The State Explosion Problem

Antti Valmari

Tampere University of Technology, Software Systems Laboratory,
PO Box 553, FIN-33101 Tampere, FINLAND,
email: ava@cs.tut.fi

Abstract. *State space methods* are one of the most important approaches to computer-aided analysis and verification of the behaviour of concurrent systems. In their basic form, they consist of enumerating and analysing the set of the states the system can ever reach. Unfortunately, the number of states of even a relatively small system is often far greater than can be handled in a realistic computer. The goal of this article is to analyse this *state explosion problem* from several perspectives. Many advanced state space methods alleviate the problem by using a subset or an abstraction of the set of states. Unfortunately, their use tends to restrict the set of analysis or verification questions that can be answered, making it impossible to discuss the methods without some taxonomy of the questions. Therefore, the article contains a lengthy discussion on alternative ways of stating analysis and verification questions, and algorithms for answering them. After that, many advanced state space methods are briefly described. The state explosion problem is investigated also from the computational complexity point of view.

1 Introduction

There are two main approaches to checking that a concurrent system is correct with respect to a formal specification: *theorem proving* and *state space* methods.

Theorem proving is based on formulating the correctness claim as a mathematical theorem. The theorem is then proven either manually or with the help of a theorem proving tool. The proof usually uses several *invariants* and *variant functions* or *bound functions*. An invariant states a property that all states that the system can reach must have. Invariants are used for showing that the system does only correct or "acceptable" things. A variant states an upper limit to the number of times that something can happen before something else happens. Variants are used to show that a system makes progress, and eventually does the things it should do. Because verification of concurrent systems by theorem proving is a major paradigm, the amount of literature on it is huge and cannot be surveyed here. As a starting point for reading one can use the textbooks and surveys [10, 19, 29, 56, 60], for instance.

Because manual proving of theorems takes time, requires highly skilled personnel, and is prone to human errors, several more or less automatic *theorem prover* tools have been developed. Although such a tool is sometimes able to prove the correctness of an invariant or variant fully automatically, it is more

common that lots of human assistance is needed. Furthermore, the invariants and variants must almost always be provided by the user. Theorem-proving-based verification is thus very seldom fully automatic. In practice, theorem prover tools can be used only by specially trained personnel, and the construction of a proof may take lots of time, even weeks.

Another problem with theorem proving is that it is geared towards proving correctness. As a consequence, it is not particularly good in providing *debugging information*, that is, information on the nature and location of errors. Basically, an error manifests itself by causing all proof attempts to fail. The error can often be traced by locating the invariant or variant that could not be proven, and analysing where attempts to prove it fail. This is, however, an indirect and sometimes unpleasant way of debugging. A related problem is that theorem proving is clumsy in the *analysis* of system behaviour. That is, it is not good in answering questions of the type "how does the system behave" instead of "is it certain that the system behaves so-and-so". These features make theorem proving somewhat ill-suited to development work, such as experimenting with new system design ideas, and fixing an incorrect design.

The other side of the coin is that theorem proving is very generally applicable. For instance, unbounded data types (such as the ordinary integers) can often be handled without particular difficulties. To give another example, theorem proving techniques work well with systems whose process structure evolves, that is, where new processes are added and old processes aborted during an execution of the system.

State space methods aim at more automatic analysis and verification of the behaviour of systems. In their basic form, they are based on constructing a structure that consists of all states that a system can reach, and all transitions that the system can make between those states. This structure is often called the *state space*. The construction of state spaces can be fully automated. Furthermore, practical algorithms are known for answering various verification and analysis questions, given the state space of a system. No invariants or variants need be provided by the user. Therefore, at best, the "only" task that the user of a state space method has to be able to accomplish is to formulate an analysis or verification question and start a tool. As a consequence, state space methods can be used by less trained personnel than theorem proving. When state space methods apply, they are also usually fast to use.

Many state space methods are capable of producing good debugging information if the system proves incorrect. Furthermore, if a system cannot be fully investigated for some reason, state space methods are often able to give partial answers. Although an incomplete analysis cannot prove the correctness of a correct system, it can point out an error in an incorrect system. These properties make state space methods good for experimenting and as an advanced form of testing. Another advantage of state space methods is that they are flexible: a large set of analysis and verification questions of many different kinds can be answered from a single state space. This is different from, say, *place invariants* and *structure theory* [73, 17] of Petri nets, that can answer only certain kinds of

questions (although sometimes very efficiently). State space methods also support the analysis of behaviour well.

State space methods are thus free from most of the problems of theorem proving, and sound like an almost ideal behavioural analysis and verification technique. Unfortunately, they suffer from one problem that is so big and fundamental that it has led many to believe that state space methods will never work well enough for large-scale practical use: *state explosion*. The number of states of almost any system of interest is huge. To get an appreciation of how serious the state explosion problem is, it suffices to investigate the number of states of some rather trivial systems:

- The system consisting of n non-interacting processes, each with k local states, has k^n states.
- The classic dining philosophers system with n 4-state philosophers (Figure 1 left) has $3^n - 1$ states.
- A simple token ring protocol described in [38] (Figure 1 right) has $9n2^{n-2}$ states, where n is the number of stations.

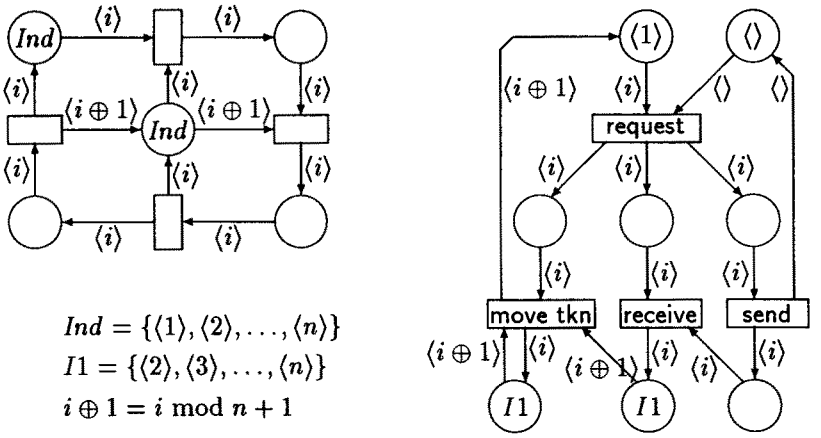


Fig. 1. The dining philosophers system (left) and the Graf-Steffen token ring (right) as high-level nets.

These examples reveal a common pattern: in a family of similar systems of different size, the number of states tends to grow exponentially in the parameter that describes the size ("n" in the above examples). More generally, intuitively speaking, the size of a state space of a system tends to grow exponentially in the number of its processes and variables, where the base of the exponentiation depends on the number of local states a process has and the number of values a

variable may store, and on some kind of "tightness" of the connection between the components of systems (that is, the extent to which the local states of components are determined by the local states of other components). In the above examples, the bases are small because the systems are simple and rather tightly coupled. In many practical analysis or verification tasks the situation is much worse.

At first sight, the state explosion problem looks so formidable that it seems to make state space methods useless for analysis and verification of systems in practice. However, the great advantages of state space methods have motivated researchers to try to find ways of alleviating the problem. During the last decade, many methods have been suggested that reduce the number of states that need to be constructed for answering certain verification or analysis questions. Such enhanced state space methods increase substantially the size of systems that can be analysed or verified, while preserving most of the advantages of state space methods. The goal of this article is to discuss many issues arising in attempts to cure the state explosion problem, and to briefly introduce several enhanced state space methods.

Unfortunately, most state space reduction techniques sacrifice one or another of the advantages of state space methods. Most often, an advanced state space method can answer only certain kinds of analysis or verification questions without losing its ability to reduce the number of states. In some cases the algorithm for constructing the reduced state space takes advantage of details of the particular verification question, and may run simultaneously and interact with the algorithm that answers the question. As a consequence, advanced state space methods cannot be properly discussed without first making a classification of different kinds of analysis and verification questions, and investigating a bit the algorithms with which they can be solved.

The next section introduces many commonly used concepts that relate to state spaces, system behaviour, and the abstractions used for extracting the latter from the former. After that, the structure of typical analysis and verification problems is examined in Section 3. Section 4 is devoted to a discussion of various kinds of analysis and verification questions, and state space algorithms for answering them. Then the state explosion problem is analysed from the complexity theory point of view in Section 5. Some strategies to state space reduction are discussed in Section 6, and many individual reduction techniques in Section 7. The conclusions are in Section 8.

2 Basic Concepts

2.1 State Spaces and Executions

To capture those aspects of state spaces that are used by the state space methods discussed later, we define a *state space* as the tuple (S, T, Δ, S_I), where

- S is a set of *states*.
- T is a set of *structural transitions*.

- Δ is a set of *semantic transitions* or *edges*. It satisfies $\Delta \subseteq S \times T \times S$.
- S_I is a set of *initial states*. It has the properties $S_I \subseteq S$ and $S_I \neq \emptyset$.

"States" are simply (a subset of) the global states of the system under analysis. Often, but not always, only those states are included that the system can reach during an execution that starts in an initial state. In the case of Petri nets, states are called *markings*, and they are usually denoted with the letter M (perhaps with extensions, like M', M_{i+1}).

"Structural transitions" are entities whose execution causes the system under analysis to change its state. They may be, for instance, atomic statements in a concurrent programming language, or Petri net transitions. They are thus present in the program or system under analysis.

"Semantic transitions", on the other hand, model actual changes of state by the system. They relate to the *behaviour* of the system, not the system itself. They can be called *occurrences* of structural transitions. Formally, a semantic transition is a triple (s, t, s') consisting of a *start state* $s \in S$, structural transition $t \in T$, and *end state* $s' \in S$. As we will see below, sometimes the middle component t is omitted or replaced by a more abstract transition label. To help reading, specific notation is often used for saying that a triple is a semantic transition, such as $M[t\rangle M'$ in the case of Petri nets, and $s \xrightarrow{a} s'$ or $s -a\rightarrow s'$ in process algebras. In other words, $s -t\rightarrow s' \overset{\text{def}}{\iff} (s, t, s') \in \Delta$.

"Initial states" are the states the system may be in when its execution starts. It is customary to define only one initial state. Petri nets, for instance, are typically defined with one initial marking, which we will denote with M_I. However, the possibility of more than one initial state is sometimes important. For instance, one might want to show that a system is correct independently of the initial value of some variable. Then one can use a set of initial states, consisting of one initial state for each possible value of the variable.

A state space with many initial states can be transformed to a state space with one initial state by adding a new state s_{new}, declaring it as the initial state, and adding the semantic transition (s_{new}, t, s) for every transition (s_I, t, s) that starts in any initial state s_I. If there is an initial state that has no incoming transitions, it can be used instead of s_{new}. It is intuitively easy to accept that this transformation does not change the "behaviour" of the state space, and it is also mathematically true for almost all precise notions of "behaviour" or "behavioural property" in this article (home state properties are an exception). Although this description was at the level of the global state space, this construction produces the desired result also when applied to each individual component of the system. Therefore, it is often implementable in a verification model in practice.

A state space is *finite* if and only if S and T are finite — then also Δ and S_I are finite. For talking about the computational complexity of state space algorithms it is handy to define the *size* of a finite state space as $|S| + |T| + |\Delta|$. In all but exceptional cases $|S| + |T| + |\Delta| = \Theta(|\Delta|)$, and the presence of $|S|$ and $|T|$ in the measure of size simplifies the treatment of the exceptional cases. The state space is *finitely branching* if and only if each state has a finite number of output edges; that is, for every $s \in S$, the set $(\{s\} \times T \times S) \cap \Delta$ is finite.

The notion of the "universe" U of "syntactically possible states" is often useful when discussing the construction of state spaces from a description of the system in some modelling formalism, such as Petri nets or a parallel programming language. In such a description, the (global) state of the system is a combination of the local states of its components, values in its variables, etc. The universe is simply the set of all such combinations. It is common that it contains many states that the system cannot reach. The set S of the states of a state space is always a subset of U.

Sequences of semantic transitions are often abbreviated, and the structural transitions that label them can be omitted:

- $s_0 -t_1\rightarrow s_1 -t_2\rightarrow \cdots -t_n\rightarrow s_n$ means that $s_0 -t_1\rightarrow s_1$, $s_1 -t_2\rightarrow s_2$, ..., and $s_{n-1} -t_n\rightarrow s_n$.
- $s_0 -t_1 t_2 \cdots t_n\rightarrow s_n$ claims that there are states $s_1, s_2, \ldots, s_{n-1}$ such that $s_0 -t_1\rightarrow s_1 -t_2\rightarrow \cdots -t_n\rightarrow s_n$. By choosing $n = 0$ we see that $s -\varepsilon\rightarrow s$ holds for every $s \in S$, where ε denotes the empty sequence.
- $s \rightarrow s'$ says that there is some $t \in T$ such that $s -t\rightarrow s'$.
- $s \rightarrow^* s'$ holds if and only if there is a (possibly empty) sequence $t_1 t_2 \cdots t_n$ of structural transitions such that $s -t_1 t_2 \cdots t_n\rightarrow s'$.

Structural transitions are not needed by all state space methods. If they are absent, then a state space is defined as the triple (S, Δ, S_I), where $\Delta \subseteq S \times S$ and $\emptyset \neq S_I \subseteq S$. A semantic transition $(s, s') \in \Delta$ may then be written as $s \rightarrow s'$, and sequences of zero or more semantic transitions as $s \rightarrow^* s'$.

A structural transition t is *enabled* in a state s, if and only if there is a state s' such that $(s, t, s') \in \Delta$. This is often written in process algebras as $s -t\rightarrow$, and in Petri nets as $M[t\rangle$. If $s -t\rightarrow s'$, then it is said that t may *occur* in s yielding s'. We say that a state is a *deadlock* if no structural transition is enabled in it.

A structural transition t is *deterministic* if and only if the state resulting from its occurrence in any given state is unique, that is, $\forall s, s_1, s_2 \in S : (s -t\rightarrow s_1 \wedge s -t\rightarrow s_2 \Rightarrow s_1 = s_2)$. If determinism is not guaranteed, then structural transitions are said to be *nondeterministic*. For example, place/transition net (that is, "ordinary" Petri net) transitions are deterministic, but transition labels in process algebras are nondeterministic.

When modelling a system at a low level of abstraction, it is usually possible to get rid of any nondeterministic structural transition t by replacing it with a set of deterministic structural transitions, which together represent exactly the possible outcomes of the occurrences of t. At higher levels of abstraction this is not always possible. For instance, assume that a communication protocol may reply ok or error to a transmission request. At a detailed level of abstraction, the choice between ok and error is determined by a loss or no loss of a message in a channel. If we look at the *service* provided by the protocol (that is, how its users see it), the transitions corresponding to the operation of the channel are not shown, so the choice between ok and error looks nondeterministic. Because many advanced state space methods rely on these kinds of abstractions, we cannot avoid nondeterministic transitions in the sequel.

An *execution* of a system is a finite or infinite sequence $\langle s_0, t_1, s_1, t_2, \ldots, t_n, s_n \rangle$ or $\langle s_0, t_1, s_1, t_2, \ldots \rangle$ such that $s_0 \in S_I$ and $s_0 -t_1\rightarrow s_1 -t_2\rightarrow \cdots -t_n\rightarrow s_n$ (or $s_0 -t_1\rightarrow s_1 -t_2\rightarrow \cdots$). A finite execution may be *incomplete* in the sense that it may end in a state with enabled structural transitions. Thus every prefix of an execution that ends with a state (instead of a transition) is an execution. In particular, a single state s is an execution if and only if $s \in S_I$. An execution is *deadlocking*, if and only if it is finite and its last state is a deadlock state. An execution is *complete*, if and only if it is infinite or deadlocking. The set of executions of a system is the same as the set of all prefixes of the complete executions of the system. We denote the set of complete executions of the system in question by CEx.

We can talk about executions that *start at some given state* s. The definitions are otherwise the same as above, but the requirement $s_0 \in S_I$ is replaced by $s_0 = s$.

A state s' is *reachable from the state* s, if and only if it is the last state of some finite execution starting at s, that is, $s \rightarrow^* s'$. The set of Petri net markings reachable from the marking M is denoted by $[M\rangle$. A semantic transition is reachable from s, if and only if its start state is — then also its end state is. A state or semantic transition is *reachable*, if and only if it is reachable from some initial state. By the *reachable part* of a state space (S, T, Δ, S_I) we mean the tuple (S', T, Δ', S_I), where S' is the set of reachable states and Δ' is the set of reachable semantic transitions. Most state space tools construct only the reachable part of the state space,[1] and it is intuitively clear that the "behaviour" of a system depends only on it. Indeed, all formal definitions of behavioural properties in this article take only the reachable part into account.

A state space represents an *interleaving* semantics of a system. That is, it does not model the possibility that two or more structural transitions occur simultaneously. Semantic structures where this possibility is modelled are often called *true concurrency* models. An obvious way of extending state spaces to a true concurrency model would be to add semantic transitions labelled by nonempty sets of simultaneously occurring structural transitions. The corresponding semantics is often called *step semantics*.

Opinions differ as to whether interleaving or true concurrency semantics should be used in verification. If the property in question is inherently truly concurrent, then, of course, interleaving models are not appropriate. On the other hand, deadlocks, livelocks, formulae in typical temporal logics — perhaps most of the properties that people want to analyse or verify — are insensitive to the difference between interleaving and truly concurrent models.[2]

[1] Not all, though. Symbolic model checking with BDDs is one exception.

[2] This is one of the statements that loses its validity in weird enough situations. Consider a parallel composition of infinitely many non-interacting processes that make one step each and then terminate. In a truly concurrent model the system as a whole can terminate, but not in the interleaving model, because in it only finitely many transitions can be executed in a finite time.

Truly concurrent models are sometimes suggested as a way of avoiding state explosion. Step semantics does not seem promising in this respect, because it does not affect the number of states and increases the number of semantic transitions. However, it is not the only possible truly concurrent semantics of systems. Indeed, we will see in Section 7.2 that states can be reduced with another truly concurrent semantic model, namely the *unfoldings* of Petri nets. Interestingly, concurrency can also be taken advantage of when constructing a reduced interleaving state space, as will be explained in Section 7.4.

2.2 Abstractions of States and Transitions

It is often reasonable to define the correctness of a system at a higher level of abstraction than the system itself. (As a matter of fact, this is what the famous software engineering principle of keeping "what" and "how" apart from each other recommends.) Instead of formulating correctness claims and analysis questions in terms of the details of individual states and structural transitions, more abstract notions may be defined that capture those properties of states and structural transitions that are relevant for the user of the system. For instance, it is more pleasant to require "processes 1 and 2 should not be in their critical sections simultaneously" than "processes 1 and 2 should not be in line 7 simultaneously".

At first sight this kind of abstraction might seem a minor user-friendliness issue, but it is actually crucial for many advanced state space methods. This is because they are based on throwing away information on those aspects of system behaviour that are not relevant for the specification or analysis questions. Without abstraction, an analysis question can potentially refer to just anything, leaving the state space method little or no room for obtaining reduction. In the presence of the abstraction, one can stipulate that the question may refer to the system *only* with the abstract concepts, and the state space method can take advantage of this fact. (This is an example of "information hiding", another famous principle in software engineering.)

For the purpose of discussing typical abstraction mechanisms, we employ two sets of "observables", Π and Σ, together with a special symbol "τ". The intention is that the properties of the system may be referred to only with these observables.

- Π is a set of (atomic) *propositions*. Propositions refer to the properties of individual states. More formally, a proposition φ is a function from S to the set $\{\mathsf{False}, \mathsf{True}\}$ of truth values. Examples: $\mathsf{Critical2} \stackrel{\text{def}}{\Longleftrightarrow}$ "Process 2 is in its critical section", $\mathsf{AB\_empty} \stackrel{\text{def}}{\Longleftrightarrow}$ "There are no messages in the channel AB", $\mathsf{p1\_is\_good} \stackrel{\text{def}}{\Longleftrightarrow} M(p_1) \geq M(p_2)$.
- Σ is a set of *observable transition labels*, also called *observable* or *visible actions*. The set Σ is often called the *alphabet*. Transition labels are, in essence, user-oriented names of individual (structural or semantic) transitions or groups of them. Examples: "press_start_button", "send_message⟨12345⟩".

τ is a special *unobservable* or *invisible action.* It is used to label those transitions that the specification should not talk about, because they are intended to model implementation details and be internal to the system. The formula $\tau \notin \Sigma$ is assumed to hold.

The values of the observables on a given state space (S, T, Δ, S_I) are determined by the following two evaluation functions.

$\mathcal{E}_\Pi : S \mapsto 2^\Pi$ assigns to each $s \in S$ the set $\mathcal{E}_\Pi(s) \subseteq \Pi$ of propositions that hold in s. That is, $\varphi(s) = \mathsf{True}$ (i.e. φ holds in s, often written as $s \models \varphi$) if and only if $\varphi \in \mathcal{E}_\Pi(s)$.

$\mathcal{E}_\Sigma : T \mapsto \Sigma \cup \{\tau\}$ gives new names to structural transitions. Several different structural transitions may have the same name. The structural transitions whose occurrences are intended to be unobservable are given the name τ, that is, $\mathcal{E}_\Sigma(t) = \tau$.

One might ask for the reason of having both structural transitions T, and labels of transitions $\Sigma \cup \{\tau\}$. The two play a different role. Structural transitions come from the formalism used for describing the system — they tell something about how the system has been put together. Transition labels, on the other hand, reflect the meaning of a semantic transition at some higher level of abstraction, or from the point of view of the user of the system. It is common that an operation is modelled with more than one structural transition, especially if the system description formalism is not rich.

For instance, the assignment of zero tokens to a place/transition net place with capacity n in one transition occurrence requires $n+1$ alternative transitions, one for each possible number of tokens before the assignment. This is because a place/transition net transition can change the number of tokens only by a constant value, and the operation requires $n+1$ different changes to be possible. However, all the alternative transitions model the same operation, so it is good if they can be given a common name such as "**reset_x**".

Another reason for having the two levels is that some advanced verification methods can take advantage of the low-level information provided by structural transitions, while answering verification questions stated in terms of transition labels. The CSP-preserving stubborn set method in Section 7.4 is an example. It uses Σ to determine whether the swapping of successive transition occurrences affects the property that is being verified, and T to find out whether both orderings will lead to the same future states.

If transition labels are needed but structural transitions are not, then it is customary to replace T in the definition of state spaces by Σ or $\Sigma \cup \{\tau\}$, yielding (S, Σ, Δ, S_I). Then $\Delta \subseteq S \times (\Sigma \cup \{\tau\}) \times S$, and $\mathcal{E}_\Sigma$ becomes the identity function and is discarded. This is the usual case with process algebras, for instance.

The functions $\mathcal{E}_\Pi$ and $\mathcal{E}_\Sigma$ can be extended to executions in a natural way:

- $\mathcal{E}_\Pi\big(\langle s_0, t_1, s_1, t_2, \ldots, t_n, s_n\rangle\big) = \big\langle \mathcal{E}_\Pi(s_0), \mathcal{E}_\Pi(s_1), \ldots, \mathcal{E}_\Pi(s_n)\big\rangle$,
- $\mathcal{E}_\Sigma\big(\langle s_0, t_1, s_1, t_2, \ldots, t_n, s_n\rangle\big) = \big\langle \mathcal{E}_\Sigma(t_1), \mathcal{E}_\Sigma(t_2), \ldots, \mathcal{E}_\Sigma(t_n)\big\rangle$, and

- $\mathcal{E}_{\Pi+\Sigma}(\langle s_0, t_1, s_1, t_2, \ldots, t_n, s_n\rangle) =$
  $\langle \mathcal{E}_\Pi(s_0), \mathcal{E}_\Sigma(t_1), \mathcal{E}_\Pi(s_1), \mathcal{E}_\Sigma(t_2), \ldots, \mathcal{E}_\Sigma(t_n), \mathcal{E}_\Pi(s_n)\rangle$.

Stuttering. An important issue arising with $\mathcal{E}$-abstracted executions is that of *stuttering*. Stuttering means that a semantic transition in an execution has no observable effect. Let $\xi = \langle s_0, t_1, s_1, t_2, \ldots, t_n, s_n\rangle$ be a finite or infinite execution. When Π is used stuttering means that $\mathcal{E}_\Pi(s_{i+1}) = \mathcal{E}_\Pi(s_i)$ for some i, and with Σ that $\mathcal{E}_\Sigma(t_i) = \tau$ for some i.

The exact number of transitions a concurrent system takes to accomplish some task is usually considered irrelevant in verification. For instance, we do not usually care whether it took 5 or 10 internal transitions for a protocol to deliver a message. Therefore, executions that differ only in the amount of stuttering are usually considered equivalent in verification, with the exception that infinite stuttering (that is, $\mathcal{E}_\Pi(s_j) = \mathcal{E}_\Pi(s_i)$ or $\mathcal{E}_\Sigma(t_j) = \tau$ for every $j \geq i$) is commonly distinguished from finite stuttering.

A property is *stuttering-insensitive* if and only if its truth value never changes when finite stuttering is added to or removed from a system. Most properties we want to verify are stuttering-insensitive. When verifying such properties, if $\mathcal{E}_\Pi(\xi) = \langle P_0, P_1, \ldots, P_n\rangle$ is finite, we could remove all P_i that satisfy $P_i = P_{i-1}$ without modifying verification results. From an infinite $\langle P_0, P_1, \ldots\rangle$ we can remove all P_i such that $P_i = P_{i-1}$ and there is $j > i$ such that $P_i \neq P_j$. Similarly, we could remove all τ-symbols from $\mathcal{E}_\Sigma(\xi)$, as long as we do not remove any infinite suffix consisting only of τs.

State-based and action-based formalisms. In many formalisms, one can refer to the properties of a system only with the elements of Π. Because Σ is not used, structural transitions cannot be referred to, and semantic transitions can be referred to only indirectly, as changes of state. Such formalisms can be called *state-based*. Most (but not all) temporal logics are in this category. In *action-based* formalisms Π is not used but Σ is. Most, if not all, process algebras are action-based. So are also some temporal logics, most notably those that have been intended to be used in connection with process algebras.

It would be possible to use both Π and Σ in the same abstraction formalism, but such formalisms seem to be rare. One reason for this is that in many cases, information about states is redundant if information about actions (i.e. transition labels) is available, and vice versa. Let $s \xrightarrow{t} s'$. State information can be encoded into actions by replacing $\mathcal{E}_\Sigma(t)$ with the pair $(\mathcal{E}_\Sigma(t), \mathcal{E}_\Pi(s'))$, and storing $\mathcal{E}_\Pi(s_I)$ separately for each initial state s_I. This approach has, unfortunately, a problem: it is not clear which pairs could take the role of the invisible action, because the set $\mathcal{E}_\Pi(s')$ may be important in the pair $(\tau, \mathcal{E}_\Pi(s'))$. Furthermore, this idea is difficult to implement at the level of the modelling formalism (such as Petri nets), because the global state is needed for computing $\mathcal{E}_\Pi(s')$.

These problems can be solved by using the triples $(\mathcal{E}_\Sigma(t), P_{\mathsf{on}}, P_{\mathsf{off}})$ where $P_{\mathsf{on}} = \mathcal{E}_\Pi(s') - \mathcal{E}_\Pi(s)$ and $P_{\mathsf{off}} = \mathcal{E}_\Pi(s) - \mathcal{E}_\Pi(s')$ as the new transition labels. The idea is that P_{on} and P_{off} describe the *change* in the $\mathcal{E}_\Pi$-abstracted state

caused by the occurrence of t, so the triple $(\tau, \emptyset, \emptyset)$ can be used as the invisible action. Furthermore, the information that P_{on} and P_{off} contain is local in the sense that it can be computed based on knowledge of the neighbourhood of t. It is worth noticing that if originally all transitions are labelled with τ, then this mapping transforms state stuttering (i.e. repetition of the same $\mathcal{E}_\Pi$-abstracted state) into action stuttering (i.e. occurrence of the invisible action), and vice versa.

Similarly, action information can be encoded into states by adding the label of the most recently executed transition into each state. This may divide the state to several copies, one for each possible input transition label, but this is usually pretty harmless. If insensitivity to stuttering is desired, then one may store the most recent *observable* transition label together with a bit that alternates its value each time a transition with an observable label occurs. The purpose of the bit is to make it possible to distinguish repeated occurrences of the same visible action. Repetition of the same visible action is not stuttering and should not be ignored in verification. For instance, if a protocol may execute, in a row, two receive$\langle msg \rangle$-actions with the same message *msg*, then we would probably want to know it, because it may be a sign of erroneous duplication of a message. Also this mapping makes state-based and action-based stuttering match each other.

The above encodings of state information to transitions and transition information to states are important, because they allow the use of state-based methods for action-based verification tasks and vice versa. For instance, the stubborn set methods (Section 7.4) are inherently action-based because they rely on analysing relations between structural transitions, but thanks to the above mapping they can be applied to state-based linear temporal logic.

2.3 Linear and Branching Time

Let us assume that we have decided to use an abstraction mechanism consisting of Π and $\mathcal{E}_\Pi$, Σ and $\mathcal{E}_\Sigma$, or both. The mechanism specifies what we can say about the properties of *individual* states and transitions. Another important question is: What do we want to be able to say about their *relations over time*?

Linear time. One popular answer is that it suffices to look separately at each complete execution of the system or, more precisely, what can be seen of it through the abstraction mechanism. In other words, "$\xi \models \varphi$" (what it means for an execution ξ to satisfy a property φ) is defined as $\mathcal{E}_\Pi(\xi) \models \varphi$ or $\mathcal{E}_\Sigma(\xi) \models \varphi$ or $\mathcal{E}_{\Pi+\Sigma}(\xi) \models \varphi$, and the system has the property φ if and only if $\xi \models \varphi$ for every $\xi \in CEx$. Any property whose validity is defined in this way is a *linear-time* property.

For instance, as we will soon demonstrate, reachability of a deadlock, and 4-boundedness of a Petri net place p (that is, $\forall M \in [M_I\rangle : M(p) \leq 4$) are linear-time properties. So is the property that at any instant of time, the sequence of messages that have been output from an alleged fifo queue is a prefix of the input sequence; and the property that the length of the output sequence will eventually be at least the same as the present length of the input sequence.

Reachability of a deadlock can be checked (in the mathematical, not necessarily computational, sense) by including the proposition is_deadlock $\stackrel{\text{def}}{\Longleftrightarrow} \forall t \in T : \neg M[t\rangle$ in Π, going through all abstracted states in the sequences in $\mathcal{E}_\Pi(CEx) = \{ \mathcal{E}_\Pi(\xi) \mid \xi \in CEx \}$, and checking whether is_deadlock holds in any of them. The same principle applies to 4-boundedness. To analyse the properties of the fifo, one may include in Σ the actions input$\langle x \rangle$ and output$\langle x \rangle$ for every possible message type x. Then the "prefix" property can be checked by searching $\mathcal{E}_\Sigma(CEx) = \{ \mathcal{E}_\Sigma(\xi) \mid \xi \in CEx \}$ for a prefix of an abstracted execution that violates the property. Finally, the "length" property is violated if and only if $\mathcal{E}_\Sigma(CEx)$ contains an abstracted complete execution with a finite number of output messages that is smaller than the number of input messages (the latter may be infinite).

As an example of a property that, making reasonable assumptions about the abstraction (we will return to this at the end of this section), is *not* linear-time we may take "liveness" of structural transitions in the Petri net sense of the word. A structural transition $t \in T$ is *Petri-net-live*, if and only if $\forall M \in [M_I\rangle : \exists M' \in [M\rangle : M'[t\rangle$ (or, in the CTL logic that will be introduced in Section 4.3, AG EF "t is enabled").

Figure 2 shows a Petri net and its state space abstracted such that $\Pi = \{t\_\mathsf{enab}\}$, where t_enab holds in exactly those states where t is enabled. The transition t is not Petri-net-live, but if the dashed transition is removed from the net, then t becomes Petri-net-live. The set $\mathcal{E}_\Pi(CEx)$ of complete abstracted executions of the net consists of the sequences $\langle P_0, P_1, P_2, \ldots \rangle$, where $P_i = \emptyset$ whenever $i = 0$ or i is odd, and each of the remaining P_i is either $\{t\_\mathsf{enab}\}$ or $\emptyset$. The same set is obtained if the dashed transition is removed. So we see that the removal of the dashed transition does not affect the linear-time properties of the net with respect to the abstraction Π, although it affects the Petri-net-liveness of t. Therefore, Petri-net-liveness of t cannot be reasoned from $\mathcal{E}_\Pi(CEx)$, and Petri-net-liveness is not a linear-time property.

It is worth noticing that the fact that Π refers to the marking of only one place (namely the input place of t) was not important in the above example. The example remains valid for every Π that contains t_enab and does not refer to $M(p_1)$.

Branching time. On the other hand, Petri-net-liveness can be determined if $t\_\mathsf{enab} \in \Pi$ and all *execution trees* of the system are known. Just like the definition of an execution, the definition of an execution tree requires that one initial state $s_I \in S_I$ is chosen as the starting point. Consider the part of the state space that consists of the states and semantic transitions that are reachable from s_I. An execution tree is an unfolding of that part into a tree. It represents all executions that start at s_I, and it also records all the positions where any two executions separate from each other.

An execution tree of the state space (S, T, Δ, S_I) with the start state $s_I \in S_I$ may be defined formally as the rooted edge-labelled graph (V, E, s_I) such that

- V is the set of finite executions $\langle s_0, t_1, \ldots, t_n, s_n \rangle$ of the system where $s_0 = s_I$

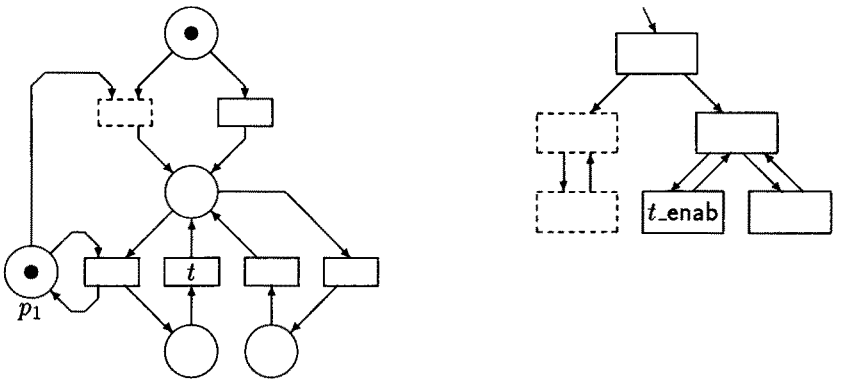


Fig. 2. A Petri-net-liveness example.

