Symbolic Determinisation of Extended Automata

Thierry Jéron, Hervé Marchand, Vlad Rusu

1

Symbolic determinisation of extended automata

Thierry Jéron, Hervé Marchand, and Vlad Rusu First.Last@irisa.fr
Irisa/Inria Rennes, Campus de Beaulieu, 35042 Rennes France

Summary. We define a symbolic determinisation procedure for a class of infinite-state systems, which consists of automata extended with symbolic variables that may be infinite-state. The subclass of extended automata for which the procedure terminates is characterised as *bounded lookahead extended automata*. It corresponds to automata for which, in any location, the observation of a bounded-length trace is enough to infer the first transition actually taken. We discuss applications of the algorithm to the verification, testing, and diagnosis of infinite-state systems.

Key words: symbolic automata, determinisation

1.1 Introduction

Most existing models of computation are nondeterministic, but they include restricted, deterministic versions as subclasses. A natural question is comparing the expressiveness of the general, nondeterministic class with that of the corresponding restricted, deterministic subclass. For example, it is well known that nondeterministic and deterministic *finite automata* on *finite words* are equivalent, but for finite automata on *infinite* words, the equivalence depends on the acceptance condition (e.g., *Müller* versus *Büchi* acceptance); and for *pushdown* and *timed automata*, the nondeterministic version is strictly more expressive than the deterministic one [2, 1].

Besides this theoretical interest, the distinction between nondeterministic and deterministic models has practical consequences. For example, *verification* consists in checking whether an *implementation* of a system satisfies a *specification*; both views of the system are modeled by automata of some kind. This problem can be seen as a language inclusion problem, which in turn can be encoded into a language emptyness problem (i.e., checking the emptyness of the language recognised by a product between the implementation and the *complement* of the specification). The complement of the specification is an automaton that accepts exactly the words that are rejected by the specification, and is easily computed if the specification is deterministic (by complementing the specification's acceptance condition). Otherwise, if the specification is nondeterministic, it has to be *determinised*, i.e., turned into an equivalent deterministic machine.

Hence, determinisation is an important operation in formal verification. It is also important in other fields such as *conformance testing* and *fault diagnosis* where deterministic testers (resp. diagnosers) have to be derived from specifications that are, in general nondeterministic due to, e.g., partial observation.

In this paper we define a determinisation operation for a class of infinite-state systems, which consists of extended automata operating on symbolic variables and communicating with the environment via synchronising actions. Variants of this model are often encountered in the literature and can be used, e.g., for the formal specification of reactive systems. The determinisation procedure consists in iterating a sequence of *local determinisation steps*, which postpone operations on the variables until it becomes clear which exact operations should have been performed. The subclass of extended automata on which the procedure terminates is characterised as *bounded-lookahead* automata, for which the observation of a bounded-length trace is enough to infer the first transition actually taken.

The result is nontrivial because the order in which local determinisation steps are iterated has a strong influence on termination. The main difficulty was to find an order for which the bounded lookahead decreases at each iteration, thus ensuring termination of the procedure.

The rest of the paper is organised as follows. We first introduce extended automata and the determinisation operation by means of examples. Then, in Section 1.2 we formally define the syntax and semantics of extended automata, and in Section 1.3 the determinisation operation is formally defined. The operation may not terminate in general, and in Section 1.4 the subclass for which the procedure does terminate is precisely characterised via necessary and sufficient conditions. However, these conditions are undecidable, hence, we also provide sufficient, decidable conditions for termination. In Section 1.5 we discuss applications of our procedure to the verification, testing, and diagnosis of reactive systems, and conclude in Section 1.6. The full paper available at `http://www.irisa.fr/vertecs/Equipe/Rusu/rr1776.ps` contains proofs of all the results.

Example 1 (extended automata, determinisation). Figure 1.1 (left) depicts an extended automaton $\mathcal{S}$. In location l_0, the action a occurs. If $x \geq 0$ then the control goes to location l_1 and the variable x is decreased by 1, and if $x \leq 0$ then the control goes to location l_2 and x is increased by 2. Clearly, if $x = 0$ then the next control location and the next value of x are not uniquely defined: the system is nondeterministic.

The right-hand side of Figure 1.1 depicts the automaton $det(\mathcal{S})$ obtained after determinising $\mathcal{S}$. Intuitively, the locations l_1 and l_2, which could be nondeterministically chosen as the next control location after an action a, are merged into one new location denoted by $\langle l_0, \langle l_1, l_2 \rangle \rangle$. A new transition labeled by a goes from l_0 to $\langle l_0, \langle l_1, l_2 \rangle \rangle$. This transition is taken if a occurs, and if x satisfies the *disjunction* $x \geq 0 \vee x \leq 0$ (which actually simplifies to *true*). This condition is the disjunction of the guards of the two transitions involved in the nondeterministic choice in $\mathcal{S}$.

Note, however, that those transitions perform different assignments to variables: $x := x - 1$ for one, and $x := x + 2$ for the other. Hence, the new transition from l_0 to $\langle l_0, \langle l_1, l_2 \rangle \rangle$ of $det(\mathcal{S})$ does not "know" which assignment to perform. To solve this problem, the idea is to *postpone* assignments until it becomes clear which one of the transitions of the nondeterministic choice was actually taken, and then to "catch up" with the assignments in order to preserve the semantics.

