symbolic representation of I_s in Figure 1.9. The only rule "matching" I_s is $\langle \overline{q_{up}}, main_0 \rangle \longrightarrow \langle \overline{q_{up}}, s_0 main_1 \rangle$, because in I_s we have $\overline{q_{up}} \rightsquigarrow^* q$ reading $main_0$. Therefore, according to saturation rule (3) we add the transitions $(\overline{q_{up}}, s_0, \langle \overline{q_{up}}, s_0 \rangle)$ and $(\langle \overline{q_{up}}, s_0 \rangle, main_1, q)$ to I_s and obtain (an automaton for) I^1_s depicted in Figure 1.12.

I^1_s accepts all configurations reachable within one step: $\langle \overline{q_{up}}, main_0 \rangle$ and $\langle \overline{q_{up}}, s_0 main_1 \rangle$.

[b] For presentation reasons our notation differs; in particular, we use Q and P for the states of A and B, respectively, while in [3] it is the other way round.

Fig. 1.12. Automata representing the set I_s^1 and I_s^2 of skyline configurations reachable within one, respectively two, steps from the initial configuration.

Now, two further rules match in I_s^1: $\langle \overline{q_{up}}, s_0 \rangle \longrightarrow \langle \overline{q_{up}}, s_1 \rangle$ and $\langle \overline{q_{up}}, s_0 \rangle \longrightarrow \langle \overline{q_{up}}, s_4 \rangle$. Applying saturation rule (2) we introduce s_1 and s_4 labeled transitions in parallel to s_0 to arrive at I_s^2 also shown in Figure 1.12. New reachable configurations are thus $\langle \overline{q_{up}}, s_1 main_1 \rangle$ and $\langle \overline{q_{up}}, s_4 main_1 \rangle$. The first is reached when the nondeterministic choice in `s` evaluates to true, and the second when the same choice yields false.

Before we comment on the final outcome of the $post^*_{A'_s}(I_s)$ computation depicted in Figures 1.13 and 1.14, note that we will add $(\overline{q_{up}}, \epsilon, \langle \overline{q_{up}}, s_0 \rangle)$ due to rule $\langle \overline{q_{up}}, s_4 \rangle \longrightarrow \langle \overline{q_{up}}, \epsilon \rangle$, that is, procedure `s` returning. Thanks to this new transition the configuration $\langle \overline{q_{up}}, main_1 \rangle$ becomes reachable. This, in turn, allows us to add an ϵ-transition from $\overline{q_{up}}$ to q indicating program termination.

Finally, observe that rule $\langle \overline{q_{up}}, s_1 \rangle \longrightarrow \langle q_{up}, up_0 s_2 \rangle$ matches in I_s^2. Using saturation rule (2) we add $(q_{up}, s_0, \langle q_{up}, s_0 \rangle)$ and $(\langle q_{up}, s_0 \rangle, s_2, \langle \overline{q_{up}}, s_2 \rangle)$ to I_s^2. We hereby obtain the first reachable configuration with control location q_{up}: $\langle q_{up}, up_0 s_2 main_1 \rangle$, which corresponds to entering `s`, taking the true branch of the choice, and calling `up`.

The automaton accepting $post^*_{A'_s}(I_s)$ is computed as delineated above. It does not fit in one figure, and so it is shown in Figures 1.13 and 1.14. Figure 1.13 shows the reachable configurations of the form $\langle q_{up}, \cdot \rangle$ and Figure 1.13 those of the form $\langle \overline{q_{up}}, \cdot \rangle$. The complete automaton obtained by merging the vertical "backbones" of the two automata.

In order to decide $post^*_{A'_s}(I_s) \cap D_S \stackrel{?}{=} \emptyset$ one can construct the product of the automata for D_s and $post^*_{A'_s}(I_s)$, and check for emptiness. But in our case this is not necessary. It suffices to check whether $post^*_{A'_s}(I_s)$ accepts anything starting in q_{up} with a subsequent $down_0$ labeled transition. To the skyline programmer's luck, it is easy to see that this is not the case, which means that a plotter driven by this program will not draw degenerate skylines.

