Composition and Abstraction

Antti Valmari

This text appeared as Valmari, A.: "Composition and Abstraction", Modeling and Verification of Parallel Processes, 4th Summer School, MOVEP 2000, Nantes, France, June 2000, Revised Tutorial lectures, *Lecture Notes in Computer Science* 2067, Springer-Verlag 2001, pp. 58–98. The division of text to pages is different between this and the printed version. I am not aware of any other differences.

Composition and Abstraction

Antti Valmari

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

Abstract. This article is a tutorial on advanced automated process-algebraic verification of concurrent systems, and it is organised around a case study. The emphasis is on verification methods that rely on the inherent compositionality of process algebras. The fundamental concepts of labelled transition systems, strong bisimilarity, synchronous parallel composition, hiding, renaming, abstraction, CFFD-equivalence and CFFD-preorder are presented as the case study proceeds. The necessity of presenting assumptions about the users of the example system is discussed, and it is shown how CFFD-preorder supports their modelling. The assumptions are essential for the verification of so-called liveness properties. The correctness requirements of the system are stated, presented in linear temporal logic, and distributed to a number of more "localised" requirements. It is shown how they can be checked with the aid of suitably chosen CFFD-abstracted views to the system. The state explosion problem that hampers automatic verification is encountered. Compositional LTS construction, interface specifications and induction are used to solve the problem and, as a result, an infinite family of systems is verified with a limited amount of effort.

1 Introduction

The goal of this article is to introduce some advanced automated process-algebraic verification methods that are based on abstraction and compositionality. The article is organised around a case study: the verification of a demand-driven token-ring mutual exclusion system. The verification methods and their underlying theory are introduced as the case study proceeds.

The article has been primarily intended for potential users of the methods, rather than verification theory researchers or tool developers. Therefore, although the article contains mathematical definitions and theorems, it presents neither proofs nor algorithms, with the exception that the basic reason why a theorem holds is sometimes explained.

Section 2 presents the system and its verification model. The structure of the system is described in Section 2.1, as is also the basic idea of the operation of the system. The system consists of n *servers* and n *clients*. The servers are modelled in Section 2.2 in *ARA Lotos*. "ARA" [37] is an abbreviation for "Advanced Reachability Analysis", and it is the name of the main tool used in the case

study. (The other tool that was used is *Ltspar* [19].) "ARA Lotos" is the input language of the tool, and it resembles closely the internationally standardised specification language Lotos [2, 13].

A *labelled transition system*, or *LTS* for brevity, is a graph-like representation of the behaviour of a system or its component. The concept is introduced and defined in Section 2.3, where also LTSs that show the behaviours of the servers are shown. Section 2.4 discusses what it means for two LTSs to represent the "same" behaviour, and introduces the important notions of *isomorphism* and *strong bisimilarity* between LTSs.

The clients that use the system are presented in Section 2.5. Special attention is paid to the subtle issue of modelling the *progress properties* of the clients in an adequate way.

The system is put together with the *parallel composition* operator. It and two other useful operators, namely *renaming* and *hiding*, are defined in Section 2.6. Hiding and renaming are presented in a slightly more general form than usually in process-algebraic literature. The section also introduces the notorious *state explosion* phenomenon. The parallel composition operator used in this article is *synchronous*, while real systems are often based on *asynchronous* communication. This apparent incompatibility is discussed and the use of a synchronous operator is justified in Section 2.7. The operator is also compared to two well-known synchronous alternatives, namely the Lotos and CCS [23] parallel composition operators. It is shown that the latter two can be simulated with the operator in this article, so its use does not restrict generality. Also the fact that *interleaving semantics* is used instead of *true parallelism* is discussed a bit.

Section 3 is devoted to the verification of the system. It starts by stating the requirements of the system verbally and in *linear temporal logic* [22], and by transforming them into a total of $\frac{1}{2}n(n+1)$ requirements that refer to at most two clients at a time.

Then, in Section 3.2, process-algebraic *abstraction* is introduced by showing two *reduced* abstract views to the system. It is discussed how the validity of the requirements can be checked from such abstract views. The particular formal notion of abstraction used in this case study is the *CFFD semantics* [40, 41]. It is defined and its relevance to the verification of the requirements of the system is discussed in Section 3.3. Special attention is paid to how the properties of *CFFD-preorder* can be taken advantage of in the modelling of the clients. Also, what can and what can not be read from an LTS obtained with the CFFD semantics is discussed. The reason for the name "CFFD" is explained in Section 3.4, where the CFFD semantics is compared to the main semantic model in the famous *CSP* theory [28, 4, 11]. (The version of CSP semantics described in [3] was not the final one.)

The first of the advanced verification methods introduced in this article, namely *compositional LTS construction*, is presented in Section 3.5. The method is a direct application of the congruence property of many process-algebraic equivalences, and has been in use since [21], or perhaps earlier. Section 3.6 shows how *interface specifications* can be used to improve the compositional method in

situations where a subsystem has much more behaviour in isolation than it has in its intended environment. Interface specifications were first presented in [9], and they have not been applied to the CFFD semantics before this article. Finally, *induction* in the style of [44, 20, 5] is applied to the system in Section 3.7. As a result, the demand-driven token-ring system becomes verified for any $n \geq 2$ with a limited finite amount of effort.

Despite of its length, the case study cannot be used for illustrating all advanced verification methods. Further reading is suggested and other final remarks are made in Section 4.

2 Systems and Their Behaviour

2.1 The architecture of the example system

Figure 1 shows the architecture of the example system and the environment where it is used. The system is a ring of n processes Server_1, Server_2, ..., Server_n. Each server has one *client*. The purpose of the system is to act as an arbiter between the clients, so that at most one client at a time can perform some critical activity. The system is thus a distributed mutual exclusion system.

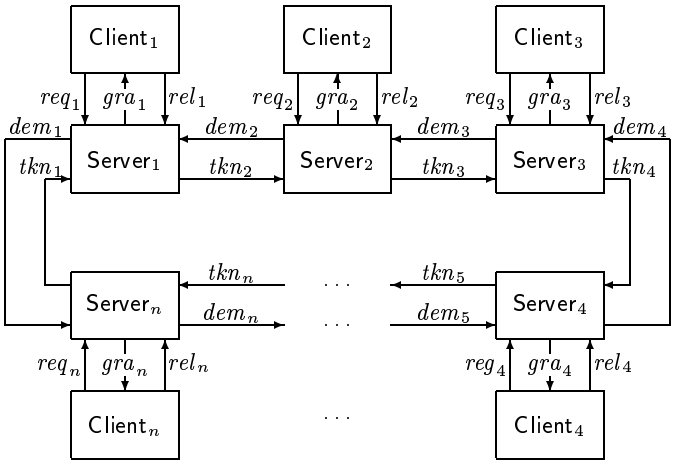

Fig. 1. The architecture of the example system.

When a client wants to perform the critical activity, it requests permission from its server by executing req_i, where i is the number of the client. The server may grant the permission by executing gra_i. After the client has finished with the critical activity, it indicates this to its server by executing rel_i (for "release"), so that the system knows that now some other client can be given the permission.

The ring of servers contains precisely one *token*. A server can grant permission to its client only when it is in possession of the token. If a server needs the token but does not have it, then the server demands it from the previous server in the ring by executing dem_i. Here i is the number of the server that needs the token, and the number of the previous server is $i-1$, except that the predecessor of Server_1 is Server_n. If the previous server does not have the token, then it forwards the demand to its predecessor, and so on.