Hence, if b occurs after a, then the transition from l_0 to l_1 was taken (hence, $x := x - 1$ sould have been performed), but if c occurs after a, the transition from l_0 to l_2 was taken (hence $x := x + 2$ should have been performed). Note how the assignments are simulated in $det(\mathcal{S})$: the transition labeled by b (resp. by c) has $x - 1$

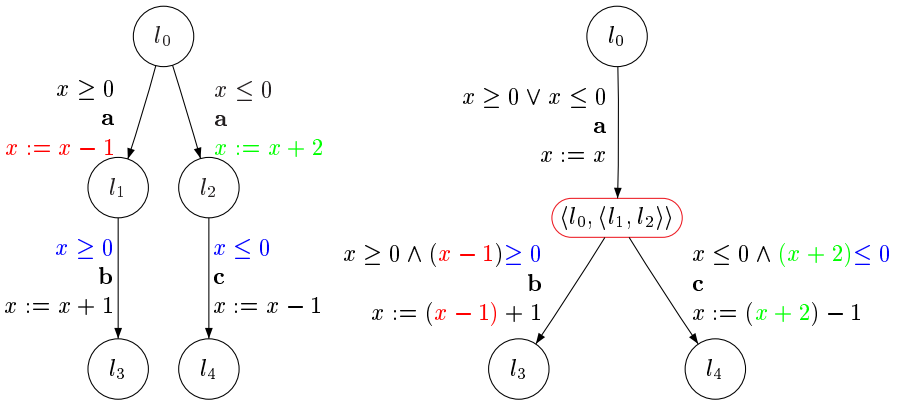

Fig. 1.1. Left: extended automaton $\mathcal{S}$ Right: extended automaton $det(\mathcal{S})$

(resp. $x+2$) substituted for x in its guard and assignments. To match the behaviour of $\mathcal{S}$, in which the transition labeled by b (resp. c) are fireable only after a transition labeled a has been fired with $x \geq 0$ (resp. $x \leq 0$) holding, the guard of the transition labeled by b (resp. c) in $det(\mathcal{S})$ is strengthened by $x \leq 0$ (resp. $x \geq 0$).

1.2 Extended automata

Extended automata consist of a finite control structure and a finite set of typed variables V. Each variable $x \in V$ takes values in some domain dom_x. A *valuation* ν of the variables V is a function that associates to each variable $x \in V$ a value $\nu(x) \in dom_x$. The set of valuations of the variables V is denoted by $\mathcal{V}$. In the sequel, a predicate P over variables V is often identified with its set of "solutions", i.e., the set of valuations $\mathcal{V}' \subseteq \mathcal{V}$ of the variables V for which P is *true*.

Definition 1 (extended automaton). *An extended automaton (sometimes refered to simply as an* automaton*) is a tuple* $\mathcal{S} = \langle V, \Theta, L, l^0, \Sigma, \mathcal{T} \rangle$*:*

- V *is a finite set of typed* variables
- Θ *is the* initial condition*, a predicate on* V*, assumed to have a unique solution* $\nu_0 \in \mathcal{V}$
- L *is a nonempty, finite set of* locations *and* $l^0 \in L$ *is the* initial location*,*
- Σ *is a nonempty, finite* alphabet of actions*,*
- $\mathcal{T}$ *is a set of* transitions*. Each transition* $t \in \mathcal{T}$ *is associated with a tuple* $\langle \mathsf{o}_t, G_t, a_t, A_t, \mathsf{d}_t \rangle$*, where*
  - $\mathsf{o}_t \in L$ *is called the* origin *of the transition,*
  - G_t *is a Boolean expression over variables* V*, called the* guard*,*
  - $a_t \in \Sigma$ *is called the* action *of the transition,*
  - A_t *is the* assignment *of the transition: a set of expressions of the form* $(x := A^x)_{x \in V}$ *where, for each* $x \in V$*, the right-hand side* A^x *of the assignment* $x := A^x$ *is an expression on* V*,*
  - $\mathsf{d}_t \in L$ *is called the* destination *of the transition.*

We sometimes write $t : \langle \mathsf{o}, G, a, A, \mathsf{d} \rangle$ to emphasise the tuple associated to t. By slight abuse of notation, we shall denote by $\circ$ an operation of syntactical substitution: a guard G (or an assignment A) is composed with another assignment A' by replacing

in G (resp. in the right-hand side of A) all the variables by their corresponding right-hands sides from A'. Examples of such substitutions in guards and assignments have been given in Example 1 above.

The semantics of extended automata is described by labelled transitions systems.

Definition 2 (Labelled Transition System (LTS)). *A Labelled Transition System is a tuple $S = \langle Q, Q^0, \Lambda, \rightarrow \rangle$ where Q is a set of* states*, $Q^0 \subseteq Q$ is the set of* initial states*, Λ is a set of* labels*, and $\rightarrow \subseteq Q \times \Lambda \times Q$ is the* transition relation*.*

The LTS semantics of an extended automaton enumerates the valuations $\mathcal{V}$ of its variables V. For an expression E involving (a subset of) V, and for $\nu \in \mathcal{V}$, we denote by $E(\nu)$ the value obtained by substituting in E each variable x by its value $\nu(x)$.

Definition 3 (Semantics of extended automata). *The semantics of an extended automaton $\mathcal{S} = \langle V, \Theta, L, l^0, \Sigma, \mathcal{T} \rangle$ is an LTS $[\![\mathcal{S}]\!] = \langle Q, Q^0, \Lambda, \rightarrow \rangle$, where*

- *the set of states is $Q = L \times \mathcal{V}$,*
- *the initial state is $q^0 = \langle l_0, \nu_0 \rangle$, where ν_0 is the unique valuation satisfying Θ,*
- *the set of labels is $\Lambda = \mathcal{T}$,*
- *$\rightarrow$ is the smallest relation in $Q \times \Lambda \times Q$ defined by the following rule:*

$$\frac{\langle l, \nu \rangle, \langle l', \nu' \rangle \in Q \quad t : \langle l, G, a, A, l' \rangle \in \mathcal{T} \quad G(\nu) = true \quad \nu' = A(\nu)}{\langle l, \nu \rangle \xrightarrow{t} \langle l', \nu' \rangle}$$

The rule says that the transition $t : \langle l, G, a, A, l' \rangle$ is *fireable* in a state $\langle l, \nu \rangle$ if the guard G evaluates to $true$ when the variables evaluate according to ν; then the transition takes the system to the state $\langle l', \nu' \rangle$ where the assignment A of the transition maps the valuation ν to ν'.