We conclude this section by inviting the interested reader to undertake the instructive exercise of picking a reachable configuration from Figures 1.13 and 1.14 and finding an execution of A'_s that reaches it. As an example, Figure 1.15 shows an execution leading to the configuration $\langle q_{up}, up_0 m_6 m_8 s_3 main_1 \rangle$. Each plotter move–that is, each line segment–is annotated with the configuration at which it was drawn. One can also check that all these configurations are accepted by one of

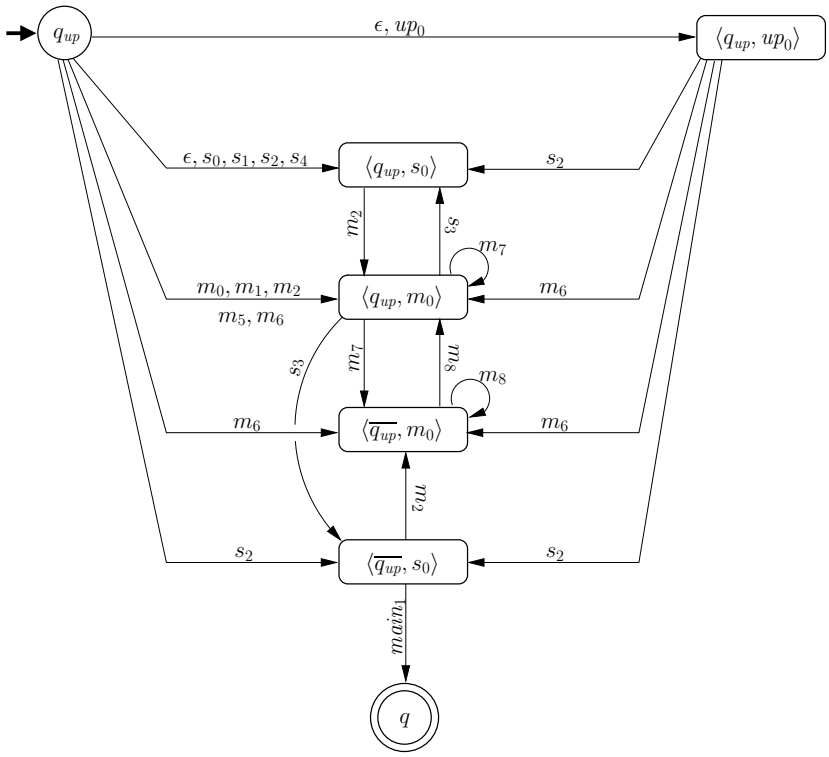

Fig. 1.13. An automaton representing the configurations of $post^*_{A'_s}(I_s)$ that are of the form $\langle q_{up}, \cdot \rangle$. Note that none of the dangerous configurations can be accepted by this automaton, because there is no accepting path starting with $down_0$.

the automata in Figures 1.13 and 1.14.

1.5.5. *Conclusion and Further Reading*

Modelling sequential procedural programs by pushdown automata is the basis of the Moped model checker developed by Schwoon [3, 15]. Suwimonteerabuth et al. have implemented a Java front end for this model checker, called jMoped [16, 17]. Recursive state machines are a model of computation equivalent to pushdown systems; model-checking algorithms for them have been proposed by several authors (see for instance [18, 19]).

Backward search with regular sets of configurations can also be accelerated to guarantee termination, and the algorithm is even slightly simpler. The saturation algorithms were presented in [20]. Efficient algorithms for both forward and backward search can be found in [21].

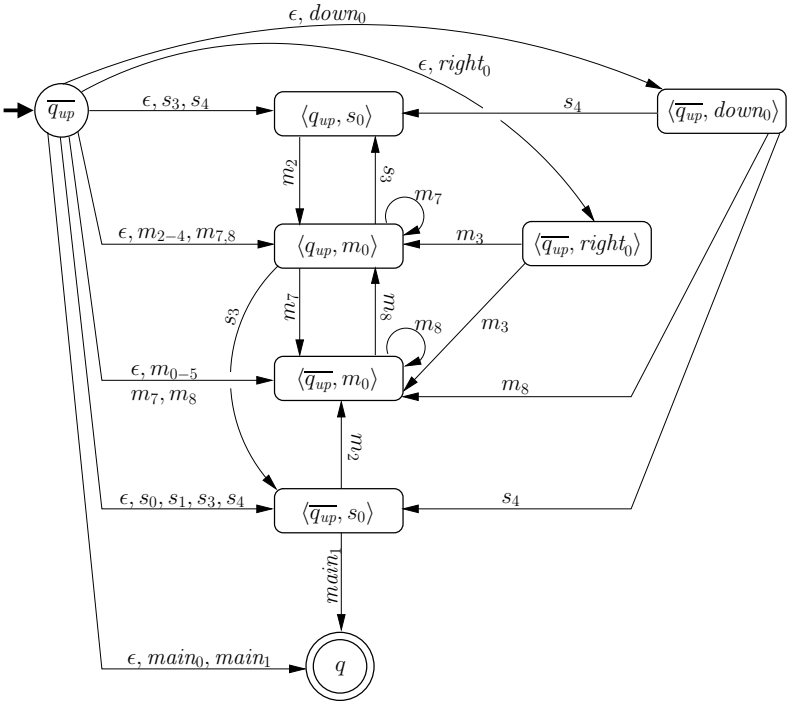

Fig. 1.14. An automaton representing the configurations of $post^*_{A'_s}(I_s)$ that are of the form $\langle \overline{q_{up}}, \cdot \rangle$.