When the demand reaches the server Server_j that has the token, this server (if necessary) first waits until its client is not doing the critical activity. Then it delivers the token to the next server in the ring by executing $tkn_{j\oplus 1}$ (where $n \oplus 1 = 1$, and $x \oplus 1 = x+1$ when $1 \leq x < n$). The next server forwards the token to its successor either immediately or after first serving its own client. In this way the token eventually propagates to the server who made the original demand.

In order to prevent the system from serving only one client and ignoring the rest, any server that has just served its client gives the token to the next server even if that server has not demanded it. As a consequence, the server can serve its client again only after the token has made a full circle in the ring, and thus all the other clients have had a chance to get service.

2.2 The servers

Although the description of the system might sound simple, the server is actually difficult to specify precisely. It has to communicate with three other processes: its client, and the previous and the next server in the ring. A message may come from one direction while the server is engaged in an interaction in another direction, and this message may have an effect on the interaction. For instance, while the server is waiting for the token to satisfy a pending demand by the next server, the client may execute req_i, so that the server must serve its client before delivering the token to the next server.

Figure 2 shows ARA Lotos code that specifies a server. The first line declares a new enumerated type that consists of five values, and gives it the name **tkn_status**. The next non-empty line specifies that the name of the process is Server and the actions with which it communicates with the rest of the world are *tl*, *tr*, ..., and *rel*. In the code, *tl* replaces tkn_i ("l" is a mnemonic for "left") and *tr* replaces $tkn_{i\oplus 1}$ ("r"for "right"), and similarly with *dem*. Furthermore, the subscripts of *req*, *gra* and *rel* have been omitted.

The next line declares three local variables t_st, is_req and is_dem. They are used for keeping track of what the server knows about the location of the token, and whether a request by the client or a demand by the next server is pending. The latter two variables are of the type **Bool**, so they may assume the values False and True, and they can be used in conditions as such. The variable t_st is of the type **tkn_status** that has been defined in the first line as the set { t_out, t_dem, t_in, t_down, t_used }. The first two of these values mean that the server does not have the token and has (t_dem) or has not (t_out) requested it. In the last three cases the server has the token, and the client has not started the

```
$sort tkn_status is ( t_out, t_dem, t_in, t_down, t_used )

process Server[ tl, tr, dl, dr, req, gra, rel ](
  t_st : tkn_status, is_req, is_dem : Bool
) :=
  [ ( is_req or is_dem ) and ( t_st == t_out ) ]->
  dl; S[...]( t_dem, is_req, is_dem )
    []
  [ ( t_st == t_dem ) or ( t_st == t_out ) ]->
  tl; S[...]( t_in, is_req, is_dem )
    []
  [ not( is_req ) ]-> req; S[...]( t_st, True, is_dem )
    []
  [ ( t_st == t_in ) and is_req ]->
  gra; S[...]( t_down, is_req, is_dem )
    []
  [ t_st == t_down ]-> rel; S[...]( t_used, False, is_dem )
    []
  dr; S[...]( t_st, is_req, True )
    []
  [ ( t_st == t_used ) ]-> tr; S[...]( t_out, is_req, False )
    []
  [ is_dem and ( t_st == t_in ) and not( is_req ) ]->
  tr; S[...]( t_out, is_req, False )
endproc
```

Fig. 2. A server specified in ARA Lotos.

critical activity (t_in), is performing it (t_down), or has finished it (t_used). The figure does not show the initial values of the variables. Initially is_req and is_dem are False, and t_st is t_in in the server that initially has the token, and t_out in the other servers.

Between the symbols ":=" and "**endproc**" the specification lists eight possible ways the server can make a transition, separated with "[]". The notation "[*cond*]→" specifies a condition *cond* that must hold for the transition to be possible. In the one place where it is missing, the corresponding transition is always enabled. Then comes the name of the action that is executed during the transition, and finally the new values of the local variables are listed in parenthesis. The text "S[...]" is neither ARA Lotos nor ordinary Lotos; it is an abbreviation used only in Figure 2 to make it more readable. In ARA Lotos and ordinary Lotos the string "`Server[ tl, tr, dl, dr, req, gra, rel ]`" occurs in its place. It denotes Server calling itself recursively.

For example, the first transition specifies that the server can make a *dl*-transition whenever there is a pending request from the client or from the next server, and token status is t_out. When the transition is made, the token status

changes to t_dem, so the transition cannot be made immediately again. The other two local variables retain their values.

As another example, the last two transitions specify that *tr* can be executed because of two reasons, and in both cases it sets the token status to t_out and marks off any pending demand from the next server. The first reason is that the client has done its critical activity. It implements the requirement that after serving the client, the server pushes the token to the next server even if it has not demanded it. The second reason is that the server has the token, the next server has demanded it, and the client has not. This last conjunct implies that the client is given priority, if both it and the next server need the token at the same time.

Although the basic idea of the server is simple, its design contains many details, and some of them are subtle. It is not easy to convince oneself merely by looking at the code that the server is correct. In the remainder of this article we will verify that it indeed is.

2.3 Labelled transition systems

The behaviour of a process-algebraic system can be represented as a *labelled transition system*, abbreviated *LTS*. It is a directed graph whose edges are labelled with action names. The vertices are called *states*, and one of them is distinguished as the *initial state*.[1] The edges together with their labels are called *transitions*.

There are two kinds of actions: the system communicates with its environment by executing *visible actions*, and it can also execute *invisible actions* that the environment cannot directly observe. The set of the names of the visible actions of the system is called the *alphabet*, and it is also considered as a part of the LTS. Invisible actions do not have names. Instead, a special symbol "τ" that is not in the alphabet is used to denote them.

Definition 1. *A* labelled transition system*, abbreviated* LTS*, is a four-tuple* $(S, \Sigma, \Delta, \hat{s})$*, where*

- S *is the set of* states.
- Σ*, also known as the* alphabet*, is the set of* visible actions*. It satisfies* $\tau \notin \Sigma$.
- $\Delta \subseteq S \times (\Sigma \cup \{\tau\}) \times S$ *is the set of transitions.*
- $\hat{s} \in S$ *is the* initial state.

The definition implies that the alphabet contains at least those visible actions that the system can execute, but it can contain more. When we later define parallel composition we will see that the "extra" elements in the alphabet are significant.

In drawings, the states of an LTS are usually denoted with circles, and the initial state is indicated with a small arrow that starts at no state. Any transition

[1] Sometimes it is useful to specify more than one initial state, or specify no initial states.

(s, a, s') is represented as an arrow from s to s' with a written somewhere near the arrow. Sometimes an arrow is labelled with several action names; then it represents several transitions between the same states. Unless otherwise specified either in the drawing or in its accompanying text, the alphabet of the LTS is precisely the set of the visible action names that occur as transition labels in the drawing. The names of the states are often considered insignificant, and are omitted in the drawing.

Figure 3 shows the behaviours of the servers as an LTS with two initial states. The initial state of the server that has the token initially is the third state in the top row, and the initial state of the other servers is the second state in the same row. Several easily understandable transition sequences can be found in the figure, such as the *dr*-*dl*-*tl*-*tr*-sequence from the leftmost initial state to itself. In it, the next server demands the token (dr), the current server forwards the demand to the previous server in the ring (dl), the token arrives from the previous server (tl), and is delivered to the next server (tr). More complicated behaviour arises if a request from the client is interleaved with the demand by the next server, or if the token arrives unexpectedly. The servers have been designed to treat reasonably all these possibilities.