We extend this notion to sequences of transitions $\sigma = t_1 \cdot t_2 \cdots t_n \in \mathcal{T}^*$, saying that σ is *fireable* in a state $q \in Q$ if there exists states $q_1 = q, q_2, \ldots q_n \in Q$ such that $\forall i = 1 \ldots n-1$, $q_i \xrightarrow{t_i} q_{i+1}$. We then write $q \xrightarrow{\sigma}$ to say that σ is fireable in q. The transition sequence σ is *initially fireable* if it is fireable in the initial state q_0. A state q is *reachable* if there exists an initially fireable transition sequence σ leading to it, i.e., $\exists \sigma \in \mathcal{T}^*, q_0 \xrightarrow{\sigma} q$. We denote by $Reach(\mathcal{S})$ the set of reachable states. For a sequence $\sigma = t_1 \cdots t_n \in \mathcal{T}^n$ $(n \geq 1)$, we let $first(\sigma) \triangleq t_1$.

Definition 4 (trace). *The* trace *of a transition sequence $\sigma = t_1 \cdot t_2 \cdots t_n$ is the projection $trace(\sigma) = a_{t_1} \cdot a_{t_2} \cdots a_{t_n}$ of σ on the set Σ of actions. The* set of traces *of an extended automaton $\mathcal{S}$ is the set of traces of initially fireable transition sequences and is denoted by $Traces(\mathcal{S})$.*

1.3 Local Determinisation

Intuitively, an extended automaton is *deterministic* if in each location, the guards of the transitions labeled by the same action are mutually exclusive. *Determinising* an extended automaton $\mathcal{S}$ means computing a deterministic extended automaton $det(\mathcal{S})$ with the same traces as $\mathcal{S}$.

Definition 5 (deterministic extended automaton). *An extended automaton $\langle V, \Theta, L, l^0, \Sigma, \mathcal{T} \rangle$ is deterministic in a location $l \in L$ if for all actions $a \in \Sigma$ and each pair $t_1 : \langle l, G_1, a, A_1, l_1 \rangle$ and $t_2 : \langle l, G_2, a, A_2, l_2 \rangle$ of transitions with origin l and labeled by a, the conjunction of the guards $G_1 \wedge G_2$ is unsatisfiable. The automaton is deterministic if it is deterministic in all locations $l \in L$.*

It is assumed that the guards are written in a theory where satisfiability is decidable, such as, e.g., combinations of quantifier-free Presurger arithmetic formulas, arrays, and lists. Such formulas are expressive enough to encode the most common data structures, and their satisfiability is decidable using, e.g., the classical Nelson-Oppen combination of decision procedures [5]. Note that determinism does not take reachability of states into account. However, since extended automata have a unique initial state, the definition of determinism is equivalent to the fact that the semantics of a deterministic extended automaton is a deterministic LTS in the usual sense.

Exemple 1 shows that determinising two transitions consists in merging the two transitions into a new one, and propagating guards and assignments onto transitions following them (cf. Figure 1.1). Formally, $follow(t) \triangleq \{t' \in \mathcal{T} | \mathsf{o}_{t'} = \mathsf{d}_t\}$. We also denote by Id_V the *identity* assignments over variables V, i.e., $x := x$ for each $x \in V$.

Definition 6 (determinising two transitions). *Let $\mathcal{S}$ be an extended automaton, and let $t_1, t_2 \in \mathcal{T}$ be two transitions with same origin $\mathsf{o} = \mathsf{o}_{t_1} = \mathsf{o}_{t_2}$ and same action $a = a_{t_1} = a_{t_2}$. The automaton $det_2(\mathcal{S}, t_1, t_2)$ is defined as follows. If $\mathsf{G}_{t_1} \wedge \mathsf{G}_{t_2}$ is unsatisfiable then $det_2(\mathcal{S}, t_1, t_2) = \mathcal{S}$, otherwise,*

- $V_{det_2(\mathcal{S},\, t)} = V_{\mathcal{S}}$
- $\Theta_{det_2(\mathcal{S},\, t)} = \Theta_{\mathcal{S}}$
- $L_{det_2(\mathcal{S},\, t)} = L_{\mathcal{S}} \cup \{\langle \mathsf{o}, \langle \mathsf{d}_{t_1}, \mathsf{d}_{t_2} \rangle \rangle\}$, *where* $\langle \mathsf{o}, \langle \mathsf{d}_{t_1}, \mathsf{d}_{t_2} \rangle \rangle$ *is a new location*
- $l^0_{det_2(\mathcal{S},\, t)} = l^0_{\mathcal{S}}$
- $\Sigma_{det_2(\mathcal{S},\, t)} = \Sigma_{\mathcal{S}}$
- $\mathcal{T}_{det_2(\mathcal{S},\, t)} = \mathcal{T}_{\mathcal{S}} \setminus \{t_1, t_2\} \cup \{t_{1,2}\} \cup T_1 \cup T_2$, *where*
  - $t_{1,2} = \langle \mathsf{o}, \mathsf{G}_{t_1} \vee \mathsf{G}_{t_2}, a, Id_V, \langle \mathsf{o}, \langle \mathsf{d}_{t_1}, \mathsf{d}_{t_2} \rangle \rangle \rangle$,
  - *for* $i = 1, 2$, $T_i = \bigcup_{t' \in follow(t_i)} \{modify_i(t')\}$, *with the transitions* $modify_i(t')$: $\langle \langle \mathsf{o}, \langle \mathsf{d}_{t_1}, \mathsf{d}_{t_2} \rangle \rangle,\ \mathsf{G}_{t_i} \wedge \mathsf{G}_{t'} \circ \mathsf{A}_{t_i}, a_{t'}, \mathsf{A}_{t'} \circ \mathsf{A}_{t_i}, \mathsf{d}_{t'} \rangle$.