– $E = \{ (\langle s_I, \ldots, s \rangle, t, \langle s_I, \ldots, s, t, s' \rangle) \mid \langle s_I, \ldots, s \rangle \in V \wedge (s, t, s') \in \Delta \}$

Although the nodes of the execution tree are defined above as executions and thus each node contains complete information of its history, one should consider nodes as anonymous and void of any information other than that provided by $\mathcal{E}_\Pi$. Unifying nodes with their histories is just a mathematical trick that ensures that the result is a tree. The edges of an execution tree may be defined also as

$E = \{ (\langle s_I, \ldots, s \rangle, \mathcal{E}_\Sigma(t), \langle s_I, \ldots, s, t, s' \rangle) \mid \langle s_I, \ldots, s \rangle \in V \wedge (s, t, s') \in \Delta \}$
if structural transitions are abstracted with Σ, or
$E = \{ (\langle s_I, \ldots, s \rangle, \langle s_I, \ldots, s, t, s' \rangle) \mid \langle s_I, \ldots, s \rangle \in V \wedge \exists t \in T : (s, t, s') \in \Delta \}$
if structural transitions are abstracted away totally.

Any property whose validity is defined on $\mathcal{E}$-abstracted execution trees is a *branching-time* property. We may call a branching-time property *proper* if it is not also a linear-time property. Thus Petri-net-liveness is a proper branching-time property.

Even the set of branching-time properties does not cover all properties of interest. Consider the net in Figure 3, and assume that $\Pi = \{\mathsf{p1}, \mathsf{p2}, \mathsf{p3}, \mathsf{p4}\}$, where $\mathsf{p}i \stackrel{\text{def}}{\Longleftrightarrow} M(p_i) = 1$. A marking M_H is a *home marking* if and only if it is reachable from all reachable markings, that is, $\forall M \in [M_I\rangle : M_H \in [M\rangle$, or AG EF "the marking is M_H". The example net does not have home markings. However, if p_5 and its adjacent arc are removed, then $M(p_4) = 1 \wedge M(p_1) = M(p_2) = M(p_3) = 0$ becomes a home marking.

On the other hand, because we assumed that $\mathsf{p5} \notin \Pi$, the $\mathcal{E}_{\Pi+\Sigma}$-abstracted execution trees of the net with and without p_5 are the same. As a consequence, under this abstraction the property "M_H is a home marking" is not a branching-time property. That it was possible to describe it with the above CTL formula is because the formula implicitly refers to p_5 in "the marking is M_H".

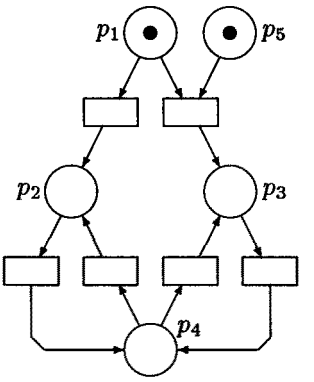


Fig. 3. A home marking example.

In the above examples we assumed that an abstraction mechanism was used that prevented us from reading the marking of one place of the net. This was necessary, because if no abstraction mechanism is used, then the distinction between linear-time, branching-time and even more general state space properties disappears in a certain theoretical sense. Namely, then we would have access to the properties of the states and transitions in full detail, and could reconstruct the reachable part of the state space from the set of complete executions by merging all occurrences of the same state. As a consequence, any state space property that does not depend on unreachable states could be checked from the set of complete executions. In the presence of an abstraction mechanism this trick does not work, because then we cannot be sure that s_1 and s_2 are the same state even if we know that $\mathcal{E}_\Pi(s_1) = \mathcal{E}_\Pi(s_2)$.

2.4 Safety, Liveness, and Fairness

In the verification of sequential programs the distinction between *partial correctness* and *termination* has proven useful. Partial correctness means that the program does not ever do anything illegal such as deliver incorrect results or crash. Partial correctness requirements depend on the task of the program, and are often formulated in some programming logic. The requirement of termination is the same for all programs, and it is simple to state: the program should not run forever. Partial correctness is proven with *weakest preconditions*, *loop invariants*, etc., while proofs of termination are usually based on *bound* or *variant functions* whose values decrease as the program continues execution but cannot decrease without limit. Thus the techniques used for stating and proving termination are very different from the techniques used for stating and proving partial correctness.

A similar distinction has been made in the world of concurrent systems between *safety* and *liveness* (or *progress*) properties [55]. Safety plays the role of

partial correctness and liveness that of termination. Since concurrent programs are usually not intended to terminate, liveness consists of a set of requirements of the kind "these things should eventually happen". Unlike termination, liveness properties are system-specific and often difficult to formulate.

Linear-time safety properties can be defined as those properties of $\mathcal{E}$-abstracted executions that have *finite* counterexamples. If a prefix of an execution matches any one of the counterexamples, then the execution violates the property, no matter how it continues. A system has a safety property, if and only if all its executions have it. The "prefix" property of a fifo queue and 4-boundedness of a Petri net place (Section 2.3) are safety properties. Namely, to demonstrate that either one of them does not hold, it suffices to give an example of a finite, perhaps incomplete execution in which a message is output that has not been input, or the place contains at some point more than four tokens. Also the property "the program will not terminate" is a safety property, if we assume that $\mathcal{E}_\Pi$ contains the proposition **stopped** $\stackrel{\text{def.}}{\Longleftrightarrow} \forall t \in T : \neg(s\ -t\rightarrow\ )$.

The precise meanings of the terms "liveness" and "progress" vary. In this article we roughly follow [2, 59] and define linear-time liveness properties as the properties such that $\mathcal{E}$-abstractions of only *complete* executions qualify as counterexamples. Furthermore, if the complete execution is finite and its $\mathcal{E}$-abstraction is $\langle P_0, a_1, P_1, a_2, \ldots, a_n, P_n\rangle$, then also its infinite completion should be a counterexample, where the completion is $\langle P_0, a_1, P_1, a_2, \ldots, a_n, P_n, \tau, P_n, \tau, P_n, \ldots\rangle$ (that is, the last abstracted state is repeated forever with invisible transitions in between). This extra condition ensures that no property is simultaneously a safety and liveness property.

For instance, the property "the program will eventually terminate" is a liveness property, because to demonstrate that it does not hold, an infinite, and thus complete, execution is needed. If we have observed the execution of a system only a finite time and the system has not terminated, we cannot claim that the execution violates any liveness property, because it is possible that the execution continues in a way that makes the property valid. The "length" property of fifo queues is a liveness property. This is because if n messages have been input to the queue and less than n have been output, it is possible that the remaining messages will come out if we wait long enough.

The definition of safety can be naturally applied to branching time by replacing "execution trees" for "executions". Because an execution tree may be simultaneously infinite and incomplete, it is not immediately clear how branching-time liveness should be defined. For instance, a counterexample for the Petri-net-liveness of a transition t has to contain a complete branch where t is never enabled, but other branches of the tree may be discarded.

If a linear-time safety property does not hold, then there is a finite execution that works as a counterexample. A state space tool can print that execution, and the user can use it to trace the reason of the error. If a linear-time liveness property does not hold, then counterexamples are usually infinite. An infinite execution cannot be printed in full, but almost always a counterexample can be made to consist of a finite prefix followed by an infinitely repeating finite

cycle, and the prefix and the cycle can be printed. Counterexamples for proper branching-time properties are more complicated. Let us again use Petri-net-liveness as an example. It is easy to print an execution that leads to a state after which t cannot any more become enabled, but it is not clear what the tool should print to show that t indeed stays disabled from that state on.

Safety properties are easier to handle in verification than liveness properties in at least two ways. First, there are verification approaches and algorithms that work for safety properties, but not for liveness properties. We will see examples of this in Section 7. Second, validity of a liveness property tends to depend on subtle assumptions about the scheduling policy of the system and liveness properties of its components, while safety properties are totally insensitive to them. For instance, a mutual exclusion algorithm cannot guarantee that each customer that has requested for the shared resource will eventually get it, unless it is assumed that each customer who gets the resource will eventually release it. (The usefulness of Petri-net-liveness in verification is increased by the fact that although it fulfills to some extent similar verification needs as liveness properties, it does not depend on fairness assumptions.)

Fairness is a particular type of assumption that is very often needed for ensuring liveness. Numerous different notions of fairness have been defined (see the book [28], for instance). Perhaps the following two from [59] are the most well known. *Weak fairness* (also known as *justice*) towards a structural transition t promises that if t is enabled in every state from some point on, then it will eventually occur. The definition rules out executions where, after some state, t never occurs although it is always ready to occur. Thus weak fairness is suitable for modelling assumptions such that each process of a system gets processor time.

Strong fairness (or *compassion*) requires that if t is enabled infinitely many times, then it should also occur infinitely many times. With strong fairness one can specify, for instance, that a server does not systematically disfavour any one of its clients.

3 Structure of the Analysis or Verification Problem

State space methods are good both in the *verification*, *analysis*, *validation* and *error detection* of the behaviour of systems. In this article, these terms have the following meanings.

Verification is the act of proving or checking that a formal system has a formally stated property. In the strictest interpretation of the word, the goal is just to find rigorous evidence to the claim that the system is correct in the sense of having the property. If the system is not correct, then it suffices that a verification technique fails either by not terminating or by giving a failure report that leaves the correctness question unanswered. This kind of a verification technique may be called *one-sided* — it can answer "yes, the system is correct" and perhaps also "cannot say", but it cannot answer "no, it is incorrect".

In a more permissive interpretation, a verification technique may also give the answer "incorrect" and, better still, give some debugging information, such as an example of an execution that violates the specification. A verification technique that, given enough time and memory, is guaranteed to eventually terminate with the answer "yes" or "no" is a *verification algorithm*. A *one-sided verification algorithm* will eventually terminate and answer "yes" or "cannot say".

Analysis means finding answers to formal questions about the behaviour of a system. Analysis differs from verification in that

- the question is not necessarily a "yes"/"no" question, and
- even if it is, the answer "yes" is not given priority over "no".

An *analysis algorithm* is guaranteed to eventually terminate with a correct answer (other than "cannot say"), unless it runs out of computer resources. Examples of analysis questions: "What is the maximum number of messages simultaneously in this queue?", "Give me a list of those Petri net places that have more than one token in some reachable marking."

Validation is the process of obtaining confidence that a system behaves as intended. The behaviour of the system is compared to the expectations of the human being who is validating the system. Validation is thus always inherently informal.

Verification does not guarantee that a system behaves as we want, because we might have made an error when formulating the correctness criteria (not to mention the problem that the object of verification is never the real system, but only a formal model of it, that may or may not represent the system accurately enough). Formal analysis or verification may, however, be used as a means of validation. Knowing that a system does not deadlock and that its input and output are related in a certain way may increase our reliance on it quite a lot. Some other, widely used forms of validation are ordinary testing and reviews of program code.

Error detection is, in a sense, opposite to verification and validation: its purpose is to find errors. In most cases the goal of making a flawless system is beyond reach. Therefore, the less ambitious goal of finding as many of the remaining errors as possible within reasonable time and costs is taken. Ordinary testing is widely used for this purpose. Analysis and verification algorithms are also useful, because they are powerful in detecting errors, and they tend to find different errors than testing.

With this attitude, the failure of one individual analysis or verification run because of state explosion is a pity, but not a catastrophe — the search for errors may be continued with another, perhaps simpler analysis question. The verification system is successful if it reveals errors, even if it falls far short of full coverage of the state space. Furthermore, the capability of finding errors is more important than the ability of proving correctness when trying to convince engineers of the benefits of verification algorithms and tools. An engineer does not necessarily know what verification means and is perhaps not much impressed when told "I verified your design and it is correct", but

pointing out an error that the engineer was not aware of gives the engineer concrete evidence of the power and usefulness of state space techniques.

To some extent, the difference between verification and analysis is a matter of taste, or point of view. For instance, a verification algorithm (in the above sense) is also an analysis algorithm, and the production of debugging information can be thought of as analysis. One practical difference in the use of these terms is that in analysis, one does most of the thinking *after* running the tool, while in verification most of the thinking is done *before* running the tool. Formulating a reasonable correctness claim for verification takes a lot of effort, whereas the answer can be interpreted easily, especially if it happens to be "yes". In analysis, on the other hand, a question may be thrown in just to see what happens, and interpreting the answer may take quite some time.

A typical framework for computer-aided analysis or verification has the following four components:

1. **A formalism for modelling the system.** It may be, for instance, a parallel programming language, some class of Petri nets, a process-algebraic language, or a formalism consisting of finite-state automata communicating via first-in-first-out queues. In the sequel we will call it *modelling formalism.*
2. **A formalism for stating analysis questions or properties for verification.** We will call the former *query formalism* and the latter *specification formalism.* Examples are state space query languages and temporal logics. A specification or query formalism may also contain aspects that are related to the modelling formalism, such as, in the case of Petri nets, fact transitions[3] and lists of places that are expected to be bounded. A query formalism may be rudimentary, and may reside in part or in full in the names and options of the commands used for invoking analysis tools. A document in a specification formalism is a *specification.*
3. **A formal meaning for the relation "the system has the properties".** The system is given in a modelling formalism, and "the properties" mean either the specification, or the results of an analysis. This relation is known as *satisfies* in the remainder of this article. In the case of temporal logics the satisfies-relation is usually the "is-a-model-of"-relation denoted by "$\models$". With process algebras it is commonly some process equivalence or preorder. A Petri net satisfies a fact transition if and only if the fact transition is disabled in all reachable markings.
4. **An algorithm for checking whether a given system satisfies a given specification.** Even if we restrict ourselves to state-space-based algorithms, the algorithm and its computational complexity depend heavily on the specification formalism. Sections 6 and 7 are devoted to a discussion of various ways of organising a verification algorithm, and of techniques that can be used to avoid or alleviate the effects of state explosion.

[3] In Petri net terminology, a *fact transition* is a structural transition that is never expected to be enabled. Thus the negation of its enabling condition is expected to always hold, to be a persistent fact.

Some major specification formalisms and the corresponding satisfies-relations and algorithms will be discussed in Section 4.

4 Specification and Query Formalisms

4.1 Statistics at the Level of Modelling Formalism

Many state space tools can produce various statistics on the state space that are formulated in the terminology of the modelling formalism. A Petri net tool, for instance, may generate lists of deadlock markings, maximum numbers of tokens in each place, transitions that never occur, transitions that are not Petri-net-live, etc. To get the lists, the user needs not do more than tell the tool to print them. On the other hand, after the lists have been printed, the user has to think a lot to check whether the results are acceptable and the system is working correctly. So this approach is more oriented towards analysis than verification.

Because a typical state space contains vast amounts of information, lists of the above kind may be rather long. Therefore, and also to facilitate more detailed investigation of the state space, some state space tools allow the user to make various queries on the state space. For instance, the user may ask for a list of markings where a certain transition is enabled. Advanced state space query languages allow the user to talk about the properties of a marking in great detail, investigate the immediate predecessor and successor markings of a given marking, pose questions about the reachability relation, etc.

State space statistics can usually be produced and queries of this kind answered efficiently with rather simple algorithms. The properties that can be checked with a typical state space query language concentrate on linear-time safety properties. Home markings, questions about the reachability relation, etc. allow the analysis of some other types of properties, but these facilities tend to be somewhat ad-hoc and clumsy. This observation has an explanation: we will see in Section 4.3 that it is very unlikely that simple, efficient verification algorithms exist for a certain powerful specification formalism that is capable of expressing liveness and branching-time properties (namely CTL*). A formalism for specifying a wide range of properties has thus to be chosen carefully, in order to make it expressive enough without making it too expensive to use.

Another problem with statistics and queries that are at the level of the modelling formalism is that they are sometimes not "semantic" enough. They do not use a terminology that is relevant for the system, but the terminology of the modelling formalism. For instance, a tool can produce a list of halted states, but cannot divide them into illegal deadlock and legal termination states unless the notion of "legal" is somehow explained to the tool. Furthermore, sometimes the statistics answer a slightly wrong question. As an example we will discuss home states.

As was defined in Section 2.3, a *home state* is a state that is reachable from all reachable states of the system. If a system has a home state, then it has some mode of operation that it can always enter but never exit. The existence of a

home state is usually a good sign, because it reveals that the system cannot do anything irrevocable after an initialisation stage; whatever is the service it provides in and around the home state, it can and will provide it forever. It is, however, possible that the home state belongs to a livelock or even deadlock, in which case the intended service is provided in the "initialisation stage", if at all. This is not possible if all initial states are home states, which is, therefore, considered as a particularly good sign. Home states are a special case of the analysis of strongly connected components of the state space.

Consider any Petri net that has a home state. Let us add to it the isolated net fragment that is shown on the left in Figure 4. This addition has absolutely no effect on the behaviour of the rest of the net. The only thing it does is to introduce exactly one extra semantic transition that is chosen from two possibilities, may take place at any instant of time, and occurs only once. Therefore, making the usual assumption that stuttering is insignificant, the addition does not affect the correctness of the net. However, after the addition the net has no home states. So we see that home states are "fragile" in the sense that a modification that is totally irrelevant for the correct functioning of the net can make a significant change to its home state properties.

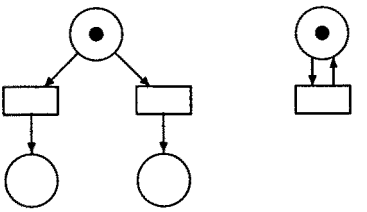


Fig. 4. Two isolated net fragments.

From the point of view of this article, comprehensive statistics and versatile query languages have yet another drawback. Namely, they do not go together well with techniques for alleviating state explosion. Many state space reduction methods are based on throwing some information away. To obtain correct results, a method should preserve those pieces of information that may be needed for producing the answers. When comprehensive statistics are produced or versatile query languages are used, the method should preserve lots of information from all over the net and all over its state space. As a consequence, good reduction results cannot be obtained.

On the other hand, the user seldom needs all that information. If we restrict ourselves to certain parts of the net by, say, letting the analysis questions only refer to certain places and transitions, then the reduction methods can throw away information on the uninteresting places and transitions and yield better results. This can be done with the abstraction mechanisms of Section 2.2.

4.2 Instrumenting the Model

Software modules are often tested in a test bed that sends them input, and receives and checks their output. The same idea can be applied to verification. For instance, a high-level Petri net verification model of a communication protocol may be augmented with the net fragment in Figure 5. As long as the protocol works correctly, the test bed keeps on sending it new messages for transmission. If the protocol ever delivers a wrong message, then t_3 occurs, and if it duplicates a message, then t_4 may occur. If it fails to deliver a message then the system either deadlocks or livelocks depending on whether the protocol part stops.

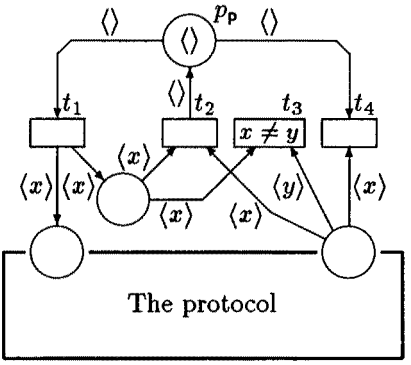


Fig. 5. A protocol test bed.

Deadlock and the occurrence of t_3 or t_4 can be easily checked from typical state space statistics of the kind in Section 4.1. If the initial states of the system as a whole are home states, then it is certain that the protocol cannot enter a livelock that it could not exit. On the other hand, the absence of home states does not necessarily mean that the protocol is incorrect in this respect.

The absence of livelocks that the protocol can exit cannot be verified with home states, occurrence checks and Petri-net-liveness checks. To see this, it suffices to take a net without such livelocks and add to it the net fragment on the right in Figure 4. (If we consider that this addition should not cause a livelock because the added transition cannot prevent the rest of the net from making progress, then this example demonstrates the need for fairness assumptions.)

On-the-fly verification of linear-time safety properties. In the above example, t_3 and t_4 were used as *fact transitions*, that is, Petri net transitions which are never expected to be enabled. Because the enabling condition of a fact transition may refer to several places all over the net, and because one can add extra places to keep track of useful information, fact transitions offer a rather versatile way of specifying verification questions.

It is actually easy to see that fact transitions can be used to check any linear-time property whose counterexamples can be expressed as a regular language over T or $\Sigma \cup \{\tau\}$. For every such property there is a finite automaton that accepts exactly the finite sequences of transition occurrences that violate the property. This automaton can be easily built from Petri net places and transitions and connected to the system with transition fusion.[4] The arrival of a token to a place that corresponds to an acceptance state of the automaton can be detected with a fact transition.

If the properties of interest are stuttering-insensitive, then τ-transitions of the system need not be fused with any transitions of the automaton. This is important for some advanced state space methods, such as the compositional analysis of Section 7.3.

There is an alternative way of connecting the test automaton to the system that is handy for state-based specifications, and used especially with the Büchi automata that are described later in this section (Section 4.2). In it, the test automaton makes its first transition according to what elements of Π are valid in the initial state of the system. Then, for each semantic transition (s, t, s') of the system, the test automaton makes a transition according to what elements of Π are valid in s'. (A more formal definition will be given when Büchi automata are discussed in more detail.) This way of connection requires support from the state space construction tool, as it cannot be implemented in typical modelling formalisms.

Can fact transitions and finite test automata express *all* linear-time safety properties? There is a classic simple argument from set theory that answers "no, no formalism can". Let $\Sigma = \{a, b\}$. Consider any $X \subseteq \{0, 1, 2, \ldots\}$, and let φ_X be the property "the number of a-transitions before the first b-transition is in X". The number of such properties is clearly uncountable. On the other hand, any object or property we ever specify in any formalism must have a finite description in that formalism (although the object itself may be infinite), otherwise we cannot write the specification in full. A finite description is a finite string of characters, and there are only countably many of them. Therefore, there are much more properties than specifications.

Because of the above, it is more meaningful to ask whether all those linear-time safety properties can be specified with fact transitions and finite test automata that can with some other given formalism. The answer is obviously "yes" for all process-algebraic semantics of Section 4.4 provided that the specification LTS is finite, because the automata-theoretic complement of the specification LTS can be used as the test automaton. The answer is also "yes" for all the linear-time safety properties that can be expressed in the temporal logics of Section 4.3, assuming either that the modelling formalism allows the encoding of

[4] *Transition fusion* means that all transitions t_S in the system and t_A in the automaton such that $\mathcal{E}_\Sigma(t_S) = \mathcal{E}_\Sigma(t_A)$ are replaced by transitions t_{SA}, one for each possible (t_S, t_A)-pair. The input and output arcs of t_S and t_A are copied to t_{SA} with their possible arc inscriptions, and the guard of t_{SA} is the conjunction of the guards of t_S and t_A.

atomic propositions on states into Σ in the sense of Section 2.2, or that the automaton is connected to the system with the alternative state-based mechanism discussed above. So we see that fact transitions and finite test automata are actually quite expressive (although they have also limitations, as will be discussed below).

A larger set of properties is obtained if the test automaton is allowed to be unbounded — any Turing machine, for instance. However, although many modelling formalisms allow the implementation of infinite-state automata, such automata tend to cause serious problems to verification algorithms and tools.

The fact that fact transitions allow the verification of any property whose counterexamples can be expressed as a regular language over T or Σ does not imply that this is always practical, or compatible with an advanced verification method one plans to use, or sufficient for verifying even simple properties stated in terms of Π. Representing the condition "no structural transition is enabled" as the disjunction of the enabling conditions of a set of fact transitions is possible in the case of Petri nets where each place may store at most one token at any instant of time, but it may be awkward and require an exponential number of fact transitions. With ordinary Petri nets it is not even possible, because it would make it possible to implement a transition that is enabled exactly when a certain place is empty. This, in turn, would make it possible to simulate two-counter machines, but that is impossible, because two-counter machines can simulate Turing machines, and ordinary Petri nets cannot. If an advanced verification method relies on the assumption that the model has a certain special property — that it is a *free-choice* Petri net, for instance — then it may be that the addition of the fact transition destroys this property, making it impossible to use the method.

When a fact transition becomes enabled it is known that the system is not correct. A state space tool may then immediately stop and print a message. This may save lots of analysis time. Such use of fact transitions is a rudimentary form of *on-the-fly verification*, a powerful idea that we will investigate in more detail in Section 6.

On-the-fly livelock detection. Fact transitions can be used only for linear-time safety properties. We mentioned earlier that livelocks that the protocol can exit cannot be checked with them. In general, a livelock corresponds to an infinite execution that produces no useful result.

In the protocol example, it is reasonable to expect that for each message given for transmission, the protocol computes only a finite amount of time. As a consequence, every infinite execution of the system should have an infinite number of transmission requests, that is, occurrences of t_1. This is an example of a general technique for specifying absence of livelocks: a set of *progress transitions* is specified, and a livelock is reported if and only if the system has an infinite execution that contains only a finite number of occurrences of progress transitions. If the state space is finite, then a livelock of this kind corresponds to a cycle where no edge is labelled by a progress transition.

Instead of progress transitions, it is possible to define *progress states* as the states that satisfy some condition, and interpret livelocks as cycles which do not contain progress states. Such a cycle is called *non-progress cycle* in [43]. In the protocol test bed example, one can define that progress states are those where p_{p} is marked. At first it might seem that this technique would lose those livelocks where the protocol is not transmitting a message, because then p_{p} is continuously marked. This is not the case, however, because in such a situation t_1 is enabled, and the state space tool will investigate also the sequence consisting of the livelock preceded by an occurrence of t_1. In this execution p_{p} is empty, so the livelock is detected.

We see from this example that a test bed need not necessarily be deterministic: it suffices that at least one nondeterministic alternative leads to the detection of the error, and no alternative gives a false alarm. A deterministic test bed is, however, often better than a nondeterministic one in that with it, the state space tool will report an error immediately when enough of the behaviour of the system has been investigated. When a nondeterministic test bed is being used, it may be that an alternative that does not give an alarm is investigated first. Then it is possible that erroneous behaviour and after it many more states are investigated before an error-detecting branch of the test bed is finally chosen and the error is detected.

In the action-based case it is natural to declare all transitions with a visible label as progress transitions. Then livelocks correspond to $\mathcal{E}_{\Sigma}$-abstracted infinite executions that end up with an infinite sequence of τ-transitions. It is not necessarily the case that all such executions are errors. If the error executions are taken and all τ-transitions are removed from them, a language consisting of finite strings and representing some stuttering-insensitive liveness property is obtained. If that language is regular, it can be represented as a finite automaton.

In [87] that automaton was connected to the system with transition fusion just like above with safety properties. Non-progress cycles were reported if and only if the automaton was in an acceptance state. Thus progress transitions (visible transitions) were used simultaneously with progress states (global states where the automaton is not in an acceptance state) for efficient on-the-fly detection of a large set of livelock errors. We may call finite automata that are used in this way *livelock detection automata*. They are particularly useful for certain process-algebraic verification tasks (Section 4.4).

Cycles of a finite directed graph can be detected easily and efficiently with the ordinary depth-first search. If the first search does not cover the graph in full, then a new search is initiated in some ignored vertex and so on, until all vertices have been investigated. Following the colour encoding in [15], let us call a vertex *gray* if the search has entered it but not yet backtracked from it. The graph has a cycle if and only if the search ever encounters an edge from the current vertex to a grey vertex. Non-progress cycles can be found from a finite state space by first removing all edges that correspond to occurrences of progress transitions or are adjacent to progress states, and then seeking for cycles.

It is also possible to integrate the detection of non-progress cycles with the

construction of the state space by processing states and their output (semantic) transitions in a certain order, as was explained in [87]. The basic idea is to investigate progress transitions or semantic transitions that start in a progress state only when there are no other uninvestigated semantic transitions, and investigate all other semantic transitions in depth-first order. This yields an on-the-fly algorithm for detecting livelocks.[5]

Progress transitions or states can also be used in the presence of certain fairness assumptions. For instance, we may want to check that the protocol always eventually delivers the correct message, assuming that the channel loses only a finite number of messages in a row. This can be done by declaring both t_2 and $t_{\mathbf{loss}}$ as progress transitions, where $t_{\mathbf{loss}}$ is the transition that corresponds to loss of messages in the channel. Then a non-progress cycle corresponds to an execution where the last message is not delivered, although only a finite number of messages is lost. The remaining executions either contain an infinite number of deliveries, or violate the fairness assumption.

On-the-fly error detection with Büchi automata and tester processes. With progress transitions or states, an error is declared if an infinite execution contains only a finite number of them. There is an alternative, even more versatile way of detecting errors: states that are visited or transitions that occur an infinite number of times in an erroneous execution. A *Büchi automaton* $\mathcal{B} = (Q, \Sigma, \Delta, q_I, F)$ is otherwise like a finite automaton (thus $\Delta \subseteq Q \times \Sigma \times Q$, $q_I \in Q$ and $F \subseteq Q$), but the notion of "acceptance" has been defined in a different way. Namely, $\mathcal{B}$ accepts an infinite string $a_1 a_2 a_3 \cdots$ of elements of Σ if and only if there is an infinite sequence $q_0 q_1 q_2 \cdots$ of states such that $q_0 = q_I$, $(q_{i-1}, a_i, q_i) \in \Delta$ for $i \geq 1$, and $q_i \in F$ for infinitely many values of i.

A Büchi automaton $\mathcal{B}$ is usually connected to the system such that the alphabet of $\mathcal{B}$ is 2^{Π}, that is, the set of all subsets of the set of atomic propositions. Initially and immediately after each semantic transition of the system, $\mathcal{B}$ makes a transition according to the propositions that hold in the system state. More formally, the joint state space of a state space (S, T, Δ, S_I) and the Büchi automaton $(Q, 2^{\Pi}, \Delta_{\mathcal{B}}, q_I, F)$ has

- $\{ (s_I, q) \mid s_I \in S_I \wedge (q_I, \mathcal{E}_{\Pi}(s_I), q) \in \Delta_{\mathcal{B}} \}$ as its initial states, and
- the rule $(s, q) -t\rightarrow (s', q') \stackrel{\text{def}}{\Longleftrightarrow} (s, t, s') \in \Delta \wedge (q, \mathcal{E}_{\Pi}(s'), q') \in \Delta_{\mathcal{B}}$ determines the set of transitions.[6]

[5] Also [43] describes an algorithm for the task, but it is based on investigating each state twice. The algorithm in [87] does it only once, and is thus usually more efficient. "Usually", because the two algorithms differ in the order in which the state space is investigated, making it possible for the [43] algorithm to detect an error earlier in some cases.

[6] The connection may be defined also such that $\mathcal{B}$ determines its move according to s instead of s'. Then the initial states would be $\{ (s_I, q_I) \mid s_I \in S_I \}$. This formulation does not, however, work appropriately in the presence of deadlock states.

Because 2^{Π} might be big and not all propositions $P \in \Pi$ are always relevant, the transitions of $\mathcal{B}$ may be labelled also by two sets $P_{\mathsf{on}} \subseteq \Pi$ and $P_{\mathsf{off}} \subseteq \Pi$. The transition $(q, P_{\mathsf{on}}, P_{\mathsf{off}}, q')$ is then an abbreviation of the set of the transitions (q, P, q') such that $P_{\mathsf{on}} \subseteq P$ and $P_{\mathsf{off}} \cap P = \emptyset$. That is, $\mathcal{B}$ may make the transition $(q, P_{\mathsf{on}}, P_{\mathsf{off}}, q')$, if and only if at least the propositions in P_{on} evaluate to True and at least the propositions in P_{off} evaluate to False in the system state according to which $\mathcal{B}$ chooses its transition.

Because a Büchi automaton expects an infinite input string, deadlock states have to be handled as a special case. The usual assumption is to add a τ-transition from the deadlock state to itself, or simulate such a transition in the state space construction tool or the algorithm that checks whether the automaton accepts. We saw a similar trick already in Section 2.4 when defining liveness properties.

With this approach, the detection of an error corresponds to acceptance by the Büchi automaton. This acceptance can be checked by seeking for a strongly connected component in the state space where in at least one of the states the local state of the Büchi automaton is an acceptance state. The cost of doing this is proportional to the size of the joint state space of the system and the Büchi automaton which, in turn, is in the worst case proportional to the product of the size of the system state space and the size of the Büchi automaton. The checking of a state space against a Büchi automaton is thus inexpensive.

There is also a nice and efficient algorithm for detecting Büchi acceptance on the fly [16]. It is based on investigating the state space in depth-first order, and starting a second search on already investigated states each time the primary search is about to backtrack from a Büchi acceptance state.

A Büchi automaton may be connected to the system also with other methods, such as transition fusion. The automaton may be connected to all or just visible transitions. In the latter case only stuttering-insensitive properties can be checked. On the other hand, fusion with all transitions is bad for some important methods of alleviating state explosion.

Fusion with only visible transitions makes it possible that an execution is infinite, but the Büchi automaton does not participate it from some point on. This raises a question: if the Büchi automaton stops in an acceptance state, should the corresponding execution be declared as erroneous? It may be reasonable to do so. On the other hand, this is exactly the situation for which we introduced livelock detection automata above. Therefore, it may also be reasonable to let the Büchi automaton ignore all executions where it stops, and handle them with a separate livelock detection automaton.

Of course, the same automaton may have acceptance states of both kinds, and even more. In [87] the use of four kinds of acceptance states was suggested. An error is declared if

- the automaton ever reaches a *reject state*,
- the system stops while the automaton is in a *deadlock monitor state*,
- the system livelocks while the automaton is in a *livelock monitor state*, or

– the automaton passes infinitely many times through an *infinite trace monitor state.*

So these automata combine the acceptance states of Büchi automata, livelock detection automata, and the finite automata that were used above for on-the-fly verification of safety properties, and treat deadlocks as a special case. They were called *tester processes* in [87] because of their intended application in process-algebraic verification (Section 4.4).

It is easy to add error detection by reject and deadlock monitor states to any state space construction algorithm, including the above-mentioned on-the-fly algorithms for detecting non-progress cycles and Büchi acceptance. Thus the first, the second, and one of the latter two kinds of errors can be detected during the same construction of the state space. Unfortunately, the on-the-fly algorithms for non-progress cycles and Büchi acceptance require the construction of the state space in different order, and are thus difficult to combine. On the other hand, we will see in Section 4.4 that the combination of the first three kinds of acceptance states suits well for process-algebraic verification.

It is known that everything that can be specified in the linear temporal logic of Section 4.3 can be checked with Büchi automata. (Actually, Büchi automata are strictly more expressive than the logic.) One may therefore ask: what need is there for reject and livelock monitor states? In the case of reject states the answer lies in the fact that with them, errors are found much earlier than with the above on-the-fly algorithm for Büchi acceptance. The situation with livelock monitor states is less straightforward, but there is a heuristic argument that suggests that also then Büchi acceptance tends to be significantly slower. It is as follows.

The above on-the-fly algorithm for Büchi acceptance is based on a depth-first search, and finds errors when backtracking. Thus an error cannot be found before some branch of the depth-first search tree has been constructed in full. On the other hand, the above on-the-fly algorithm for non-progress cycle detection detects an error immediately when it has constructed all states and semantic transitions of a non-progress cycle. Furthermore, because it disfavours progress states and transitions, it will complete any non-progress cycle rather soon after finding a state in it. One has to notice, however, that this reasoning does not take into account all factors that affect the relative speeds of the algorithms. Deeper theoretical analysis or experimental results would be needed for checking whether non-progress cycle detection really is faster in practice, but for the time being none is available.