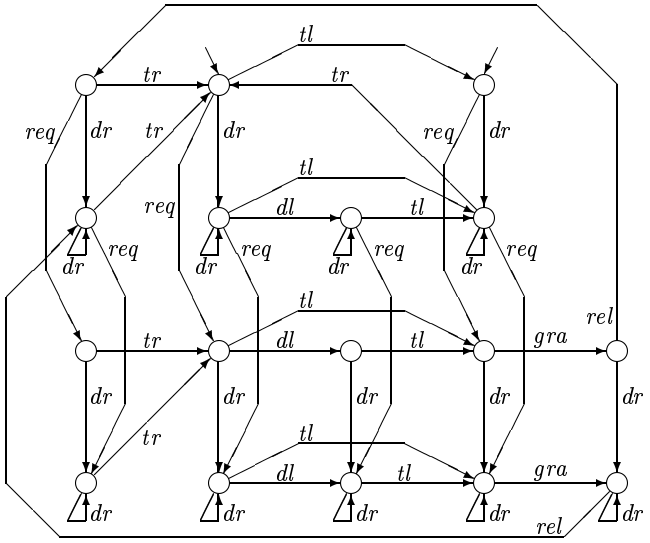

Fig. 3. The LTS of a server that initially has ("↙") or has not ("↘") the token.

The states in Figure 3 are anonymous, and thus do not make the contents of the local variables t_st, is_req and is_dem explicit. However, the figure has been drawn such that the location of a state reveals the values of the local variables. The value of t_st is t_used in the first column, t_out in the second, and t_dem, t_in and t_down in the third, fourth and last column. The bit is_req is True in

precisely the two bottom rows, and is_dem = True in the second row and bottom row.

The "$(s, a, s') \in \Delta$" notation is inconvenient for talking about sequences of transitions. We therefore define a notation where only the first and optionally also the last state are mentioned, and the actions executed along a path from the first to the last state are listed within an arrow symbol. We will also need symbols for the sets of finite and infinite sequences of elements of a given set.

Definition 2. *Let $(S, \Sigma, \Delta, \hat{s})$ be an LTS, $s, s' \in S$, $n \geq 0$, and $a_1, a_2, \ldots \in \Sigma \cup \{\tau\}$.*

- $s -a_1 a_2 \cdots a_n\rightarrow s'$ *if and only if*
  $\exists s_0, s_1, \ldots, s_n \in S : s_0 = s \wedge s_n = s' \wedge \forall i \in \{1, 2, \ldots, n\} : (s_{i-1}, a_i, s_i) \in \Delta$.
- $s -a_1 a_2 \cdots a_n\rightarrow$ *if and only if* $\exists s' : s -a_1 a_2 \cdots a_n\rightarrow s'$.
- $s -a_1 a_2 a_3 \cdots\rightarrow$ *if and only if*
  $\exists s_0, s_1, s_2, \ldots \in S : s_0 = s \wedge \forall i \in \{1, 2, 3, \ldots\} : (s_{i-1}, a_i, s_i) \in \Delta$.

Let A be a set.

- ε *denotes the empty sequence, that is,* $a_1 a_2 \cdots a_n = \varepsilon$ *whenever* $n = 0$.
- A^* *is the set of finite sequences of elements of A, that is,*
  $A^* = \left\{ a_1 a_2 \cdots a_n \;\middle|\; n \geq 0 \wedge a_1 \in A \wedge \cdots \wedge a_n \in A \right\}$.
- A^ω *is the set of infinite sequences of elements of A, that is,*
  $A^\omega = \left\{ a_1 a_2 a_3 \cdots \;\middle|\; \forall i \in \{1, 2, 3, \ldots\} : a_i \in A \right\}$.

It follows from the definitions that $s -\varepsilon\rightarrow s$ holds for every $s \in S$.

If $s -a_1 a_2 \cdots a_n\rightarrow s'$ holds for some $a_1, a_2, \ldots, a_n$, then we say that s' is *reachable* from s. The *reachable part* of an LTS is obtained by discarding those states and transitions that are not reachable from the initial state.

Definition 3. *Let $P = (S, \Sigma, \Delta, \hat{s})$ be an LTS. The* reachable part *of P is* $\mathsf{repa}(P) = (S', \Sigma', \Delta', \hat{s}')$, *where*

- $S' = \left\{ s \in S \;\middle|\; \exists \sigma \in (\Sigma \cup \{\tau\})^* : \hat{s} -\sigma\rightarrow s \right\}$,
- $\Sigma' = \Sigma$,
- $\Delta' = \Delta \cap (S' \times (\Sigma \cup \{\tau\}) \times S')$, *and*
- $\hat{s}' = \hat{s}$.

As defined above, the arrow notation does not make explicit the LTS whose transition relation is in question. Therefore, we sometimes decorate the arrow in the same way as the LTS, such as in $s -a\rightarrow' s'$ and $s -a\rightarrow_1 s'$, when the LTSs are P' and P_1.

2.4 "Sameness" of LTSs

The obvious mathematical definition of the equivalence of two LTSs is equality: $(S, \Sigma, \Delta, \hat{s}) = (S', \Sigma', \Delta', \hat{s}')$ if and only if $S = S'$, $\Sigma = \Sigma'$, $\Delta = \Delta'$, and $\hat{s} = \hat{s}'$. However, the names of the states of an LTS are usually considered insignificant

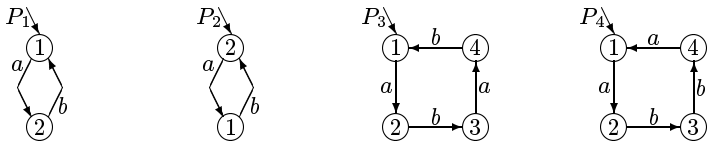

Fig. 4. Four LTSs with varying level of differences.

in process-algebraic verification, and this definition distinguishes between two LTSs that are otherwise the same but happen to use different names for states. For instance, it distinguishes between P_1 and P_2 in Figure 4. In the figure, the names of the states are the numbers drawn within the circles.

What mathematicians usually do in this kind of a situation is to use *isomorphism* instead of equality as the notion of "sameness". Two LTSs are isomorphic, if and only if one can be converted to the other by changing the names of the states.

Definition 4. *Let $P = (S, \Sigma, \Delta, \hat{s})$ and $P' = (S', \Sigma', \Delta', \hat{s}')$ be LTSs.*

- *An* isomorphism *is a bijection (that is, a 1–1 correspondence) $f : S \to S'$ such that $f(\hat{s}) = \hat{s}'$ and $(s_1, a, s_2) \in \Delta \Leftrightarrow (f(s_1), a, f(s_2)) \in \Delta'$.*
- *P and P' are* isomorphic*, denoted $P =_{\mathsf{iso}} P'$, if and only if $\Sigma = \Sigma'$ and there is an isomorphism $f : S \to S'$.*

Although we will use isomorphism of LTSs a couple of times in this article, it is usually considered too strict a notion of "sameness" of LTSs. It unites P_1 and P_2 in Figure 4, but distinguishes between P_1 and P_3. However, the behaviours represented by P_1 and P_3 look the same to an external observer that cannot directly see the state the LTS is in, but sees the actions it makes. On the other hand, P_1 should be distinguished from P_4, because the latter can do two b-actions in a row, but the former cannot.

Strong bisimilarity [23, 25] (or just *bisimilarity*) is a relation that is well-suited for comparing LTSs. It unifies the above-mentioned P_1, P_2 and P_3, but distinguishes them from P_4. Its definition resembles the definition of the isomorphism of LTSs, but instead of a bijection f, a binary relation "$\sim$" called *strong bisimulation* is used.