The transitions t_1 and t_2 in $\mathcal{S}$ are replaced in $det_2(\mathcal{S}, t_1, t_2)$ by the set of transitions $\{t_{1,2}\} \cup T_1 \cup T_2$. The transition $t_{1,2}$ leads from the common origin o of t_1, t_2 to the new location $\langle \mathsf{o}, \langle \mathsf{d}_{t_1}, \mathsf{d}_{t_2} \rangle \rangle$; its guard is the *disjunction* of those of t_1, t_2; hence, $t_{1,2}$ can be fired whenever t_1 or t_2 can. However, $t_{1,2}$ does not perform any of the assignments of t_1, t_2 because it does not "know" which ones to perform. The assignments are postponed onto *copies* of the transitions $t' \in follow(t_i)$ $(i = 1, 2)$, *modified* in order to "catch up" with the effect of transition t_i:

- the guard $\mathsf{G}_{modify_i(t')}$ equals $\mathsf{G}_{t_i} \wedge \mathsf{G}_{t'} \circ \mathsf{A}_{t_i}$. Intuitively, this amounts to *firing the transition* $modify_i(t')$ in $det_2(\mathcal{S}, t_1, t_2)$, *under exactly the same conditions* as the transition t' in $\mathcal{S}$: the conjunct G_{t_i} "recalls" that t_i should have been fired before t', and by composing $\mathsf{G}_{t'}$ with A_{t_i}, the effect of t_i on the variables is simulated before the guard of t' is evaluated.
- $\mathsf{A}_{modify_i(t')}$ performs the assignments of $\mathsf{A}_{t'}$ *composed* with the assignments A_{t_i}. In this way, the cumulated effect on the variables of firing in sequence t_i then t' in $\mathcal{S}$ is simulated.

Definition 7 (Local determinisation in location). *The* local determinisation in location l *of an extended automaton $\mathcal{S} = \langle V, \Theta, L, l^0, \Sigma, \mathcal{T} \rangle$, where $l \in L$ and $a \in \Sigma$, is defined as follows. Let $\mathcal{T}_l \subseteq \mathcal{T}$ be the set of all transitions with origin l, then:*

- $det(\mathcal{S}, l) = \mathcal{S}$ *if for every pair of same-labeled distinct transitions t_1, $t_2 \in \mathcal{T}_l$, the formula $\mathsf{G}_{t_1} \wedge \mathsf{G}_{t_2}$ is unsatisfiable;*
- *otherwise, choose two distinct transitions $t_1, t_2 \in \mathcal{T}_l$ such that $\mathsf{a}_{t_1} = \mathsf{a}_{t_2}$, $\mathsf{G}_{t_1} \wedge \mathsf{G}_{t_2}$ is satisfiable, and let $det(\mathcal{S}, l) = det(det_2(\mathcal{S}, t_1, t_2), l)$.*

The operation terminates: the set of pairs of nondeterministic transitions decreases.

1.4 Bounded-Lookahead Extended Automata

We now know hot to eliminate nondeterminism from a location $l \in L_S$. Then, to eliminate the nondeterminism globally from $\mathcal{S}$, one should iterate $det(\mathcal{S}, l)$ for all $l \in L_S$. However, local determinisation creates new locations, which may themselves be nondeterministic and have to be determinised, which may give rise to yet another set of nondeterministic locations, etc. This raises the question of whether the global determinisation process ever terminates. In this section we define a global determinisation procedure that we show to terminate exactly for the class of *bounded lookahead* extended automata. Intuitively, an automaton is deterministic with lookahead n if any nondeterministic choice can be resolved by looking n actions ahead.

Definition 8 (bounded lookahead). *An automaton $\mathcal{S} = \langle V, \Theta, L, q^0, \Sigma, \mathcal{T} \rangle$ has lookahead $n \in \mathbb{N}$ in a state $q \in Q_{[\![S]\!]}$ if $\forall \sigma_1, \sigma_2 \in \mathcal{T}^{n+1}$. $q \xrightarrow{\sigma_1} \wedge q \xrightarrow{\sigma_2} \wedge trace(\sigma_1) = trace(\sigma_2) \Rightarrow first(\sigma_1) = first(\sigma_2)$. The automaton has lookahead n in a set $Q' \subseteq Q_{[\![S]\!]}$ of states if it has lookahead n in every $q \in Q'$. Finally, $\mathcal{S}$ has bounded lookahead if, for some $n \in \mathbb{N}$, $\mathcal{S}$ has lookahead n in the whole set $Q_{[\![S]\!]}$.*

We shall find it convenient to define the lookahead of a *location* of an automaton.

Definition 9 ((smallest) lookahead in location). *An automaton $\mathcal{S}$ has lookahead n in location $l \in L$ if $\mathcal{S}$ has lookahead n in the set $\{\langle l, \nu \rangle | \nu \in \mathcal{V}\}$. $\mathcal{S}$ has* smallest *lookahead $n \in \mathbb{N}$ in a given location l if it has lookahead n in l, and does not have lookahead $n-1$ in l. We denote by $look(l, \mathcal{S}) \in \mathbb{N}$ the smallest lookahead of location l in $\mathcal{S}$ (if it exists), otherwise, $look(l, \mathcal{S}) \triangleq \infty$.*

For example, the automaton depicted in the left-hand side of Figure 1.2 has $look = 1$ in l_0, because, when e occurs, the left-hand side a-labeled transition must have been fired, but when b occurs, the right-hand side a-labeled transition has been fired.

On the other hand, the automaton depicted in the left-hand side of Figure 1.3 does not have $look = 1$ in l_0, because the occurence of b does not reveal which of the a-labeled transitions was fired. However, the following action (either c or d) reveals all the past trace, hence, $look = 2$ in l_0 for the given automaton.

Definition 10 (global lookahead of automaton). $look(\mathcal{S}) \triangleq max_{l \in L_S}\{look(l, \mathcal{S})\}$.

Clearly, a location l is deterministic in an automaton $\mathcal{S}$ iff $look(l, \mathcal{S}) = 0$; and the automaton $\mathcal{S}$ itself is deterministic iff $look(\mathcal{S}) = 0$.

The following proposition says that the lookahead of an automaton does not increase by local determinisation (all proofs can be found in the full paper).

Proposition 1 (Global lookahead does not increase). $look(\, det(\mathcal{S}, l)) \leq look(\mathcal{S})$.

The following examples show that $look(\mathcal{S})$ may or may not decrease with local determinisation. Consider the automaton on the left-hand side of Figure 1.2, which has global lookahead 1. Determinising in l_0 leaves the automaton in the right-hand side, which still has the same global lookahead! The determinisation in l_0 in Figure 1.3, however, decreases the global lookahead of the automaton from 2 to 1.