The theoretical properties of Büchi and many other types of automata on infinite objects are surveyed in [80].

4.3 Temporal Logics

Temporal logics are ordinary logics augmented with operators for specifying temporal relationships. The temporal logics used in the verification of systems are usually state-based, but also action-based logics have been suggested, especially

in connection with process algebras. Temporal logics are often very expressive, but specifications written in them tend to be somewhat cryptic.

The syntax of a typical temporal logic consists of the following components (we restrict ourselves to *propositional* temporal logics, that is, logics without variable symbols and the quantifiers "$\forall$", "$\exists$"):

- atomic propositions,
- ordinary propositional operators, and
- temporal operators.

The set of atomic propositions of a state-based logic is what we called Π in Section 2.2. It is the link that connects the logic to the application domain. From the point of view of the logic Π is just some abstract set whose content is not important. From the point of view of a user of a temporal logic verification tool the contents of Π are important, because they specify the elementary terms in which the user can talk about the properties of the system. This makes it reasonable to divide a temporal logic verification tool into two parts:

- A modelling-formalism-dependent part that gives the user versatile facilities for writing expressions that describe the properties of individual states, such as $\neg M[t_1\rangle \wedge M(p_2) > M(p_3)$.
- A temporal part that treats the expressions of the above part as atomic propositions, and analyses the temporal relations of states.

The ordinary propositional operators are and "$\wedge$", or "$\vee$", not "$\neg$", implies "$\Rightarrow$", etc. They have their usual meanings.

The first temporal operators that we will discuss, namely the *path operators*, talk about the properties of infinite sequences $\sigma = \langle P_0, P_1, P_2, \ldots\rangle$ of subsets of Π. In a typical application, the sequence σ is the $\mathcal{E}_\Pi$-abstraction of a complete execution of the system in question. If the execution is finite, it is completed to infinite by letting its last state repeat forever. The most important path operators and their meanings are:

If $\varphi \in \Pi$, then $\langle P_0, P_1, P_2, \ldots\rangle \models \varphi$ if and only if $\varphi \in P_0$ (i.e. φ holds in P_0).

$\langle P_0, P_1, P_2, \ldots\rangle \models \Box\varphi$ if and only if $\langle P_i, P_{i+1}, P_{i+2}, \ldots\rangle \models \varphi$ for every $i \geq 0$. "$\Box\varphi$" thus means that φ holds continuously from now on, and is pronounced "always φ" or "henceforth φ".

$\langle P_0, P_1, P_2, \ldots\rangle \models \Diamond\varphi$ if and only if $\langle P_i, P_{i+1}, P_{i+2}, \ldots\rangle \models \varphi$ for at least one $i \geq 0$. "$\Diamond\varphi$" thus means that φ holds now or will hold at least once in the future, and is pronounced "eventually φ" or "sometimes φ". The formula $\Diamond\varphi \Leftrightarrow \neg\Box\neg\varphi$ holds.

$\langle P_0, P_1, P_2, \ldots\rangle \models \varphi\, \mathcal{U}\, \psi$ if and only if there is $i \geq 0$ such that $\langle P_i, P_{i+1}, P_{i+2}, \ldots\rangle \models \psi$, and $\langle P_j, P_{j+1}, P_{j+2}, \ldots\rangle \models \varphi$ for every j such that $0 \leq j < i$. "$\varphi\, \mathcal{U}\, \psi$" is pronounced "$\varphi$ until ψ" and means that ψ will eventually hold, and φ holds until then. The formula $\Diamond\varphi \Leftrightarrow \mathsf{True}\, \mathcal{U}\, \varphi$ holds.

$\langle P_0, P_1, P_2, \ldots \rangle \models \bigcirc\varphi$ if and only if $\langle P_1, P_2, P_3, \ldots \rangle \models \varphi$. "$\bigcirc\varphi$" thus means that φ will hold in the next state. The use of this operator in specification is often discouraged, because, unlike with "$\Box$", "$\Diamond$" and "$\mathcal{U}$", with it one can specify properties that are sensitive to stuttering. However, "$\bigcirc$" is important for the development of temporal logic verification algorithms, because formulas such as $\varphi \,\mathcal{U}\, \psi \Leftrightarrow \psi \vee \varphi \wedge \bigcirc(\varphi \,\mathcal{U}\, \psi)$ use it and have proven useful in that work.

Many other path operators have been defined in the literature, such as "weak until" that is equivalent to $(\varphi \,\mathcal{U}\, \psi) \vee \Box\varphi$, "$\rightsquigarrow$" or "leads to" $\Box(\varphi \Rightarrow \Diamond\psi)$, and operators that refer to the past instead of the future.

The infinite repetition of the last state of a finite complete execution guarantees that $\bigcirc\varphi$ has a well-defined meaning also in the last state of the execution. The meanings of "$\Box$", "$\Diamond$" and "$\mathcal{U}$" are not affected by it. Terminating executions were extended to infinite also in the definition of liveness properties in Section 2.4, and of acceptance of Büchi automata in Section 4.2.

The extension has the consequence that unless Π contains enough information for characterising deadlock states, the temporal logics in this section do not suffice for distinguishing between a deadlock and livelock. At first this might seem a deficiency, but it is actually in harmony with the "philosophy" of abstraction: all that matters is when and how the $\mathcal{E}_\Pi$-abstracted state may change, and if it does not change from some point on, it is not important whether the system has terminated or is running an endless invisible loop.

LTL. So far we have defined what it means for a *complete execution* to satisfy a temporal logic formula. Now we need to define it for a *system*. The simplest way to do that is to say that a system satisfies a formula built of the above operators, if and only if all of its complete executions satisfy it. With this definition, the logic consisting of the above operators can be only used to specify properties that are linear-time in the sense of Section 2.3. Correspondingly, it is called *(propositional) linear temporal logic*, and often abbreviated as "LTL".

LTL has been studied extensively in numerous articles, and the textbooks [59, 60]. The sublogic of LTL where the use of "$\bigcirc$" is forbidden is denoted by LTL$_{-\mathsf{X}}$ in this article. With it, only stuttering-insensitive properties can be specified.

If φ is an LTL formula and σ is an infinite sequence of subsets of Π, then σ satisfies either φ or $\neg\varphi$. However, the same is not true for systems: it is possible that neither φ nor $\neg\varphi$ holds in a given system. This is because the system may have one execution that satisfies φ thus violating $\neg\varphi$, and another one that violates φ. This induces a natural preorder relation between systems for any fixed Π: Sys_1 *implements* or *is more deterministic than* Sys_2, if and only if $\varphi \models Sys_2$ implies $\varphi \models Sys_1$ for every LTL-formula φ whose atomic propositions are from Π.

A *model checking* algorithm inputs a state space and a temporal logic formula φ and checks whether φ is a true statement about the state space. LTL

has a model checking algorithm whose worst-case time consumption is linear in the size of the state space, and exponential in the length of the formula [57]. It is thus feasible for short LTL formulae, but may run very long with long formulae. Fortunately, the time consumption of the algorithm depends on what operators and how are used in the formula, so not every long formula causes the algorithm to slow down. Furthermore, the LTL formulae needed in verification are often rather short. It is very unlikely that a worst-case polynomial time LTL model checking algorithm could ever be found, because the LTL model checking problem is known to be PSPACE-complete [78].

The above-mentioned model checking algorithm is not particularly popular in automatic verification as such, but its basic techniques have been used in an alternative, more popular approach. Namely, every LTL formula can be compiled to a Büchi automaton that accepts exactly the executions described by the formula. One advanced practical algorithm for this was given in [32]. As was told in Section 4.2, Büchi automata can be efficiently and even on the fly used for verifying that no execution has the property described by the automaton. Therefore, the validity of any LTL formula φ can be checked by constructing a Büchi automaton for $\neg\varphi$ and checking the system with it. This technique is called the *automata-theoretic approach to model checking* [97], and it suits well both ordinary and on-the-fly verification.

Unfortunately, the PSPACE-hardness of LTL model checking must manifest itself somewhere also in the automata-theoretic approach: the size of a Büchi automaton may be exponential compared to the size of the LTL formula from which it was constructed.

Assume that t_enab denotes that the structural transition t is enabled, and t_occ that it has just occurred. Weak fairness towards t can be expressed with the LTL formula $\Diamond\Box t\_\mathsf{enab} \Rightarrow \Box\Diamond t\_\mathsf{occ}$, and strong fairness with $\Box\Diamond t\_\mathsf{enab} \Rightarrow \Box\Diamond t\_\mathsf{occ}$. As a consequence, a fairness assumption could be taken into account in the verification of the LTL formula φ by expressing the assumption with an LTL formula γ, and checking the validity of $\gamma \Rightarrow \varphi$. However, because fairness assumptions are very common, special techniques for handling them have been developed and integrated to LTL model checking algorithms.

CTL and CTL*. Another possibility of extending the logic to apply to systems relies on the following two new operators. They say whether a formula should be valid in one or all possible executions that start at a given state s:

$s \models \mathsf{A}\varphi$, if and only if φ holds in all paths of the state space that start at s.
$s \models \mathsf{E}\varphi$, if and only if φ holds in at least one path of the state space that starts at s.

The system satisfies a formula, if and only if all of its initial states satisfy it.

Because the validity of A- and E-formulae has been defined for every individual state of the state space, it is meaningful to apply path operators on formulae built with A and E. In this way one gets formulae like $\mathsf{A}\,\Box\,\mathsf{E}\,\Diamond\, t\_\mathsf{enab}$, which says that in all possible futures, there is always a future where eventually

t_enab holds. If t_enab means that t is enabled, then this formula expresses the Petri-net-liveness of t. In this context the operators "$\Box$", "$\Diamond$", "$\mathcal{U}$" and "$\bigcirc$" are usually written as "G", "F", "U" and "X", respectively. Thus the above formula is more commonly written as $\mathsf{A\,G\,E\,F}\,t\_\mathsf{enab}$.

This logic was developed by Emerson and Halpern [20] on the basis of LTL and the earlier CTL logic of Clarke and Emerson [11], and it is known as *CTL** ("CTL" stands for *computation tree logic*). The validity of CTL*-formulae could also be defined on execution trees instead of state spaces, but that would not change the meaning of the logic. Properties expressible with CTL* are thus branching-time in the sense of Section 2.3. Any LTL formula φ can be represented as the CTL* formula $\mathsf{A}\varphi$, so CTL* is an extension of LTL. As a consequence, also CTL* model checking is PSPACE-hard. It is not harder, though, as was proven in [21].

The older logic CTL is a restriction of CTL*. It has a fast model checking algorithm [11]. In CTL, every path operator must be immediately preceded by A or E. Thus $\mathsf{AGEF}\,t\_\mathsf{enab}$ is a valid CTL formula (and more commonly written as $\mathsf{AGEF}\,t\_\mathsf{enab}$), but $\mathsf{E}(\varphi\,\mathsf{U}\,\mathsf{G}\psi)$ is not. CTL has become very popular in automatic verification. Although not all LTL formulae can be expressed in CTL, CTL seems to have enough expressive power for many verification tasks, and its efficient model checking algorithm is definitely an advantage. Like with LTL, the use of the next state operator is sometimes forbidden, yielding the logics $\mathrm{CTL}_{-\mathsf{X}}$ and $\mathrm{CTL}^{*}_{-\mathsf{X}}$.

The basic idea of the CTL model checking algorithm can be illustrated with the case of the AU operator; the remaining operators can be handled with similar methods. Let us say that a state is *φ-marked*, if and only if the algorithm has found out that φ holds in it. Consider a formula of the form $\theta = \mathsf{A}(\varphi\ \mathsf{U}\ \psi)$. The algorithm is first run recursively to φ-mark every state where φ holds, and similarly with ψ. Then every ψ-marked state is marked with θ. Finally the following is repeated as long as possible: if there is a state s that is φ-marked but not θ-marked, and each of its immediate successor states is θ-marked, then s is marked with θ. This last stage can be implemented efficiently with a suitable backwards search.

In order to compare CTL* and $\mathrm{CTL}^{*}_{-\mathsf{X}}$ to the branching-time process-algebraic semantic models discussed in Section 4.4, we define two equivalence notions between the state spaces $Sys_1 = (S_1, \Delta_1, S_{I1})$ and $Sys_2 = (S_2, \Delta_2, S_{I2})$ [7]:

1. Sys_1 and Sys_2 are *Π-bisimilar* if and only if there is a relation "$\sim$" $\subseteq S_1 \times S_2$ such that the following hold for every $s_1, s_1' \in S_1$ and $s_2, s_2' \in S_2$ (the prefix "Π" was added to the name of the relation to avoid confusion with the "strong bisimilarity" relation in Section 4.4):
   - If $s_1 \sim s_2$ and $\pi \in \Pi$, then $s_1 \models \pi$ if and only if $s_2 \models \pi$.
   - If $s_1 \in S_{I1}$, then there is $s \in S_{I2}$ such that $s_1 \sim s$.
   - If $s_2 \in S_{I2}$, then there is $s \in S_{I1}$ such that $s \sim s_2$.
   - If $s_1 \sim s_2$ and $(s_1, s_1') \in \Delta_1$,
     then there is s such that $s_1' \sim s$ and $(s_2, s) \in \Delta_2$.

- If $s_1 \sim s_2$ and $(s_2, s_2') \in \Delta_2$,
  then there is s such that $s \sim s_2'$ and $(s_1, s) \in \Delta_1$.

2. The complete executions $\langle s_{1,0}, s_{1,1}, s_{1,2}, \ldots \rangle$ and $\langle s_{2,0}, s_{2,1}, s_{2,2}, \ldots \rangle$ *stutter-simulate* each other according to the relation "$\simeq$" $\subseteq S_1 \times S_2$, if and only if $\langle s_{1,0}, s_{1,1}, s_{1,2}, \ldots \rangle$ has a partition $B_{1,0}, B_{1,1}, B_{1,2}, \ldots$ and $\langle s_{2,0}, s_{2,1}, s_{2,2}, \ldots \rangle$ has a partition $B_{2,0}, B_{2,1}, B_{2,2}, \ldots$ such that for every $i \geq 0$ the following hold: $0 < |B_{1,i}| < \infty$, $0 < |B_{2,i}| < \infty$, and $\forall s_1 \in B_{1,i} : \forall s_2 \in B_{2,i} : s_1 \simeq s_2$. The state spaces Sys_1 and Sys_2 are *stuttering equivalent* if and only if there is a relation "$\simeq$" $\subseteq S_1 \times S_2$ such that the following hold for every $s_1 \in S_1$ and $s_2 \in S_2$:
   - If $s_1 \simeq s_2$ and $\pi \in \Pi$, then $s_1 \models \pi$ if and only if $s_2 \models \pi$.
   - If $s_1 \in S_{I1}$, then there is $s \in S_{I2}$ such that $s_1 \simeq s$.
   - If $s_2 \in S_{I2}$, then there is $s \in S_{I1}$ such that $s \simeq s_2$.
   - If $s_1 \simeq s_2$, then for every complete execution of Sys_1 that starts at s_1 there is a complete execution of Sys_2 starting at s_2 such that the two executions stutter-simulate each other according to "$\simeq$".
   - If $s_1 \simeq s_2$, then for every complete execution of Sys_2 that starts at s_2 there is a complete execution of Sys_1 starting at s_1 such that the two executions stutter-simulate each other according to "$\simeq$".

It is relatively easy to see that if Sys_1 and Sys_2 are Π-bisimilar and Sys_1 satisfies a CTL* formula φ, then also $Sys_2 \models \varphi$. This was proven in [7] together with a reverse result where CTL (without the "*") suffices: if Sys_1 and Sys_2 are finite and satisfy the same CTL formulae, then they are Π-bisimilar. A similar pair of results holds for stuttering equivalence, CTL$^*_{-X}$, and CTL$_{-X}$.

The article [19] is an excellent survey on various temporal logics and their theoretical properties, including model checking.

4.4 Process-Algebraic Semantics

A process algebra, such as the *Calculus of Communicating Systems* (*CCS*) [65] and *Communicating Sequential Processes* (*CSP*) [6, 42, 75] consists of a language for specifying systems, and a theory of the behaviour of the systems specified in that language. Most (or perhaps all) process algebras are action-based. Instead of structural transitions, emphasis is put on *actions* that are $\mathcal{E}_\Sigma$-abstractions (Section 2.2) of structural transitions. Correspondingly, in the context of process algebras, state spaces are usually defined as (S, Σ, Δ, S_I), where Σ does not contain the invisible action symbol τ, and $\Delta \subseteq S \times (\Sigma \cup \{\tau\}) \times S$. Such structures are called *labelled transition systems*, abbreviated *LTS*. Occurrences of actions (i.e. semantic transitions) are sometimes called *events*.

Process composition operators. The language of a process algebra is a "modelling formalism" in the sense of Section 3. It practically always contains some operators for *parallel composition* of processes and for *hiding* of actions. (The CCS language does not have a separate hiding operator, but its parallel

composition is capable of hiding.) Communication between parallel processes is synchronous, like Ada rendez-vous. The meanings of operators are usually defined in terms of expressions of the language, and their details vary, but the basic ideas can be illustrated with state-space-level definitions.

The following definition corresponds to a popular version of parallel composition. Let $L_1 = (S_1, \Sigma_1, \Delta_1, S_{I1})$ and $L_2 = (S_2, \Sigma_2, \Delta_2, S_{I2})$ be LTSs. Their parallel composition $L_1 || L_2$ is the LTS (S, Σ, Δ, S_I) such that

- $S = S_1 \times S_2$
- $\Sigma = \Sigma_1 \cup \Sigma_2$
- $\big((s_1, s_2), a, (s_1', s_2')\big) \in \Delta$ if and only if either
  - $(s_1, a, s_1') \in \Delta_1$, $a \notin \Sigma_2$, and $s_2' = s_2$,
  - $a \notin \Sigma_1$, $s_1' = s_1$, and $(s_2, a, s_2') \in \Delta_2$, or
  - $(s_1, a, s_1') \in \Delta_1$, $(s_2, a, s_2') \in \Delta_2$, and $a \neq \tau$.
- $S_I = S_{I1} \times S_{I2}$

Since τ is never a member of Σ, this definition implies that a component of a parallel composition can do its τ-transitions independently of and without affecting the state of the other component, and this is the only way in which τ-transitions can be executed. The same holds for transitions that are labelled with a visible action that is not in the alphabet of the other component, because $(s, a, s') \in \Delta$ implies $a \in \Sigma$ or $a = \tau$. If a visible action is in the alphabet of both components, then a corresponding transition must be executed by both components simultaneously. This parallel composition is essentially the same thing as the transition fusion of Section 4.2.

This definition extends easily to more than two components. Alternative definitions of parallel composition may specify the synchronisation of visible actions in a different way, but it is probably universal that τ-transitions do not synchronise. A parallel composition computed according to the above definition contains usually many unreachable states. Since their presence is most of the time either insignificant or harmful, the definition of parallel composition is often formulated in such a way that unreachable states and their adjacent semantic transitions are not generated or are removed from the result. This is not an important issue, however, because, with reasonable notions of "behaviour", the removal of unreachable states does not affect the behaviour of the system.

The hiding operator converts visible actions into τ. A popular version of it can be defined as **hide** A **in** $(S, \Sigma, \Delta, S_I) = (S, \Sigma', \Delta', S_I)$, where A is some set of actions, and

- $\Sigma' = \Sigma - A$, and
- $\Delta' = \big\{ (s, a, s') \;\big|\; (s, a, s') \in \Delta \wedge a \notin A \big\}$
  $\cup \big\{ (s, \tau, s') \;\big|\; \exists a \in A : (s, a, s') \in \Delta \big\}$.

Strong bisimilarity. As was mentioned above, a theory of the behaviour of systems is an essential component of a process algebra. Because the theory of CCS was originally presented in the context of the CCS language and similarly

for CSP, it is common to associate each theory to a particular language. This is not necessary, however, and we can and will discuss the theories of behaviour at the level of state spaces in the sequel.

To jump from a language to state spaces it suffices that an operational semantics for process-algebraic expressions is defined which, given any well-defined expression in the language, produces a labelled transition system that represents the behaviour of the object described by the expression. For a reason that will be described later in this section, there is significant freedom in the choice of the technical details of this semantics. For instance, even if it is intuitively clear that the system has only a finite number of states, there is no compelling reason to require that the corresponding LTS must be finite.

Behavioural equivalence is a central notion in process-algebraic theories of behaviour. Behavioural equivalences can be divided to two categories according to how they treat τ-transitions. In the first category τ is handled in the same way as all visible actions. This category contains only one important equivalence: *strong bisimilarity* [66]. The LTSs $L_1 = (S_1, \Sigma, \Delta_1, S_{I1})$ and $L_2 = (S_2, \Sigma, \Delta_2, S_{I2})$ that have the same alphabet are *(strongly) bisimilar*, denoted in this article by $L_1 \simeq_{\mathbf{sb}} L_2$, if and only if there is a relation "$\sim$" $\subseteq S_1 \times S_2$ such that the following hold for every $s_1, s_1' \in S_1$, $s_2, s_2' \in S_2$, and $a \in \Sigma \cup \{\tau\}$:

- If $s_1 \in S_{I1}$, then there is $s \in S_{I2}$ such that $s_1 \sim s$.
- If $s_2 \in S_{I2}$, then there is $s \in S_{I1}$ such that $s \sim s_2$.
- If $s_1 \sim s_2$ and $(s_1, a, s_1') \in \Delta_1$,
  then there is s such that $s_1' \sim s$ and $(s_2, a, s) \in \Delta_2$.
- If $s_1 \sim s_2$ and $(s_2, a, s_2') \in \Delta_2$,
  then there is s such that $s \sim s_2'$ and $(s_1, a, s) \in \Delta_1$.

The relation "$\sim$" is called a *strong bisimulation*. The basic idea of this definition is that if the LTSs are in strongly bisimilar states and one of them makes a transition, the other can simulate it with a transition of its own such that the labels of the transitions are the same, and after both transitions the systems are again in strongly bisimilar states. Since it is assumed that every initial state of each LTS is simulated by some initial state of the other LTSs, a simple induction argument reveals that each system can simulate everything the other does. It is difficult to think of any property of LTSs that could reasonably be called "behavioural" and that would hold in one and not in the other of two strongly bisimilar LTSs. (Two strongly bisimilar LTSs can have different numbers of states, for instance, but the number of states can hardly be called "behavioural".)

Two strongly bisimilar systems thus have "the same behaviour" in a very strong sense. Indeed, strong bisimilarity is the strongest equivalence that is in common use in process algebras. It is useful for technical purposes, such as abstracting away from irrelevant details of the mapping that associates to each process-algebraic expression the LTS that represents its behaviour. It is needed because it is common that two process-algebraic expressions that, according to our intuition, have obviously the same behaviour are nevertheless formally

different because of syntactic details, and cannot thus be unified outright. For instance, according to the above definition, $L_1||L_2$ is not the same as $L_2||L_1$, because their states list their components in opposite order.

The *Lotos* specification language [44, 4] provides another, more fundamental example. Its operational semantics is defined in such a way that a cyclic process "remembers" the number of times the cycle has been executed. As a consequence, the state after the cycle is formally different from the state before the cycle although intuitively it is "the same state", and the resulting LTS is not a cycle but an infinite chain. However, the chain can be folded into a cycle on the basis that the start states of each round are strongly bisimilar. Strong bisimilarity gives a sound mathematical notion for unifying states.[7]

Algorithms for strong bisimilarity. We may say that two states s_1 and s_2 of an LTS $L = (S, \Sigma, \Delta, S_I)$ are strongly bisimilar, if and only if there is a strong bisimulation relation between $(S, \Sigma, \Delta, \{s_1\})$ and $(S, \Sigma, \Delta, \{s_2\})$. Let $[[s]]$ denote the set of states that are strongly bisimilar with s, and let $L_{\mathsf{min}} = (S', \Sigma, \Delta', S'_I)$, where $S' = \{\, [[s]] \mid s \in S \,\}$, $S'_I = \{\, [[s]] \mid s \in S_I \,\}$, and $([[s]], a, [[s']]) \in \Delta'$ if and only if there is $s'' \in [[s']]$ such that $(s, a, s'') \in \Delta$ (this definition of Δ' is independent of the choices of the states s that span the sets $[[s]]$). Then $L \simeq_{\mathsf{sb}} L_{\mathsf{min}}$.

Furthermore, it is possible to show that if L is finite, then L_{min} is minimal (that is, has the smallest possible number of states and transitions) among the LTSs that are strongly bisimilar to L, and all LTSs that are strongly bisimilar to L and have as few states as L_{min} are the same as L_{min} except for the names of states. A similar result holds also when L is infinite, although then one has to be careful with what is meant by "minimal".

There is a very efficient algorithm that constructs L_{min} from any given finite LTS [26]. It resembles the classic "block splitting" algorithm for the minimisation of deterministic finite automata. It can also be used for checking strong bisimilarity of two finite LTSs simply by minimising their disjoint union. The LTSs are strongly bisimilar, if and only if each block that contains an initial state of one of them also contains an initial state of the other.

Strong bisimilarity is almost the same as the Π-bisimilarity of Section 4.3. The only difference is that now the labels of matching transitions are compared instead of the sets of propositions that hold in matching states. This difference can be easily taken into account in the minimisation of an LTS with block splitting. As a consequence, algorithms for strong bisimilarity can be taken advantage of in the verification of CTL* formulae.

Abstract process equivalences. The second category of behavioural equivalences consists of those where most information about τ-transitions is abstracted away in one way or another. We will call them *abstract process equivalences*. It

[7] Mathematicians often use isomorphism for these kinds of tasks. Isomorphism is a strictly stronger notion than strong bisimilarity. It is too strong for the above Lotos example, because it does not unify a cycle with the infinite chain that is its unfolding.

is often desirable to preserve some information on the indirect consequences of τ-transitions. However, opinions differ regarding how much and what kind of information should be preserved, which has led to the development of many different equivalences. Furthermore, the desire of making equivalences to have certain mathematical properties that are useful either for theory development (such as unique minimal solutions to recursive equations) or for applications has further increased the number of equivalences that have been presented in the literature. The survey [95] lists 155. In this article we can mention only some. Fortunately, many of the others are small variations of what we will present.

An important requirement, especially from the point of view of advanced verification methods, is that the equivalence should be a *congruence* with respect to parallel composition, hiding, and whatever other process operators are used. This means that if "$\simeq$" denotes the equivalence, then $L_1 \simeq L_2$ should guarantee that **hide** A **in** $L_1 \simeq$ **hide** A **in** L_2, $L_1||L \simeq L_2||L$, $L||L_1 \simeq L||L_2$, and similarly for the other operators, where L_1, L_2 and L are arbitrary LTSs. The congruence property guarantees that any LTS that is a component of a bigger system can freely be replaced by an equivalent LTS, and the behaviour of the bigger system does not change. Although the congruence property seems a natural one, it is difficult to obtain, and many otherwise nice equivalences lack it.

Trace semantics and verification with a process equivalence. The simplest widely used abstract process equivalence is *trace equivalence*. In process algebras, a *trace* is the sequence of visible actions obtained from a finite execution by removing all states and all τ-symbols. In other words, it is the $\mathcal{E}_\Sigma$-abstraction of a finite execution with all stuttering removed. (Process-algebraic traces have thus nothing to do with Mazurkiewicz traces [61].)

The set of traces of an LTS (which we will denote with $Tr(L)$) is called its *trace semantics*, and two LTSs that have a common alphabet are *trace equivalent* (written as $L_1 \simeq_{\mathsf{tr}} L_2$ in the sequel) if and only if they have the same trace semantics. *Trace preorder* "$\sqsupseteq_{\mathsf{tr}}$" is defined by $L_1 \sqsupseteq_{\mathsf{tr}} L_2$ if and only if $Tr(L_1) \subseteq Tr(L_2)$.[8] Trace equivalence is a congruence with respect to most, or perhaps all, process operators that have been suggested in the literature.

Trace semantics can be used in the verification of a system *Sys* in at least three different ways. For the sake of examples, let us assume that *Sys* is a model of a communication protocol implementation that consists of a protocol sender process and a protocol receiver process connected to each other via a bidirectional channel, such that all actions are hidden except those with which the protocol sender inputs transmission requests from and reports success or failure to the sending client, and those with which the protocol receiver delivers arrived messages to the receiving client. Assume further that *Spec* is a single LTS that models the service that the protocol is supposed to provide to the clients. We presume that *Spec* has the same alphabet as *Sys*; if necessary, the hiding operator is first used to convert the extra actions in either alphabet to τ.

[8] This direction of "$\sqsupseteq$" seems to be more common in the literature, although also the opposite direction has supporters.

1. One can compare whether *Sys* $\simeq_{\mathsf{tr}}$ *Spec*. If not, then *Sys* either has an illegal sequence of communication with the clients of the protocol (such as the delivery of a wrong message), or it lacks some desired sequence (for instance, fails to deliver a message). If yes, then *Sys* provides exactly the required service as far as one can say on the basis of the trace semantics.
2. One can compare whether *Sys* $\sqsupseteq_{\mathsf{tr}}$ *Spec*. It is common that a specification does not define a system in full, but only states minimal requirements that the system must satisfy. In the case of a protocol, *Spec* may allow the protocol sender to give up and report failure to the sending client, if it does not receive an acknowledgement from the protocol receiver within a specified time. However, if the channel and protocol receiver are reliable and fast enough, then the acknowledgement is never lost or delayed, and *Sys* never reports failure. In that case *Sys* lacks some traces that *Spec* has, but for a very acceptable reason. In this kind of a situation the requirement that *Sys* $\simeq_{\mathsf{tr}}$ *Spec* is too stringent, but *Sys* $\sqsupseteq_{\mathsf{tr}}$ *Spec* works well (except that for many applications, it is not stringent enough, but other process equivalences that are described later in this section will solve this problem).
3. One can construct a small LTS that is trace-equivalent to *Sys* and investigate its properties, for example, with model checking tools. Of course, one should investigate only those properties φ that trace equivalence *preserves* in the sense that if $L_1 \models \varphi$ and $L_1 \simeq_{\mathsf{tr}} L_2$, then $L_2 \models \varphi$.

In the terminology of Section 3, *Spec* is the "specification". As a consequence, the "specification formalism" is the formalism in which *Spec* was written, that is, a process-algebraic language or LTSs. It may very well be the same formalism in which *Sys* was written! The "satisfies" relation that says what it means for the system to satisfy its specification is $\simeq_{\mathsf{tr}}$ in the first case above, and $\sqsupseteq_{\mathsf{tr}}$ in the second case.

Let φ be any stuttering-insensitive linear-time safety property over Σ. Let L_φ be an LTS whose finite executions are exactly those which do not violate φ. The LTS L_φ is not necessarily finite, but it exists, at least in a theoretical sense. Now, an LTS L has the property φ, which we may write as $L \models \varphi$, if and only if $L \sqsupseteq_{\mathsf{tr}} L_\varphi$. We see that preorder checking is conceptually close to model checking. This fact has useful consequences in automatic verification: thanks to it, similar techniques can be used for model checking and preorder checking in many cases.

Because trace semantics is defined on the basis of finite executions, it preserves only stuttering-insensitive linear-time safety properties. On the other hand, as the following simple argument shows, it preserves all of them. If L violates a stuttering-insensitive linear-time safety property φ, then it has a finite execution ξ that acts as a counterexample. Let σ be the trace corresponding to that execution. If $L' \simeq_{\mathsf{tr}} L$, then also L' has a finite execution that has σ as its trace; call it ξ'. The executions ξ and ξ' differ only in the amount of finite stuttering. Thus also ξ' is a valid counterexample to φ, because φ is stuttering-insensitive. So L' violates φ. This argument is valid also for state-based properties, if the transformation in Section 2.2 is used for encoding state information into actions.

In Section 4.2 we pointed out that no formalism can specify all safety properties, because there are uncountably many of them. This might seem to be in contradiction with the above claim that trace semantics preserves all safety properties, because trace semantics was used above in specification by describing an LTS *Spec* and requiring that *Sys* $\simeq_{\mathsf{tr}}$ *Spec* or *Sys* $\sqsupseteq_{\mathsf{tr}}$ *Spec*. There is no contradiction, however, because trace semantics was not used as the specification formalism but as an abstract mathematical concept that serves as the "satisfies" relation. The specification formalism is the formalism used for describing *Spec*, and it allows only countably many specifications to be written.

Algorithms for trace semantics. Every finite LTS L can be interpreted as a finite automaton in the classic automata-theoretic sense — we only need to specify which states are accepting. If we declare that all states are accepting, then $Tr(L)$ is the same as the language accepted by the automaton. As a consequence, classic algorithms can be used for determinising LTSs, and minimising deterministic LTSs and comparing the languages they accept. Unfortunately, these algorithms consume exponential time (and space) in the worst case, because the minimal deterministic finite automaton that accepts the same language as a given finite automaton may be exponentially larger than the latter.

That *Sys* $\sqsupseteq_{\mathsf{tr}}$ *Spec* holds can be checked on the fly as follows. First *Spec* is determinised yielding *DSpec*. Then the LTS *Sys*||*DSpec* is computed. If the result has at least one state where *Sys* is ready to execute a visible action that *DSpec* is not ready to execute, then *Sys* $\sqsupseteq_{\mathsf{tr}}$ *Spec* does not hold, otherwise it does.

Instead of comparing the next possible actions of *Sys* and *DSpec* in each joint state, the errors can also be detected with the machinery of tester processes in Section 4.2. It suffices to add one extra reject state s_R, and extra transitions (s, a, s_R) to *DSpec* for those $s \in S_{DSpec}$ and $a \in \Sigma$ such that *DSpec* has no a-transition out of s. Then violations against *Sys* $\sqsupseteq_{\mathsf{tr}}$ *Spec* are detected as *Sys*||*DSpec* reaching any state of the form (s, s_R). This algorithm consumes exponential time and memory in the size of *Spec*, but only linear time in the size of *Sys*. If *Spec* is deterministic, then this algorithm is linear in the sizes of both of its arguments.

It is known well that the problem of checking whether a finite automaton $\mathcal{A} = (Q, \Sigma, \Delta, q_I, F)$ accepts all strings in Σ^* is PSPACE-complete (see e.g., [1, 30]). Consider the LTS $L = (S, \Sigma', \Delta', S_I)$ that is obtained from $\mathcal{A}$ by choosing a new state $s_{\mathsf{new}} \notin Q$ and a new action $\delta \notin \Delta$ and letting $S = Q \cup \{s_{\mathsf{new}}\}$, $\Sigma' = \Sigma \cup \{\delta\}$, $\Delta' = \Delta \cup \{ (s, \delta, s_{\mathsf{new}}) \mid s \in F \} \cup \{ (s_{\mathsf{new}}, a, s_{\mathsf{new}}) \mid a \in \Sigma' \}$, and $S_I = \{q_I\}$. The automaton $\mathcal{A}$ accepts all strings in Σ^* if and only if $Tr(L) = \Sigma'^*$. Thus the question "are all finite strings over the alphabet of a given LTS traces of that LTS" is PSPACE-hard.