Definition 5. *Let $P = (S, \Sigma, \Delta, \hat{s})$ and $P' = (S', \Sigma', \Delta', \hat{s}')$ be LTSs.*

- *The relation "$\sim$" $\subseteq S \times S'$ is a* strong bisimulation*, if and only if for every s_1, $s_2 \in S$, s_1', $s_2' \in S'$ and $a \in \Sigma \cup \{\tau\}$:*
  - *If $s_1 \sim s_1'$ and $(s_1, a, s_2) \in \Delta$, then there is an $s' \in S'$ such that $s_2 \sim s'$ and $(s_1', a, s') \in \Delta'$.*
  - *If $s_1 \sim s_1'$ and $(s_1', a, s_2') \in \Delta'$, then there is an $s \in S$ such that $s \sim s_2'$ and $(s_1, a, s) \in \Delta$.*
- *P and P' are* strongly bisimilar*, denoted $P =_{\mathsf{sb}} P'$, if and only if $\Sigma = \Sigma'$, and there is a strong bisimulation "$\sim$" $\subseteq S \times S'$ such that $\hat{s} \sim \hat{s}'$.*

In Figure 4, the relation "$\sim$" that satisfies $1 \sim 2$, $1 \sim 4$, $2 \sim 1$, $2 \sim 3$ and $x \not\sim y$ in all the remaining cases, is a strong bisimulation between the states of P_2 and the states of P_3, thus $P_2 =_{\mathsf{sb}} P_3$.

If two LTSs are isomorphic, then they are also strongly bisimilar, but the opposite does not always hold. This is because a relation between S and S' is an isomorphism if and only if it is simultaneously a strong bisimulation and a bijection, and $\hat{s} \sim \hat{s}'$.

Now that our relation of "sameness" is no more the equality, we have to be careful that the relation is appropriate for its purpose. Namely, the relation must be an *equivalence*.

Definition 6. *The relation "$=_{\mathsf{x}}$" is an* equivalence*, if and only if for every P_1, P_2 and P_3:*

- $P_1 =_{\mathsf{x}} P_1$ *(*reflexivity*).*
- *If $P_1 =_{\mathsf{x}} P_2$, then $P_2 =_{\mathsf{x}} P_1$ (*symmetry*).*
- *If $P_1 =_{\mathsf{x}} P_2$ and $P_2 =_{\mathsf{x}} P_3$, then $P_1 =_{\mathsf{x}} P_3$ (*transitivity*).*

Both isomorphism and strong bisimilarity are equivalences. This holds because if $(S_1, \Sigma_1, \Delta_1, \hat{s}_1)$, $(S_2, \Sigma_2, \Delta_2, \hat{s}_2)$ and $(S_3, \Sigma_3, \Delta_3, \hat{s}_3)$ are LTSs, then

- The identity function $id : S_1 \to S_1 : id(s) = s$ is an isomorphism.
- If $f : S_1 \to S_2$ is an isomorphism, then its inverse function $f^{-1} : S_2 \to S_1$ that satisfies $f^{-1}(f(s_1)) = s_1$ and $f(f^{-1}(s_2)) = s_2$ exists and is an isomorphism.
- If $f : S_1 \to S_2$ and $g : S_2 \to S_3$ are isomorphisms, then $h : S_1 \to S_3 : h(s) = g(f(s))$ is an isomorphism.
- The identity relation "$=_{\mathsf{id}}$" $\subseteq S_1 \times S_1 : s =_{\mathsf{id}} s' \Leftrightarrow s = s'$ is a strong bisimulation such that $\hat{s}_1 =_{\mathsf{id}} \hat{s}_1$.
- If "$\sim$" $\subseteq S_1 \times S_2$ is a strong bisimulation between the states of P_1 and P_2 such that $\hat{s}_1 \sim \hat{s}_2$, then "$\sim^{-1}$" $\subseteq S_2 \times S_1 : s_2 \sim^{-1} s_1 \Leftrightarrow s_1 \sim s_2$ is a strong bisimulation between the states of P_2 and P_1 such that $\hat{s}_2 \sim^{-1} \hat{s}_1$.
- If "$\sim_{12}$" $\subseteq S_1 \times S_2$ and "$\sim_{23}$" $\subseteq S_2 \times S_3$ are strong bisimulations such that $\hat{s}_1 \sim_{12} \hat{s}_2$ and $\hat{s}_2 \sim_{23} \hat{s}_3$, then "$\sim_{13}$" $\subseteq S_1 \times S_3 : s_1 \sim_{13} s_3 \Leftrightarrow \exists s_2 \in S_2 : s_1 \sim_{12} s_2 \wedge s_2 \sim_{23} s_3$ is a strong bisimulation such that $\hat{s}_1 \sim_{13} \hat{s}_3$.

Another important feature that strong bisimilarity shares with isomorphism is that the ability to simulate extends from individual transitions to sequences of transitions of any finite or even infinite length.

Theorem 1. *Let $(S, \Sigma, \Delta, \hat{s})$ and $(S', \Sigma', \Delta', \hat{s}')$ be LTSs, $s_0, s_1, s_2, \ldots \in S$, a_1, a_2, $\ldots \in \Sigma \cup \{\tau\}$, and $s'_0 \in S'$. Assume that $s_0 \,{-a_1\to}\, s_1 \,{-a_2\to}\, \cdots \,{-a_n\to}\, s_n \,{-a_{n+1}\to}\, \cdots$.*

- *If $f : S \to S'$ is an isomorphism, then $f(s_0) \,{-a_1\to'}\, f(s_1) \,{-a_2\to'}\, \cdots \,{-a_n\to'}\, f(s_n) \,{-a_{n+1}\to'}\, \cdots$.*
- *If "$\sim$" $\subseteq S \times S'$ is a strong bisimulation and $s_0 \sim s'_0$, then there are s'_1, s'_2, $\ldots$, s'_n, $\ldots$ such that $s'_0 \,{-a_1\to'}\, s'_1 \,{-a_2\to'}\, \cdots \,{-a_n\to'}\, s'_n \,{-a_{n+1}\to'}\, \cdots$ and $s_1 \sim s'_1$, $s_2 \sim s'_2$, $\ldots$, $s_n \sim s'_n$, $\ldots$.*

The claim remains valid if the sequences are ended at the nth state.

An important difference between strong bisimilarity and isomorphism is that strong bisimilarity of an LTS with another LTS depends only on those states and transitions that are reachable from the initial state of the LTS. This is because a strong bisimulation remains a strong bisimulation, when the pairs (s, s') are removed where s or s' is not reachable from the corresponding initial state.

Some above-mentioned and other facts about isomorphism and strong bisimilarity are collected into the following theorem.

Theorem 2. *Let P and Q be LTSs.*

- *"$=_{\mathsf{iso}}$" and "$=_{\mathsf{sb}}$" are equivalences.*
- *If $P =_{\mathsf{iso}} Q$, then $P =_{\mathsf{sb}} Q$.*
- *$P =_{\mathsf{sb}} Q$ if and only if $\mathsf{repa}(P) =_{\mathsf{sb}} \mathsf{repa}(Q)$.*
- *If $P =_{\mathsf{iso}} Q$, then $\mathsf{repa}(P) =_{\mathsf{iso}} \mathsf{repa}(Q)$.*

As defined above, strong bisimilarity is a relation between two LTSs. It can, however, be applied also to two states of the same LTS $P = (S, \Sigma, \Delta, \hat{s})$ by defining that s_1 and s_2 are strongly bisimilar, denoted by $s_1 \sim_{\mathsf{sb}} s_2$, if and only if $(S, \Sigma, \Delta, s_1) =_{\mathsf{sb}} (S, \Sigma, \Delta, s_2)$. The relation "$\sim_{\mathsf{sb}}$" is a strong bisimulation between the states of P and the states of P such that $\hat{s} \sim_{\mathsf{sb}} \hat{s}$. Furthermore, it is the *largest* such strong bisimulation: If "$\sim$" $\subseteq S \times S$ is a strong bisimulation, then $s \sim s'$ implies $s \sim_{\mathsf{sb}} s'$.