The difference between these situations is the following: in Figure 1.2, the determinisation step has merged the *nondeterministic* location l_2 into the *new* location $\langle l_0, \langle l_1, l_2 \rangle \rangle$, hence, the resulting automaton has *inherited* (in a sense that will be

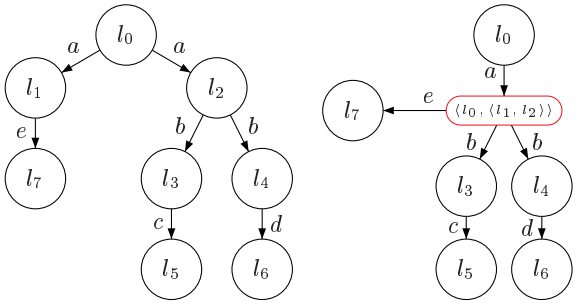

Fig. 1.2. Inherited nondeterminism may not decrease global lookahead.

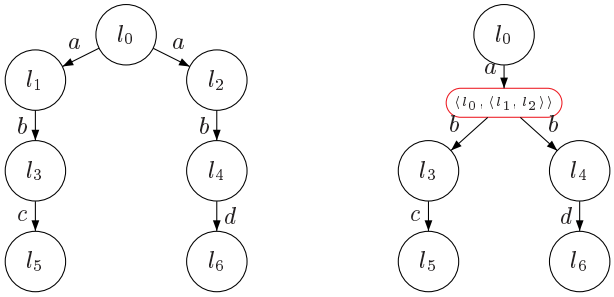

Fig. 1.3. Created nondeterminism has decreased global lookahead.

made precise below) the nondeterminism that l_2 had; because of that nondeterminism, the global lookahead has not decreased. On the other hand, the determinisation step in Fig. 1.3 does not have this problem: both l_1, l_2 are deterministic, and, even though the new location $\langle l_0, \langle l_1, l_2\rangle\rangle$ is nondeterministic, the nondeterminism is *created* by the fact that l_1, l_2 bring one b-labeled transition each.

Definition 11 (created/inherited nondeterminism). *Let $\mathcal{S}$ be an extended automaton and t_1, t_2 be two transitions of $\mathcal{S}$ involved into a nondeterminism in $\mathsf{o}_{t_1} = \mathsf{o}_{t_2} = \mathsf{o}$. Let $\langle \mathsf{o}, \langle \mathsf{d}_{t_1}, \mathsf{d}_{t_2}\rangle\rangle$ be the new location resulting from the determinisation $det_2(\mathcal{S}, t_1, t_2)$, and assume that $\langle \mathsf{o}, \langle \mathsf{d}_{t_1}, \mathsf{d}_{t_2}\rangle\rangle$ is nondeterministic in $det_2(\mathcal{S}, t_1, t_2)$. We say that this nondeterminism is* created *if both d_{t_1}, d_{t_2} are deterministic in $\mathcal{S}$, otherwise, the nondeterminism is* inherited.

Now, consider a global determinisation procedure that performs local determinisation steps in a *breadth-first* order: the first iteration determinises the nondeterministic locations of the original automaton, and each subsequent iteration determinises the new nondeterministic locations, generated during the iteration that preceded it. Figure 1.2 also illustrates the first iteration of such a *breadth-first* procedure on the automaton in the left-hand side. The resulting automaton is depicted on the right-hand side. Both automata have the same global lookahead =1. Hence, the lookahead cannot be used as a decreasing measure to ensure the termination of the procedure.

Even worse, applying local determinisations in a *depth-first* order (i.e., determinising new nondeterministic locations as soon as they are created) may not terminate, even when the automaton has bounded lookahead. An example is shown in Figure 1.4: the automaton in the left-hand side has global lookahead 1, and, by

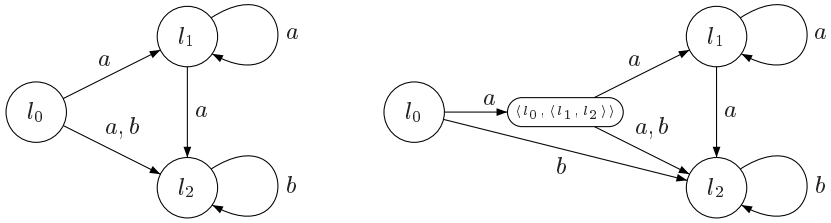

Fig. 1.4. Depth-first determinisation may not terminate.

determinising in l_0, one obtains the automaton depicted in the right-hand side of the figure, which contains a sub-automaton isomorphic the automaton in the left-hand side, with global lookahead still 1. After determinising in the newly created location, the sub-automaton is still there, and remains present all through the process of *depth-first* determinisation, which, in this case, clearly does not terminate.

Hence, applying local determinisation steps in depth-first or in breadth-first order does not lead, in general, to a terminating global determinisation procedure.

However, Proposition 2 below shows that if an iteration of a breadth-first procedure only gives rise to *created* nondeterminism, the global lookahead does decrease.

Proposition 2 (Global lookahead decreases if all new nondeterminism is created). *Let $\mathcal{S}'$ be an automaton obtained by determinising all nondeterministic locations $\{l_1, \ldots l_k\}$ of an automaton $\mathcal{S}$ in an arbitrary order, (i.e., $\mathcal{S}_0 = \mathcal{S}$, $\forall i \leq k-1, \mathcal{S}_{i+1} = det(\mathcal{S}_i, l_i)$, and $\mathcal{S}' = \mathcal{S}_k$). If none of these local determinisation steps gave rise to inherited nondeterminism, then $look(\mathcal{S}') < look(\mathcal{S})$.*

To ensure that all new nondeterminism is created, one must determinise locations whose *direct successors* are deterministic. But now we are faced with another difficulty: if the automaton has cycles in which *every location is nondeterministic*, it is impossible to choose a location on the cycle to start determinising with! This will lead us to "breaking" such cycles by determinising one location on each of them.