The problem "does *Sys* $\sqsupseteq_{\mathsf{tr}}$ *Spec* hold" is in PSPACE, because an execution σ of *Sys* can be guessed that produces a trace that is not in $Tr(Spec)$, and "$\sigma \notin Tr(Spec)$" can be verified by maintaining the set of all the states where *Spec* could be when simulating the trace. Polynomial space does not necessarily

suffice for storing the trace, but it need not be stored, if simulation is interleaved with the guessing of the execution.

From the above results we can reason that the problems "does $Sys \simeq_{\mathrm{tr}} Spec$ hold" and "does $Sys \sqsupseteq_{\mathrm{tr}} Spec$ hold" are PSPACE-complete, even if Sys is replaced by the one-state LTS All_Σ that has a transition from that state to itself for every member of Σ. Namely, the question "does $All_\Sigma \sqsupseteq_{\mathrm{tr}} Spec$ hold" is the same as "is $Tr(Spec) = \Sigma^*$". Therefore, checking "$Sys \sqsupseteq_{\mathrm{tr}} Spec$" is PSPACE-complete in the size of $Spec$. On the other hand, the above algorithm checks it in time that is linear in the size of Sys. Indeed, the question "does $Sys \sqsupseteq_{\mathrm{tr}} All_\Sigma$ hold" is computationally very easy: its answer is always "yes".

In conclusion, we can check $Sys \sqsupseteq_{\mathrm{tr}} Spec$ inexpensively in terms of the size of Sys, but expensively in the size of $Spec$. It is common that Sys, representing the implementation, consists of several parallel processes and has a huge state space, while $Spec$ consists of only one process and has a small state space. This is fortunate, because it means that expensive operations are done to small and inexpensive operations to big objects. This observation generalises to the verification of linear-time properties in many formalisms: we already saw a similar situation with LTL (Section 4.3), and we will soon see it again with certain other linear-time abstract process semantics. If we wanted to verify $Sys \simeq_{\mathrm{tr}} Spec$ or $Spec \sqsupseteq_{\mathrm{tr}} Sys$, then we would have to do expensive operations also to Sys, leading soon to the exhaustion of computational resources. This suggests that we can attack the state explosion problem much better if we prefer "$\sqsupseteq$"-type notions of "system satisfies specification" over "$\simeq$"-type.

Another operation used frequently in process-algebraic verification is *LTS reduction*. It means the construction of an LTS that is equivalent to the input LTS, but as small as the algorithm can produce. If the result is the smallest possible equivalent LTS, then this operation is known as *minimisation* of the LTS. Unfortunately, as the counterexample in Figure 6 shows, trace semantics does not guarantee the existence of a unique minimal equivalent LTS. Even the problem of finding some equivalent LTS with the smallest possible number of states is PSPACE-hard, because its solution can obviously be easily used to solve "does $Sys \simeq_{\mathrm{tr}} All_\Sigma$ hold". (It is also in PSPACE because it can be solved by constructing LTSs of increasing size until an equivalent one is found.)

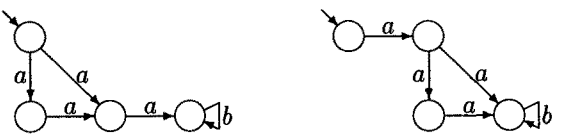


Fig. 6. Two non-isomorphic trace-equivalent minimal LTSs.

Fortunately, various heuristics can be used for trace-equivalence-preserving

LTS reduction. Examples include minimisation with respect to weak bisimilarity, and the construction of the minimal equivalent deterministic LTS with rejection of the result if it is not smaller than the input LTS. These kinds of techniques have proven to work reasonably well in practice.

Refusals, failures and divergence. Because trace semantics does not preserve any liveness properties, other semantics have been developed. Unlike in temporal logics, in process algebras it is common (but not ubiquitous) to distinguish between deadlock and livelock. This is partly due to a reason illustrated by the following example. If L is ready to do an a-transition and only it, a is not in the alphabet of either L_1 or L_2, and L_1 is in deadlock and L_2 in livelock, then $L||L_1$ is guaranteed to eventually do an a-transition, but $L||L_2$ is not. In terms of a single-processor multi-process computer system, the system running $L||L_1$ will eventually execute L because L_1 is not executable, but the system running $L||L_2$ might spend all processor time on L_2 unless we make some fairness assumption about the scheduling policy. So we see that although we cannot detect directly whether an LTS has deadlocked or livelocked, we can find it out by putting the LTS into a suitable context.

A livelock corresponds to an infinite execution that has only a finite number of visible transitions. The removal of all τ-symbols from the $\mathcal{E}_\Sigma$-abstraction of the execution yields a finite sequence. That sequence is called a *divergence trace*. We will denote the set of the divergence traces of an LTS L with $Divtr(L)$.

To obtain a congruence that handles deadlocks properly, a more general notion of *refusal* has been defined in process algebras. The need for more generality can be illustrated with the following example. Consider two LTSs L_1 and L_2, both of which have $\{a, b\}$ as their alphabet. It may be that in some situation, L_1 is ready to execute an a-transition but not any b-transition, while L_2 is ready for b but not a. Then $L_1||L_2$ is in deadlock, although neither process would be in deadlock if it were alone.

Let $L = (S, \Sigma, \Delta, S_I)$ be an LTS, $A \subseteq \Sigma$, and $s \in S$. We say that s is *stable*, or L *is stable in* s, if and only if $\neg(s \xrightarrow{\tau})$, that is, s has no outgoing τ-transitions. Furthermore, s *refuses* A, or L *refuses* A *in* s, if and only if s is stable and has no outgoing transitions labelled with an element of A.[9] This definition has the consequence that $L_1||L_2$ deadlocks in (s_1, s_2) if and only if s_1 and s_2 are stable, and there are sets A_1 and A_2 such that $A_1 \cup A_2 = \Sigma_1 \cup \Sigma_2$ and L_i refuses A_i in s_i for $i \in \{1, 2\}$. It is worth noticing that if a belongs to both Σ_1 and Σ_2, then it suffices that a is in one of A_1 and A_2; that is, one of the LTSs is not ready for a.

A *stable failure* of an LTS $L = (S, \Sigma, \Delta, s_I)$ is any pair (σ, A) such that L has a stable state s that refuses A, and a finite execution that ends at s and has σ as the corresponding trace. The set of the stable failures of L is denoted by $Sfail(L)$. The set of traces that lead to a deadlock is $\{\, \sigma \mid (\sigma, \Sigma) \in Sfail(L) \,\}$, and we have the identity $Tr(L) = Divtr(L) \cup \{\, \sigma \mid (\sigma, \emptyset) \in Sfail(L) \,\}$.

[9] Not all authors require the stability of s in this definition.

Finally, consider an infinite execution that contains an infinite number of occurrences of visible transitions. The result of the removal of all τ-symbols from its $\mathcal{E}_\Sigma$-abstraction is an infinite sequence of elements of Σ, and it is called an *infinite trace*. The notation $Inftr(L)$ denotes the set of the infinite traces of L. Let $Tr\omega(L) = \{\, \omega \in \Sigma^\omega \mid \forall \sigma < \omega : \sigma \in Tr(L) \,\}$, that is, $Tr\omega(L)$ is the set of those infinite sequences of elements of Σ whose every proper prefix is a trace of L. It is clear that $Inftr(L) \subseteq Tr\omega(L)$. If L is finite, then also $Tr\omega(L) \subseteq Inftr(L)$.

Because $Tr(L)$, $Divtr(L)$ and $Inftr(L)$ are defined on the basis of executions, they are linear-time notions. The set $Sfail(L)$ can be thought of as having some branching-time aspect, because its elements talk about all possible next visible actions after a particular execution. However, whether or not a state refuses a set of actions can be thought of as a property of the state, and thus a member of Π. In this interpretation, the property "L has the stable failure (σ, A)" is clearly a linear-time property. Therefore, we will classify as linear-time all abstract process semantics that are only built from $Tr(L)$, $Sfail(L)$, $Divtr(L)$ and $Inftr(L)$.

Failure-based semantic models. The most well-established linear-time abstract process semantics that preserves some liveness properties is certainly the standard *failures–divergences* model of CSP [6, 42, 75]. Because of reasons that at least partly depend on the mathematical approach used in the development of the theory of CSP, this semantics uses different notions of failures and divergences from what we defined above.

Let $CSPdivtr(L) = \{\, \sigma \in \Sigma^* \mid \exists \rho : \rho \leq \sigma \wedge \rho \in Divtr(L) \,\}$. That is, a *CSP-divergence trace* of L is any divergence trace of L continued with any finite sequence of actions. *CSP-failures* of L are all stable failures of L, and all pairs (σ, A) where σ is a CSP-divergence trace of L and $A \subseteq \Sigma$. That is, $CSPfail(L) = Sfail(L) \cup (CSPdivtr(L) \times 2^\Sigma)$. The *CSP-semantics* of L is the pair $(CSPfail(L), CSPdivtr(L))$, and CSP-preorder and CSP-equivalence of two LTSs that have the same alphabet are defined as

$$L_1 \sqsupseteq_{\mathsf{CSP}} L_2 \stackrel{\text{def}}{\Longleftrightarrow} CSPfail(L_1) \subseteq CSPfail(L_2) \wedge CSPdivtr(L_1) \subseteq CSPdivtr(L_2), \text{ and}$$

$$L_1 \simeq_{\mathsf{CSP}} L_2 \stackrel{\text{def}}{\Longleftrightarrow} L_1 \sqsupseteq_{\mathsf{CSP}} L_2 \wedge L_2 \sqsupseteq_{\mathsf{CSP}} L_1.$$

The CSP-semantics of L has the property that if σ is a divergence trace, then $\sigma\rho$ is a CSP-divergence trace and $(\sigma\rho, A)$ is a CSP-failure, for every $\rho \in \Sigma^*$ and $A \subseteq \Sigma$. This implies that CSP-semantics does not preserve any information about the behaviour of the system after it has executed σ. This property of CSP-semantics is called *catastrophic divergence*. Any system C that has ε as a divergence trace has $CSPdivtr(C) = \Sigma^*$ and $CSPfail(C) = \Sigma^* \times 2^\Sigma$, and is called *chaos*. The catastrophic divergence property makes CSP-semantics useless for analysing the behaviour of a system after a divergence trace. Despite this, CSP-semantics has been very successful both in terms of theory development and practical use. The textbook [75] is a thorough treatment of the CSP language, semantic theory of CSP, and automatic verification in the context of CSP.

The catastrophic divergence problem has motivated researchers to develop alternative semantic models based on some kinds of failures and/or divergence traces. Among them, the *Chaos-Free Failures Divergences* (*CFFD*) semantics [94] is interesting because of its special relation to LTL$_{-X}$ (Section 4.3). It is the triple $(Sfail(L), Divtr(L), Inftr(L))$. (Originally it had also a fourth *initial stability* component. It was needed for ensuring the congruence property in the presence of the so-called *choice* process composition operator.) CFFD-preorder and -equivalence are defined just like the corresponding CSP notions. If the LTSs are finite, then the sets of infinite traces need not be compared, because then $Inftr(L) = Tr\omega(L)$ and $Tr\omega(L)$ is uniquely determined by $Sfail(L)$ and $Divtr(L)$.

CFFD-equivalence implies CSP-equivalence, trace equivalence and the NDFD-equivalence mentioned below in the sense that if $L_1 \simeq_{\mathsf{CFFD}} L_2$, then $L_1 \simeq_{\mathsf{CSP}} L_2$, $L_1 \simeq_{\mathsf{tr}} L_2$ and $L_1 \simeq_{\mathsf{NDFD}} L_2$. Furthermore, NDFD-equivalence implies CSP- and trace equivalences.

Assume that Π is encoded into Σ as was discussed in Section 2.2. Then CFFD-equivalence is the weakest possible (that is, makes least distinctions between systems) congruence that (1) preserves the validity of formulae written in LTL$_{-X}$, and (2) distinguishes between deadlock and livelock. Furthermore, a slight modification of CFFD-equivalence called *nondivergent failures divergences* (*NDFD*, it takes into account only those stable failures (σ, A) where σ is not a divergence trace) equivalence is the weakest congruence that has the property (1). These facts were proven in [47], and elaborated a bit further in [89]. Due to them, CFFD- and NDFD-semantics provide a means for applying process-algebraic verification methods to the verification of LTL$_{-X}$ formulae.

CFFD- and NDFD-equivalences do, however, suffer from a problem that to some extent restricts their applicability to the verification of liveness properties in general and LTL$_{-X}$ formulae in particular: they do not handle fairness assumptions in a satisfactory way. As was mentioned in Section 4.3, fairness assumptions can be encoded in the formula whose validity is being verified. However, doing that leads to a big Σ, and that is bad for process-algebraic methods of alleviating state explosion. This problem is not a deficiency of only the CFFD- and NDFD-equivalences, but is present in most or all abstract process semantics that are both congruences and strong enough for handling liveness properties. Some ways of working around it have been found [71, 92], but more work needs to be done in this field.

Algorithms for failure-based semantics. Most complexity results concerning trace semantics generalise relatively easily to CSP-, CFFD- and NDFD-semantics. For instance, the LTSs in Figure 6 are CSP-, CFFD- and NDFD-equivalent, thus also these equivalences fail to have unique minimal LTSs.

The paper [14] shows how algorithms for strong bisimilarity can be used for failure-based semantic models by first transforming the LTSs into *acceptance graphs* that are based on the *acceptance trees* of [41]. Acceptance graphs are, in essence, deterministic LTSs augmented with a special representation for refusal and divergence information. Acceptance graphs were applied in a slightly differ-

ent way in [93], where algorithms for LTS equivalence comparison and reduction according to CFFD semantics were developed. The results of [93] can be adapted to CSP- and NDFD-semantics.

The *tester processes* presented towards the end of Section 4.2 can be used for checking CFFD-preorder, and even without infinite trace monitor states [87]. The basic idea is that illegal traces are caught with reject states, illegal stable failures with deadlock monitor states, and illegal divergence traces with livelock monitor states. Illegal infinite traces need not be worried of, because the presence of such a trace in a finite LTS implies the presence of an illegal stable failure or illegal divergence trace. Reject states are not absolutely necessary because every illegal trace can be detected as an illegal deadlock or illegal divergence trace. Reject states are, however, very easy to implement efficiently in a state space tool, and they detect an error immediately when the error trace has been executed, while deadlock and livelock monitor states require continuation to a deadlock or livelock. Reject states thus improve efficiency.

There is an algorithm that, given an LTS *Spec*, produces a tester process that checks $Sys \sqsupseteq_{\mathsf{CFFD}} Spec$. It is based on the construction of the *DSpec* with an extra reject state s_R that was defined in "Algorithms for trace semantics", the taking of "mirror images" of refusal sets as has been described in [5], and the marking of those states as deadlock and livelock monitor states which correspond to traces after which *Spec* cannot deadlock or livelock, respectively. The algorithm consumes exponential time and space in the worst case, both because the number of states may grow exponentially, and because the handling of refusal sets may be expensive with pathological inputs. In practice, however, the algorithm often runs reasonably fast. This approach can be adapted to NDFD- and CSP-preorders with small changes. Like with trace semantics, the use of tester processes is inexpensive in the size of *Sys*.

Another on-the-fly method for checking CSP-preorder was described in [74, 75]. It also relies on constructing *DSpec*, but from then on it works differently from the previous one. It augments *DSpec* with an acceptance-tree-type representation of refusal and divergence information, making it possible to compare on the fly each state of *Sys* directly with the corresponding state in *DSpec*. (The "corresponding state" is the unique state of *DSpec* that is reached with the same trace as with which the state of *Sys* was reached.) Again, small changes suffice to apply this algorithm to CFFD- and NDFD-preorders.

Branching-time abstract semantics. *Weak bisimilarity* or *observation equivalence* [65] is certainly the most well known abstract branching-time semantic model in process algebras. Its definition resembles that of strong bisimilarity, but has the difference that when simulating a transition, the simulating LTS may do any number of transitions (including zero) as long as the resulting sequence of the visible actions is the same in both sides.

Let $s -\tau^* \rightarrow_i s'$ denote that s' can be reached from s with zero or more τ-transitions in the LTS in question. The LTSs $L_1 = (S_1, \Sigma, \Delta_1, S_{I1})$ and $L_2 = (S_2, \Sigma, \Delta_2, S_{I2})$ are *weakly bisimilar*, denoted here $L_1 \simeq_{\mathsf{wb}} L_2$, if and only if

there is a relation "$\sim$" $\subseteq S_1 \times S_2$ such that for every $s_1, s_1' \in S_1$, $s_2, s_2' \in S_2$, and $a \in \Sigma$:

- If $s_1 \in S_{I1}$, then there is $s \in S_{I2}$ such that $s_1 \sim s$.
- If $s_2 \in S_{I2}$, then there is $s \in S_{I1}$ such that $s \sim s_2$.
- If $s_1 \sim s_2$ and $(s_1, \tau, s_1') \in \Delta_1$,
  then there is s such that $s_2 -\tau^*\!\rightarrow_2 s$ and $s_1' \sim s$.
- If $s_1 \sim s_2$ and $(s_2, \tau, s_2') \in \Delta_2$,
  then there is s such that $s_1 -\tau^*\!\rightarrow_1 s$ and $s \sim s_2'$.
- If $s_1 \sim s_2$ and $(s_1, a, s_1') \in \Delta_1$, then there are s, s' and s'' such that $s_2 -\tau^*\!\rightarrow_2 s'$, $(s', a, s'') \in \Delta_2$, $s'' -\tau^*\!\rightarrow_2 s$, and $s_1' \sim s$.
- If $s_1 \sim s_2$ and $(s_2, a, s_2') \in \Delta_2$, then there are s, s' and s'' such that $s_1 -\tau^*\!\rightarrow_1 s'$, $(s', a, s'') \in \Delta_1$, $s'' -\tau^*\!\rightarrow_1 s$, and $s \sim s_2'$.

The definition allows the simulation of a local τ-loop (that is, a transition of the form (s, τ, s)) by no transition at all. As a consequence, weak bisimilarity does not distinguish deadlock from livelock. If this is considered a deficiency, an extra requirement may be added saying that if $s_1 \sim s_2$, then $s_1 -\tau^\omega\!\rightarrow_1 \Leftrightarrow s_2 -\tau^\omega\!\rightarrow_2$, where $s -\tau^\omega\!\rightarrow$ means that there is an infinite execution that starts in s and contains only τ-transitions. This modification of weak bisimilarity implies CFFD-equivalence (excluding the "initial stability" component). Its theory and algorithms were investigated in [18].

Assume that an LTS $L = (S, \Sigma, \Delta, S_I)$ is converted to $Sat(L) = (S, \Sigma, \Delta', S_I)$ such that $\Delta' = \Delta \cup Sc_1 \cup Sc_2$, where *Sat* and *Sc* stand for "saturate" and "shortcut", $Sc_1 = \{\, (s, a, s') \mid (s \neq s' \vee a \neq \tau) \wedge \exists s_1, s_2 : s -\tau^*\!\rightarrow s_1 \wedge (s_1, a, s_2) \in \Delta \wedge s_2 -\tau^*\!\rightarrow s \,\}$, and $Sc_2 = \{\, (s, \tau, s) \mid s \in S \,\}$. Clearly $L' \simeq_{\mathsf{wb}} L$. What is more, $L_1 \simeq_{\mathsf{wb}} L_2 \Leftrightarrow Sat(L_1) \simeq_{\mathsf{sb}} Sat(L_2)$. The computation of $Sat(L)$ is essentially the problem of computing the transitive closure of a relation. It can be solved in cubic time (even faster, but the significance of the known faster algorithms is mostly theoretical).

In this way minimisation and comparison of LTSs according to weak bisimilarity can be converted to the related problems with strong bisimilarity, which, as we have seen, have very efficient algorithms. This approach is from [48], and yields cubic equivalence checking and minimisation algorithms for weak bisimilarity. Unfortunately, the number of semantic transitions in a saturated LTS tends to be high, so this approach consumes a lot of memory. This can be avoided by simulating the "$-\tau^*\!\rightarrow$"-relation as needed instead of storing it in an explicit form, but then the algorithm becomes slower.

It is clear that weak bisimilarity does not preserve the validity of all CTL$_{-\mathsf{X}}$-formulae, because it discards livelock information. However, as the following example demonstrates, weak bisimilarity does not even preserve the validity of all those CTL$_{-\mathsf{X}}$ properties that do not depend on livelocks. The LTSs in Figure 7 are weakly bisimilar, but the formula $\mathsf{EG}(\neg\alpha \wedge (\mathsf{EF}\alpha \vee \neg\mathsf{EF}\beta))$ holds only in the rightmost one.

In Section 4.3 a *stuttering equivalence* was defined that preserves the validity of CTL$^*_{-\mathsf{X}}$-formulae. It requires that each state of an execution at one side is

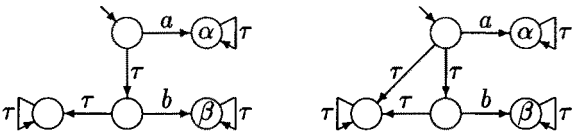


Fig. 7. Weak bisimilarity does not preserve CTL.

simulated by one or more states at the opposite side. Therefore, according to it, the execution where the rightmost LTS in Figure 7 jumps to the bottom left state without visiting the bottom middle state cannot be stutter-simulated by the leftmost LTS, because this execution has no state that could be the pair of the bottom middle state.

Branching bisimilarity [96] is a process-algebraic equivalence that is between weak bisimilarity and stuttering equivalence. Branching bisimilarity implies weak bisimilarity, and differentiates the LTSs in Figure 7 from each other. It, too, discards livelock information, and therefore fails to preserve the validity of CTL$_{-X}$ formulae. It can be modified to preserve CTL$_{-X}$, but then it becomes essentially the same as stuttering equivalence. One attractive feature of branching bisimilarity is that its algorithms are similar to the algorithms for weak bisimilarity, but need less memory, because the saturation of LTSs is not needed.

The article [90] is a tutorial on the basic ideas of process-algebraic semantic models and automatic verification with them.

5 The Complexity of Verification

5.1 Complexity in Terms of the State Space

In the previous section we mentioned the computational complexity of some verification tasks as a function of the size of the state space $|\mathcal{S}|$ and the size of the property or specification $|\varphi|$:

- The checking of various individual properties, such as the absence of deadlocks, 4-boundedness of a Petri net place and Petri-net-liveness of a transition are linear in $|\mathcal{S}|$.
- LTL and CTL* model checking are linear in $|\mathcal{S}|$ and PSPACE-complete in $|\varphi|$.
- CTL model checking is linear in $|\mathcal{S}|$ and $|\varphi|$.
- Minimisation with respect to trace, CSP-, CFFD- and NDFD-semantics is not well-defined, because there is no unique minimum. Finding some equivalent LTS with a minimal number of states is PSPACE-complete.
- Trace, CSP-, CFFD- and NDFD-equivalence checking are PSPACE-complete.

- Trace, CSP-, CFFD- and NDFD-preorder checking are linear in the alleged smaller LTS, and PSPACE-complete in the other argument.
- An LTS can be minimised with respect to weak or branching bisimilarity in $O(|\mathcal{S}|^3)$ time.
- Equivalence according to weak or branching bisimilarity can be checked in $O(|\mathcal{S}_1|^3 + |\mathcal{S}_2|^3)$ time.

Excluding CTL* model checking, the above results suggest a general tendency: verification tasks related to linear-time properties tend to be PSPACE-complete, whereas similar tasks for branching time can be performed in low-order polynomial time. This may feel counterintuitive, because branching time seems more complicated than linear time and, for instance, weak bisimilarity implies trace equivalence. The case of LTL vs. CTL has an easy explanation: CTL restricts seriously what one can say about the properties of individual executions, and when those restrictions are removed (yielding CTL*), model checking becomes PSPACE-hard again. In the case of weak bisimilarity vs. trace semantics the explanation lies in the fact that checking equivalence under two different semantics is two distinct problems that do not necessarily have much in common, even if one of the equivalences implies the other. We will now illustrate this with an example.

Consider the following seven different equivalences, each of which implies the next: identity ($(S_1, \Sigma_1, \Delta_1, S_{I1}) = (S_2, \Sigma_2, \Delta_2, S_{I2})$ if and only if $S_1 = S_2$, $\Sigma_1 = \Sigma_2$, $\Delta_1 = \Delta_2$, and $S_{I1} = S_{I2}$), isomorphism, strong bisimilarity, branching bisimilarity, weak bisimilarity, trace equivalence, and the "universal" equivalence that says that all LTSs are equivalent. Identity is easy: it can be checked in linear time on the average by inputting the LTSs into a hash table, and in worst-case linear time by replacing the hash table with the (impractical) data structure in exercise 12.1-4 of [15]. Isomorphism is clearly in NP, but has not been proven NP-complete — as a matter of fact, it is a strong candidate for a problem that is not solvable in polynomial time but is not NP-hard either [30]. It is thus rather complex. Strong bisimilarity, despite of its resemblance to isomorphism, can be checked in low-order polynomial time. Known algorithms for branching bisimilarity take some more time and weak bisimilarity still some more, but they are still within cubic time. Trace equivalence is PSPACE-complete and thus (apparently) the hardest in this list. On the other hand, the universal equivalence is the easiest of all: the algorithm **print** "yes" solves it in constant time.

We see that the strength of an equivalence and the complexity of its equivalence checking have low correlation. The fact that branching-time semantic models have fast equivalence checking algorithms seems thus coincidental. Intuitively, their low complexity compared to equivalence checking in linear-time models is due to the fact that branching-time tasks rely on *local* information, while linear-time verification requires simultaneous information about all possible executions that have the same $\mathcal{E}$-abstraction.

We will see in Section 5.3 that there is, however, a sense in which the weakness of an equivalence is beneficial regarding the cost of verification.

5.2 Complexity in Terms of the Modelling Formalism

In the previous section we discussed the complexity of verification as a function of the size of the state space of the system. In reality, the system is usually not originally given as a state space, but as a program in some concurrent programming language, Petri net, parallel composition of LTSs, etc. This representation is typically much, much smaller than the state space — after all, that is what the state explosion problem is about. As a consequence, the complexity of a problem as a function of the size of the state space does not give complete picture of the cost of verification.

Numerous results exist about the complexity of the verification of some property from a system represented in some formalism. One can derive more with the following technique. (More detail can be found in [89], for instance.)

A *linear bounded automaton* is a nondeterministic single-tape Turing machine whose use of the tape is restricted as follows. The input string is surrounded by two endmarkers that the machine cannot write over or bypass. The tape alphabet may be much larger than that used in the input string, and the machine is allowed to write over the input. Therefore, the machine can use the part of the tape that originally contains the input also as working storage. As a consequence, the working storage of the machine is linear in the size of the input string (while with the ordinary nondeterministic Turing machine it is unbounded).

The question whether a given linear bounded automaton accepts a given string is PSPACE-complete ([30], p. 265). A linear bounded automaton is basically a finite automaton that has access to a fixed amount of memory that it can address only by jumping from a memory cell to the next or previous cell. Given a typical formalism that is intended for modelling concurrent systems, it is usually relatively easy to design a system that simulates an arbitrary linear bounded automaton with an arbitrary input, and whose size is polynomial in the size of the description of the automaton and input. The simulator can be made such that when the automaton reaches an acceptance state, the simulator does something that it otherwise cannot do, such as stops or executes a certain transition. Checking the ability of doing "that something" is thus PSPACE-hard for the formalism in question. In this way it is possible to show that the detection of deadlocks, checking whether a given structural transition may ever become enabled, and so on are PSPACE-hard for Petri nets, parallel labelled transition systems, and so on.

To prove that something is PSPACE-complete instead of just PSPACE-hard, it suffices to additionally show that an execution leading to a deadlock, ending up with the occurrence of a given structural transition, etc. can be simulated using at most a polynomial amount of space as a function of the size of the description of the system. Because of the famous result by Savitch that PSPACE = NPSPACE ([76], [30] p. 176, [1] p. 395), the simulator needs not "know" what step it should take next if there are many choices; we may assume that it can always "guess" it. However, when making proofs of this kind, one has to remember that polynomial space does not usually suffice for storing the guesses or the simulated sequence, so the sequence can be used only once in the proof.

The detection of deadlocks, checking whether a given structural transition may ever become enabled, and so on are indeed PSPACE-complete for parallel labelled transition systems and many other formalisms. The result is not valid for ordinary Petri nets because polynomial space does not suffice for storing the marking, as the number of tokens in a place may grow very high. It becomes valid if a fixed upper bound to the number of tokens is enforced for each place.

We can make the conclusion from the above that an interesting verification problem for an interesting modelling formalism can be easier than PSPACE-hard only in exceptional cases. Excluding those exceptional cases (and ignoring the very unlikely possibility that PSPACE = P), *every* verification algorithm for an interesting task must have bad worst-case performance. Thus each verification method must contain at least one potentially expensive step. With ordinary state spaces, this step is obviously the construction of the state space. The same holds for most advanced methods discussed in Section 7. We will see in Section 7.2 that there are methods that can sometimes pack full state space information into a very small representation, but with them the problem of extracting eventual answers from the packed state space tends to be complicated.

Fortunately, the above complexity results apply only to the worst case. They do not prevent a verification method from being reasonably efficient in a large set of practical cases. They do, however, imply that for each method there are systems with which the method becomes very slow (unless PSPACE = P).

5.3 How Much Information to Preserve?

An advanced state space method can reduce the size of the state space either by throwing some information away, or by representing the information in a denser form. Throwing information away implies that some verification questions cannot any more be answered. In theory, packing the information more densely does not rule out the ability to answer any verification question. In practice, both the above complexity results and experience with advanced state space methods suggest that if the packed representation is significantly denser than the ordinary state space, then extracting the answers to certain verification questions becomes so difficult that it becomes a bottleneck. It is thus reasonable to accept the idea that state explosion cannot be significantly alleviated without losing some analysis capability.

Consider two sets Γ_1 and Γ_2 of analysis questions such that $\Gamma_1 \subseteq \Gamma_2$ and $\Gamma_1 \neq \Gamma_2$. Any method that is capable of answering all questions in Γ_2 can be used also for Γ_1. Furthermore, there may be advanced methods that are legal for Γ_1, but fail to answer correctly some verification question in $\Gamma_2 - \Gamma_1$. Therefore, at least in principle, Γ_1 allows the use of at least the same and may allow more tools for attacking the state explosion problem than Γ_2. Therefore, *the more information we are willing to give away, the more we can do to avoid state explosion.*

As an example of the above principle, we will now explain how the fact that some equivalence (say, trace equivalence) preserves strictly less information than some other (weak bisimilarity, for instance) can be used to reduce the cost of

verification, although equivalence checking and minimisation are expensive for trace equivalence and cheap for weak bisimilarity. To compare the uses of two different equivalences in the same verification task it is necessary to assume that both equivalences apply to that task, that is, they preserve the property φ in question. For instance, if φ is any linear-time safety property that is stated in terms of Σ, then both trace equivalence and weak bisimilarity preserve it. In such a situation the equivalences are just tools in the verification of φ, not goals in themselves.

As will be discussed in more detail in Section 7.3, equivalences can be used to avoid the construction of the full state space of the system, and produce a smaller but equivalent state space instead. Because any two weakly bisimilar state spaces are also trace-equivalent but the opposite does not hold, trace semantics allows the production of smaller state spaces than weak bisimilarity. Computing the smallest possible trace-equivalent state space is expensive, but it is not necessary in this kind of an application. It is perfectly legal to first run the inexpensive algorithm that produces the unique minimal weakly bisimilar state space, and then apply some inexpensive trace-equivalence-preserving heuristics to reduce the result further still, and perhaps minimise again with respect to weak bisimilarity. In this way a state space is constructed that is never bigger and may be much smaller than the smallest weakly bisimilar state space, but is still valid for the verification of any linear-time safety property φ.

5.4 How Much Memory is Really Needed?

Consider the problem of detecting deadlocks from an ordinary Petri net that has the property that no place ever contains more than one token. This problem can be easily solved by constructing the state space, if enough time and memory is available. The number of the reachable states of the net may be exponential, so this method uses exponential space and consequently also exponential time in the worst case. Even so, unless the net belongs to some special subclass, state spaces are the most practical deadlock detection method known today. We saw above, however, that deadlock detection is PSPACE-complete for this type of Petri nets. Deadlocks can thus be detected in polynomial space, at least in theory. Why are polynomial space deadlock detection algorithms not used in practice?

The result that deadlocks can be detected in polynomial space relies on a theorem by Savitch ([76], [30] p. 176, [1] p. 370). From the proof that Savitch gave to his theorem, it is possible to derive the following polynomial space algorithm "is_reachable(M_1, M_2, k)" for checking whether M_2 is reachable from M_1 in at most k steps.

If $k = 0$ the algorithm returns True if $M_1 = M_2$, and False otherwise. If $k = 1$ the algorithm returns True if there is a structural transition t such that $M_1 [t\rangle M_2$, and False otherwise. For bigger values of k, the algorithm constructs and tests one at a time every possible marking M that assigns to each place either 0 or 1 tokens. A marking is tested by calling "is_reachable(M_1, M, $\lfloor \frac{k}{2} \rfloor$)" and "is_reachable($M$, M_2, $\lceil \frac{k}{2} \rceil$)". If both calls return True, then it is known that there is a path from M_1 to M_2 that goes through M, so the algorithm returns

True. If either call returns False, the test of M fails, and the algorithm proceeds to the next value of M. If the test fails for all values of M, then the algorithm returns False. This is correct, because if there were a path from M_1 to M_2 whose length is at most k, then the middlemost marking of the path would make both "is_reachable(M_1, M, $\lfloor \frac{k}{2} \rfloor$)" and "is_reachable($M$, M_2, $\lceil \frac{k}{2} \rceil$)" to return True.

Let the number of the places in the net be n. The net has at most 2^n reachable markings, because we assumed that $M(p) \leq 1$ for every place p and reachable marking M. Therefore, if M_2 is reachable from M_1, it is reachable via a path whose length is less than 2^n. As a consequence, reachability of a deadlock can be checked by letting M_1 scan through all initial markings and M_2 through all markings, and by calling "is_reachable(M_1, M_2, 2^n)" for each M_2 that is a deadlock. Each such call creates a tree of recursive calls of "is_reachable" whose height is at most $n+1$. Each invocation of "is_reachable(M_1, M_2, k)" consumes only $\Theta(n)$ bits of memory, because each of M, M_1, M_2 and k fits $n+1$ bits. Thus the total memory consumption is $O(n^2)$, which is low-order polynomial in n.

The memory consumption of the above algorithm is actually quite reasonable. Even a straightforward non-optimised stack-based (instead of recursive) implementation survives with $4n+2$ bits of memory per recursion level. If, for instance, $n = 1000$, then roughly 4 000 000 bits or 500 kilobytes of memory suffices for checking the reachability of a deadlock.