If $s \in S$, then let $[[s]] = \left\{ s' \in S \;\middle|\; s \sim_{\mathsf{sb}} s' \right\}$, that is, $[[s]]$ is the set of the states of P that are strongly bisimilar to s. For any finite LTS $P = (S, \Sigma, \Delta, \hat{s})$, there is a unique (excluding the naming of states) LTS $P_{\mathsf{min}} = (S_{\mathsf{min}}, \Sigma, \Delta_{\mathsf{min}}, \hat{s}_{\mathsf{min}})$ that has as few states as possible, and is strongly bisimilar to P. It is the following:

- $S_{\mathsf{min}} = \left\{ [[s]] \;\middle|\; s \in S \wedge \exists \sigma \in (\Sigma \cup \{\tau\})^* : \hat{s} \stackrel{\sigma}{\longrightarrow} s \right\}$.
- $\Delta_{\mathsf{min}} = \left\{ ([[s]], a, [[s']]) \;\middle|\; [[s]] \in S_{\mathsf{min}} \wedge (s, a, s') \in \Delta \right\}$.
- $\hat{s}_{\mathsf{min}} = [[\hat{s}]]$.

To be more precise, $P_{\mathsf{min}} =_{\mathsf{sb}} P$, and any LTS that is strongly bisimilar to P is either isomorphic to P_{min}, or has more states than P_{min}. A basic property of this construction is that if $(s, a, s') \in \Delta$, then for *every* $s_1 \in [[s]]$ there is $s_1' \in [[s']]$ such that $(s_1, a, s_1') \in \Delta$; this follows directly from the definitions of $[[s]]$ and strong bisimulation. Therefore, the relation "$\sim$" $\subseteq S \times S_{\mathsf{min}}$: $(s_1 \sim [[s]] \Leftrightarrow s_1 \in [[s]])$ is a strong bisimulation. The purpose of the condition "$\exists \sigma \in (\Sigma \cup \{\tau\})^* : \hat{s} \stackrel{\sigma}{\longrightarrow} s$" is to restrict P_{min} to the reachable part or, equivalently, avoid unnecessary simulation of the unreachable part of P. Without it, the result would be the smallest possible LTS that simulates every state of P. (The definition of P_{min} is valid even if P is not finite, although then one has to be careful with what is meant by "as few states as possible".)

P_{min} can be constructed rather efficiently with the algorithms presented in [17, 7]. A closely related algorithm can be used for testing whether two finite LTSs $(S_1, \Sigma, \Delta_1, \hat{s}_1)$ and $(S_2, \Sigma, \Delta_2, \hat{s}_2)$ are strongly bisimilar. That algorithm applies the construction of P_{min} to both LTSs simultaneously, such that any $[[s]]$ may

contain states from both LTSs. That is, $[[s]] = \{\, s' \in S_1 \cup S_2 \mid \text{there is a strong bisimulation ``}\sim\text{'' such that } s \sim s' \,\}$. The LTSs are strongly bisimilar if and only if $\hat{s}_1$ and $\hat{s}_2$ end up in the same $[[s]]$.

The minimal LTS that is strongly bisimilar to the servers is shown in Figure 5. Compared to Figure 3, the bottom row and the first state of the second row have disappeared, and some transitions have been re-directed accordingly. This has an intuitive explanation: if there is a pending request from the client or such a request has just been served, then the server will push the token to the next server independently of whether the next server has requested it, so the value of is_dem need not be preserved in those cases.

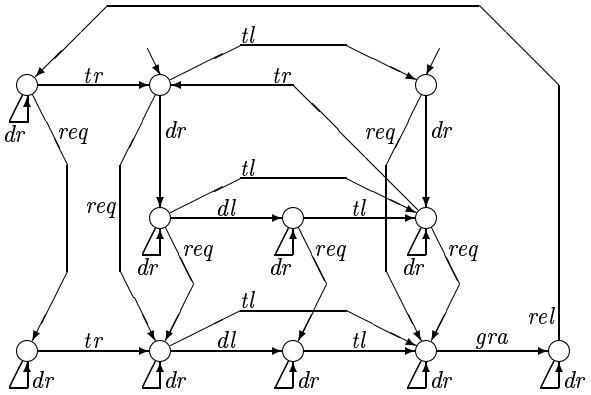

Fig. 5. Minimised version of the servers as an LTS.

2.5 The Clients

Often in process-algebraic verification, the users of a system are not modelled, although, of course, the system is. This is called an *open* model of the system. A *closed* model contains both the system and its users, and it is the norm in most other types of verification. The use of open models is possible in process algebras because of the type of compositionality found in them. We will see in Section 3.5 that the behaviour of any subsystem can be projected to its interface with the rest of the system and then reduced. One possibility is to project the behaviour of the system without the users to its interface with the users. The result shows how arbitrary users would see the system, and is thus often suitable for verification.

In this article, however, there are more clients than we will be able to handle simultaneously. We will, therefore, use projections to one or two clients at a time. For these projections to provide valid information, the remaining clients must be somehow presented at the other side of the interface. For this purpose we will

need an adequate model of the clients. Such a model is shown in Figure 6. Since we will have to model almost all clients, it is natural to model them all, that is, use a closed model of the system. That is what we will do.

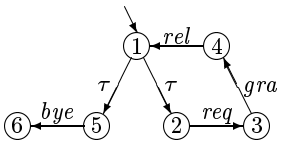

Fig. 6. A client.

A client starts its operation by executing one of the two alternative τ-transitions. If the client executes the τ-transition that leads to state 2, then it requests the permission to do the critical activity, waits in state 3 until it gets the permission, and does the critical activity in state 4. Then it indicates the server that it has stopped the critical activity and returns to its initial state. In the opposite branch, the client executes *bye* and stops for good.

To understand the reason for the presence of the initial τ-transitions and the *bye*-branch, let us consider the purpose of mutual exclusion systems for a while. The purpose is to ensure that (1) no two clients are simultaneously doing their critical activities, and (2) any client that has requested permission to do the critical activity will eventually get the permission. Assume that a client that has got the permission does the critical activity forever, and another client asks for the permission. Then the mutual exclusion system cannot satisfy the specification: if it ever gives the permission to the second client, then it violates (1), and if it does not, it violates (2).

We see that no mutual exclusion system can work correctly unless the clients obey some discipline. If a client has started doing the critical activity — that is, executed *gra* — then it must eventually stop it — that is, execute *rel*. It is also reasonable to require that if it has executed *req*, and *gra* is executable, then it must eventually execute *gra*, because the opposite would mean that it has requested the permission but then refuses to take it when it becomes available. Therefore, to be able to verify the system, we have to assume that the clients follow these rules, and we have to somehow take the assumption into account in the verification process. But we have actually done that (or done as well as can be done in this verification theory): Figure 6 enforces these rules by giving the client no other way to continue after *req* than by executing *gra*, and similarly with *rel* after *gra*.