Definition 12. *A location l' is a direct successor of a location l in $\mathcal{S}$ if there exists $t \in \mathcal{T}_\mathcal{S}$ such that $\mathsf{o}_t = l$ and $\mathsf{d}_t = l'$. A* cycle *is a sequence $c = t_1 \cdot t_2 \cdots t_n \in \mathcal{T}^*$ such that $\forall i = 1, \ldots n-1, \mathsf{d}_{t_i} = \mathsf{o}_{t_{i+1}}$, and $\mathsf{d}_{t_n} = \mathsf{o}_{t_1}$. The cycle is* elementary *if moreover $\forall i, j = 1, \ldots n-1, i < j \Rightarrow \mathsf{d}_{t_j} \neq \mathsf{o}_{t_i}$ holds. We say $l \in c$ if $\exists i \in \{1, \ldots n\}. l = \mathsf{d}_{t_i}$, denote by $\mathcal{C}(\mathcal{S})$ the set of cycles of $\mathcal{S}$, and by $\mathcal{C}(\mathcal{S}, l) = \{c \in \mathcal{C}(\mathcal{S}) | l \in c\}$.*

Definition 13 (nondeterministic cycle). *A cycle c is* nondeterministic *if $\forall l \in c$, l is nondeterministic. We denote by $\mathcal{N}(\mathcal{S})$ the set of nondetermnistic cycles of $\mathcal{S}$.*

Lemma 1. *For $\mathcal{S}$ an automaton and all locations $l \in L_\mathcal{S}$, $\mathcal{C}(det(\mathcal{S}, l), l) \cap \mathcal{N}(det(\mathcal{S}, l)) = \emptyset$, and $\forall c' \in \mathcal{C}(\mathcal{S}). c' \notin \mathcal{C}(\mathcal{S}, l) \wedge c' \notin \mathcal{N}(\mathcal{S}) \Rightarrow c' \in \mathcal{C}(det(\mathcal{S}, l)) \setminus \mathcal{N}(det(\mathcal{S}, l))$.*

Proof. For the first statement, note that l is deterministic in $det(\mathcal{S}, l)$, hence, by definition, a cycle $c \in \mathcal{C}(det(\mathcal{S}, l), l)$ cannot be nondeterministic in $det(\mathcal{S}, l)$, i.e., it cannot be in $\mathcal{N}(det(\mathcal{S}, l))$. For the second statement, the left-hand side of the implication means that the cycle $c' \in \mathcal{C}(\mathcal{S})$ does not visit l, but visits some other location l' which is *deterministic* in $\mathcal{S}$. Determinisation in l leaves c' unchanged, thus, $c' \in \mathcal{C}(det(\mathcal{S}, l))$, and l' is still deterministic in $det(\mathcal{S}, l)$, hence, $c' \notin \mathcal{N}(det(\mathcal{S}, l))$.□

Lemma 1 says that cycles visiting l in $det(\mathcal{S}, l)$ are *not* nondeterministic, and cycles c' that do not visit l and that are *not* nondeterministic in $\mathcal{S}$ are still *not* nondeterministic cycles of $det(\mathcal{S}, l)$. *The consequences are that determinising one location per* elementary *nondeterministic cycle generates an automaton without* any *nondeterministic cycles, and determinisation does not add new nondeterministic cycles.*

We now introduce our global determinisation procedure, which starts by "breaking" all elementary nondeterministic cycles, by determinising one location on each.

```
Procedure det(S)
while C := {c ∈ N(S) | c elementary} ≠ ∅ do
  choose c ∈ C; choose l ∈ c; S := det(S, l)
endwhile
n := 0; S_n := S
while S_n is nondeterministic do
    S'_n := S_n
    while L' := {l ∈ L_{S_n} | S'_n is nondeterministic in} ≠ ∅ do
        L'' := {l' ∈ L' | S'_n is deterministic in all direct successors of l'})
        choose l ∈ L''
        S'_n := det(S'_n, l)
    endwhile
    S_n := S'_n; n := n + 1
endwhile
return S_n.
```

Fig. 1.5. Global determinisation procedure $det()$

Theorem 1 (termination, sufficient condition). *$det(\mathcal{S})$ terminates if $look(\mathcal{S})<\infty$.*

Proof. By Lemma 1 and Proposition 1, the elimination of nondeterministic cycles (first `while` loop in Figure 1.5) terminates and does not increase $look(\mathcal{S})$. Consider the sets $L'' \subseteq L'$ computed at each new iteration of the inner (third) `while` loop.

Note that $L' \neq \emptyset$ and $L'' = \emptyset$ implies that there exists a nondeterministic cycle in $\mathcal{S}_n$. Indeed, assume $l_1 \in L'$, then $L'' = \emptyset$ implies $l_1 \notin L''$, which implies that l_1 has a direct successor $l_2 \in L_{\mathcal{S}_n}$ where $\mathcal{S}'_n$ is also nondeterministic, which implies again $l_2 \in L'$. The process continues, and we eventually build a nondeterministic cycle in $\mathcal{S}_n$, which is impossible since all nondeterministic cycles were eliminated.

Inside the inner `while` loop, $L' \neq \emptyset$, and by the above reasoning, $L'' \neq \emptyset$. Hence, the `choose` l operation (from L'') inside the loop is always possible, and then determinising in location l decreases the cardinal of L' by one. Since L' is finite ($L' \subseteq L_{\mathcal{S}_n}$) and its cardinal decreases, eventually $L' = \emptyset$ and the inner `while` loop terminates. $L' = \emptyset$ also means that at the end of the inner `while` loop, $\mathcal{S}'_n$ is deterministic in all locations $L_{\mathcal{S}_n}$, hence, nondeterministic locations in $\mathcal{S}'_n$ are *new*.

For termination of the outer `while` loop, we prove $look(\mathcal{S}_{n+1}) < look(\mathcal{S}_n)$. We know that after the inner loop, the nondeterministic locations in $\mathcal{S}'_n$ are *new* (in $L_{\mathcal{S}'_n} \setminus L_{\mathcal{S}_n}$) and cannot have inherited nondeterminism, because they were generated by determinising locations in L'', whose direct successors are, by construction, deterministic. Finally, by Proposition 2, $look(\mathcal{S}'_n) < look(\mathcal{S}_n)$, and $\mathcal{S}_{n+1}$ becomes $\mathcal{S}'_n$ after n is incremented, and the proof is done. □

The fact that bounded lookahead is *necessary* for termination is based on:

Proposition 3 (*look*() decreases by at most 1). *look*($det_2(\mathcal{S}, t_1, t_2)$)$\geq$*look*($\mathcal{S}$)$-1$.