On the other hand, the time consumption of the above algorithm is woeful. If no marking is reachable from another, then the bottom level of recursion is called 2^{n^2} times. That is $2^{1\,000\,000}$ or roughly $10^{300\,000}$ times for $n = 1000$. The estimated age of the universe is only about 10^{27} nanoseconds.

The reason why the above algorithm consumes so little space and so much time is that it throws away almost all subresults it has computed, and recomputes them again and again. Significant savings in time may be obtained by storing the subresults and fetching them from the store as needed instead of recomputing them. But that would require exponential memory. This observation is valid for PSPACE-complete problems in general: although they can be solved in polynomial space in theory, in practice exponential space is used because it is much faster.

In conclusion, it is in theory possible to solve interesting verification tasks in relatively small memory. However, the known algorithms that do that consume unimaginable amounts of time. The possibilities of making a big enough improvement to the speed of the algorithms while keeping the memory consumption polynomial seem minuscule. On the other hand, a much, much faster algorithm is obtained, one that is often fast enough in practice, if one is willing to use much more memory and store intermediate results in it. This is exactly what the state space methods do.

6 Reduction Strategies

An advanced verification algorithm that tries to alleviate state explosion may be organised in many different ways. In this section we discuss the most important of them.

Transparent construction-time reduction. Perhaps the easiest way of connecting a state explosion alleviation method to verification with state spaces is *transparent construction-time reduction.* In it, instead of the full state space, a reduced state space is constructed that gives the same answers to a predefined class of verification questions as the full state space would give. Specifications may be checked and analysis questions may be answered with the same algorithms and tools as with full state spaces.

In order to avoid incorrect answers, the user has to know what properties the reduced state space has preserved, or the analysis tool must refuse to answer illegal questions. The latter may be implemented by adding to the state space some information on what properties it preserves.

The basic stubborn set method that preserves deadlocks and non-termination (Section 7.4) is an example of transparent construction-time reduction. So is also the compositional LTS construction method that is commonly used in the verification of process-algebraic equivalences between systems (Section 7.3).

Guided construction-time reduction. *Guided construction-time reduction* is an elaboration of transparent construction-time reduction, where information about the analysis or verification question or special knowledge on the system is used for guiding the construction of the reduced state space. The abstraction mechanisms of Section 2.2 are one simple way of doing this: the set Π or Σ is specified before the construction of the reduced state space, the state space reduction algorithm uses knowledge of that set to decide where it may reduce, and only the elements in the set can be used as basic components in the verification questions. When the user wants to ask a question that refers to details that are not present in Π or Σ, a new state space has to be constructed.

For instance, the $\mathrm{LTL}_{-\mathrm{X}}$ and $\mathrm{CTL}^{*}_{-\mathrm{X}}$-preserving stubborn set methods (Section 7.4) work in this way. Also compositional LTS construction falls into this category when it is used as an aid in the verification of individual properties (such as mutual exclusion) instead of checking process equivalence.

If the user specifies a small Π or Σ, then the state space can answer only a small set of verification questions. On the other hand, if Π or Σ is made larger, then also the state space becomes larger and may soon exceed the capacity of the verification tools. The user is thus in a trade-off situation.

The user may have special knowledge of the system that is useful in alleviating state explosion. For instance, the user may know that a correct token-ring system contains at most one token at any instant of time. Because an incorrect system does not necessarily have this property, one has to be careful when taking advantage of this kind of information in verification in order to avoid circular reasoning. In Sections 7.1 and 7.3 we will see examples of how this can be done (the elimination of remnant variable values with **uninitialise**-statements, and the interface processes of Graf and Steffen).

Preprocessing the model. State explosion can be alleviated also by modifying the system description before starting the construction of the state space, or by taking the needs of state space methods into account already when modelling the system. For instance, it is customary to use in a verification model as few variables as possible and to restrict their types to as small as possible. It is also customary to make the degree of atomicity of transitions as coarse as possible.

The correctness of abstractions of this kind is in the responsibility of the author of the verification model. It is also possible to apply sound theories and automatic tools for preprocessing the model such that it is guaranteed that the answers to certain verification questions are not changed. This is the topic of Section 7.1.

Packed state spaces. A *packed state space* is a nonstandard, dense way of storing the state space or a part of it. The packed state space may contain full or incomplete information on the interleaved (or even true concurrency) behaviour of the system.

The analysis algorithms and tools have to be modified to work on packed state spaces instead of ordinary ones. Because the information on states and semantic transitions is to some extent implicit in a packed state space, extracting an answer to a verification question is sometimes hard. This leads to slow and complicated algorithms for certain verification questions. As a consequence, methods based on packed state spaces tend to work well only for certain types of analysis questions, even if no information is lost in the packing. Fortunately, the most important types of packed state spaces support well quite large types of analysis questions.

The symmetry method, the unfolding method, and BDDs are examples of packed state space methods that preserve full information. Petri net coverability graphs preserve incomplete information, and allow the verification of only a restricted class of properties. Also Holzmann's supertrace algorithm throws information away. It allows the handling of a rather large set of properties, but gives only approximate answers. It is thus not a verification method but a validation and error detection method. All these methods are described in Section 7.2.

The construction of an ordinary state space is based on the execution of individual structural transitions in individual states, that is, the construction of semantic transitions one at a time. With packed state spaces this scheme has to be modified to at least some extent, because the processing of semantic transitions one by one would lead to a total amount of work of the order of the size of the full state space, which would largely destroy the benefits of packing. With the symmetry, coverability graph and supertrace methods, the construction of semantic transitions does not differ much from the construction of ordinary state spaces (except that semantic transitions are not stored when supertrace is used). On the other hand, when the set of reachable states is represented with a BDD, semantic transitions are handled with an entirely different technique that will be discussed in Section 7.2.

A packed state space that preserves full information on the behaviour of

a system must rely on some regularity in the behaviour, otherwise the packed representation could not be essentially smaller than an explicit representation. The symmetry method, for instance, relies on the assumption that the system consists of several components that are identical except the name or index of the component. Therefore, intuitively speaking, the storing of many behaviours can be replaced by a remark that says that those behaviours are the same as certain stored behaviours except the naming of entities.

The regularity on which a packed state space relies needs not be well-specified. For example, the type of regularity that BDDs rely on is very difficult to characterise.

On-the-fly verification. In *on-the-fly verification* the algorithm that checks the validity of a property is integrated to the algorithm that constructs the state space. The construction of the state space is stopped immediately when an error against the property is found. The state space is usually thrown away when the algorithm has finished. We have seen several on-the-fly algorithms and techniques already in Sections 4.2, 4.3 and 4.4.

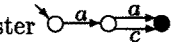
To appreciate the importance of on-the-fly verification, let us consider the introduction of an error to an originally correct system. In some cases the error makes the state space of the system smaller — this happens, for example, if it causes the system to deadlock right at the start. It is more common, however, that the error greatly increases the size of the state space. This is because a correct system almost always obeys some state invariant that significantly restricts the number of states that it can reach, and is violated by the broken system.

For instance, the alternating bit values of messages in a fifo channel of the well-known alternating bit protocol are not arbitrary, but they can change in at most one point within the fifo. If the capacity of the channel is k and the actual contents of the message are not modelled, arbitrary values of the alternating bits allow $2^0 + 2^1 + \cdots + 2^k = 2^{k+1} - 1$ different contents of the channel, whereas only $1 + 2 \cdot 1 + 2 \cdot 2 + \cdots + 2 \cdot k = k^2 + k + 1$ value combinations satisfy the invariant. Once the broken system has entered a state that violates the invariant, usually nothing stops it from slowly corrupting the state more and more, and thus eventually reaching a great number of states that are disallowed by the invariant.

It is obvious that stopping when an error is found does not speed up the verification of a correct system. On the other hand, we argued above that an incorrect system tends to have a much larger state space than the corresponding correct system. If the validity of the above invariant is monitored on the fly with a method that detects violations without delay, then analysis is stopped at the latest when one more state has been constructed than what the invariant permits. On-the-fly verification thus tends to reduce the number of states of the erroneous system back to roughly the same level with the correct system. This is of great help when testing design ideas during system development.

According to the above reasoning, the ability of an on-the-fly method to keep the state space small depends on how soon the method detects the error after

all states of an erroneous execution have been constructed. Because of this, we recommended in Section 4.2 the use of reject states and livelock monitor states for checking linear-time safety properties and livelocks, even if they could be checked also with Büchi automata (or infinite trace monitor states).

On-the-fly verification may reduce the number of states in also another way, by avoiding the construction of those parts of the state space that are not relevant for the property. This may be thought of as an instance of guided construction-time reduction, and it reduces states also when analysing a correct system. For the sake of an example, consider a tester process (Section 4.2) that is connected to the system with transition fusion. Assume that the alphabet of the system is $\Sigma = \{a, b, c\}$, and the property to be checked is "if the first visible action is a, then the next one is b". The 3-state tester  (the black state is a reject state) suffices for checking the property. Due to transition fusion, the tester prevents the system from executing b or c as the first visible action, thus pruning a potentially big part of the state space.

On-the-fly verification can often be combined to advanced state space methods of other kinds, and some advanced methods (most notably Holzmann's supertrace, Section 7.2) actually require the use of on-the-fly verification. However, the addition of extra components to the system that is required by most on-the-fly methods sometimes more or less destroys the ability of an advanced state space method to reduce states.

For instance, the addition of a Büchi automaton by synchronising with every structural transition of the system makes stubborn-set-type methods (Section 7.4) and process-algebraic compositionality (Section 7.3) more or less useless, because that requires the making of every transition visible, and (significant) reduction is obtained only from invisible transitions. This problem can be avoided by using an automaton that represents a stuttering-insensitive property, and either connecting it to only visible transitions (then one has to be careful with infinite stuttering, because the automaton does not even see stuttering), or synchronising it to each state and using a method that preserves the property without getting guidance from the automaton.

The lesson to be learnt here is that although it is often technically possible to use more than one advanced verification paradigm or method simultaneously, it may lead to bad results if done without good enough understanding of the situation.

7 Advanced State Space Methods

Many different methods for coping with state explosion have been suggested in the literature, and the field is still evolving rapidly. In this section the basic ideas of a dozen or so methods are described. The discussion concentrates on methods and ideas whose importance has already been proven by practical use, influence on ongoing research, or some other way. Unfortunately, it would have required too much effort to carefully assess the merits of all the numerous approaches suggested in the literature. It is thus possible that some important ideas may

be missing from this section. Stubborn-set-type methods are presented in more detail than other methods because there was a wish that this article would also serve as a stubborn set tutorial.

7.1 Preprocessing Methods

Every experienced user of state space tools knows that the way in which the system is modelled for the tool may have a great effect on the size of the state space. Some of the tricks that (assuming sufficient tool support) the modeller of the system can use to alleviate state explosion without affecting verification results are discussed in this section.

Elimination of remnant variable values. The state of a system is determined by the history of the system, and usually most of the state information affects the future behaviour of the system. With some systems it is, however, possible that some variable may contain several different values due to different histories, but all values lead to the same possible future behaviours. We might then say that the variable contains a *remnant value*.

A good example of remnant values can be found from within a communication protocol with a bounded number of retransmissions. The protocol sender process contains a counter for keeping track of transmission attempts. After the sender has received an acknowledgement, the counter will not be touched until the sender receives a new transmission request from the sending client. Then it resets the counter to zero. Thus the value of the retransmission counter is unnecessary between the reception of the acknowledgement and the start of the transmission of a new message.

Although remnant values do not affect future behaviour, they contribute to the global state of the system, and thus increase the number of reachable states. This effect can be prevented by adding to the verification model of the system statements that reset the variable to a specified value immediately after the value of the variable becomes unnecessary.

If the reset statements are added manually, there is the risk that a variable is reset although its value is not unnecessary, which may lead to incorrect verification results. Fortunately, this risk can be avoided if the input language of the verification tool contains the following feature. In addition to the "normal" values, the type of each variable should contain a special value undefined, and there should be a statement **uninitialise**(v) that assigns undefined to the variable v. Each attempt to use the value of v when it is undefined is immediately reported by the verification tool as a run-time error (like division by zero). Thus the modeller can add **uninitialise**-statements wherever the value of the variable is believed to be remnant, and the tool reveals if it was not remnant after all.

Coarsening of atomicity. The sizes of atomic actions have a great effect on the size of the state space of a system. As a consequence, system modellers often try to make the actions as large as possible. Unfortunately, careless coarsening

of atomicity often removes important behavioural errors that we would like to find with verification algorithms. A widely used way of avoiding this problem is to adhere to the following "critical reference" rule.

Let us say that a reference to a variable (or any other block of memory) that is shared by two or more parallel processes is *critical*, if and only if some other process can write to that variable (that is, change its value), or the reference is a writing reference and some other process has access to the variable. If a sequence of statements contains at most one critical reference, may be blocked only at the beginning, and does not contain a potentially non-terminating loop, then the sequence may be collapsed to one atomic action without affecting the answer to almost any verification question. There are special cases where this rule can be made more liberal. For instance, if only one process can write to an unbounded fifo and no process can test the emptiness of the fifo, then the writing reference needs not be considered critical.

The coarsening of atomicity often requires support from the modelling language. For instance, the *Promela* modelling language of the *Spin* tool contains an "**atomic**" statement with which an arbitrarily long sequence of successive transitions can be glued into one atomic transition [43]. If such a sequence is deterministic, then coarsening may be implemented in a state space construction tool by executing the whole sequence as one semantic transition. If the sequence contains a nondeterministic choice, then it may simplify the implementation to store the state where the choice is made. In this "intermediate" state no other transitions than those that belong to the process in question are investigated.

A simple technique that facilitates both the implementation of the above intermediate states and some other modelling tricks is to assign priorities to structural transitions. A low-priority transition may occur only if no higher-priority transition is enabled. An example of other uses of priorities is the modelling of the assumption that a timer does not occur prematurely, that is, while some other processing is still going on. This is achieved be giving the expiration of the timer lower priority than anything else.

The stubborn-set-type methods of Section 7.4 have a similar but usually much stronger effect than the coarsening of atomicity. Coarsening is thus more or less unnecessary when those methods are used. Coarsening is, however, much easier to implement, and, unlike stubborn-set-type methods, can be used with almost any verification question. Only questions that refer to states in the middle of a coarsened action or are sensitive to the disappearance of such states cannot be answered when actions are coarsened.

System transformations. The effect of the elimination of remnant variable values and coarsening of atomicity may to some extent be obtained by applying suitable transformations to the system model before the construction of the state space. Examples of such transformations of Petri nets are given in [40]. Unfortunately, system transformations tend to either preserve only a restricted set of behavioural properties, or contain complicated conditions on their use. An example of good use of net reductions is contained in [77].

System transformations work at the structural level. They cannot thus take advantage of situations where, for instance, a variable contains a remnant value (Section 7.1) but this fact cannot be seen by investigating the structure of the system. Process-algebraic equivalence-preserving reductions of state spaces (Section 7.3) can almost always do the same as system transformations and much more. Their use, however, requires that the state space of a suitably chosen part of the system is first constructed. Sometimes such a part cannot be identified, or has too big a state space. In such a situation process-algebraic reductions do not make system transformations unnecessary.

Data-independence. Many systems are *data-independent* in the sense that they just move data around without looking at it or modifying it. Most obvious examples of this are communication protocols and cache memories. In such a case it is clearly unnecessary for verification to model the data in full detail. However, if the model contains only one data value, then errors such as the swapping of messages in a protocol cannot be detected. This raises the question: what is the minimum (or at least a sufficient) number of data values for detecting all behavioural errors of a given data-independent system?

This question is usually answered with relatively simple manual reasoning. For instance, two different values suffice for detecting the swapping of two successive messages in a data-independent protocol. To see this, assume that the protocol may swap the nth and $(n+1)$th message in a sequence of m messages, where m may be infinite or any finite integer greater than n. (We have to give n and m this much freedom, unless we know that the possibility of swapping two messages does not depend on the total length of the sequence and the location of the swapping within it.)

Because of data-independence, the swapping may occur also in the sequence that consists of n messages of type m_1 and then $m - n$ (or infinitely many, if $m = \infty$) messages of type m_2. In that case the output of the protocol contains at least one instance of m_2 being immediately followed by m_1. On the other hand, such an output is impossible for the given input sequence if the protocol never swaps messages.

Swapping may thus be detected by adding to the system a data source process that gives the protocol an arbitrary sequence of the above kind, and a tester process that checks the output of the protocol. Only two values of messages are needed. State explosion is further reduced by the fact that the two values are not sent in arbitrary combinations. The error is detected on-the-fly which, as was discussed in Section 6, also helps to alleviate state explosion.

The theory of data-independence was carefully investigated in [99]. Recently, several theorems giving sufficient numbers of data values in CSP-type preorder verification were developed by Ranko Lazić; these results are summarised in [75]. A theory that allows the postponing of the instantiation of data values in process-algebraic compositional verification (Section 7.3) until doing it causes less state explosion was developed in [51].

Further remarks on modelling a system for verification. The size of a state space of a system is at worst proportional to the product of the numbers of different local states and values that its component processes and variables may have. For simplicity, we will discuss only the values of variables in the sequel. This does not imply loss of generality, because the set of local states of a process can be thought of as a variable whose value specifies the current local state.

The more a variable v contains "arbitrary" information that does not depend on the values of other variables, the bigger is the contribution of v to the size of the state space. In particular, if the value of v can be uniquely determined from the values of the other variables, then v is harmless from the point of view of state explosion. (Due to this, tester processes used in on-the-fly verification like in Section 4.2 do not usually increase the size of the state space very much.) Therefore, it is not essential to keep the number of variables small; what is essential is to ensure that the variables do not store arbitrary information.

It is common that a system contains more than one variable of some data type, and that the values of those variables may be partially unrelated. For instance, one of the variables may contain the value that is currently being processed, while another contains the previously processed value as a remnant value. In such a situation, making the data type grow affects the size of the state space more than linearly. Because of this, it is often important for state space methods to keep data types very small. This also explains why elimination of remnant values sometimes helps a lot in fighting state explosion.

The power of coarsening of atomic actions has a similar explanation. Coarsening reduces the extent to which component processes may proceed independently of each other and without affecting non-local variable values. It thus reduces significantly the amount of arbitrary information in component processes.

Many systems contain one or more parameters which may assume different values. Examples of such parameters are the number of philosophers in the dining philosophers system, and the window size, maximum number of retransmissions and queue capacities in a sliding window protocol. Strictly speaking, such "systems" are actually infinite *families* of similar systems of different size. Most state space methods require that all such parameters are given a constant value, and usually the values must be small to avoid state explosion. Fortunately, it seems that the majority of design errors manifest themselves already with small values of parameters. Therefore, verification with small parameter values can be used as an error detection and validation method for systems with large parameter values.

7.2 Methods Based on Packed State Spaces

Holzmann's supertrace. Most modelling formalisms allow the representation of the state of the system as a bit vector of fixed length, at least if integers, queues, etc. are replaced with bounded data types such as 8-bit integers and queues of fixed capacity. A bit vector of length k has 2^k different value combinations. These combinations comprise a "universe" U of all "syntactically possible" states, of which the set S of reachable states is a subset. It is common that only

a small fragment of syntactically possible states are reachable. Consider the n dining philosophers system in Figure 1. The state of a philosopher can be encoded to two bits, and the state of a fork to one bit. This makes a total of $3n$ bits and thus $2^{3n} = 8^n$ value combinations. However, only $3^n - 1$ of them are reachable. We can say that S is usually *sparse*.

The most memory-efficient way of storing an entirely arbitrary subset X of a set $U = \{u_1, u_2, \ldots, u_n\}$ of n elements is a bit vector of length n, where the value of the ith bit tells whether $u_i \in X$. To avoid confusion with bit vectors that represent states, we will use the term *bitset* of a bit vector that stores a set. With sparse sets, the bitset approach leads to the storing of numerous "no" bits, making the explicit enumeration of all elements of X as the keys of the records of some data structure (such as a linked list, hash table, or binary tree) much more memory-efficient. For instance, the state of the 10 philosophers system fits 30 bits or 4 bytes. If we assume that the data structure requires two additional 4-byte pointers per record, the total amount of memory becomes 59 048 states $\times$ 12 bytes/state $\approx$ 700 kilobytes. The bitset representation requires $2^{30}/8 =$ 128 megabytes.

Because of this, the majority of traditional state space tools store states in explicit form. The use of bitsets is possible only if the state can be encoded to a number of bits that is at most the base 2 logarithm of the amount of available memory in bits.

The classic state space construction algorithm needs to store two sets of states: the states *Found* that have been found so far, and its subset *Incomplete* consisting of those states that have been found but not yet fully processed. When the algorithm is run, *Found* keeps on growing until it contains all reachable states, while *Incomplete* alternates between growing and shrinking. It is rather common that the size of *Incomplete* stays small compared to the set of reachable states. Thus the bottleneck is the storing of *Found*.

The basic idea of Holzmann's *supertrace* algorithm [43] is to replace *Found* with an approximation that is obtained by artificially mapping the set U of all syntactically possible states to a small enough set H, and store the approximation as a bitset $H[]$. The mapping is implemented as a hash function $h : U \mapsto H$ that takes the original representation of a state s as a long bit vector and produces a short enough bit vector $h(s)$. The test $s \in$ *Found* is replaced by the test $H[h(s)] = 1$.

Because $|H| < |U|$ (and $|H| << |U|$), the risk of two different reachable states mapping to the same element of H (hash collisions, in other words) cannot be eliminated. As a consequence, every now and then the supertrace algorithm treats a newly found state as an old state and fails to add it to *Incomplete*. This causes the algorithm to ignore the output transitions of some or many states, and leave a part of the state space uninvestigated. Supertrace is thus *not* a verification algorithm, but a validation and error detection algorithm.

The bitset approximation of *Found* is not particularly useful after supertrace has terminated, because it is neither a lower (an unreachable state may hash to a marked bit) nor an upper (supertrace did not investigate all reachable

states) approximation of the set of reachable states. It is difficult to think of any way how supertrace could store reasonable additional information on the sets of states and semantic transitions that it has investigated without increasing the memory requirements at least by an order of magnitude. Therefore, with supertrace, errors must be detected on the fly. Fortunately, as we saw in Section 4.2, violations against linear-time safety properties, linear-time livelock properties and even arbitrary LTL properties can be detected on-the-fly. Supertrace has thus wide applicability.

Perhaps the most important advantage of supertrace is that it can be used almost independently of the size of the state space of the system in question and the amount of available memory. Supertrace never investigates more states than fits the bitset. This also sets an upper limit to the time consumption of supertrace. If the bitset is small, hash collisions start to occur early in the analysis, and only a small fraction of the state space is investigated. If more memory is given, the bitset may be made larger, leading to the investigation of a bigger part of the state space.

In this way supertrace almost always gives an answer within the resources given to it, and the quality of the answer improves if the resources are increased. Furthermore, set membership can be tested and a new element added very efficiently with bitsets, so supertrace uses the resources given to it in an exceptionally efficient way. It may, however, run out of memory, because the set *Incomplete* is stored in the ordinary way, and its maximum size may be difficult to predict in advance.

Supertrace is thus not a verification technique, but it is a very pragmatic validation and error detection technique. In addition, it is easy to implement.

Petri net coverability graphs. A marking M' of a place/transition net *covers* a marking M, denoted by $M' \geq M$, if and only if $M'(p) \geq M(p)$ for every place p. The notation $M' > M$ means that $M' \geq M$ and $M' \neq M$. If M' covers M and $M\,[t\rangle\,M_1$, then there is a marking M_1' such that $M'\,[t\rangle\,M_1'$ and M_1' covers M_1. As a matter of fact, $M_1' - M_1 = M' - M$, where $M_x - M_y$ denotes the function from places to integers such that $(M_x - M_y)(p) = M_x(p) - M_y(p)$ for every place p. This fact implies that if $M_0\,[t_1 t_2 \ldots t_n\rangle\,M_1$ and $M_1 \geq M_0$, then $M_1\,[t_1 t_2 \ldots t_n\rangle\,M_2\,[t_1 t_2 \ldots t_n\rangle\,M_3\,[t_1 t_2 \ldots t_n\rangle \cdots$, where $M_k = M_0 + k(M_1 - M_0)$. If, furthermore, $M_0 \in [M_I\rangle$ and p is a place such that $M_1(p) > M_0(p)$, then the number of tokens in p may grow without limit, and $[M_I\rangle$ is infinite.

On the other hand, if a place/transition net has a finite number of places and transitions but an infinite number of reachable markings, then it can be proven to have an infinite execution with infinitely many different markings. Furthermore, any such execution has a marking M_0 and a later marking M_1 such that $M_1 > M_0$.

Let us use the term *ω-marking* of any function from the set of places to $\{0, 1, 2, \ldots\} \cup \{\omega\}$, where ω is a special symbol that denotes unboundedness. The notion of covering can be extended to ω-markings by defining that $i < \omega$ (and thus $i \leq \omega$) for every integer i. An ω-marking M_ω denotes a set $[[M_\omega]]$ of

ordinary markings such that

- for every place p and every $M \in [[M_\omega]]$, if $M_\omega(p) \neq \omega$, then $M(p) = M_\omega(p)$; and
- for every ordinary marking M such that $M \leq M_\omega$, $[[M_\omega]]$ contains an ordinary marking M' such that $M \leq M'$.

The set denoted by an ω-marking M_ω that contains no ω-symbols is thus $\{M_\omega\}$. If M_ω contains ω-symbols, $[[M_\omega]]$ is not uniquely defined, but it is certainly infinite. If M and M' are ordinary markings, then $M \leq M_\omega \Leftrightarrow \exists M' \in [[M_\omega]] : M \leq M'$.

What it means for a transition to be enabled in an ω-marking, and what is the result of its occurrence are defined like with place/transition nets, except that a place marked with ω always contains enough tokens, and is marked with ω also after the transition occurrence. This implies that if M_ω contains at least one ω-symbol, then $M_\omega\ [t\rangle\ M'_\omega$ represents an infinite number of occurrences of t from a marking in $[[M_\omega]]$ to a marking in $[[M'_\omega]]$.

A *coverability graph* of a place/transition net is constructed like the ordinary state space with the following exception. Each time it is found out that a newly constructed ω-marking (remember that also ordinary markings are ω-markings) M' covers and is reachable from an older ω-marking M, M' is replaced by the unique ω-marking M_ω such that $M \in [[M_\omega]]$, $M' \in [[M_\omega]]$, and M_ω contains as few ω-symbols as possible. The details of when and to which ω-markings M' is compared may vary, and it is also possible to replace M_ω for M instead of or in addition to M'. To guarantee termination it suffices that each newly constructed M' is compared to each ω-marking in the path along which M' was found. Figure 8 shows an example of a Petri net and its coverability graph.

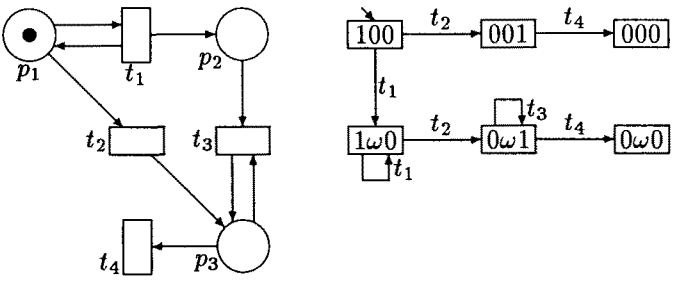


Fig. 8. A coverability graph example.

Coverability graphs can be easily used to detect unbounded places of a place/transition net. Because the set represented by an ω-marking is not uniquely defined, coverability graphs cannot be used for checking the reachability of a

marking. For instance, if the weight of the arc from t_1 to p_2 were changed to 2 in Figure 8, then the coverability graph would not change, but the marking 110 would become unreachable. This implies that there are simple state-based properties that cannot be verified from coverability graphs. On the other hand, coverability graphs can be used for checking reachability of a state where a transition is enabled. As was discussed in Section 4.2, this suffices for checking a large set of action-based linear-time safety properties.

Regarding liveness properties, the example net cannot execute t_3 an infinite number of times. However, if a new arc were drawn from t_3 to p_2, then t_3 would become infinitely executable without any change in the coverability graph. Thus coverability graphs throw away essential information regarding the verification of liveness properties.

Coverability graphs are a technique for handling infinite state spaces. They are of no help if the state space is known to be finite. A coverability graph of a Petri net with a finite number of reachable markings is the same as its state space.

With coverability graphs it is possible to answer many questions that are undecidable for Turing machines, such as the possibility of non-termination. This implies that place/transition nets are computationally strictly weaker than Turing machines, and that the coverability graph construction cannot be fully transformed to Turing-strong formalisms such as automata that communicate through unbounded fifo-queues. On the other hand, the ideas of coverability graphs could perhaps be of use also with fifo systems. They provide an incomplete test for detecting that the state space is infinite, and allow some reasoning from an infinite state space. Because of undecidability, the coverability graph of a fifo system will every now and then be infinite and thus impossible to construct, but state spaces that are too big to be constructed have always been a part of everyday life with state space methods.

More information on coverability graphs can be found in many books on the theory or analysis of Petri nets. The usual definition of coverability graphs is somewhat liberal, making it possible to obtain different coverability graphs from the same Petri net. The article [27] defines and analyses *minimal* coverability graphs and shows how they can be constructed. The minimal coverability graph of a Petri net is unique.

The symmetry method. Many systems exhibit symmetry in one form or another. For instance, a system may contain several identical components that are coupled to each other and the rest of the system in a regular way. Consider a system with the syntactically possible states U and the state space (S, T, Δ, S_I). The notion of symmetry may be formalised with the aid of a set of bijections $f : (U \cup T) \mapsto (U \cup T)$ that have for every states s and s' and structural transition t the properties that $f(s) \in U$, $f(t) \in T$ and $s \,{-t\!\rightarrow}\, s' \Leftrightarrow f(s) \,{-f(t)\!\rightarrow}\, f(s)'$. Furthermore, if $s \in S_I$, then also $f(s) \in S_I$ and vice versa. We may call such bijections *symmetry bijections*.

It is clear that the identity function $Id : (U \cup T) \mapsto (U \cup T) : Id(x) = x$

is a symmetry bijection. Furthermore, if f and g are symmetry bijections, then so are also the inverse f^{-1} of f and the function composition $g \circ f$ defined by $(g \circ f)(x) = g(f(x))$. Let B be a nonempty set of symmetry bijections, and assume that B^* is the set of all symmetry bijections that can be constructed from the elements of B by repeated application of the inverse and/or function composition operations. For instance, in the case of a token-ring protocol, if B contains only one element and that element is the rotation of the ring one step to the right, then B^* is the set of all rotations of the protocol. If B consists of all possible ways of swapping two clients in a star-shaped client-server system with one server and many clients, then B^* is the set of all permutations of the client processes.

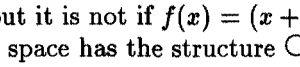
The pair $(B^*, \circ)$ is a mathematical *group*, and the relation $s \simeq s' \stackrel{\text{def}}{\Longleftrightarrow} \exists f \in B^* : s' = f(s)$ is an equivalence. Let us denote its equivalence classes with $[[s]]$. That is, $[[s]] = \{ f(s) \mid f \in B^* \}$. It follows from the above definitions that if any state in $[[s]]$ is reachable, then every state in $[[s]]$ is reachable. Then we may say that $[[s]]$ is reachable.

The basic idea of the reduction of state spaces with symmetries is to store only one state from each reachable $[[s]]$. We may call that state the *representative* for $[[s]]$. If s has been stored and $s \xrightarrow{t} s'$, then an edge is stored that starts from s, is labelled with t, and ends in the state s_r that represents $[[s']]$. (Some versions of the symmetry method store as an additional label of the edge the symmetry bijection h or h^{-1} such that $s_r = h(s')$.) It may or may not be the case that $s_r = s'$. This means that for every reachable semantic transition (s, t, s'), the symmetry method stores the semantic transition $\big(f(s), f(t), g(s')\big)$, where f and g are some symmetry bijections. Furthermore, if (s, t, s') is an edge in the symmetry state space, then there is a symmetry bijection g such that $\big(f(s), f(t), f(g(s'))\big)$ is a reachable semantic transition for every symmetry bijection f. Therefore, assuming that the set of symmetry bijections is known, the symmetry method preserves full information on the reachable part of the state space.

We pointed out in Section 5.3 that even if a packed state space preserves full information on the behaviour, it is not guaranteed that answers to verification questions can be extracted efficiently enough. Fortunately, if each $[[s]]$ is finite (which is certainly the case if the ordinary state space of the system is finite), then the symmetry method can be used to verify all LTL properties with the on-the-fly techniques in Section 4.2. In order to not break the symmetry and thus invalidate the symmetry method, one has to add to the system the necessary number of symmetric copies of the fact transition, Büchi automaton, or tester process with which the property is represented. Then errors can be detected with reject states, deadlock monitor states, livelock monitor states and Büchi acceptance states (or infinite trace monitor states) exactly like with ordinary state spaces. We will now show that this is the case with Büchi acceptance; the other forms of error detection can be handled similarly (and do not need the finiteness assumption in the case of reject and deadlock monitor states).

Consider an infinite execution $s_0 \xrightarrow{t_1} s_1 \xrightarrow{t_2} \cdots$ that goes infinitely

many times through some Büchi acceptance state s_B. The symmetry state space contains the path $f_0(s_0) \xrightarrow{-f_0(t_1)} f_1(s_1) \xrightarrow{-f_1(t_2)} \cdots$, where f_0, f_1, ... are some symmetry bijections. This path contains infinitely many occurrences of elements of $[[s_B]]$. Because $[[s_B]]$ is finite, at least one element in it occurs infinitely many times, and the path $f_0(s_0) \xrightarrow{-f_0(t_1)} f_1(s_1) \xrightarrow{-f_1(t_2)} \cdots$ is Büchi-accepted. If, on the other hand, the symmetry state space contains a path $s_0 \xrightarrow{-t_1} s_1 \xrightarrow{-t_2} \cdots$ where some Büchi acceptance state s_B occurs infinitely many times, then $f_0(s_0) \xrightarrow{-f_0(t_1)} f_1(s_1) \xrightarrow{-f_1(t_2)} \cdots$ is an execution for at least some symmetry bijections f_0, f_1, ... Due to the same reason as above, some $s \in [[s_B]]$ occurs infinitely many times in that execution. Let f be the symmetry bijection that maps s to its corresponding acceptance state in the Büchi automaton that represents the original property. Then $f(f_0(s_0)) \xrightarrow{-f(f_0(t_1))} f(f_1(s_1)) \xrightarrow{-f(f_1(t_2))} \cdots$ is an execution that violates the property.