The *req* action is, however, different. It is up to the client to decide whether it ever wants to ask for the permission. The client must be able to never execute *req* if that is what it wants. The τ-transition from the initial state to state 5 is there to express this possibility. By executing it the client can avoid executing *req*.

The τ-transition from state 1 to state 2 ensures that the client can commit itself to executing *req*. The semantics of interaction in the theory, to be discussed in Sections 2.6 and 2.7, is such that if the *req*-transition started at the initial state, then the server could force the client to execute the τ-transition leading to state 5 by refusing *req* forever. An incorrect server could then do that whenever it is unable to provide service, and thus mask its deficiency.

The *bye*-transition is not really necessary. It makes the interpretation of some of the verification results easier by stating explicitly that the client has decided not to request the permission any more. It would also have been possible to start the *bye*-transition at the initial state and remove state 5 and its adjacent transitions. This would not have given the servers the power to drive the client to state 2 by preventing the execution of the *bye*-transition, because the servers do not interact via *bye*, and thus cannot prevent its execution. However, the environment surrounding both the system and the clients would have got this power. That would not have caused errors to verification results, because in this case there is no such environment. It would, however, have made the results a bit more difficult to read, because the reader would have had to manually apply the piece of information that there will be no processes in addition to the servers and the clients.

The presence of a τ-transition both before the *req*-transition and before the *bye*-transition makes it explicit that it is the client who chooses between requesting and not requesting the permission, and nobody can force it to make one of the choices by preventing the other.

One may wonder whether this model of clients is fully appropriate. Permanent refusal from *req* might seem a bit harsh; would it not be better to refuse for some time and execute *req* again at some later time? After discussing abstract semantic models in Section 3.3 we will see that the model is very appropriate.

2.6 Putting the system together

All the components of the system and its users have now been specified. They cannot, however, be put together as they are, because they use different action names from those in Figure 1, and the clients share common action names although they have not been intended to communicate directly with each other.

This problem is, of course, easy to solve: the server and client LTSs can be used as models from which LTSs with the same structure but different, appropriate action names are constructed by changing the labels of transitions. In this case visible action names have to be changed to other visible action names, but we will later also need to *hide* visible actions, that is, change them to τ. A general operator that serves both purposes and can even be used to duplicate transitions is defined below.

Definition 7. *Let $P = (S, \Sigma, \Delta, \hat{s})$ be an LTS. A binary relation Φ is an* action transformer *for P, if and only if it satisfies the following conditions:*

- $(\tau, \tau) \in \Phi$.

- *If* $(\tau, a) \in \Phi$, *then* $a = \tau$.
- $\{\, a \mid a \neq \tau \wedge \exists b : (a, b) \in \Phi \,\} = \Sigma$.

If Φ *is an action transformer for* P, *then the LTS* $P\Phi$ *is defined as* $(S, \Sigma', \Delta', \hat{s})$, *where*

- $\Sigma' = \{\, b \mid b \neq \tau \wedge \exists a : (a, b) \in \Phi \,\}$.
- $\Delta' = \{\, (s, b, s') \mid \exists a : (s, a, s') \in \Delta \wedge (a, b) \in \Phi \,\}$.

The first two conditions of the action transformer relation specify that the invisible action symbol τ is transformed to τ and nothing else. The idea is that τ is totally untouchable: it cannot be affected by any operation. The last condition says that in addition to τ, Φ specifies new names for precisely the visible actions of P. Φ may duplicate transitions by specifying more than one new name for a visible action. When an LTS is transformed with the action transformer operator, its set of states and the initial state stay the same. The new alphabet consists of the new names other than τ of the visible actions, and for each transition (s, a, s'), and each b that a is transformed to, there is the transition (s, b, s').

The operator allows the transformation of an action to itself, and the transformation of a visible action to many actions. The simplest example of an action transformer is the *identity action transformer* $\Phi_{\mathsf{id}} = \{\, (a, a) \mid a \in \Sigma \cup \{\tau\} \,\}$. It does not change P at all.

The action transformer operator obeys some useful laws.

Theorem 3. *Let* P *and* Q *be LTSs, and* Φ *be an action transformer for* P.

- *If* Ψ *is an action transformer for* $P\Phi$, *then* $(P\Phi)\Psi = P\Xi$, *where* $\Xi = \{\, (a, c) \mid \exists b : (a, b) \in \Phi \wedge (b, c) \in \Psi \,\}$ *is an action transformer for* P.
- $P\Phi_{\mathsf{id}} = P$.
- *If* $P =_{\mathsf{iso}} Q$, *then* $P\Phi =_{\mathsf{iso}} Q\Phi$.
- *If* $P =_{\mathsf{sb}} Q$, *then* $P\Phi =_{\mathsf{sb}} Q\Phi$.

The first of these laws tells how two action transformers can be combined into one. The last two state the non-surprising fact that isomorphism and strong bisimilarity are *congruences* with respect to action transformation.

A more concrete syntax for two special cases of the action transformer is given in the next definition. For convenience, first an operator is defined which converts any binary relation to an action transformer. The resulting transformer inherits from the relation the changes that affect the required alphabet, and specifies no change for τ and the actions that are in the alphabet but are not affected by the relation. The first special case is called the *hiding* operator. It converts explicitly mentioned actions to τ and leaves the rest intact. The second special case can be used to express any action transformer that makes a finite number of changes, and we call it *renaming*. Unlike Φ, neither new operator requires the listing of those visible actions that are not changed. Furthermore, they allow irrelevant visible actions among the actions to be changed, but do nothing with them.

Definition 8. *Let $P = (S, \Sigma, \Delta, \hat{s})$ be an LTS, $A = \{a_1, \ldots, a_n\}$ such that $\tau \notin A$, and let $b_1, \ldots, b_n$ be any action names, X any set, and Φ any binary relation.*

- $\Phi{\downarrow}X = \big\{\, (a,b) \;\big|\; (a,b) \in \Phi \wedge a \in X - \{\tau\} \,\big\} \cup \big\{\, (a,a) \;\big|\; a \in X \wedge \neg\exists b : (a,b) \in \Phi \,\big\} \cup \big\{ (\tau,\tau) \big\}$.
- **hide** A **in** $P = P\big(\big\{\, (a,\tau) \;\big|\; a \in A \,\big\}{\downarrow}\Sigma\big)$.
- $P[b_1/a_1, \ldots, b_n/a_n] = P\big(\big\{(a_1,b_1), \ldots, (a_n,b_n)\big\}{\downarrow}\Sigma\big)$.

The servers and clients of the system can now be represented as

$\mathsf{Server\_tkn}_1 =$
$\quad \mathsf{Server\_tkn}[tkn_1/tl, tkn_2/tr, dem_1/dl, dem_2/dr, req_1/req, gra_1/gra, rel_1/rel]$
$\mathsf{Server}_i =$
$\quad \mathsf{Server}[tkn_i/tl, tkn_{i\oplus 1}/tr, dem_i/dl, dem_{i\oplus 1}/dr, req_i/req, gra_i/gra, rel_i/rel]$
$\mathsf{Client}_i = \mathsf{Client}[req_i/req, gra_i/gra, rel_i/rel, bye_i/bye]$.

where "_tkn" refers to the server that has the token initially.

The servers and clients are connected together with the *parallel composition* operator. The state of the parallel composition is a vector consisting of the states of the component processes. The initial state is, of course, the vector that consists of the initial states of the components. The parallel composition has all the visible actions of the components as its visible actions.

Each component can make an invisible transition independently of the other components, and at the level of the composed system the transition appears as an invisible transition. The other components stay where they are during the transition.