Then, a finite sequence of $det_2()$ operations cannot decrease lookahead from ∞ to 0:

Theorem 2 (necessary condition). *If* $det(\mathcal{S})$ *terminates then* $look(\mathcal{S}) < \infty$.

This concludes the study of the procedure's *termination*. It also *preserves traces:*

Theorem 3. *If* $det(\mathcal{S})$ *terminates then* $Traces(det(\mathcal{S})) = Traces(\mathcal{S})$.

All proofs can be found in the full version of the paper.

The determinisation procedure can be improved using approximate reachability analysis. Assume that an over-approximation $Reach^\alpha \supseteq Reach(\mathcal{S})$ of the reachable set of states is known (e.g., by abstract interpretation). Moreover, assume that this set is described using a formula in the same logic as the autoamaton's guards, which we have assumed to be decidable for satisfiability (cf. Section 1.2). Then, Definition 5 of a deterministic extended automaton can be weakened, by requiring that $Reach^\alpha \wedge \mathsf{G}_{t_1} \wedge \mathsf{G}_{t_2}$ be unsatisfiable (instead of $\mathsf{G}_{t_1} \wedge \mathsf{G}_{t_2}$ unsatisfiable).

This new definition of determinism increases the suclass of extended automata on which the determinisation procedure terminates. The procedure now terminates for automata satisfying a modified definition of bounded lookahead, which, intuitively, requires only states in the set $Reach^\alpha$ (instead of $Q_{[\![\mathcal{S}]\!]}$) to have bounded lookahead.

Checking for Bounded Lookahead. The bounded lookahead condition is clearly undecidable for extended automata. We now give a sufficient criterion for this condition. We need a notion of *product* of extended automata:

Definition 14 (Synchronous Product). *Two automata* $\mathcal{S}_j = \langle V_j, \Theta_j, L_j, l_j^0, \Sigma_j, \mathcal{T}_j \rangle$ $(j = 1, 2)$ *are* compatible *if* $V_1 \cap V_2 = \emptyset$ *and* $\Sigma_1 = \Sigma_2$. *The synchronous product* $\mathcal{S} = \mathcal{S}_1 || \mathcal{S}_2$ *of two compatible automata* $\mathcal{S}_1, \mathcal{S}_2$ *is the automaton* $\langle V \Theta, L, l^0, \Sigma, \mathcal{T} \rangle$ *with:* $V = V_1 \cup V_2$, $\Theta = \Theta_1 \wedge \Theta_2$, $L = L_1 \times L_2$, $l^0 = \langle l_1^0, l_2^0 \rangle$, $\Sigma = \Sigma_1 = \Sigma_2$, *and the set* $\mathcal{T}$ *of transitions of the composed system is the smallest set defined by the rule:*

$$\frac{t_1 : \langle l_1, a, G_1, A_1, q_1' \rangle \in \mathcal{T}_1 \quad t_2 : \langle l_2, a, G_2, A_2, l_2' \rangle \in \mathcal{T}_2}{t : \langle \langle l_1, l_2 \rangle, a, G_1 \wedge G_2, A_1 \cup A_2, \langle l_1', l_2' \rangle \rangle \in \mathcal{T}}$$

Then, the bounded lookahead condition for an extended automaton can be equivalently formulated as follows. Consider an extended automaton $\mathcal{S} = \langle V, \Theta, L, l^0, \Sigma, \mathcal{T} \rangle$, and let the *primed copy* $\mathcal{S}'$ of $\mathcal{S}$ be the automaton obtained by "priming" all the components of $\mathcal{S}$ except the alphabet Σ, i.e., $\mathcal{S}' = \langle V', \Theta', L', l'^0, \Sigma, \mathcal{T}' \rangle$, where $V' = \{v' | v \in V\}$, $L' = \{l' | l \in L\}$, and for states $q' = (\langle l, \nu \rangle)' = \langle l', \nu' \rangle$ where ν' is the same valuation as ν, but for variables V', i.e., $\forall x' \in V'$, $\nu'(x') \triangleq \nu(x)$.

Proposition 4 (checking for bounded lookahead). *An extended automaton* $\mathcal{S}$ *has bounded lookahead iff, for all* $q, q_1, q_2 \in Q_{[\mathcal{S}]}$ *and distinct transitions* $t_1, t_2 \in \mathcal{T}_\mathcal{S}$ *with* $\mathsf{a}_{t_1} = \mathsf{a}_{t_2}$, *if* $q \xrightarrow{t_1}_\mathcal{S} q_1 \wedge q \xrightarrow{t_2}_\mathcal{S} q_2$ *then there exists no infinite executioin in* $\mathcal{S} || \mathcal{S}'$ *starting from* (q_1, q_2'), *where* $\mathcal{S}'$ *denotes the primed copy of* $\mathcal{S}$.

The conditions of Proposition 4 are decidable if $\mathcal{S}$ is finite-state but are not decidable in general. For infinite-state extended automata $\mathcal{S}$, we can build finite-state abstractions $\mathcal{S}^\alpha$ that simulate the transition sequences σ of $\mathcal{S}$ (i.e., whenever $q \xrightarrow{\sigma} q'$ holds in $\mathcal{S}$, $\alpha(q) \xrightarrow{\alpha(\sigma)} \alpha(q')$ holds in $\mathcal{S}^\alpha$). The bounded lookahead conditions of Proposition 4 can be then automatically checked on $\mathcal{S}^\alpha$, and, if they hold, the simulation property guarantees that they also hold on $\mathcal{S}$. This gives a sufficient criterion for bounded lookahead, which is, in general, not necessary ($\mathcal{S}^\alpha$ may contain cycles not present in $\mathcal{S}$), and whose precision can be improved by taking more precise abstractions $\mathcal{S}^\alpha$.