Efficient verification of proper branching-time properties with the symmetry method is possible, but less straightforward. The problem can be illustrated with the net  , where x may get the values 0, 1, 2 and 3. If $f(x) = (x + 1) \bmod 4$, then the occurrence mode $x = 1$ of the rightmost transition is Petri-net-live, but it is not if $f(x) = (x + 2) \bmod 4$. However, in both cases the symmetry state space has the structure  . To decide Petri-net-liveness it is thus necessary to look at the actual symmetry bijections in addition to the structure of the symmetry state space, making verification more difficult.

The state space reduction power of the symmetry method depends crucially on the set of symmetry bijections that is used. The application of rotation symmetry to a ring-like system of n components can divide the number of states by at most n, which is usually not sufficient for curing state explosion. On the other hand, the symmetry consisting of all permutations can give much better reduction results.

The addition of (all symmetric copies of) a Büchi automaton or tester process to a system increases the amount of information that is stored of each symmetric component. States that would be symmetric without the extra information may become asymmetric in the presence of the extra information. Because of this effect, on-the-fly verification with symmetries does not always work well. This effect can be fought against when designing the automaton by paying careful attention to the amount and nature of the information that it stores.

The construction of symmetry state spaces and their use in the verification of Petri nets has been discussed in [46]. The symmetry method was applied to CTL* model checking in [12, 22], of which [12] used also BDDs. The use of symmetries when model checking with Büchi automata in the presence of fairness assumptions was investigated in [23, 39]. The combination of the symmetry method with the stubborn set method (Section 7.4) was investigated in [84, 53] with a small number of case studies, and was found to be better than either method alone.

The unfolding method. Consider an ordinary Petri net $\mathcal{N}$ that has the property that $M(p) \leq 1$ for every place p in every reachable marking M. Let M_I be an initial marking of $\mathcal{N}$. The *unfolding* of $\mathcal{N}$ from M_I is a (usually infinite) unmarked acyclic Petri net $\mathcal{U}$ that represents the truly concurrent behaviour of $\mathcal{N}$ in a certain way. Each place b of $\mathcal{U}$ corresponds to some place $fb(b)$ (b "*f*olded *b*ack") of $\mathcal{N}$, and similarly for transitions. The unfolding of $\mathcal{N}$ is obtained as follows.

To start with, a place b_i^I is added to $\mathcal{U}$ for each place p_i of $\mathcal{N}$ that is marked in M_I, and, naturally, $fb(b_i^I)$ is chosen to be p_i. We will call these places the *initial places* of $\mathcal{U}$. Then new places and transitions are added according to the following rule as long as possible. Let t be a transition of $\mathcal{N}$, and let the numbers of input and output places of t be denoted by k and h. If $b_1, \ldots, b_k$ are places of $\mathcal{U}$ such that they are *concurrent* in the sense explained soon, and $fb(b_1), \ldots, fb(b_k)$ are exactly the input places of t, then a new transition e and new places $b'_1, \ldots, b'_h$ are added, and arcs are drawn from each b_i to e and from e to each b'_j, provided that such a transition and places have not already been added. The function fb is extended to the newly added elements such that $fb(e) = t$ and $\{fb(b'_1), \ldots, fb(b'_h)\}$ is the set of the output places of t.

In the unfolding, a place can never have more than one input transition. Furthermore, every place is reachable from at least one initial place. Let b_1 and b_2 be two different places of $\mathcal{U}$. They are *causally related*, if and only if there is a path from one of them to the other. They are in *conflict*, if and only if $\mathcal{U}$ has a place b such that $b \neq b_1$, $b \neq b_2$, and there is a path from b to b_1 and another path from b to b_2 such that the paths have nothing else in common than b. Finally, the places $b_1, \ldots, b_k$ of $\mathcal{U}$ are *concurrent*, if and only if no two of them are causally related or in conflict. It follows from the construction of $\mathcal{U}$ that no two of its places can simultaneously be causally related and in conflict.

Let $\mathcal{M}_I$ be the marking of $\mathcal{U}$, where each initial place contains exactly one token, and the remaining places contain no tokens. Consider the set of markings that $\mathcal{U}$ can reach starting from $\mathcal{M}_I$. If $\mathcal{M} \in [\mathcal{M}_I\rangle$ is the marking where exactly the places $b_1, \ldots, b_n$ are marked, then we let $fb(\mathcal{M})$ denote the marking of $\mathcal{N}$ where exactly $fb(b_1), \ldots, fb(b_n)$ are marked. For instance, $fb(\mathcal{M}_I) = M_I$ (that is, the initial marking of $\mathcal{N}$). The construction of $\mathcal{U}$ guarantees that if $\mathcal{M} \in [\mathcal{M}_I\rangle$, then $fb(\mathcal{M})$ is a reachable marking of $\mathcal{N}$. If, furthermore, $\mathcal{M}\,[e\rangle\,\mathcal{M}'$, then $fb(\mathcal{M})\,[fb(e)\rangle\,fb(\mathcal{M}')$.

In the reverse direction, if M is a reachable marking of $\mathcal{N}$, then $\mathcal{U}$ has a reachable marking $\mathcal{M}$ such that $fb(\mathcal{M}) = M$. If, furthermore, $M\,[t\rangle\,M'$, then $\mathcal{U}$ has a transition e and a marking $\mathcal{M}'$ such that $\mathcal{M}\,[e\rangle\,\mathcal{M}'$, $fb(e) = t$ and $fb(\mathcal{M}') = M'$. (To see that this holds it is helpful to first notice that the property "all simultaneously marked places are concurrent" is an invariant property of $\mathcal{U}$ with the initial marking $\mathcal{M}_I$.) Thus $\mathcal{U}$ contains complete information on the reachable part of the state space of $\mathcal{N}$.

The unfolding $\mathcal{U}$ is usually infinite and cannot be constructed as such. However, if $\mathcal{N}$ is finite, then $\mathcal{U}$ has at least one finite prefix that represents all reachable states of $\mathcal{N}$. An algorithm for constructing one such prefix was presented

in [62]. Its result — let us call it the *finite unfolding* and denote it with $\mathcal{U}^{\text{fin}}$ — may be much smaller than the state space of $\mathcal{N}$. Some properties of $\mathcal{N}$ can be checked easily from $\mathcal{U}^{\text{fin}}$, such as the reachability of a state where a certain transition is enabled. Unfortunately, the checking of many seemingly simple properties from $\mathcal{U}^{\text{fin}}$ is quite hard, as the following example from [62] demonstrates.

Let $\varphi = \varphi_1 \wedge \cdots \wedge \varphi_n$ be a Boolean formula in 3-conjunctive normal form (that is, each φ_j is a disjunction of three different *literals*, where a literal is either a variable symbol or a negated variable symbol). Let the variables that occur in φ be $v_1, \ldots, v_m$. We will now construct a Petri net $\mathcal{N}_\varphi$ from φ. For each variable v_i the net contains two transitions t_i^{F} and t_i^{T} and three places p_i^I, p_i^{F} and p_i^{T}. There is an arc from p_i^I to t_i^{F}, from t_i^{F} to p_i^{F}, from p_i^I to t_i^{T}, and from t_i^{T} to p_i^{T}. For each conjunct $\varphi_j = l_{j,1} \vee l_{j,2} \vee l_{j,3}$ there are two transitions t_j^φ and t_j^∞ and a place p_j^φ. The place p_j^φ has input arcs from both t_j^φ and t_j^∞, and an output arc to t_j^∞. Finally, for each $k \in \{1, 2, 3\}$, t_j^φ has an input arc from p_i^{F} if and only if $l_{j,k} = v_i$, and p_i^{T} if and only if $l_{j,k} = \neg v_i$ (here "$\neg$" indeed goes together with "T"). The places p_i^I are initially marked, and the other places are not.

The unfolding of $\mathcal{N}_\varphi$ is the same as $\mathcal{N}_\varphi$ without the initial marking, except that each p_j^φ-t_j^∞-pair is replaced by an infinite linear chain of places and transitions. By cutting these chains after the second place, a finite unfolding $\mathcal{U}_\varphi^{\text{fin}}$ is obtained that represents all reachable markings of $\mathcal{N}_\varphi$. It can thus be constructed from φ in polynomial time.

Consider any marking where for each i, either p_i^{F} or p_i^{T} is marked, and the remaining places are empty. This marking is clearly reachable from the initial marking of $\mathcal{N}_\varphi$. If that marking corresponds to an assignment of truth values to $v_1, \ldots, v_m$ that makes φ false, then at least one φ_j is false, and t_j^φ is enabled. Otherwise all φ_j are true, and $\mathcal{N}_\varphi$ is in a deadlock. It is clear that in all other reachable markings some t_i^{T} or some t_j^∞ is enabled. Therefore, $\mathcal{N}_\varphi$ may deadlock if and only if φ is satisfiable. As a consequence, detecting deadlock from $\mathcal{N}_\varphi$, and thus also from $\mathcal{U}_\varphi^{\text{fin}}$, is NP-hard. So the existence of a worst-case polynomial time algorithm for deadlock detection from the finite unfolding is very unlikely.

On the other hand, it is trivial to check reachability of a deadlock from the state space of a system in linear time — linear *in the size of the state space*. In the case of $\mathcal{N}_\varphi$ (and many other systems, for that matter), linear in the size of the state space is exponential in the size of the system, because the state space clearly has an exponential number of states. So we see that the detection of deadlocks (most likely) contains an exponential step both with unfoldings and with state spaces, it is just that with unfoldings this step is after and with state spaces during the construction of the object that represents the behaviour of the system. As a matter of fact, making the very likely assumption that NP $\neq$ PSPACE, *every* deadlock detection method must contain a step that cannot be performed in (even nondeterministic) polynomial time.

The difficulty of finding a deadlock from the finite unfolding is thus a consequence of the facts that finding a deadlock is hard with *any* method, and, for a significant class of systems, the unfolding is small and can be constructed in polynomial time. We can interpret this by saying that the finite unfolding is

an "intermediate" representation of behaviour that is between a Petri net and its state space. Because it is sometimes much smaller than the state space, and not everything is hard to check from it, it provides answers to some questions on some systems with much less time and memory than state spaces do. There are, however, many systems for which the finite unfolding is of exponential size. This is the case at least when an exponential number of transition occurrences is needed before a certain transition becomes enabled. A binary counter is a simple example.

For the practical use of unfoldings, it is important to distinguish properties that can be checked easily and efficiently enough. Although deadlock detection is hard in the worst case, [62] presents a heuristic algorithm that works often well. An alternative algorithm that uses integer linear-algebraic techniques was given in [64].

An algorithm for checking the validity of formulae in a certain simple temporal logic from the finite unfolding was developed in [24]. The logic consists of atomic propositions of the form "$M(p) = 1$", the Boolean connectives "$\wedge$" and "$\neg$", and the CTL operator "EF" (from which "$\vee$" and "AG" can be constructed). The publication [24] also distinguishes a non-trivial class of Petri nets for which the finite unfolding is always small and constructible in polynomial time, yielding polynomial time verification algorithms for formulae of the above logic that are in a certain form. (Of course, the binary counter cannot be modelled with such Petri nets, and the conversion of an arbitrary formula to the required form may increase its length exponentially.)

The original algorithm [62] for constructing a finite unfolding may produce an unnecessarily large result if the Petri net contains lots of conflicts, leading sometimes to finite unfoldings that are significantly bigger than the corresponding state spaces. An improved algorithm that avoids this problem was presented in [25].

Binary decision diagrams. An *(ordered) binary decision diagram* or *(O)BDD* [8] is a data structure for representing a set of bit vectors of equal length or, equivalently, a Boolean formula. It is a directed acyclic graph whose each vertex has either zero or exactly two successor vertices. Vertices with no output edges are labelled by "F" or "T". Each of the remaining vertices is labelled by a variable, and its output edges are labelled by "0" and "1". Exactly one vertex, the *root*, has no incoming edge. Figure 9 shows a BDD that represents the set $\{0011, 0111, 1011, 1100, 1101, 1110, 1111\}$, or the formula $(v_1 \wedge v_2) \vee (v_3 \wedge v_4)$. A vector $v_1v_2v_3v_4$ is in the set if and only if the corresponding path from the root down through the BDD ends with "T", where the "corresponding" path is the one where the output edge from each node v_i is selected according to the value of v_i.

BDDs are usually made as small as possible by merging any two nodes that are roots of isomorphic sub-BDDs, and removing all nodes whose both output edges lead to isomorphic sub-BDDs. This can be done very efficiently by working bottom-up. The ordering in which the variables occur in a BDD is fixed.

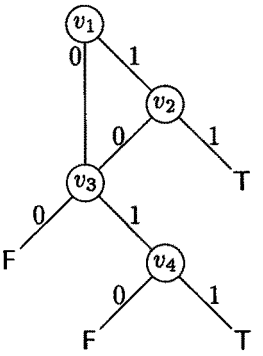


Fig. 9. A BDD example: $(v_1 \wedge v_2) \vee (v_3 \wedge v_4)$.

This has the consequence that each set has a unique fully reduced BDD representation and, furthermore, basic set operations including union, intersection, complementation, and equivalence test can be done efficiently with fully reduced BDDs. On the other hand, the size of a BDD (that is, the number of nodes in it) may depend crucially on the ordering of the variables, and it is difficult to know in advance whether a particular ordering would be good.

Several BDDs over the same set of variables and with the same ordering of those variables can be represented efficiently in one data structure by sharing identical sub-BDDs. This is useful in algorithms that manipulate BDDs.

With BDDs, it is natural to think of each structural transition t as defining a relation in $S \times S$ or, if the structural transitions are deterministic, a partial function $S \mapsto S$. This relation can be represented as a Boolean formula $R_t(s, s')$ where s and s' are the state before and after the occurrence of t. In these terms, the enabling condition of t is $\exists s' : R_t(s, s')$. The set of all structural transitions corresponds to the formula $R(s, s') \stackrel{\text{def.}}{\Longleftrightarrow} \bigvee_{t \in T} R_t(s, s')$.

Assuming that the formula $S(s)$ describes some set of states, the formula $\exists s' : S(s') \wedge R(s', s)$ describes the set of those states that are reachable from the present set with one transition occurrence. A formula representing the set of all reachable states may be constructed by starting with a formula $S(s)$ that represents the set of initial states, and repeating the operation $S(s) := S(s) \vee \exists s' : S(s') \wedge R(s', s)$ until $S(s)$ does not change any more. (Alternatively, one can restrict the application of $R(s', s)$ to those reachable s' to which it has not yet been applied, or those that the previous application of $R(s', s)$ produced.) All these formulae can be represented with BDDs, and the operations needed in this approach can be performed efficiently in the sizes of the BDDs.

When the set $S(s)$ of reachable states is obtained, one can check whether it contains a forbidden state by describing forbidden states with a formula $F(s)$, computing $S(s) \wedge F(s)$, and checking whether the result is False. Other properties

may be checked with more complicated algorithms, such as the algorithm for Petri net liveness in [67]. Perhaps the most important such algorithm is the *symbolic model checking* algorithm for CTL, LTL, weak bisimilarity and some other properties described in [9]. In it, the construction of the BDD proceeds backwards guided by the formula that is being checked. A BDD representing the set $G(s)$ of syntactically possible states that satisfy the formula is obtained as a result. One can then compute $I(s) \wedge \neg G(s)$ where $I(s)$ models the set of initial states to verify that the system satisfies the formula.

As was mentioned in Section 6, to obtain a small BDD it is necessary that the set of the reachable states of a system is regular in some informal and not well understood sense. (In the case of symbolic model checking it is the set of syntactically possible states that satisfy the formula that must be regular.) On the other hand, the total number of states is seldom an important factor of the BDD size. As a consequence, intermediate BDDs may be much larger than the final BDD in the above algorithm for constructing a BDD for the set of reachable states. For instance, [67] contains case studies where the biggest BDD has more than ten times as many nodes as the final BDD. Furthermore, systems exist which have no small BDDs. A combinatorial multiplier circuit is an often cited example. This problem has been attacked by developing variants of BDDs.

It is unclear whether it is useful to combine BDDs to other state space reduction methods. Most methods discussed in this article aim at reducing the number of states, but the number of states is not important with BDDs. On the other hand, the simultaneous use of another method may well pay off, if it makes the set of states more regular in the above informal sense. For instance, [3] contains a successful example of the combination of BDDs to a stubborn-set-type method.

BDDs have found lots of use especially in circuit design, as can be seen from the survey [63].

7.3 Methods Based on Process-Algebraic Compositionality

Compositional LTS construction. The goal of *compositional LTS construction* is to construct a reduced state space that is equivalent to the full state space in the sense of some process equivalence. The method is usually applied to process-algebraic verification of the kind in Section 4.4, but it can also be used for state-based verification.

The method can be illustrated with the example in Figure 10. The ordinary LTS of the example system can be computed according to the process-algebraic expression

$$P = \mathbf{hide}\, u, v, w, x, y\, \mathbf{in} (P_1 || P_2 || P_3 || P_4).$$

Due to the properties of hiding and parallel composition, the expression

$$\mathbf{hide}\, w, x\, \mathbf{in}\, (\mathbf{hide}\, u, v\, \mathbf{in} (P_1 || P_3) \,||\, \mathbf{hide}\, y\, \mathbf{in} (P_2 || P_4))$$

produces the same result (excluding the names of states). Let $\simeq$ be a process equivalence that is a congruence with respect to "**hide**" and "||". If *Red* is an algorithm that reduces an LTS but preserves $\simeq$, then $P \simeq P_{\mathbf{red}}$, where

$$P_{\mathbf{red}} = \mathbf{hide}\ w, x\ \mathbf{in}\big(\mathit{Red}(\mathbf{hide}\ u, v\ \mathbf{in}(P_1||P_3))\,||\,\mathit{Red}(\mathbf{hide}\ y\ \mathbf{in}(P_2||P_4))\big).$$

Thus $P_{\mathbf{red}}$ can be used for verifying all the properties that are preserved by $\simeq$. On the other hand, the LTS $P_{\mathbf{red}}$ may be much smaller than the LTS P. (Suitable equivalences $\simeq$and reduction algorithms *Red* were discussed in Section 4.4).

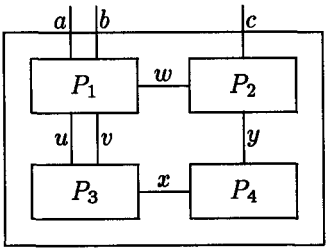


Fig. 10. A system consisting of four processes.

Compositional LTS construction can be applied hierarchically. For instance, if P_1 consists of several processes, then a compositionally constructed reduced LTS of it can be used in its place. Even if P_1 is a single process, it may be worthwhile to replace it by its reduced LTS.

Compositionality is inherent in process-algebraic theories, so it is difficult to give anyone credit of inventing compositional LTS construction. An early article where its use was explicitly suggested in a modern form is [58]. The article [90] is a tutorial on compositional LTS construction.

Compositional LTS construction is most often used for the verification of action-based properties of systems with synchronous interprocess communication, but it can be used also for state-based properties and with shared-variable communication. Shared variables can be handled by thinking of them as processes in their own right. Statements that read, test or modify the values of a shared variable are interpreted as actions that synchronise with the variable. When all processes that touch a shared variable have been added to the system, a special *initialisation* operator is used to give the variable its initial value and hide it from the interface of the (sub)system. This trick was applied to Petri net places in [88].

State-based properties can be handled by encoding state properties into action names as was described in Section 2.2. It is also possible to proceed as follows [88, 89]. Consider a state-based temporal logic formula φ such that its atomic propositions refer to the local states of processes $P_1, \ldots, P_n$, but not to the local states of processes $Q_1, \ldots, Q_m$. Then the validity of φ on $P_1|| \ldots ||P_n\ ||$

$Q_1||\ldots||Q_m$ can be checked by compositionally constructing a reduced LTS Q_{red} of $Q_1||\ldots||Q_m$, and checking the validity of φ on $P_1||\ldots||P_n \,||\, Q_{\mathsf{red}}$. Of course, the equivalence that is preserved by the reduction must preserve the temporal logic in which φ was written. NDFD-equivalence preserves $\mathrm{LTL}_{-\mathsf{X}}$ and branching bisimilarity has a variant that preserves $\mathrm{CTL}_{-\mathsf{X}}$ and even $\mathrm{CTL}^*_{-\mathsf{X}}$.

The interface processes of Graf and Steffen. The ability of compositional LTS construction to save effort relies on the assumption that the intermediate LTSs are smaller than the full LTS of the whole system. This is not always the case. A typical example is a token-ring protocol where one token circulates in a ring consisting of n processes. To keep the example simple to discuss, we assume that a process without a token can be in one and the process with the token in two different states. Then the system as a whole has only $2n$ different states. The number of states is that small, because the system maintains the property that there is only one token in the ring. Any process that is not in the possession of the token is willing to input the token, but in all but one case the previous process in the ring does not have the token and is thus unable to give it.

Consider then an open-ended segment of the token ring consisting of $k < n$ processes. The input actions of the first process of the segment are now connected to nowhere. Therefore, if the first process does not have a token, then it can input one even if some other process in the segment has a token. As a consequence, the segment may contain up to k tokens, and may be in 3^k different states. Even for relatively small values of k, 3^k is much bigger than $2n$.

The problem in this example is that a subsystem that is isolated from its proper context may exhibit lots of "spurious" behaviour that it does not have when it is a part of the system as a whole. This problem was pointed out in [38], and essentially the following solution was suggested. The fact that the full system allows the segment to contain only one token at a time is modelled by augmenting the segment with an extra process I that monitors the input and output of the segment, and lets the segment contain at most one token at a time. That process might be called an *interface process*, because it monitors and restricts the interface of the subsystem. The process I allows the segment to input a token even if it already contains one, but then the input action leads to a specially marked "cut" state where no actions are possible. This restricts the number of states of the segment to $2k + 2$ — one state with no tokens, $2k$ states with one token and the special cut state with two tokens. Whenever the segment contains at most one token, I does not prevent any actions of the segment.

An LTS L of the system as a whole is constructed compositionally in the usual way, except that wherever needed, subsystems are augmented with suitable interface processes, and the special cut states are treated during LTS reduction as equivalent to each other and different from all other states. In L, inputting a second token should be impossible and the cut states should thus have disappeared. If this did not happen, then the user knows that some I was incorrect, and L is not valid for verifying the system. If the cut states did disappear, then the interface processes have no effect on the behaviour of the complete system,

and L is equivalent to what would have been obtained without them.

Instead of cut states, [38] developed and used a theory of "undefinedness" predicates. The theory allows the merging of any states whose S_i-components are the same during the computation of the parallel composition of a subsystem S_i and its interface process I_i.

The use of compositional LTS construction in this example is clearly an overkill, because the system as a whole has so few states. The purpose of the example was only to illustrate the spurious behaviour problem, and the use of interface processes for avoiding it. In [38] the token ring in Figure 1 was used as an example. It has $9n2^{n-2}$ states, but if all actions except **move tkn** are hidden and the full LTS is reduced preserving weak bisimilarity, then the result contains only n states. Any attempt to use the ordinary compositional LTS method leads to LTSs that are bigger than the full LTS, but the biggest LTS encountered with interface processes has (according to the numerical experiments in [38]) only $4n + 4$ states.

The interface processes resemble on-the-fly verification in that they allow the user to give the verification algorithm information about the *expected* (not necessarily *real*) behaviour of the system that helps to keep the number of states small. The user cannot fool the algorithm to give wrong answers by giving incorrect information, because verification results are obtained only if cut states disappear and the information has thus proven correct.

Behavioural fixed points and network invariants. Sometimes process-algebraic compositionality makes it possible to verify an infinite family of similar systems of increasing size. The dining philosophers system provides a simple example. It was observed in [93] that when a CFFD-semantics-preserving reduced LTS was constructed for it with the compositional approach, the open-ended segment PF_4 with four philosophers and forks where only the interface actions at the ends are visible is CFFD-equivalent to the segment PF_3 consisting of three philosophers and forks. By adding k philosophers and forks to PF_4 and similarly to PF_3 we get the result that $PF_{k+4} \simeq_{\mathsf{CFFD}} PF_{k+3}$, which implies that $PF_i \simeq_{\mathsf{CFFD}} PF_3$ for any $i \geq 3$.

The system of n dining philosophers can be composed of one philosopher and fork that are connected to both ends of PF_{n-1}. As a consequence, the analysis results obtained from the four-philosopher system are valid also for all larger philosopher systems, provided that the analysis questions refer only to the state of one philosopher and to the fork on her left side (the fork that he takes first). However, that suffices for detecting deadlock and starvation in the philosopher system.

The situation where the behaviour of a system does not change when more components are added to it may be called a *behavioural fixed point*. The stronger a behavioural equivalence is, the less likely it is to yield a behavioural fixed point in a given system. One would thus expect behavioural fixed points to be found more often with trace, CFFD- and CSP-equivalences than with weak bisimilarity.

The behavioural fixed point method can also be used for the verification

of systems with integer parameters that represent some upper bounds, as was demonstrated in [92]. The maximum number of transmission attempts that a sender process of a communication protocol will make before giving up is an example of such an upper bound. The basic idea is to model a counter that counts towards the upper bound as a stack of processes that may be added to the system one by one, thus increasing the upper bound. If a behavioural fixed point is found when the bound is k, then the system is independent of the value of the bound provided that it is at least k. This method was also used in [92] to model a certain fairness assumption by letting the counter count the number of times a certain unfavourable action (namely the loss of a message in a channel) has been chosen in a row instead of a favourable action.

Other methods that make it possible to verify an infinite family of systems of similar structure include [13, 54, 100].

7.4 Methods Based on Commutativity

Introduction to stubborn-set-type methods. The total effect of a set of concurrent actions is independent of the order in which the actions occur. As is obvious from the trivial "n non-interacting k-state processes" example of Section 1, the fact that the ordinary state space contains all possible orderings of concurrent actions is a major source of state explosion.

This observation has led many researchers to develop advanced state space methods where only some of the orderings are investigated, ideally only one ordering for each set of concurrent actions. In these methods, at each state that has been added to the reduced state space, only a subset of the semantic transitions out of that state are investigated. In other words, only a subset of structural transitions is used when constructing output edges for the state. The subset is chosen such that the occurrences of the remaining structural transitions can be postponed, or perhaps ignored altogether, without modifying the answers to the verification questions at hand.

Although the above idea is natural, a number of problems arise when trying to turn it into a working verification method. We will illustrate them with the aid of Figure 11.

- An enabled structural transition t may be in conflict with another structural transition t' that is disabled at the moment ("conflict" means, roughly speaking, that they are not concurrent even when both of them are enabled). This is the situation, for instance, with t_2 and t_3 of the example net in the marking $M_{2,4,7}$ where exactly p_2, p_4 and p_7 are marked. Executions where t' becomes enabled and occurs (perhaps disabling t) before the occurrence of t should not be ignored. Therefore, t' must somehow be taken into account in that state although it is disabled.
- Sometimes only one of two concurrent actions is taken in an execution that is relevant for verification. This happens, for instance, if we want to check the reachability of the above-mentioned marking $M_{2,4,7}$. Namely, initially t_1 and t_4 are concurrent, but only t_1 should occur to reach $M_{2,4,7}$. The method

must thus be able to favour one transition over another if that is important for the verification question.
- Some verification tasks require that the ordering of concurrent actions is partially preserved. For instance, to preserve the set of the process-algebraic traces of a system, the ordering of all transitions t such that $\mathcal{E}_\Sigma(t) \neq \tau$ must be preserved.
- The transitions that will be investigated in a state must be chosen reasonably efficiently, and the choice cannot rely on information on states that have not yet been investigated. For instance, it is obvious for a human who knows the net well that in $M_{2,4,7}$ it suffices to investigate t_4 alone but it is not sufficient to investigate t_2 alone, although both are enabled. An algorithm is needed for this kind of an analysis.
- The *ignorance* problem: If the investigation of some structural transition is postponed in each state of a cycle, then there is the risk that the transition is postponed forever. As an extreme example, if only t_6 is investigated in the initial marking of the example net, then no other markings will be constructed, and most of the behaviour of the net will be ignored.

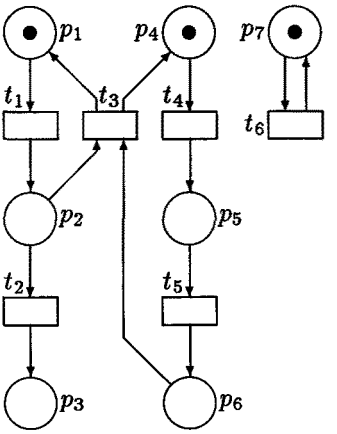


Fig. 11. A stubborn set example.

Different verification tasks require different answers to these problems, and different researchers have fine-tuned the details in different ways. This has led to the development of several methods under the names *stubborn sets* (since 1988 [81, 82]), *persistent sets* (since 1990 [33], although the term "persistent" is slightly more recent) and *ample sets* (since 1993 [68], but the idea was used in manual verification already in [49]). The persistent set approach is explained in detail in [34], and ample sets in [70]. These methods are so similar that it

would perhaps be a good idea to think of them as variants of the same method. In this article we call them *stubborn-set-type methods*. We use mostly the basic technical definitions in [72, 98], because they both cover or generalise most of the definitions in the literature, and are intuitively reasonably clear.

Stubborn-set-type methods are often classified as *partial-order methods*, but in the opinion of the present author that is somewhat misleading. "Partial order" refers to the truly concurrent semantics known as *Mazurkiewicz traces* [61], in which the ordering of actions is, indeed, partial. It would be tempting to think of stubborn-set-type methods as attempting to construct one linearisation of each Mazurkiewicz trace of the system, taking advantage of the symmetric *independency* relation in the theory of Mazurkiewicz traces.

In reality, however, the methods construct *representatives* for executions of the system, where a representative is not necessarily a member of the same Mazurkiewicz trace as an execution that it represents. The construction of representatives is based on an asymmetric relation that is more liberal than the independency relation. Furthermore, a Mazurkiewicz trace that is not relevant for the verification question at hand may be left without representatives. We will return to this issue after presenting the basic stubborn set method.

Basic stubborn set method. A set $T_s \subseteq T$ of structural transitions is *dynamically stubborn* at state s_0, if and only if the following hold:

D1 If $t \in T_s$, $t_1, \ldots, t_n \notin T_s$, $s_0 -t_1 t_2 \cdots t_n \rightarrow s_n$, and $s_n -t \rightarrow s'_n$, then there is s'_0 such that $s_0 -t \rightarrow s'_0$ and $s'_0 -t_1 t_2 \cdots t_n \rightarrow s'_n$.

D2 There is at least one transition $t_k \in T_s$ such that if $t_1, \ldots, t_n \notin T_s$ and $s_0 -t_1 t_2 \cdots t_n \rightarrow s_n$, then $s_n -t_k \rightarrow$. The transition t_k is called a *key transition* of T_s at s.

By letting $n = 0$ in D2 we see that a key transition is enabled. A set of structural transitions is *strongly* dynamically stubborn at state s, if and only if it is dynamically stubborn at s, and all of its enabled transitions are key transitions, that is, they qualify as the t_k in D2. Strongly dynamically stubborn sets are an important subclass of dynamically stubborn sets, because most (but not all) known algorithms for stubborn set construction produce them, and some analysis algorithms require them.

Because a dynamically stubborn set must contain a key transition, deadlock states have no dynamically stubborn sets. (A deadlock state is a state with no enabled structural transitions.) If s is not a deadlock state, then the set T of all structural transitions is dynamically stubborn and even strongly dynamically stubborn at s.

Let $T_s : S \mapsto 2^T$ be a function that assigns to each non-deadlock state s a dynamically stubborn set $T_s(s)$. The *basic stubborn set method* starts with the initial states, and constructs for each state s that has been found so far only those output edges and immediate successor states that are obtained by firing the enabled structural transitions in $T_s(s)$.

Let s_d be a deadlock state and s any state in the reduced state space constructed with the basic stubborn set method. If $s -t_1 t_2 \cdots t_n \rightarrow s_d$, then D2 implies that at least one of $t_1, t_2, \ldots, t_n$ must belong to $T_s(s)$, because otherwise all key transitions of $T_s(s)$ would be enabled at s_d. Let i be as small as possible such that $t_i \in T_s(s)$. D1 implies that there is s' such that $s -t_i \rightarrow s'$, and s_d is reachable from s' with $n-1$ steps, namely by firing $t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_n$. If s is an initial state, then by repeating this argument in s' and so on a total of n times we see that the reduced state space contains s_d. We have shown the hardest part of the following theorem:

> A reduced state space that has been constructed with the basic stubborn set method (that is, for each state s in it, $T_s(s)$ satisfies D1 and D2) contains all deadlock states of the system that are reachable from the initial states. Furthermore, all deadlock states of the reduced state space are deadlock states of the system.

If the transitions are deterministic, then the following can be proven from D1 and D2:

D3 If T_s is dynamically stubborn at s, $t_1, t_2, \ldots \notin T_s$, t_k is a key transition of T_s at s, and $s -t_1 t_2 \cdots \rightarrow$, then there is s' such that $s -t_k \rightarrow s'$ and $s' -t_1 t_2 \cdots \rightarrow$.

From D1, D2 and D3 it is not difficult to prove that if the system has an infinite execution, then also its reduced state space obtained with the basic stubborn set method contains an infinite execution. Therefore, the basic stubborn set method can be used for checking whether a system with deterministic transitions may fail to terminate. The result can be applied also when transitions are not deterministic by first showing that the stubborn sets that are used satisfy D3.

> A reduced state space that has been constructed with the basic stubborn set method contains an infinite execution if and only if the full state space contains an infinite execution, provided that either all structural transitions of the system are deterministic, or $T_s(s)$ satisfies D3 in each state s of the reduced state space.

On alternative definitions of stubborn sets. Several different definitions of stubborn (or ample or persistent) sets of different strength have appeared in the literature. We say that a definition of stubborn sets is *weaker* than another definition, if every set that is stubborn according to the latter is stubborn also according to the former, but not necessarily vice versa.

The weaker a definition of stubborn sets is, the more sets it classifies as stubborn, and the better are the chances that a stubborn set with very few enabled transitions may be found. As a consequence, weak definitions of stubborn sets have more potential for good reduction results than strictly stronger definitions. This has motivated researchers to try to find as weak definitions as possible without making the theory too complicated [36, 50, 72, 81, 83]. The conditions

D1 to D3 (essentially from [72]) were chosen for this article, because they are simple and among the weakest that have appeared in the literature.

As an example of the above, consider the situation where a process is ready to read from a non-empty fifo queue that other processes can access only by writing to it. Assume that the process has no alternative actions to the reading. Let t_{read} be the structural transition that corresponds to the reading. The conditions D1 to D3 classify the set $\{t_{\mathsf{read}}\}$ as dynamically stubborn (and even strongly). On the other hand, writing to and reading from a fifo are not independent in the "classic" sense used, for instance, in the theory of Mazurkiewicz traces, because sometimes the writing transition enables the reading transition. Therefore, $\{t_{\mathsf{read}}\}$ could not be stubborn according to the classic notion of independency. This problem does not affect D1 to D3, because we applied them in a state where t_{read} is known to be enabled.