Regarding visible transitions, we say that a component is *interested* in a visible action if the action is in its alphabet. A visible transition of the parallel composition consists of simultaneous transitions by one or more components. The transitions of the components must have the same label, and that label is also the label of the joint transition. All components who are interested in the label must participate the transition. This implies that any of these components can postpone or prevent the transition by not being ready for it. All the other participants must then wait or perform some other transition. Again, the transition does not affect the states of those components that are not interested in its label.

Only the states that are reachable from the initial state are taken into account in the parallel composition. In the definition below, this is achieved by using an auxiliary notion of *synchronous product*, and defining the parallel composition to be its reachable part.

Definition 9. *Let $P_1 = (S_1, \Sigma_1, \Delta_1, \hat{s}_1)$, …, $P_n = (S_n, \Sigma_n, \Delta_n, \hat{s}_n)$ be LTSs. Their* synchronous product *is the LTS $P_1 \times \cdots \times P_n = (S, \Sigma, \Delta, \hat{s})$, where*

- $S = S_1 \times \cdots \times S_n$
- $\Sigma = \Sigma_1 \cup \cdots \cup \Sigma_n$
- $\hat{s} = \langle \hat{s}_1, \ldots, \hat{s}_n \rangle$

- $(\langle s_1, \ldots, s_n \rangle, \tau, \langle s'_1, \ldots, s'_n \rangle) \in \Delta$, *if and only if there is an* i, $1 \leq i \leq n$, *such that*
  - $(s_i, \tau, s'_i) \in \Delta_i$, *and*
  - $s'_j = s_j$ *whenever* $1 \leq j < i$ *or* $i < j \leq n$.
- $(\langle s_1, \ldots, s_n \rangle, a, \langle s'_1, \ldots, s'_n \rangle) \in \Delta$ *where* $a \in \Sigma$, *if and only if whenever* $1 \leq i \leq n$, *either*
  - $a \in \Sigma_i$ *and* $(s_i, a, s'_i) \in \Delta_i$, *or*
  - $a \notin \Sigma_i$ *and* $s'_i = s_i$.
- Δ *contains no other elements than those generated by the above two rules.*

Their parallel composition *is the LTS* $P_1 \,||\, \cdots \,||\, P_n = \mathsf{repa}(P_1 \times \cdots \times P_n)$.

Tools that compute parallel compositions do not usually construct the unreachable part at all. This is because the synchronous product is often very much bigger than its reachable part. For instance, the synchronous product corresponding to the 3-server demand-driven token-ring system has $17^3 \cdot 6^3 = 1\,061\,208$ states, whereas only 1 320 of them are reachable.

Also the parallel composition obeys some useful laws.

Theorem 4. *Let* P, P', P_1, ..., P_n, Q, *and* Q' *be LTSs,* Φ *an action transformer for* $(P \,||\, Q)$, *and* $1 \leq i \leq j \leq n$. *Let the alphabets of* P *and* Q *be* Σ_P *and* Σ_Q.

- $P_1 \,||\, \cdots \,||\, P_{i-1} \,||\, (P_i \,||\, \cdots \,||\, P_j) \,||\, P_{j+1} \,||\, \cdots \,||\, P_n =_{\mathsf{iso}} P_1 \,||\, \cdots \,||\, P_n$.
- $P \,||\, Q =_{\mathsf{iso}} Q \,||\, P$.
- *If* $P =_{\mathsf{iso}} P'$ *and* $Q =_{\mathsf{iso}} Q'$, *then* $P \,||\, Q =_{\mathsf{iso}} P' \,||\, Q'$.
- *If* $P =_{\mathsf{sb}} P'$ *and* $Q =_{\mathsf{sb}} Q'$, *then* $P \,||\, Q =_{\mathsf{sb}} P' \,||\, Q'$.
- *Assume that for every* a, a' *and* b *such that* $(a, b) \in \Phi$ *and* $(a', b) \in \Phi$
  - $a \in \Sigma_P - \Sigma_Q \Rightarrow b = \tau \vee a' \notin \Sigma_Q$,
  - $a \in \Sigma_Q - \Sigma_P \Rightarrow b = \tau \vee a' \notin \Sigma_P$, *and*
  - $a \in \Sigma_P \cap \Sigma_Q \Rightarrow b \neq \tau \wedge a' = a$.

  Then $(P \,||\, Q)\Phi = (P(\Phi{\downarrow}\Sigma_P)) \,||\, (Q(\Phi{\downarrow}\Sigma_Q))$.

The first law implies that any hierarchical tree of parallel compositions is isomorphic to the flat structure where all components have been connected together in one multi-parameter parallel composition. It also implies that parallel composition is associative, provided that isomorphism is accepted as a strong enough notion of equivalence. Parallel composition is also commutative in the same sense, says the second law.

The next two laws express that isomorphism and strong bisimilarity are congruences also with respect to the parallel composition operator. The laws have been given for the case of two component processes, but they can be extended to any number of components by repeatedly applying the first law.

The fifth law allows the distribution of an action transformer over a parallel composition in certain cases. Its horrible-looking condition is there to ensure that local transitions of P and Q synchronise after the distribution if and only if they synchronise before the distribution.

The n-server demand-driven token-ring system can now be defined as

$$\mathsf{System}_n = \mathsf{Server\_tkn}_1 \,||\, \mathsf{Client}_1 \,||\, \mathsf{Server}_2 \,||\, \mathsf{Client}_2 \,||\, \cdots \,||\, \mathsf{Server}_n \,||\, \mathsf{Client}_n$$

The alphabet of System_n is

$$\Sigma_n^{\mathsf{Sys}} = \{\, tkn_1, \ldots, tkn_n, dem_1, \ldots, dem_n, req_1, \ldots, req_n, \\ gra_1, \ldots, gra_n, rel_1, \ldots, rel_n, bye_1, \ldots, bye_n \}$$

The numbers of states and transitions of the system are shown in the second and third column of Table 1.

	full system		only min-servers	
n	states	transitions	states	transitions
2	132	298	30	58
3	1 320	4 164	150	402
4	12 320	49 936	680	2 332
5	110 000	544 800	2 900	12 120
6	950 400	5 562 240	11 880	58 560

Table 1. The size of the LTS of System_n and of $\mathsf{Server\_tkn}_1^{\mathsf{min}} \,||\, \mathsf{Server}_2^{\mathsf{min}} \,||\, \cdots \,||\, \mathsf{Server}_n^{\mathsf{min}}$.

It can be seen from the table that the size of the system LTS grows very rapidly as a function of n. This phenomenon is very common with parallel composition, and it is called *state explosion*.

A part of the state explosion in this system is due to somewhat wasteful modelling. For instance, as was discussed in Section 2.4, Figure 5 could have been used as the model of the servers instead of Figure 3. However, the growth remains exponential even with the most economical modelling. Namely, the token may be in any of the n servers, and any client may have or have not requested the permission independently of the states of the other parts of the system, yielding at least $n2^n$ states. The real growth rate is much faster, as can be seen from the last two columns of Table 1. They act as a lower bound, because they show the size of the LTS of a ring, the servers of which are as in Figure 5, and that has no clients at all.

The lesson to be learnt from the table is that careful modelling helps a lot in keeping the LTS size small, but usually cannot prevent state explosion altogether.

2.7 A digression on the parallel composition operator

Although the parallel composition operator presented in Definition 9 is very common in the literature (perhaps with the difference that the restriction to the reachable part is sometimes omitted), it deserves some comments.