1.5 Applications of Determinisation

Verification. A standard verification problem is that of trace (or language) inclusion: given two systems $\mathcal{I}$ (the *implementation*) and $\mathcal{S}$ (the *specification*), decide whether $\mathit{Traces}(\mathcal{I}) \subseteq \mathit{Traces}(\mathcal{S})$. When $\mathcal{I}$, $\mathcal{S}$ are extended automata and $\mathcal{S}$ is deterministic, the problem reduces to a reachability problem in the extended automaton $\mathcal{I}||\overline{\mathcal{S}}$, where $\overline{\mathcal{S}}$ is obtained from $\mathcal{S}$ by adding a new location $\mathit{fail} \notin L$, and for each $l \in L$ and $a \in \Sigma$, a new transition with origin l, destination fail, action a, identity assignments, and guard $\bigwedge_{t:\langle l,a,G_t,A_t,l'_t\rangle\in\mathcal{T}} \neg G_t$. The new transitions allow actions in $\overline{\mathcal{S}}$ whenever they are not allowed in $\mathcal{S}$. Hence, when $\mathcal{S}$ is deterministic, $\mathit{Traces}(\mathcal{I}) \subseteq \mathit{Traces}(\mathcal{S})$ iff no location in the set $\{\langle l, \mathit{fail}|l \in L_\mathcal{I}\rangle\}$ is reachable in $\mathcal{I}||\overline{\mathcal{S}}$.

When $\mathcal{S}$ is *not* deterministic, the above statement is incorrect. Let $\mathcal{S}$ be the nondeterministic automaton in the left-hand side of Figure 1.2. A naive application of the completion operation on $\mathcal{S}$ builds a transition labeled b from l_1 to fail, suggesting that $a \cdot b$ is not a trace of $\mathcal{S}$, which is obviously false. In particular, verification would wrongly declare erroneous an implementation that exhibits the trace $a \cdot b$. Hence, to be adequate for verification, $\mathcal{S}$ has to be *determinised* before being completed.

Conformance Testing is a functional testing that consists in comparing a black-box implementation $\mathcal{I}$ to a formal specification $\mathcal{S}$ according a conformance relation. The implementation is a black box, i.e., only its interface (input and output alphabet) is known. In [6] we show that conformance of an implementation $\mathcal{S}$ to a specification according to the standard **ioco** relation [8] is equivalent to the fact that running a *canonical tester* in parallel with the implementation never reaches a certain set of locations. The tester can be automatically computed from the specification using operations similar to the completion operation, defined above, and, of course, determinisation. Without determinisation, the tester might wrongly declare non-conformant an implementation that is conformant to the specification (a phenomenon similar to that exhibited by the trace $a \cdot b$, noted in the previous paragraph).

Diagnosis The determinisation problem for extended automata also has a close relationship with diagnosis for discrete event systems [7]. For instance, an extended automaton with bounded lookahead can be seen as an automaton in which nondeterministic choices are diagnosable; and checking membership to the class of bounded lookahead automata can be reduced to a diagnosability problem in this model. Also, the sufficient criterion for bounded lookahead (around Proposition 4) was inspired by the algorithm used to check diagnosability [4], based on the search of specific cycles in a product of the specification with itself.

Conversely, it could be profitable to re-define diagnosability in terms of our *bounded lookahead* condition, in order to capture a notion of diagnosability for richer, infinite-state models. Finally, the construction of a diagnoser from an automaton specifying a plant and a fault model is based on determinisation: one has to determinise the plant "decorated" with past occurrences of (unobservable) faults. Our determinisation procedure then constitutes a basic block for the construction of diagnosers from plants specified as extended automata, thus extending the works on diagnosis to more expressive, infinite-state models.

1.6 Conclusion and Future Work

In this paper we present a determinisation procedure for extended automata and prove that the procedure terminates exactly for the class of extended automata *with*

bounded lookahead. The intuition behind this class is that in any location, for any trace, there exists a bounded number of steps after which the first transition taken is uniquely identified. Technical difficulties for proving termination arise from the fact that the order in which elementary determinisation steps are applied has a strong influence on termination. The main dificulty was to find an adequate order, for which the bounded lookahead provides a decreasing measure.

The models of extended automata considered in this paper only have observable actions. One can also consider models with *internal* (unobservable) actions. In this case, determinisation first consists in an extended ϵ-*closure* generalising that of finite automata. The extended ϵ-closure algorithm is then based on the propagation of guards and actions onto the next transitons labeled by observable actions [9], and terminates iff there are no cycles of transitions labeled by internal actions.

The present work was initially motivated by *conformance testing*, more specifically, model-based testing based on the **ioco** theory [8]. In this framework, off-line test generation (computation of test cases from specifications) involves *determinising* the specification in order to compute the next possible observable actions after each trace, and, therefore, to obtain *deterministic* test cases [3]. In that work, we consider an extension of the model presented here (actions are either inputs or outputs and may carry communication parameters), which can be handled by a small modification of our determinisation procedure. The procedure also has potentially interesting application in the verification and diagnosis of infinite-state systems.

References

1. Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.
2. John E. Hopcroft, Rajeev Motwani, Rotwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computability*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.
3. B. Jeannet, T. Jéron, V. Rusu, and E. Zinovieva. Symbolic test selection based on approximate analysis. In *11th Int. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05), Edinburgh (Scottland)*, volume 3440 of *LNCS*, April 2005.
4. S. Jiang, Z. Huang, V. Chandra, and R. Kumar. A polynomial time algorithm for diagnosability of discrete event systems. *IEEE Transactions on Automatic Control*, 46(8):1318–1321, August 2001.
5. Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst.*, 1(2):245–257, 1979.
6. Vlad Rusu, Hervé Marchand, and Thierry Jéron. Automatic verification and conformance testing for validating safety properties of reactive systems. In John Fitzgerald, Andrzej Tarlecki, and Ian Hayes, editors, *Formal Methods 2005 (FM05)*, LNCS. Springer, July 2005.
7. M. Sampath, R. Sengupta, S. Lafortune, K. Sinnamohideen, and D. Teneketzis. Failure diagnosis using discrete event models. *Proceedings of the IEEE Transactions on Automatic Control*, 4(2):105–124, 1996.
8. Jan Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software - Concepts and Tools*, 17(3):103–120, 1996.
9. E. Zinovieva. *Méthodes symboliques pour la génération de tests de systèmes réactifs comportant des données,*. PhD thesis, Univ. of Rennes, Nov. 2004.