As a consequence, a theory of stubborn sets based on the classic notion of independency would yield worse reduction results than the theory presented in this article. So the theory of Mazurkiewicz traces is not an optimal starting point for the development of stubborn-set-type methods, although it provides good background intuition.

Unlike an ample set [70] or persistent set [34], a stubborn set may contain disabled transitions. Disabled transitions might seem unnecessary, because they do not contribute to the set of output edges of a state. However, their presence simplifies the formulation of both the conditions V and L2 that will be presented later, and the notion of static stubborn sets that is useful for developing stubborn set construction algorithms. It was shown in [98] that if transitions are deterministic, then the non-empty persistent sets of [34] correspond precisely to the sets of enabled transitions of strongly dynamically stubborn sets.

Static definitions of stubborn sets. To implement the basic stubborn set method, it is necessary to design an algorithm that, given a non-deadlock state, produces the enabled transitions in a dynamically stubborn set of transitions. The set of all structural transitions is stubborn at every non-deadlock state, but it should be returned only as a last resort, because it will not yield any reduction of the number of output edges of the state. The definition of dynamical stubbornness does not directly lead to an algorithm, because it refers to states that are in the future of s and are thus not yet available.

To solve this problem, a static notion of a *stubborn set* of structural transitions has been defined. (Historically, stubborn sets were defined before dynamically stubborn sets.) This notion depends on the formalism used for modelling the system, and can be given several different definitions even for the same formalism, depending on how much effort one is willing to put into the analysis of the dependencies between transitions. What is important is that one must be able to prove for the chosen notion of stubborn sets that each stubborn set is also a dynamically stubborn set. A definition of "stubborn sets" is thus nothing but a static sufficient condition for a set being dynamically stubborn. Furthermore, if transitions are not deterministic and one wants to use stubborn sets for

the detection of failure of termination, then one should show also that the sets satisfy D3.

In the case of ordinary Petri nets, the following is a possible simple definition of stubborn sets:[10]

- If $t \in T_s$ and $\neg M[t\rangle$, then there is $p \in {\bullet t}$ such that $M(p) < W(p,t)$ and ${\bullet p} \subseteq T_s$.
- If $t \in T_s$ and $M[t\rangle$, then $({\bullet t}){\bullet} \subseteq T_s$ or ${\bullet}({\bullet t}) \subseteq T_s$.
- T_s contains a transition t_k such that $M[t_k\rangle$ and $({\bullet t_k}){\bullet} \subseteq T_s$.

This definition can be converted to a more restricted definition that allows only strongly dynamically stubborn sets simply by removing "or ${\bullet}({\bullet t}) \subseteq T_s$".

Also the following, more complicated definition from [83] (with a small improvement from [98]) implies D1 and D2. It represents a more careful analysis of the dependencies between transitions. It is implied by the above simple definition but is not equivalent. Therefore, it accepts more sets as stubborn. So it can yield smaller stubborn sets than the above simple definition and thus produce better reduction results.

- If $t \in T_s$ and $\neg M[t\rangle$, then there is $p \in {\bullet t}$ such that $M(p) < W(p,t)$ and $\{\, t' \mid W(p,t') < W(t',p) \wedge W(p,t') \leq M(p) \,\} \subseteq T_s$.
- If $t \in T_s$ and $M[t\rangle$, then for every $p \in {\bullet t}$, either $\{\, t' \mid \min\big(W(t,p), W(t',p)\big) < \min\big(W(p,t), W(p,t')\big) \,\} \subseteq T_s$, or $\{\, t' \mid \min\big(W(t,p), W(p,t')\big) < \min\big(W(p,t), W(t',p)\big) \,\} \subseteq T_s$.
- T_s contains a transition t_k such that $M[t_k\rangle$ and for every $p \in {\bullet t_k}$ $\{\, t' \mid W(t',p) < \min\big(W(p,t_k), W(p,t')\big) \,\} \subseteq T_s$.

A strong version is obtained by replacing everything after and including the "or" with "T_s contains a transition t_k such that $M[t_k\rangle$."

The following is an example of a definition of stubborn sets with nondeterministic transitions, stated in the language of labelled transition systems (Section 4.4). It applies to the state $(s_1, \ldots, s_n)$ of $L = L_1 || \cdots || L_n$, where $L_i = (S_i, \Sigma_i, \Delta_i, S_{Ii})$ for $1 \leq i \leq n$, and "$s_i - a \rightarrow_i s_i'$" denotes that $(s_i, a, s_i') \in \Delta_i$. It produces only strongly dynamically stubborn sets, and guarantees also D3. To simplify the presentation of the definition, it is assumed that each τ-action is subscribed by the index of the component process that executes it. The stubborn set T_s is a subset of $\Sigma \cup \{\tau_1, \ldots, \tau_n\}$.

- If $a \in T_s$ and $\neg(s - a \rightarrow)$, then there is $1 \leq i \leq n$ such that $a \in \Sigma_i \cup \{\tau_i\}$, $\neg(s_i - a \rightarrow_i)$, and $\{\, b \mid s_i - b \rightarrow_i \,\} \subseteq T_s$.
- If $a \in T_s$ and $s - a \rightarrow$, then for every $1 \leq i \leq n$, either $a \notin \Sigma_i \cup \{\tau_i\}$, or $\{\, b \mid s_i - b \rightarrow_i \,\} \subseteq T_s$.

[10] "${\bullet x}$" denotes the set of the input places or transitions of the transition or place x, "$x{\bullet}$" is the similar notion for output, and "${\bullet x}$" and "$x{\bullet}$" are extended to sets by taking the union of the results for each member of the set. $W(p,t)$ and $W(t,p)$ are the numbers of tokens that the transition t consumes from and produces for the place p when it occurs.

– There is $a_k \in T_s$ such that $s -a_k\rightarrow$.

As was emphasised above, each of the above definitions is *static* in the sense that, unlike the earlier definition of dynamically stubborn sets, it refers to only one state. Given a state and a set of transitions, it is possible (and easy) to check whether the set satisfies the definition in the state.

When constructing a reduced state space with stubborn-set-type methods, it is necessary at each state to construct a "good" stubborn set, given only the state. It is not known what stubborn set would be the best regarding reduction results. For instance, [82] contains an example showing that always choosing the stubborn set with the smallest number of enabled transitions does not necessarily produce the smallest reduced state space. Even so, it is easy to believe that trying to keep the number of enabled transitions small is a reasonable heuristics. In particular, if the set of enabled transitions in T_{s1} is a proper subset of the set of enabled transitions in T_{s2}, then T_{s1} is preferable, because with it the basic stubborn set method cannot produce a bigger but may produce a smaller reduced state space than with T_{s2}.

Algorithms for constructing stubborn sets. Each of the above static definitions of stubborn sets can be thought of as consisting of a requirement of the form "T_s must contain a transition t_k with a certain property k", and of conditions of the form "if $t \in T_s$, then all transitions in either $f_1(s,t)$ or $f_2(s,t)$ or ... or $f_k(s,t)$ must be in T_s", where $f_1, \ldots, f_k$ are functions of the state and t. If we are ready to give up some of the generality of the above definitions, we can introduce a more or less arbitrary rule for choosing one of the f_i, given s and t. Then the above conditions span a binary relation between transitions that we will denote with "$\rightsquigarrow$": $t_1 \rightsquigarrow t_2 \stackrel{\text{def}}{\iff} t_2 \in f_i(s,t_1)$.

For instance, in the case of the first of the above definitions, we may choose $t_1 \rightsquigarrow t_2 \stackrel{\text{def}}{\iff} {\bullet}t_1 \cap {\bullet}t_2 \neq \emptyset$ when $M[t_1\rangle$, and $t_1 \rightsquigarrow t_2 \stackrel{\text{def}}{\iff} t_2 \in {\bullet}p(M,t_1)$ when $\neg M[t_1\rangle$, where $p(M,t)$ is the smallest-numbered input place of t according to some fixed numbering of places such that $M\big(p(M,t)\big) < W\big(p(M,t),t\big)$.

The closure algorithm. The above notions lead to a simple sufficient graph-theoretic condition for stubborn sets: if a set is closed under "$\rightsquigarrow$" and it contains a transition that has the property k, then it is stubborn. Therefore, a stubborn set may be constructed by picking a transition t_k that satisfies k, and then performing a graph search (such as the depth-first search) in the graph $(T, \rightsquigarrow)$ starting at t_k.

This algorithm is easy to implement and reasonably fast: it consumes $O(|T| + |\rightsquigarrow|)$ time (but see the comment on speed below). The size of "$\rightsquigarrow$" depends on the modelling formalism. In the case of Petri nets it is at most $c^2|T|$, where c is the maximum number of input or output places or transitions of a transition or place. It can be reduced to $2c|T|$ by letting also the places act as vertices of the graph, yielding $O(c|T|)$ worst-case time.

Unfortunately, the quality of the stubborn sets this algorithm produces depends a lot on the choices of the start transitions.

The strong component algorithm. A better stubborn set is obtained by searching from $(T, \rightsquigarrow)$ for a maximal strongly connected component such that it contains a transition that satisfies k, and no other maximal strongly connected component reachable from it contains such a transition [81]. The stubborn set consists of the transitions in that component and all components reachable from it.

If the f_i are chosen such that each enabled transition automatically satisfies k, then this reduces to a search for a maximal strongly connected component that contains an enabled transition but its descendants do not (and then the resulting sets are strongly dynamically stubborn). For instance, with the above simple definition this means that if t is enabled, $(\bullet t)\bullet$ is used and not $\bullet(\bullet t)$.

In this application, it is better to search for the component with Tarjan's algorithm [79, 1] than with the more modern one described in [15]. This is because the former can be easily adapted to stop when a suitable component has been found even if most of $(T, \rightsquigarrow)$ has not been touched, whereas the latter requires that $(T, \rightsquigarrow)$ is searched completely through at least once. The former algorithm has thus much better best-case performance than the latter. Also this stubborn set construction algorithm consumes $O(|T| + |\rightsquigarrow|)$ or $O(c|T|)$ time in the worst case.

Instead of stopping immediately when the first suitable component has been found, the algorithm can be continued until it has found all suitable components. Then the one resulting in the smallest number of enabled transitions in the stubborn set may be chosen.

The deletion algorithm. The need to choose artificially one of the above-mentioned functions f_i can be avoided by spending more time in the construction of stubborn sets as follows [82]. One can take the different f_i into account by replacing $(T, \rightsquigarrow)$ by a more complicated graph containing "and"- and "or"-vertices. For instance, in the case of the above simple definition of stubborn sets for Petri nets, a disabled transition t corresponds to an or-vertex such that $t \rightsquigarrow p$ for every p such that $M(p) < W(p,t)$, and each such p corresponds to an and-vertex such that $p \rightsquigarrow t'$ for each $t' \in \bullet p$.

The algorithm starts with the full graph, and performs "removal searches". Each removal search starts by removing an enabled transition. When any vertex of the graph is removed, also those of its immediate "$\rightsquigarrow$"-predecessor vertices are removed that either are and-vertices, or are or-vertices and have lost all of their immediate "$\rightsquigarrow$"-successors. If the graph does not any more correspond to a stubborn set after a removal search, then the effects of the search are cancelled. Removal searches are continued as long as possible. The worst-case time consumption of this algorithm is $O(|T||\rightsquigarrow|)$. With the above Petri net definitions this makes $O(c|T|^2)$.

The time spent in constructing a stubborn set is an important factor in the time taken by the stubborn set method, because a stubborn set is constructed in every state that is investigated. Although all of the above algorithms have good asymptotic performance, they require a lot of analysis of "$\rightsquigarrow$".

In particular, the above definitions of stubborn sets of Petri nets may cause the algorithms to investigate long chains consisting of a disabled transition, its empty input place, a disabled input transition of that place, and so on. The above process-algebraic definition is better in this respect, because it causes the analysis to jump directly to the location where the control of the process is at the moment, instead of stepping one disabled transition at a time. It can be further improved by precomputing information on the reachability between the states of the same component LTS, and not making the jump if the start point of the jump is not reachable from the end point of the jump.

It may sometimes be more efficient to use simple heuristics for constructing a stubborn set rapidly, and use the set of all transitions when the heuristics fail. This may lead to worse stubborn sets and a bigger reduced state space, but may pay back in time consumption if the heuristics are fast enough. For instance, in [34] an algorithm is investigated that, in essence, computes the set of transitions that are reachable from an enabled transition with "$\rightsquigarrow$", until either the set is ready, or a disabled transition is encountered. In the latter case, the algorithm gives up and returns the set of all transitions.

When the stubborn set method is combined to packed state space methods such as the symmetry method, the problem of constructing good stubborn sets fast enough becomes more challenging. For instance, stubborn sets for a coloured Petri net may be constructed by first unfolding the net to an ordinary Petri net and then using the above methods, but one would not like to do this, because the ordinary Petri net may be much bigger than the original coloured Petri net. Unfortunately, in [53] it is shown that unfolding or something equally expensive is sometimes unavoidable in the construction of "good" stubborn sets for coloured Petri nets. It is, however, possible to find "good" stubborn sets in time that is proportional to the size of the coloured Petri net, if the net is given additional structure [53].

Another result of this kind is the definition of stubborn sets in [51]. It works for parallel labelled transition systems that are extended with symbolic data.

Stubborn set methods for safety properties. Because of the above-mentioned "ignorance" problem, the basic stubborn set method is not good for checking other properties than those that directly relate to the termination of the system. The addition of the following assumption makes the basic stubborn set method capable of analysing various safety properties.

S For every state s in the reduced state space and every $t \in T$, if $s \,-t\rightarrow$, then there is an execution $s_0 \,-t_1\rightarrow s_1 \,-t_2\rightarrow \cdots \,-t_n\rightarrow s_n$ such that $s_0 = s$, $t \in T_s(s_n)$, and t_i is a key transition of $T_s(s_{i-1})$ for $1 \leq i \leq n$.

We say that a structural transition t is *ignored* at s in the reduced state space, if a violation of S occurs with s and t. Ignoring a structural transition resembles being weakly unfair to it, but is not the same thing. Weak fairness states a requirement for all futures, whereas t is not ignored if there is at least one future of a particular kind.

If the conditions D1, D2 and S hold in a reduced state space, then the following claims hold [83]. They make it possible to check a number of safety properties from a reduced state space. The last three of the claims are corollaries of the first one. By a *terminal component* we mean a maximal strongly connected component such that each edge that starts in the component also ends in it.

- If $s - t_1 t_2 \cdots t_n \rightarrow s'$ is a finite execution that starts at a state s that is in the reduced state space, then there are structural transitions $t_{n+1}, \ldots, t_{n+k}$ ($k \geq 0$) and a state s'' such that $s' - t_{n+1} \cdots t_{n+k} \rightarrow s''$, and the reduced state space contains the execution $s - t_{\pi(1)} t_{\pi(2)} \cdots t_{\pi(n+k)} \rightarrow s''$ for some permutation $t_{\pi(1)}, t_{\pi(2)}, \ldots, t_{\pi(n+k)}$ of $t_1, t_2, \ldots, t_{n+k}$.
- A structural transition t labels an edge in the reduced state space if and only if t labels an edge in the full state space.
- A structural transition t is Petri-net-live (Section 2.3) if and only if t is Petri-net-live in the reduced state space.
- If the full state space is finite and contains a terminal component C, then the reduced state space contains a terminal component C_{red} such that $C_{\mathsf{red}} \subseteq C$, and a structural transition t occurs in C_{red} if and only if it occurs in C. In the reverse direction, each terminal component C_{red} of the reduced state space is a subset of some terminal component C of the full state space such that t occurs in C_{red} if and only if it occurs in C.

The second of the above results makes it possible to verify various linear-time safety properties with the fact transition technique and tester processes described in Section 4.2. In this approach it is important that the tester process synchronises only with a subset of the transitions of the system. This is because the addition of the tester introduces new dependencies between the transitions it synchronises with. If the tester is synchronised with every transition, then it is almost always the case that the stubborn set must contain all enabled transitions, leading to no reduction in the size of the state space.

A strong stubborn set algorithm for ensuring the condition S. In the case of strongly stubborn sets, the condition S reduces to "if s is in the reduced state space and $s - t \rightarrow$, then t is investigated in at least one of the states that are reachable from s in the reduced state space". This observation leads to an efficient algorithm for ensuring that S holds [83]. In it, the reduced state space is constructed in depth-first order. Tarjan's algorithm is used during the construction to recognise terminal components of the state space.

Assume that a terminal component has just been completed, and the construction of the reduced state space is just about to backtrack from a state s in the component to a state outside the component (or is about to terminate, if s has no predecessor states in the depth-first search tree). We will say that the structural transition t is *ignored in the component*, if it is enabled in s but occurs nowhere in the component. The algorithm checks that no structural transition is ignored in the component. If this does not hold, the stubborn set $T_{\mathsf{s}}(s)$ that was used in s is extended such that at least one structural transition that was

ignored in the component is included into the extended set, and the extended set is stubborn.

From the point of view of the depth-first search discipline and Tarjan's algorithm, the extension of $T_s(s)$ adds new output edges to s, and does that exactly when the original output edges have been scanned through. Because the depth-first search has not yet made any actions after investigating the original output edges of s, from its point of view the newly added output edges could have been there all the time. Therefore, the extension of $T_s(s)$ does not confuse the depth-first search discipline or Tarjan's algorithm.

One possibility of computing the extension of $T_s(s)$ is to compute a new stubborn set that contains a transition that is ignored in the component. This is correct, because it is easy to check from D1 to D3 that the union of two stubborn sets is stubborn. If both the original $T_s(s)$ and its extension are constructed with the closure or strong component algorithm using the same relation "$\rightsquigarrow$", then the construction of the extension needs not enter the original stubborn set.

More details of this algorithm are given in [83].

Preserving process-algebraic traces. The above algorithm makes it possible to check on the fly whether $L_1 \sqsupseteq_{\mathrm{tr}} L_2$ with the tester processes described in Section 4.2. It is, however, sometimes beneficial to have an algorithm that preserves process-algebraic traces and is *transparent* in the sense of Section 6. This is the case, for instance, if the LTS is intended to be used as a component in the compositional LTS construction method of Section 7.3. This goal is obtained by stating an additional requirement for the stubborn sets $T_s(s)$ that are used in the states s of the reduced state space:

V If $T_s(s)$ contains a structural transition t such that $s\ -t\rightarrow$ and $\mathcal{E}_\Sigma(t) \neq \tau$, then $T_s(s)$ contains all those structural transitions t' such that $\mathcal{E}_\Sigma(t') \neq \tau$ (even if t' is disabled).

The condition is labelled "V" because it talks about the handling of *visible* transitions, that is, those structural transitions whose $\mathcal{E}_\Sigma$-abstraction is not τ. It ensures that when the stubborn set method produces a representative for an execution, it does not change the relative ordering of visible transitions. Assuming that D1, D2, S and V are satisfied, then the reduced state space is trace-equivalent with the full state space.

There are at least two practical ways of implementing V. The more complicated one consists of first trying to find a stubborn set which contains no enabled visible transitions with, for instance, the strong component algorithm. If that fails, then the closure algorithm is applied to every visible transition and the union of the results is taken. The simpler way consists of just adding to "$\rightsquigarrow$" an edge from t_1 to t_2 for every enabled visible t_1 and visible t_2 before attempting to construct a stubborn set.

If all transitions are visible, then V forces the reduced state space to be the same as the full state space. Therefore, no reduction is obtained unless there are invisible transitions.

The above-mentioned on-the-fly method of checking "$\sqsupseteq_{\mathrm{tr}}$" with tester processes does not need the condition V. Therefore, it might seem to allow for better stubborn sets. Sometimes it indeed does that, but perhaps not as often as one might think, because the addition of the tester process to the system introduces new dependencies between transitions. In particular, if the tester process is synchronised only with the visible actions and it never refuses any visible actions, then each state s_t of the tester process has for every $a \in \Sigma$ an output edge that is labelled with a. In most (but not all) cases this implies that if one output action of s_t is taken into the stubborn set, then all of them must be taken, which is equivalent to the extension of "$\rightsquigarrow$" described above. If the tester is synchronised with all actions instead of just visible ones, then the situation is even worse, because then almost all actions become dependent of each other.

It is essential for V that a stubborn set may contain disabled transitions. Instead of V, [70] uses the strictly stronger condition V′ that either $T_s(s)$ contains no enabled visible transitions, or $T_s(s) = T$. This condition leads to worse reduction results than V, but is easier to implement.

Other ways of handling safety properties. An alternative approach to the verification of safety properties that is based on a transformation of the system description was presented in [37]. Safety properties can be analysed also with the CFFD-, LTL$_{-X}$-, and branching-time-preserving stubborn set methods that are described below.

Stubborn set methods for liveness properties. The above stubborn set methods for safety properties suffice for the verification of various properties whose validity depend on finite executions. However, they do not suffice for properties that depend on infinite executions, as the example in Figure 12 demonstrates. In the example, a and b are visible actions, and τ_1, τ_2 and τ_3 are invisible. If the dashed edges are removed, the resulting reduced state space satisfies D1, D2, D3, S and V. However, according to it it is guaranteed that if a is executed, then also b will be executed, although this is not true in the full state space.

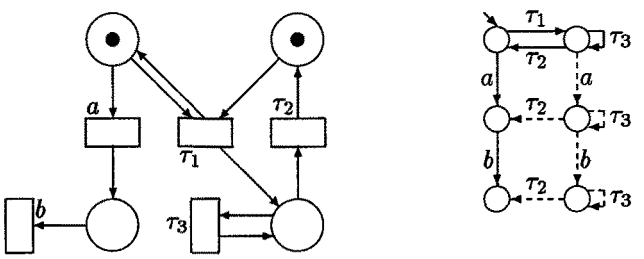


Fig. 12. Safety-preserving stubborn set methods do not suffice for liveness.

The following two conditions make the stubborn set method capable of handling various linear-time liveness properties. They replace the condition S.

L1 If s is a state in the reduced state space such that there is $t \in T$ such that $\mathcal{E}_\Sigma(t) = \tau$ and $s -t\rightarrow$ in the full state space, then $T_s(s)$ has a key transition t_k such that $\mathcal{E}_\Sigma(t_k) = \tau$.

L2 If $s_0 -t_1\rightarrow s_1 -t_2\rightarrow \cdots$ is an infinite execution in the reduced state space starting at any state s_0, then for each t_v such that $\mathcal{E}_\Sigma(t_v) \neq \tau$, there is $i \geq 0$ such that $t_v \in T_s(s_i)$. (This implies that there are actually infinitely many such i.)

Regarding the execution $M_I -\tau_1 a \tau_3^\omega\rightarrow$ of the system in Figure 12, L2 ensures that a is investigated after τ_1 (the infinite execution in L2 is $M_I -\tau_1 \tau_3^\omega\rightarrow$). L1 ensures that τ_3 is investigated after a.

The following theorem is slightly strengthened from a similar one in [86]:

- If s is a state in a reduced state space that satisfies the conditions D1, D2, D3, V, L1 and L2, and if $s -t_1 t_2 \cdots\rightarrow$ is an infinite execution that starts at s, then the reduced state space contains an infinite execution $s -t'_1 t'_2 \cdots\rightarrow$ such that $\mathcal{E}_\Sigma(t'_1 t'_2 \cdots) = \mathcal{E}_\Sigma(t_1 t_2 \cdots)$.
- If s is a state in a reduced state space that satisfies the conditions D1, D2 and V, and if $s -t_1 t_2 \cdots t_n\rightarrow s_d$ is a finite execution that starts at s and ends in a deadlock state, then the reduced state space contains a finite execution $s -t'_1 t'_2 \cdots t'_n\rightarrow s_d$ such that $\mathcal{E}_\Sigma(t'_1 t'_2 \cdots t'_n) = \mathcal{E}_\Sigma(t_1 t_2 \cdots t_n)$, and $t'_1 t'_2 \cdots t'_n$ is a permutation of $t_1 t_2 \cdots t_n$.

The condition L1 can be implemented by first constructing a stubborn set without worrying about L1, and then extending L1 to contain an invisible key transition if necessary. The closure and strong component algorithms are handy for computing the extension. If the condition V is replaced by the stronger condition V' mentioned above, then L1 holds automatically and needs not be worried of.

With finite state spaces, L2 is equivalent to the condition that for each visible transition t_v, each cycle of the state space must contain at least one state s such that $T_s(s)$ contains t_v. Perhaps surprisingly, t_v needs not be enabled in s; it suffices that it affects the construction of the stubborn set. This condition is difficult to implement in its full generality. It is, however, easily implementable, if we do not mind using the same s for each t_v. Then cycles that violate it can be recognised with the non-progress cycle detection algorithm that was explained in Section 4.2. The stubborn set of the current state is extended with the "$\rightsquigarrow$"-closure of visible transitions if necessary. More detail can be found in [85, 86].

Another possibility is to construct the reduced state space in depth-first order, and extend the stubborn set each time the current state has an output edge to any state in the depth-first stack [68, 70]. (Only the edges in the reduced state space are taken into account in this test.) This algorithm is much simpler than the previous one, but extends stubborn sets more often.

Like with V and V′, it is possible to simplify the implementation of L2 at the cost of less reduction by requiring that $T_s(s) = T$ in the states s whose stubborn sets are extended [68, 70]. We will call this condition L2′.

Stubborn set methods for LTL$_{-X}$. The above theorem yields a transparent construction-time reduction method that preserves the validity of LTL$_{-X}$-formulae (or stuttering-insensitive LTL formulae). Let Π be the set of atomic propositions that appear in the formulae. We say that a structural transition t *affects* a proposition $P \in \Pi$ if and only if the reachable part of the state space contains states s and s' such that $s \xrightarrow{t} s'$, and either $s \models P$ and $s' \models \neg P$, or $s \models \neg P$ and $s' \models P$. That is, t affects P if and only if some occurrence of t changes the truth value of P.

Let T_Π be a set of structural transitions such that it contains *at least* those transitions that affect any $P \in \Pi$. The set T_Π is allowed to be larger than the precise set of transitions that affect members of Π, because the precise set may be difficult to find, and the use of a larger set does not sacrifice correctness (although it may lead to worse reduction results). The theorem that underlies the LTL$_{-X}$-preserving stubborn set method is as follows:

> If all transitions in T_Π are treated as visible and a reduced state space is constructed such that the conditions D1, D2, D3, V, L1 and L2 are satisfied, then the truth value of any LTL$_{-X}$-formula whose atomic propositions belong to Π is the same in the reduced and full state spaces.

Due to the condition V, the larger Π is, the larger will the reduced state space usually be. If φ is of the form $\varphi_1 \wedge \cdots \wedge \varphi_k$, then this effect may be fought against by verifying each φ_i separately. Then more than one reduced state space is constructed. However, with some luck each φ_i uses a smaller set of visible transitions and leads to a much smaller reduced state space than φ, so significant savings in total effort are possible (but not guaranteed). Unfortunately, this technique does not yield correct results for formulae of the form $\varphi_1 \vee \varphi_2$.

A more general technique for distributing the set of visible transitions was suggested in [68]. That paper assumed that the underlying model of computation is fair in a certain particular sense; we will return to this assumption a bit later. The LTL$_{-X}$-formula φ in question is converted into a form $B(\varphi_1, \ldots, \varphi_k)$, where $B(x_1, \ldots, x_k)$ represents some Boolean combination of $x_1, \ldots, x_k$, and each φ_i is as small as possible. For instance, $\Box(\varphi \wedge \Diamond\psi)$ can be converted to $(\Box\varphi) \wedge (\Box\Diamond\psi)$.

In this technique, only one reduced state space is constructed. Even so, instead of one set of visible transitions, each subformula has its own set, and the condition V is applied to each set separately. If t_1 only affects atomic propositions in φ_1 and t_2 only those in φ_2 and both t_1 and t_2 are enabled, then the original condition V insists that if t_1 is in the stubborn set, then also t_2 must be. On the other hand, this requirement disappears when V is applied to each subformula separately. This leads to smaller stubborn sets and better reduction results.

If V is applied to each subformula separately, then the proper treatment of the condition L1 becomes complicated. This problem did not arise in [68],

because the fairness assumption in it made L1 unnecessary. On the other hand, the assumption rejects the execution $M_I \xrightarrow{-\tau_1 a \tau_3^\omega \rightarrow}$ of the system in Figure 12 as unfair, so we are not always willing to make it. The thesis [98] solves this problem by showing how both V and L1 can be distributed to subformulae, so that the fairness assumption of [68] is not any more needed.

Handling fairness assumptions. The correctness of a system with respect to an LTL$_{-X}$-formula that expresses some liveness property often depends on some fairness assumptions (Section 2.4) about the system. As the following examples illustrate, customary fairness assumptions are not easy to handle with the LTL$_{-X}$-preserving stubborn set method, because the method may choose an unfair representative to a fair execution that violates a property.

Consider the alleged property $\Diamond(M(p_5) = 1)$ of the net on the left in Figure 13, when strong fairness towards t_5 is assumed. In the situation where exactly p_2 and p_3 are marked, a typical implementation of the LTL$_{-X}$-preserving stubborn set method would fire t_3 and only it, because $\{t_3\}$ is then stubborn, but no set not containing t_3 is stubborn. The situation is symmetric when exactly p_1 and p_4 are marked. As a consequence, in each infinite execution in the reduced state space, t_5 is enabled infinitely often. They are thus unfair and do not qualify as counterexamples to $\Diamond(M(p_5) = 1)$. So the reduced state space would lead to the conclusion that $\Diamond(M(p_5) = 1)$ holds. On the other hand, the full state space contains the fair counterexample $M_I \xrightarrow{-(t_1 t_2 t_3 t_4)^\omega \rightarrow}$.

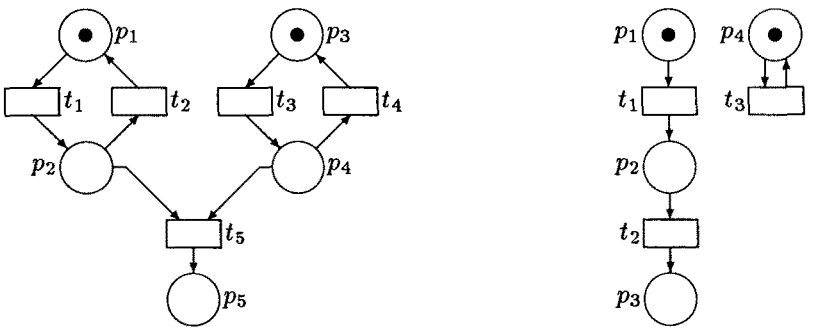


Fig. 13. Two fairness examples.

The net on the right in the figure exemplifies a similar problem with weak fairness. Let the formula be $\Diamond(M(p_3) = 1)$, and assume weak fairness towards t_2. The LTL$_{-X}$-preserving stubborn set method may choose $\{t_1\}$ as the stubborn set used in the initial marking. Unfortunately, if it does that, then it looses the only fair counterexample to the formula.

This problem can be avoided by representing the fairness assumptions as a part of the formula to be verified in the form "$(\mathsf{fair}_1 \wedge \cdots \wedge \mathsf{fair}_n) \Rightarrow \mathsf{property}$".

Then the ordinary LTL$_{-X}$-preserving stubborn set method is guaranteed to treat the fairness assumptions correctly, because they need no special treatment. This technique has, however, the problem that with it, very many transitions must be visible, which leads to bad reduction results. Fortunately, "$(\mathsf{fair}_1 \wedge \cdots \wedge \mathsf{fair}_n) \Rightarrow$ property" is a Boolean combination of "fair_1", ..., "fair_n", and "property", so the above-mentioned techniques of [68, 98] can be used to alleviate the problem.

Another possibility is to use fairness assumptions that are insensitive to the difference between a sequence and its representative chosen by the stubborn-set-type method. With this goal in mind, in [68, 70] the following atypical but natural fairness assumption was suggested. Furthermore, the LTL$_{-X}$-preserving stubborn set method was proven correct when it and L2′ are assumed and the classic notion of "dependency" is used, even if the conditions L1 and D3 are dropped:

[68, 70]-fairness If a structural transition t is enabled continuously from some state on in an execution, then some transition that is dependent on t — perhaps t itself — occurs in the execution somewhere after that state. This is required from every $t \in T$.

Stubborn set methods for failure-based process semantics. A reduced state space that is constructed according to the LTL$_{-X}$-preserving stubborn set method — that is, obeying D1, D2, D3, V, L1 and L2 — is CFFD-equivalent (and thus also CSP- and NDFD-equivalent) with the full state space, assuming that all transitions t such that $\mathcal{E}_{\Sigma}(t) \neq \tau$ are treated as visible by the conditions V, L1 and L2. Therefore, the LTL$_{-X}$-preserving stubborn set method can also be used as a method of computing reduced parallel compositions when any of these three semantics is used.

When using the method, it is useful to notice that if the system is of the form

$$\mathbf{hide}\, a_1, \ldots, a_k \,\mathbf{in}(L_1 || \cdots || L_n),$$

then the transitions of $L_1 || \cdots || L_n$ labelled with a_1 or ... or a_k need not be considered visible by V, L1 and L2, because they are invisible at the system level, although they are visible from the point of view of an individual L_i.

The use of stubborn sets is beneficial even if the resulting state space will be further reduced with some process-algebraic LTS reduction algorithm. This is because the stubborn set method reduces the risk of the parallel composition being so big that it cannot be constructed or processed by the LTS reduction algorithm.

Worth mentioning is the fact that if a reduced state space obeys D1, D2, V and L1, then it contains all the stable states and all the stable failures of the original state space. Furthermore, if it obeys D1, D2, D3, V and L1, then it is CSP-equivalent with the full state space. In other words, L2 is not needed to preserve the CSP-semantics of a state space. This is an interesting consequence of the fact that, as was explained in Section 4.4, divergence is "catastrophic" in the CSP-semantics.

Stubborn set methods for process algebras were surveyed in [91].

On-the-fly methods. The use of automata-theoretic on-the-fly verification methods simultaneously with stubborn-set-type methods was discussed already in connection with safety-property-preserving stubborn set methods. Those Büchi automata that are connected to the system with the state synchronisation technique and do not restrict the transitions of the system (Section 4.2) are invisible to the stubborn set method. Therefore, Büchi automata can be used simultaneously with the LTL$_{-X}$-preserving stubborn set method. This idea has been developed further in [69].

An alternative approach is to connect the Büchi automaton (or tester process, Section 4.2) with transition fusion to only the visible transitions of the system. The automaton is now allowed to restrict the behaviour of the system. This idea was investigated in [87].

When verifying an LTL$_{-X}$-formula, it is often the case that an atomic proposition becomes irrelevant for the formula at some point of an execution. This is the case, for instance, with the formula $\Box(P \Rightarrow \Box\neg Q)$: when a state s such that $P \in \mathcal{E}_{\Pi}(s)$ has been seen, P does not matter any more, but the only interesting thing is whether Q may become true. An advanced on-the-fly method that exploits this fact was developed in [52].