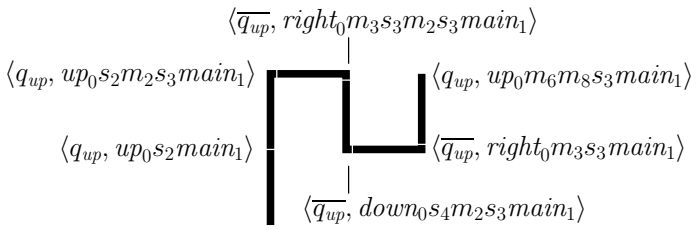

Fig. 1.15. A sample skyline and potential program configurations after each drawing step. By checking how the automata in Figures 1.13 and 1.14 accept the respective configurations, one gains quite some insight into how the program works.

1.6. Conclusion

We have presented an introduction to the verification of systems with an infinite state space. We have argued that many different sources of infinity can be modelled within the framework of extended automata and symbolic search, and have considered three different case studies involving real-time, parametric systems, and control structures. The key issue of symbolic search is finding an adequate class of symbolic configurations, or, in other words, an adequate data structure for representing infinite sets. We have presented several data structures: constraints, sets of minimal elements, and finite automata. We have also addressed the problem of guaranteeing termination of the search, and introduced three different strategies for achieving the goal.

References

[1] J. Bengtsson and W. Yi. Timed automata: Semantics, algorithms and tools. In *Lectures on Concurrency and Petri Nets*, vol. 3098, *LNCS*, pp. 87–124. Springer, (2003).
[2] G. Delzanno, Constraint-based verification of parameterized cache coherence protocols, *Formal Methods in System Design*. **23**(3), 257–301, (2003).
[3] S. Schwoon. *Model-Checking Pushdown Systems*. PhD thesis, Technische Universität München, (2002).
[4] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pp. 238–252, (1977).
[5] R. Alur and D. L. Dill, A theory of timed automata, *Theor. Comput. Sci.* **126**(2), 183–235, (1994).
[6] D. L. Dill. Timing assumptions and verification of finite-state concurrent systems. In *Automatic Verification Methods for Finite State Systems*, vol. 407, *LNCS*, pp. 197–212. Springer, (1989).
[7] R. W. Floyd, Algorithm 97: Shortest path, *Commun. ACM.* **5**(6), 345, (1962).
[8] R. Alur and P. Madhusudan. Decision problems for timed automata: A survey. In *In Proceedings of SFM'04, Lect. Notes Comput. Sci. 3185, 1–24*, pp. 1–24. Springer, (2004).
[9] J. Handy, *The Cache Memory Book*. (Academic Press, 1993).
[10] M. R. Garey and D. S. Johnson, *A guide to the theory of NP-completeness*. (W.H Freeman, 1979).
[11] E. Contejean and H. Devie, An efficient incremental algorithm for solving systems of linear diophantine equations, *Inf. Comput.* **113**(1), 143–172, (1994).
[12] P. A. Abdulla and B. Jonsson, Undecidable verification problems for programs with unreliable channels, *Inf. Comput.* **130**(1), 71–90, (1996).
[13] P. A. Abdulla, K. Cerans, B. Jonsson, and Y.-K. Tsay, Algorithmic analysis of programs with well quasi-ordered domains, *Inf. Comput.* **160**(1-2), 109–127, (2000).
[14] A. Finkel and P. Schnoebelen, Well-structured transition systems everywhere!, *Theor. Comput. Sci.* **256**(1-2), 63–92, (2001).
[15] J. Esparza and S. Schwoon. A BDD-based model checker for recursive programs. In *CAV*, vol. 2102, *LNCS*, pp. 324–336. Springer, (2001).
[16] D. Suwimonteerabuth, S. Schwoon, and J. Esparza. jMoped: A Java bytecode checker based on Moped. In *TACAS*, vol. 3440, *LNCS*, pp. 541–545. Springer, (2005).

[17] D. Suwimonteerabuth, F. Berger, S. Schwoon, and J. Esparza. jMoped: A test environment for Java programs. In *CAV*, vol. 4590, *LNCS*, pp. 164–167. Springer, (2007).
[18] R. Alur, K. Etessami, and M. Yannakakis. Analysis of recursive state machines. In *CAV*, vol. 2102, *LNCS*, pp. 207–220. Springer, (2001).
[19] M. Benedikt, P. Godefroid, and T. W. Reps. Model checking of unrestricted hierarchical state machines. In *ICALP*, vol. 2076, *LNCS*, pp. 652–666. Springer, (2001).
[20] A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model checking. In *Proceedings of CONCUR'97*, vol. 1243, *LNCS*. Springer, (1997).
[21] J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. In *CAV*, vol. 1855, *LNCS*, (2000).