A stubborn set method for branching time properties. A stubborn-set-type method that preserves the validity of CTL$^{*}_{-X}$-formulae was presented in [31], and adapted to nondeterministic structural transitions in [91]. To preserve the validity of CTL$^{*}_{-X}$-formulae, it is important that the points in time when decisions between different futures are made are located correctly with respect to each other and relative to $\mathcal{E}_{\Pi}$-abstracted states. This is exemplified by Figure 14 and the CTL-formula $\mathsf{EF}(\alpha \wedge (\mathsf{EF}\beta) \wedge (\mathsf{EG}\neg\beta))$. The formula holds in the net in the figure. The transitions t_2 and t_3 are invisible with respect to it, so the LTL$_{-X}$-preserving stubborn set method would allow to choose $\{t_2, t_3\}$ as the stubborn set that is used in the initial marking. That would, however, lead to a reduced state space where the formula does not hold, because in it the choice between t_2 and t_3 occurs too early.

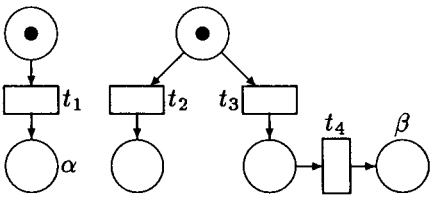


Fig. 14. A branching time example.

Because of the above problem, the CTL$^{*}_{-X}$-preserving stubborn set method

is based on the following, rather strong condition:

B If s is a state in the reduced state space, then the stubborn set $T_s(s)$ either contains only one enabled structural transition, and that transition is invisible; or it contains all enabled structural transitions.

The following theorem expresses the correctness of the CTL$^*_{\setminus X}$-preserving stubborn set method:

> If all structural transitions are deterministic, all structural transitions in T_{Π} are treated as visible, and a reduced state space is constructed such that the conditions D1, D2, S, and B are satisfied, then the truth value of any CTL$^*_{\setminus X}$-formula whose atomic propositions belong to Π is the same in the reduced and full state spaces.

The construction of stubborn sets that satisfy D1, D2 and B is simple, because only the set of all structural transitions and the sets containing exactly one enabled structural transition need be taken into account. One may first scan through all enabled invisible structural transitions with the hope of finding one that, together with a suitable set of disabled structural transitions, satisfies D1 and D2. If that fails, then all structural transitions are taken.

Also the implementation of S is easy in this context, because states that do not have all enabled structural transitions in their stubborn sets have exactly one successor state. Such states may be constructed in a linear sequence. S is violated whenever the sequence enters any of its states anew. Then an arbitrary state of the sequence can be picked, and its output edges re-examined using this time the set of all transitions.

Allowing nondeterministic transitions. The condition B does not suffice with nondeterministic structural transitions, because it does not rule out the possibility that the same transition may occur in two different ways leading to two different futures. In [91] a stronger condition was suggested that is based on the notion of *super-determinism*. We use the labelled transition system notation (Section 4.4) for explaining it, because nondeterministic transitions (or actions) are frequently used with them.

An action $a \in \Sigma$ is super-deterministic in the state s_0, if and only if the existence of an execution $s_0 -a_1\rightarrow \cdots -a_n\rightarrow s_n$ where $a_1, \ldots, a_n \neq a$ implies the existence of states $s'_0, \ldots, s'_n$ such that $s'_0 -a_1\rightarrow \cdots -a_n\rightarrow s'_n$, and for every $0 \leq i \leq n$: $s_i -a\rightarrow s'_i$ and if $s_i -a\rightarrow s''$, then $s'' = s'_i$. Less formally, a super-deterministic action is enabled, can yield only one result when it occurs even if other actions are executed before it, the execution of other actions cannot disable it, and its execution commutes with the execution of the other actions.

NB If s is a state in the reduced state space, then the stubborn set $T_s(s)$ either contains only one enabled structural transition, and that transition is invisible and super-deterministic; or it contains all enabled structural transitions.

The condition NB implies D1 and D2. Therefore, the CTL$^*_{-X}$-preserving stubborn set method can be formulated as follows in the presence of nondeterministic transitions:

> If all structural transitions in T_Π are treated as visible, and a reduced state space is constructed such that the conditions S and NB are satisfied, then the truth value of any CTL$^*_{-X}$-formula whose atomic propositions belong to Π is the same in the reduced and full state spaces.

If an LTS is constructed with the CTL$^*_{-X}$-preserving stubborn set method, then it is branching bisimilar and thus also weakly bisimilar with the ordinary LTS of the system, assuming that all transitions t such that $\mathcal{E}_\Sigma(t) \neq \tau$ are treated as visible.

The following sufficient condition for super-determinism [91] can be used in implementing the CTL$^*_{-X}$-preserving stubborn set method in the presence of nondeterministic transitions. Let $L_1 || \cdots || L_n$ be a parallel composition of LTSs, and $(s_1, \ldots, s_n)$ its state. Assume that $(s_1, \ldots, s_n) -a\rightarrow$. If for each $1 \leq i \leq n$, either $a \notin \Sigma_i \cup \{\tau_i\}$, or the local state s_i has only one output transition in the LTS L_i, then a is super-deterministic in $(s_1, \ldots, s_n)$. In other words, a is super-deterministic in $(s_1, \ldots, s_n)$, if each L_i that is interested in a-actions is ready to do only one transition, and that transition is labelled with a.

On the strength of the CTL$^*_{-X}$-preserving method. Stubborn set methods demonstrate nicely the principle that the more information one is willing to give away, the more powerful state space reduction methods are available. In this section we started with a method that is valid only for the verification of termination-related properties, and ended up with a method for full CTL$^*_{-X}$ and branching bisimilarity. Each time when we moved from a method to the next, we added one or more new restrictions on the construction of stubborn sets.

Although the relative strengths of the conditions underlying the methods are not always formally comparable, it is intuitively clear that B and NB prevent the use of a small stubborn set much more often than D3, V, L1 and L2, so the ability of the CTL$^*_{-X}$-preserving method of reducing state spaces is smaller than that of the other methods. On the other hand, it is easy to produce a fast implementation for the CTL$^*_{-X}$-preserving method without losing much of its generality, whereas the attempt to exploit the full power of the safety-property-preserving and LTL$_{-X}$-preserving methods leads to complicated and significantly slower algorithms. It is not known how significant the difference between the reduction power of the CTL$^*_{-X}$- and LTL$_{-X}$-preserving methods is in practice.

There are two important stuttering-insensitive branching-time specification formalisms that are strictly less powerful than CTL$^*_{-X}$ and branching bisimilarity, namely CTL$_{-X}$ and weak bisimilarity. It would be interesting to find a stubborn-set-type method that would preserve one of these, and would improve reduction results by not preserving CTL$^*_{-X}$ and branching bisimilarity. Unfortunately, the example in Figure 14 is valid also for CTL$_{-X}$ and weak bisimilarity, so the chances of finding such a method seem small.

Sleep sets. Sleep sets were first presented in [33], and perhaps the best source on them is [34]. Their theory has not been developed as far as that of stubborn sets. While stubborn-set-type methods save effort by postponing the investigation of structural transitions to future states, sleep sets avoid the investigation of transitions that have been investigated in the past states.

Assume that t_1 and t_2 are enabled and independent of each other in some state s, and neither of them has been put to *sleep* in the sense described below. Unlike stubborn-set-type methods, the sleep set method investigates both t_1 and t_2 at s. Assume that t_1 is investigated before t_2. Then, when investigating t_2, t_1 is put to sleep. Sleeping transitions will not be taken into account in future states until they are woken up. A sleeping transitions is woken up for certain reasons, such as the occurrence of a transition that is dependent on it.

As such, the sleep set method does not reduce the number of reachable states, it reduces only the number of edges. However, it can be used simultaneously with stubborn-set-type methods, and the combination gives better reduction results (in terms of reachable states) than the stubborn-set-type methods alone.

As was pointed out in [35], sleep sets can also be used to improve the *state space caching* method of [45]. The basic idea of state space caching is simply to start throwing reachable states away from the memory of the state space construction tool when the memory fills up. If a state has more than one input edge, as is very common in the presence of concurrency, and if it is thrown away before investigating all of its input edges, then it and at least a subset of its successor states will be constructed more than once. This is bearable if the full state space is not much bigger than what fits the memory, but leads to dramatic decrease of the performance of the algorithm if the full state space is big enough. Sleep sets reduce the number of times a state is entered, and therefore reduce the number of times a state is re-constructed. An impressive example of the power of sleep sets in this application is given in [34].

8 Conclusions

Many different techniques based on widely diverse principles have been suggested for alleviating the state explosion problem. In this article we have discussed a number of them. None of today's techniques solves the state explosion problem once and for all, and, because of the results regarding the computational complexity of typical verification tasks, it is unlikely that a perfect solution will be found in the near future. Even so, today's techniques facilitate the analysis, validation and verification of much bigger systems than ordinary state space construction. Although they are not strong and easy enough to be routinely used in all production of concurrent software and hardware, they can help a lot in finding serious design flaws in systems, and the proving of the correctness of various critical details.

An advanced state space method practically always restricts the set of properties that can be analysed or verified. This implies that the user of the method has to have at least some idea of the analysis questions before starting the tool.

If, during the analysis, new questions become relevant, it may be that a new reduced state space has to be constructed to answer them. This is a drawback compared to the use of full state spaces, where the same state space can be used for answering all analysis questions, even if the user invents new questions after obtaining the answers to the original ones. On the other hand, when the full state space of the system is too big to be constructed, it is better to get answers one at a time than to get no answers at all.

Because many advanced state space methods work well only with certain kinds of analysis questions, the user of the methods has to decide what kinds of questions will be asked. The smaller set of question types the user is happy with, the more methods are available for alleviating state explosion. Some arguments in favour of and against certain question types are given in the following.

Sensitivity to stuttering. Stuttering is usually not a problem for methods based on packed state spaces (perhaps excluding the unfolding method). In contrast, methods that are based on commutativity (Section 7.4) or process-algebraic compositionality (Section 7.3) are valid only for stuttering-insensitive properties.

Because the level of atomicity is often artificial to some extent, and because knowledge of how many actions of a parallel, independent process were scheduled between two actions of an interesting process in a one-processor machine is unimportant, many researchers consider stuttering as irrelevant in the context of concurrent systems.

Linear vs. branching time. Excluding CTL* and $\mathrm{CTL}^*_{-\mathrm{X}}$, the most common branching-time specification and query formalisms have fast algorithms for the last step of the verification of a property. With linear-time formalisms the algorithms for the last step tend to take exponential time in the length of the description of the property in the worst case (with the exception of preorder checking against a deterministic process-algebraic specification).

One may argue, however, that the bad worst-case performance of linear-time model and preorder checking is not an important problem, because the description of the property is often small, at least compared to the size of the system. Furthermore, linear-time formalisms allow the use of more powerful state space reduction methods than branching-time formalisms. For instance, we pointed out in Section 5.3 that labelled transition systems can be reduced more with failure-based semantic models than with weak bisimilarity. To give another example, the $\mathrm{CTL}^*_{-\mathrm{X}}$-preserving stubborn set method allows the use of a non-trivial stubborn set (that is, one that does not contain all enabled transitions) much less often than the $\mathrm{LTL}_{-\mathrm{X}}$-preserving method. Finally, it is difficult to think what a branching-time version of Holzmann's supertrace (Section 7.2) might look like.

As a consequence, although linear-time formalisms lose in the checking of the property from the reduced state space, they may win a lot during the construction of the reduced state space. It is thus difficult to say which approach is faster in practice. This question was analysed in more detail in [89].

Linear-time formalisms have also the advantage that if a property does not hold, then a counterexample that the user can easily simulate and analyse can be provided.

The verification of linear-time liveness properties needs often that fairness assumptions are made and taken into account. Fairness assumptions are problematic to certain advanced state space methods, including the process-algebraic compositional LTS construction in Section 7.3 and the LTL$_{-X}$-preserving stubborn set method in Section 7.4. Petri-net-liveness, or more generally the CTL "AG EF", gives to some extent similar information as linear-time liveness, although they are not precisely the same: linear-time liveness guarantees that, say, a structural transition t will occur in the future, whereas Petri-net-liveness only promises that the future occurrence of t is possible and cannot be made impossible.

Petri-net-liveness may, therefore, fall a bit short from what the user wants. It does not, however, need fairness assumptions; we saw in Section 7.4 that it is preserved by the safety-property-preserving stubborn set method; and it is within the scope of the unfolding method in Section 7.2. It is thus a particularly interesting branching-time operator from the point of view of alleviating state explosion.

State- vs. action-based. In many cases it is easy to switch from a state-based specification to an action-based one or vice versa with the mappings of Section 2.2. In those cases the choice between a state- or action-based formalism does not have much significance from the point of view of advanced verification methods.

There are, however, situations where advanced verification methods clearly favour one approach over the other. For instance, we saw in Section 7.2 that checking whether a certain transition can ever occur is easy for the unfolding method, but checking the reachability of a state with a certain property is difficult, unless the property can be handily encoded as the enabling condition of one or few transitions. It seems that the majority of advanced state space methods either handle arbitrary atomic propositions over states without difficulties (for instance, binary decision diagrams and Holzmann's supertrace), or require that the properties are ultimately stated in terms of actions (for instance, visible transitions in the stubborn-set-type methods). In the former case actions can be talked about by describing their enabling condition and the change of state they cause. It thus seems that action-based formalisms allow the use of more state space methods.

Action-based approaches often provide automatic insensitivity to stuttering. For instance, if a Büchi automaton is connected to the system with fusion with visible transitions instead of synchronisation to every state, then the automaton does not see stuttering. The acceptance condition of the automaton must then be modified so that executions where the automaton makes only a finite number of transitions are handled properly. This is not a big problem, because something similar has to be done in any case with executions that lead to deadlock states. We saw in Section 4.2 that the action-based approach allows faster on-the-fly detection of errors by facilitating the use of four different kinds of acceptance states for different types of erroneous executions.

We started this article by comparing state space methods to theorem proving. The goal of that was to highlight the advantages and disadvantages of state space methods. Despite the development of advanced state space methods, theorem proving is still much stronger in certain tasks, such as the proving of parameterised results. Sometimes best results are obtained by combining state space methods and theorem proving, for instance, by using a model checker as a sub-tool of a theorem prover, or debugging a system with state space methods and fixed parameter values, but conducting the final correctness proof with a theorem prover and symbolic parameters.

Acknowledgement Numerous researchers have contributed to this article by developing the original ideas, and communicating them to me through their books, papers, talks and discussions. I am afraid that here and there I was unable to trace the true source of an idea, and was unable to give the original inventor the recognition that (s)he deserves — please accept my apologies. Jaco Geldenhuys and Ilkka Kokkarinen helped me to reduce the number of errors in this article.

References

1. Aho, A. V., Hopcroft, J. E. & Ullman, J. D.: *The Design and Analysis of Computer Algorithms.* Addison-Wesley 1974, 470 p.
2. Alpern, B. & Schneider, F. B.: "Defining Liveness". *Information Processing Letters* 21(4), 1985, pp. 181–185.
3. Alur, R., Brayton, R. K., Henzinger, T. A., Qadeer, S. & Rajamani, S. K.: "Partial-Order Reduction in Symbolic State Space Exploration". *Proc. Computer Aided Verification (CAV) '97*, Lecture Notes in Computer Science 1254, Springer-Verlag 1997, pp. 340–351.
4. Bolognesi, T. & Brinksma, E.: "Introduction to the ISO Specification Language LOTOS". *Computer Networks and ISDN Systems* 14 (1987), pp. 25–59.
5. Brinksma, E.: "A Theory for the Derivation of Tests". *Protocol Specification, Testing and Verification VIII* (Proc. International IFIP WG 6.1 Symposium, 1988), North-Holland 1988, pp. 63–74.
6. Brookes, S. D., Hoare, C. A. R. & Roscoe, A. W.: "A Theory of Communicating Sequential Processes". *Journal of the ACM*, 31 (3) 1984, pp. 560–599.
7. Browne, M. C., Clarke, E. M. & Grumberg, O.: "Characterizing Finite Kripke Structures in Propositional Temporal Logic". *Theoretical Computer Science* 59, 1988, pp. 115–131.
8. Bryant, R. E.: "Graph-Based Algorithms for Boolean Function Manipulation". IEEE Transactions on Computers C-35 (8) 1986, pp. 677–691.
9. Burch, J. R., Clarke E. M., McMillan K. L., Dill D. L. & Hwang, L. J.: "Symbolic Model Checking: 10^{20} States and Beyond". Information and Computation 98 (2) 1992, pp. 142–170.
10. Chandy, K. M. & Misra, J.: *Parallel Program Design: A Foundation.* Addison-Wesley 1988, 516 p.

11. Clarke, E. M. & Emerson, E. A.: "Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic". *Proc. Workshop on Logics of Programs*, Lecture Notes in Computer Science 131, Springer-Verlag 1981, pp. 52–71.
12. Clarke, E. M., Filkorn, T. & Jha, S.: "Exploiting Symmetry in Temporal Logic Model Checking". *Proc. Computer-Aided Verification (CAV) '93*, Lecture Notes in Computer Science 697, Springer-Verlag 1993, pp. 450–462.
13. Clarke, E. M., Grumberg, O. & Jha, S.: "Verifying Parameterized Networks using Abstraction and Regular Languages". *Proc. CONCUR '95, 6th International Conference on Concurrency Theory*, Lecture Notes in Computer Science 962, Springer-Verlag 1995, pp. 395–407.
14. Cleaveland, R. & Hennessy, M.: "Testing Equivalence as a Bisimulation Equivalence". *Formal Aspects of Computing*, 5 (1) 1993, pp. 1–20.
15. Cormen, T. H., Leiserson, C. E. & Rivest, R. L.: *Introduction to Algorithms.* The MIT Press, 1990, 1028 p.
16. Courcoubetis, C., Vardi, M., Wolper, P. & Yannakakis, M.: "Memory-Efficient Algorithms for the Verification of Temporal Properties", *Formal Methods in System Design* 1 (1992), pp. 275–288.
17. Desel, J. & Esparza, J.: *Free Choice Petri Nets.* Cambridge Tracts in Theoretical Computer Science 40, Cambridge University Press 1995, 244 p.
18. Eloranta, J., Tienari, M. & Valmari, A.: "Essential Transitions to Bisimulation Equivalences". *Theoretical Computer Science* 179 (1997) pp. 397–419.
19. Emerson, E. A.: "Temporal and Modal Logic". *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, Elsevier Science Publishers 1990, pp. 995–1072.
20. Emerson, E. A., & Halpern, J. Y.: "'Sometimes' and 'Not Never' Revisited: on Branching Versus Linear Time Temporal Logic". *Journal of the ACM* 33 (1) 1986, pp. 151–178.
21. Emerson, E. A., & Lei, C.-L.: "Modalities for Model Checking: Branching Time Strikes Back". *Science of Computer Programming*, 8, 1987, pp. 275–306.
22. Emerson, E. A. & Sistla, A. P.: "Symmetry and Model Checking". *Proc. Computer-Aided Verification (CAV) '93*, Lecture Notes in Computer Science 697, Springer-Verlag 1993, pp. 463–477.
23. Emerson, E. A. & Sistla, A. P.: "Utilizing Symmetry when Model-Checking under Fairness Assumptions: An Automata-Theoretic Approach". *ACM Transactions on Programming Languages and Systems*, 19 (4) 1997, pp. 617–638.
24. Esparza, J.: "Model Checking Using Net Unfoldings". *Science of Computer Programming* (1994) 23: 151–195.
25. Esparza, J., Römer, S. & Vogler, W.: "An Improvement of McMillan's Unfolding Algorithm". *Proc. Tools and Algorithms for the Construction and Analysis of Systems '96*, Lecture Notes in Computer Science 1055, Springer-Verlag 1996, pp. 87–106.
26. Fernandez, J.-C.: "An Implementation of an Efficient Algorithm for Bisimulation Equivalence". *Science of Computer Programming* 13 (1989/90) pp. 219–236.
27. Finkel, A.: "The Minimal Coverability Graph for Petri Nets". *Advances in Petri Nets 1993*, Lecture Notes in Computer Science 674, pp. 210–243.
28. Francez, N.: *Fairness.* Springer-Verlag 1986, 295 p.
29. Francez, N.: *Program Verification.* Addison-Wesley 1992, 312 p.
30. Garey, M. R. & Johnson, D. S.: *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, 1979, 340 p.

31. Gerth, R., Kuiper, R., Peled, D. & Penczek, W.: "A Partial Order Approach to Branching Time Logic Model Checking". *Proc. Third Israel Symposium on the Theory of Computing and Systems*, IEEE 1995, pp. 130–139.
32. Gerth, R., Peled, D., Vardi, M. & Wolper, P.: "Simple On-the-fly Automatic Verification of Linear Temporal Logic". *Proc. Protocol Specification, Testing and Verification 1995*, Chapman & Hall 1995, pp. 3–18.
33. Godefroid, P.: "Using Partial Orders to Improve Automatic Verification Methods". *Proc. Computer-Aided Verification 90*, AMS-ACM DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Vol. 3, 1991, pp. 321–340.
34. Godefroid, P.: *Partial-Order Methods for the Verification of Concurrent Systems, An Approach to the State-Explosion Problem.* Lecture Notes in Computer Science 1032, Springer-Verlag 1996, 143 p. (Earlier version: Ph.D. Thesis, University of Liège, 1994.)
35. Godefroid, P., Holzmann, G. J. & Pirottin, D.: "State Space Caching Revisited". *Proc. Computer-Aided Verification (CAV) '92*, Lecture Notes in Computer Science 663, Springer-Verlag 1993, pp. 178–191.
36. Godefroid, P. & Pirottin, D.: "Refining Dependencies Improves Partial-Order Verification Methods". *Proc. Computer-Aided Verification (CAV) '93*, Lecture Notes in Computer Science 697, Springer-Verlag 1993, pp. 438–449.
37. Godefroid, P., & Wolper, P.: "Using Partial Orders for the Efficient Verification of Deadlock Freedom and Safety Properties". *Proc. Computer Aided Verification (CAV) '91*, Lecture Notes in Computer Science 575, Springer-Verlag 1992, pp. 332–342.
38. Graf, S. & Steffen, B.: "Compositional Minimization of Finite State Processes". *Proc. Computer-Aided Verification '90*, AMS-ACM DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Vol. 3, 1991, pp. 57–73.
39. Gyuris, V. & Sistla, P.: "On-the-Fly Model Checking Under Fairness That Exploits Symmetry". *Proc. Computer Aided Verification (CAV) '97*, Lecture Notes in Computer Science 1254, Springer-Verlag 1997, pp. 232–243.
40. Haddad, S.: "A Reduction Theory for Coloured Nets". *Advances in Petri Nets 1989*, Lecture Notes in Computer Science 424, Springer-Verlag 1990, pp. 209–235. Also in *High-level Petri Nets. Theory and Application*, Springer-Verlag 1991, pp. 399–425.
41. Hennessy, M.: "Acceptance Trees". *Journal of the ACM*, 32 (4) 1985, pp. 896–928.
42. Hoare, C. A. R.: *Communicating Sequential Processes.* Prentice-Hall 1985, 256 p.
43. Holzmann, G. J.: *Design and Validation of Computer Protocols.* Prentice-Hall 1991, 500 p.
44. *ISO 8807 International Standard: Information processing systems — Open Systems Interconnection — LOTOS — A formal description technique based on the temporal ordering of observational behaviour.* International Organization for Standardization 1989, 142 p.
45. Jard, C. & Jéron, T.: "Bounded-memory Algorithms for Verification On-the-fly". *Proc. Computer Aided Verification (CAV) '91*, Lecture Notes in Computer Science 575, Springer-Verlag 1992, pp. 192–202.
46. Jensen, K.: *Coloured Petri Nets. Volume 2, Analysis Methods.* Monographs in Theoretical Computer Science, Springer-Verlag 1995, 174 p.
47. Kaivola, R. & Valmari, A.: "The Weakest Compositional Semantic Equivalence Preserving Nexttime-less Linear Temporal Logic". *Proc. CONCUR '92, Third International Conference on Concurrency Theory*, Lecture Notes in Computer Science 630, Springer-Verlag 1992, pp. 207–221.

48. Kanellakis, P. C. & Smolka, S. A.: "CCS Expressions, Finite State Processes, and Three Problems of Equivalence". *Information and Computation* 86 (1990) pp. 43–68.
49. Katz, S. & Peled, D.: "An Efficient Verification Method for Parallel and Distributed Programs". *Proc. Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency 1988*, Lecture Notes in Computer Science 354, Springer-Verlag 1989, pp. 489–507.
50. Katz, S. & Peled, D.: "Defining Conditional Independence Using Collapses". *Theoretical Computer Science* 101 (1992), pp. 337–359.
51. Kokkarinen, I.: *A Verification-Oriented Theory of Data in Labelled Transition Systems.* Ph.D. Thesis, Tampere University of Technology Publications 234, Tampere, Finland 1998, 105 p.
52. Kokkarinen, I., Peled, D. & Valmari, A.: "Relaxed Visibility Enhances Partial Order Reduction". *Proc. Computer Aided Verification (CAV) '97*, Lecture Notes in Computer Science 1254, Springer-Verlag 1997, pp. 328–339.
53. Kristensen, L. M. & Valmari, A.: "Finding Stubborn Sets of Coloured Petri Nets Without Unfolding". To appear in *Proc. International Conference on Application and Theory of Petri Nets*, 1998, 20 p.
54. Kurshan, R. P., Merritt, M., Orda, A. & Sachs, S. R.: "A Structural Linearization Principle for Processes". *Formal Methods in System Design* 5, 1994, pp. 227–244.
55. Lamport, L.: "Proving the Correctness of Multiprocess Programs". *IEEE Transactions on Software Engineering*, SE-3(2), 1977, pp. 125–143.
56. Lamport, L. & Lynch, N.: "Distributed Computing: Models and Methods". *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, Elsevier Science Publishers 1990, pp. 1157–1199.
57. Lichtenstein, O. & Pnueli, A.: "Checking that Finite State Concurrent Programs Satisfy Their Linear Specifications". *Proc. 12th ACM Symposium on Principles of Programming Languages*, 1985, pp. 97–107.
58. Madelaine, E. & Vergamini, D.: "AUTO: A Verification Tool for Distributed Systems Using Reduction of Finite Automata Networks". *Proc. Formal Description Techniques II (FORTE '89)*, North-Holland 1990, pp. 61–66.
59. Manna, Z. & Pnueli, A.: *The Temporal Logic of Reactive and Concurrent Systems, Volume I: Specification.* Springer-Verlag 1992, 427 p.
60. Manna, Z. & Pnueli, A.: *Temporal Verification of Reactive Systems, Volume II: Safety.* Springer-Verlag 1995, 512 p.
61. Mazurkiewicz, A.: "Trace Theory". *Petri Nets: Applications and Relationships to Other Models of Concurrency*, Lecture Notes in Computer Science 255, Springer-Verlag 1987, pp. 279–324.
62. McMillan, K.: "Using Unfoldings to Avoid the State Explosion Problem in the Verification of Asynchronous Circuits". *Proc. Computer-Aided Verification (CAV) '92*, Lecture Notes in Computer Science 663, Springer-Verlag 1993, pp. 164–177.
63. Meinel, C. & Theobald, T.: "Ordered Binary Decision Diagrams and Their Significance in Computer-Aided Design of VLSI Circuits". *Bulletin of the European Association for Theoretical Computer Science* 64, 1998, pp. 171–187.
64. Melzer, S. & Römer, S.: "Deadlock Checking Using Net Unfoldings". *Proc. Computer Aided Verification (CAV) '97*, Lecture Notes in Computer Science 1254, Springer-Verlag 1997, pp. 352–363.
65. Milner, R.: *Communication and Concurrency.* Prentice-Hall 1989, 260 p.
66. Park, D.: "Concurrency and Automata on Infinite Sequences". *Theoretical Computer Science: 5th GI-Conference*, Lecture Notes in Computer Science 104, Springer-Verlag 1981, pp. 167–183.

67. Pastor, E., Roig, O., Cortadella, J. & Badia, R.: "Petri Net Analysis Using Boolean Manipulation". *Proc. Application and Theory of Petri Nets 1994*, Lecture Notes in Computer Science 815, Springer-Verlag 1994, pp. 416-435.
68. Peled, D.: "All from One, One for All: On Model Checking Using Representatives". *Proc. Computer-Aided Verification (CAV) '93*, Lecture Notes in Computer Science 697, Springer-Verlag 1993, pp. 409–423.
69. Peled, D.: "Combining Partial Order Reductions with On-the-fly Model-Checking". *Formal Methods in System Design* 8 (1) 1996: 39–64.
70. Peled, D.: "Partial Order Reduction: Linear and Branching Temporal Logics and Process Algebras". *Proc. POMIV'96, Workshop on Partial Order Methods in Verification*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science Vol. 29, American Mathematical Society 1997, pp. 233–257.
71. Puhakka, A. & Valmari, A.: "Verification of Self-Synchronizing Alternating Bit Protocols with ARA". *Proc. Fifth Symposium on Programming Languages and Software Tools*, University of Helsinki, Department of Computer Science, Report C-1997-37, pp. 167–178.
72. Rauhamaa, M.: *A Comparative Study of Methods for Efficient Reachability Analysis.* Lic.Tech. Thesis, Helsinki University of Technology, Digital Systems Laboratory, Research Report A-14, Espoo, Finland 1990, 61 p.
73. Reisig, W.: *Petri Nets, An Introduction.* EATCS Monographs on Theoretical Computer Science, Vol. 4, Springer-Verlag 1985, 161 p.
74. Roscoe, A. W.: "Model-Checking CSP". *A Classical Mind: Essays in Honour of C. A. R. Hoare*, Prentice-Hall 1994, pp. 353–378.
75. Roscoe, A. W.: *The Theory and Practice of Concurrency.* Prentice-Hall 1998, 565 p.
76. Savitch, W. J.: "Relationships Between Nondeterministic and Deterministic Tape Complexities". *Journal of Computer and System Sciences* 4, 1970, pp. 177–192.
77. Shatz, S. M., Tu, S., Murata, T. & Duri, S.: "Application of Petri Net Reduction for Ada Tasking Deadlock Analysis". *IEEE Transactions on Parallel and Distributed Systems* 7 (12) 1996, pp. 1307–1322.
78. Sistla, A. P. & Clarke, E. M.: "The Complexity of Propositional Linear Temporal Logics". *Journal of the ACM* 32 (3) 1985, pp. 733–749.
79. Tarjan, R. E.: "Depth-first Search and Linear Graph Algorithms". *SIAM Journal on Computing*, 1 (2) 1972, pp. 146–160.
80. Thomas, W.: "Automata on Infinite Objects". *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, Elsevier Science Publishers 1990, pp. 133–191.
81. Valmari, A.: "Error Detection by Reduced Reachability Graph Generation". *Proc. 9th European Workshop on Application and Theory of Petri Nets*, 1988, pp. 95–112.
82. Valmari, A.: *State Space Generation: Efficiency and Practicality.* Ph.D. Thesis, Tampere University of Technology Publications 55, Tampere, Finland 1988, 169 p.
83. Valmari, A.: "Stubborn Sets for Reduced State Space Generation". *Advances in Petri Nets 1990*, Lecture Notes in Computer Science 483, Springer-Verlag 1991, pp. 491–515.
84. Valmari, A.: "Stubborn Sets of Coloured Petri Nets". *Proc. 12th International Conference on Application and Theory of Petri Nets*, 1991, pp. 102–121.
85. Valmari, A.: *Alleviating State Explosion during Verification of Behavioural Equivalence.* Department of Computer Science, University of Helsinki, Report A-1992-4, Helsinki, Finland 1992, 57 p.

86. Valmari, A.: "A Stubborn Attack on State Explosion". *Formal Methods in System Design*, 1: 297–322 (1992).
87. Valmari, A.: "On-the-fly Verification with Stubborn Sets". *Proc. Computer-Aided Verification (CAV) '93*, Lecture Notes in Computer Science 697, Springer-Verlag 1993, pp. 397–408.
88. Valmari, A.: "Compositional Analysis with Place-Bordered Subnets". *Proc. Application and Theory of Petri Nets 1994*, Lecture Notes in Computer Science 815, Springer-Verlag 1994, pp. 531–547.
89. Valmari, A.: "Failure-based Equivalences Are Faster Than Many Believe". *Proc. Structures in Concurrency Theory 1995*, Springer-Verlag "Workshops in Computing" series, 1995, pp. 326–340.
90. Valmari, A.: "Compositionality in State Space Verification Methods". Invited talk, *Proc. Application and Theory of Petri Nets 1996*, Lecture Notes in Computer Science 1091, Springer-Verlag 1996, pp. 29–56.
91. Valmari, A.: "Stubborn Set Methods for Process Algebras". *Proc. POMIV'96, Workshop on Partial Order Methods in Verification*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science Vol. 29, American Mathematical Society 1997, pp. 213–231.
92. Valmari, A. & Kokkarinen, I.: "Unbounded Verification Results by Finite-State Compositional Techniques: 10^{any} States and Beyond". *Proc. 1998 International Conference on Application of Concurrency to System Design*, IEEE Computer Society 1998, pp. 75–85.
93. Valmari, A. & Tienari, M.: "An Improved Failures Equivalence for Finite-State Systems with a Reduction Algorithm". *Proc. Protocol Specification, Testing and Verification XI*, North-Holland 1991, pp. 3–18.
94. Valmari, A. & Tienari, M.: "Compositional Failure-Based Semantic Models for Basic LOTOS". *Formal Aspects of Computing* (1995) 7: 440–468.
95. van Glabbeek, R.: "The Linear Time — Branching Time Spectrum II: The Semantics of Sequential Systems with Silent Moves". *Proc. CONCUR '93, Fourth International Conference on Concurrency Theory*, Lecture Notes in Computer Science 715, Springer-Verlag 1993, pp. 66–81.
96. van Glabbeek, R. & Weijland, W.: "Branching Time and Abstraction in Bisimulation Semantics (Extended Abstract)". *Proc. IFIP International Conference on Information Processing '89*, North-Holland 1989, pp. 613–618.
97. Vardi, M. Y. & Wolper, P.: "An Automata-Theoretic Approach to Automatic Program Verification". *Proc. IEEE Symposium on Logic in Computer Science*, 1986, pp. 332–344.
98. Varpaaniemi, K.: *On the Stubborn Set Method in Reduced State Space Generation*. Ph.D. Thesis, Helsinki University of Technology, Digital Systems Laboratory, Research Report A-51, Espoo, Finland 1998, 105 p.
99. Wolper, P.: "Expressing Interesting Properties of Programs in Propositional Temporal Logic". *Proc. 13th ACM Symposium on Principles of Programming Languages*, 1986, pp. 184–193.
100. Wolper, P. & Lovinfosse, V.: "Verifying Properties of Large Sets of Processes with Network Invariants". *Proc. Workshop on Automatic Verification Methods for Finite State Systems*, Lecture Notes in Computer Science 407, Springer-Verlag 1989, pp. 68–80.