First, it does not allow *true parallelism.* In reality, transitions that are participated by disjoint sets of component processes may occur simultaneously, but this operator forces them to occur one at a time in an arbitrary order. However, such transitions can be executed in any order, the end result is independent of the order, and it is the same that would have resulted from simultaneous execution. As a consequence, the "one-at-a-time" convention (called *interleaving semantics*) does not affect most of the aspects of systems that people want to verify. There are theories for true parallelism, but they are mathematically complicated. Therefore, many, perhaps the majority of verification researchers use interleaving semantics.

Second, the operator uses *synchronous communication*: processes communicate by executing simultaneously transitions that are labelled with the same action. The operator even allows more than two participants in the same transition. This mode of communication has no clear notions of input and output. There is no buffering between the communicating parties, and the use of the operator leads easily to seemingly superfluous deadlocks where processes wait for synchronisation with each other but with different actions. So it might seem that, say, fifos (first-in first-out queues) would have been a more adequate form of communication.

It has turned out, however, that most, if not all, other modes of communication can reasonably easily be simulated with synchronous communication, whereas simulation in the opposite direction is usually difficult. Input of data is modelled by offering a transition for each possible data value, whereas in the case of output there is a transition only for one value, namely the value that is to be sent. Buffering can be represented by modelling buffers as processes of their own right, such as in Figure 7 that shows a fifo as an LTS. In it, i_1 and i_2 are input actions that are both offered in every state where the fifo is not full, while o_1 and o_2 are output actions, of which at most one is offered at any state.

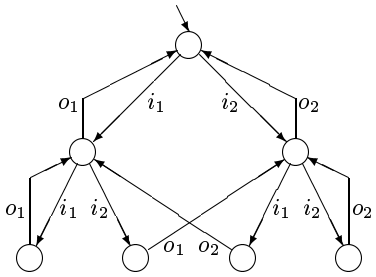

Fig. 7. A fifo of two message types and capacity two.

Although fifos might be less prone to deadlocks than synchronous communication, they have other problems: what happens to the first message in the fifo if

the recipient is not willing to read it, and what happens if messages are written to the fifo more frequently than read from it? After these questions have been appropriately answered, the fifo model starts to suffer from unbounded accumulation of messages in fifos, unexpected losses of messages, or deadlocks. Many of the seemingly superfluous deadlocks of synchronous communication would thus not go away when switching over to fifo communication, they would just change shape.

Finally, synchronisation by more than two parties makes it possible to simulate broadcasts, so it is useful.

In conclusion, the use of synchronous communication does not restrict the applicability of the theory as much as it might initially seem.

Third, the operator relies on alphabets for determining which processes synchronise for each action. This makes extra actions in an alphabet significant. This could have been avoided by using some other form of parallel composition. For instance, the main parallel composition operator of Lotos [2] lists explicitly the actions with which the processes synchronise. It looks like this: $P \,|[a_1, \ldots, a_n]|\, Q$.

The Lotos operator can, however, be simulated with our parallel composition and action transformer operators: $P \,|[a_1, \ldots, a_n]|\, Q$ behaves in the same way as $(P\Phi \,||\, Q \,||\, \mathbf{stop}_{\{a_1,\ldots,a_n\}-(\Sigma_P\cap\Sigma_Q)})\Phi^{-1}$, where

- Σ_P and Σ_Q are the alphabets of P and Q.
- $\mathbf{stop}_X$ is the one-state LTS that has no transitions, and whose alphabet is X.
- $\Phi = [b'_1/b_1, \ldots, b'_k/b_k]$, where $\{b_1, \ldots, b_k\} = (\Sigma_P \cap \Sigma_Q) - \{a_1, \ldots, a_n\}$, and $b'_1, \ldots, b'_k$ are new action names, that is, $b'_i \notin \Sigma_P \cup \Sigma_Q \cup \{\tau\} \cup \{a_1, \ldots, a_n\}$, and if $i \neq j$, then $b'_i \neq b'_j$.
- $\Phi^{-1} = [b_1/b'_1, \ldots, b_k/b'_k]$.

This formula refers to the alphabets. However, if $P = (S_P, \Sigma_P, \Delta_P, \hat{s}_P)$, any set X such that $\tau \notin X$ and $\{\, a \mid a \neq \tau \wedge \exists s, s' : (s, a, s') \in \Delta_P \,\} \subseteq X$ can be used instead of Σ_P without changing anything else than the alphabet of the result, and similarly with Q.

The CCS [23] parallel composition operator can be simulated by taking advantage of the ability of the action transformer operator to duplicate transitions. In CCS, the visible actions are divided to two disjoint sets: "names" like a, and "co-names" like $\bar{a}$. We define that $\bar{\bar{a}} = a$ and $\bar{A} = \{\, \bar{a} \mid a \in A \,\}$. The CCS parallel composition operator "|" takes precisely two operands. Synchronisation occurs when one operand executes a visible action a and the other simultaneously executes $\bar{a}$. The label of the resulting transition of $P \mid Q$ is τ. Furthermore, either operand can do just any action without synchronising with the other operand.[2]

Let $\Sigma = \Sigma_P \cup \Sigma_Q \cup \overline{\Sigma_P} \cup \overline{\Sigma_Q} = \{a_1, \ldots, a_n\}$, and let $b_1, \ldots, b_n$ and $a'_1, \ldots, a'_n$ be different from each other and from all elements of $\Sigma \cup \{\tau\}$. Then $P|Q$ behaves like $\big(\big((P \,||\, \mathbf{stop}_{\Sigma-\Sigma_P})\Phi_P\big) \,||\, \big((Q \,||\, \mathbf{stop}_{\Sigma-\Sigma_Q})\Phi_Q\big)\big)\Phi$, where Σ_P and Σ_Q are like above, and

[2] This can be prevented with the CCS *restriction* operator $P\backslash L$ that works like $P \,||\, \mathbf{stop}_{L\cup\bar{L}}$.

- $\Phi_P = [a'_1/a_1, \ldots, a'_n/a_n, b_1/\bar{a}_1, \ldots, b_n/\bar{a}_n]$,
- $\Phi_Q = [a_1/a_1, \ldots, a_n/a_n, b_1/a_1, \ldots, b_n/a_n]$, and
- $\Phi = [a_1/a'_1, \ldots, a_n/a'_n, \tau/b_1, \ldots, \tau/b_n]$.

Other commonly used parallel composition operators can thus be simulated with our "||". On the other hand, "||" is simple to define and use, is more general than the CCS "|" that allows only two-process synchronisation, and enjoys some useful mathematical properties that the Lotos "$|[\cdots]|$" lacks. Namely, "||" is associative, and if P and Q are *deterministic* (in the sense that if $s \mathrel{-}a\!\!\rightarrow s_1$ and $s \mathrel{-}a\!\!\rightarrow s_2$, then $a \neq \tau$ and $s_1 = s_2$), then also $P \,||\, Q$ is deterministic and $P \,||\, P =_{\mathsf{iso}} P$. These are good reasons for using "||" in our theory.

3 Verification

3.1 Requirements

As was mentioned in Section 2.5, the purpose of the system is to ensure two important properties:

mutual exclusion: No two clients are simultaneously doing their critical activities.

eventual access: Any client that has requested permission to do the critical activity will eventually get the permission.

To state these properties formally, we define two *state propositions* of Client_i:

- r_i = client i is in state 3.
- g_i = client i is in state 4.

A state proposition is a claim whose truth value depends only on the current state of the system, not on preceding or succeeding states. With these propositions a user of linear temporal logic [22] could formalise the requirements like this ("$\Box$" can be read as "always" and "$\Diamond$" as "eventually"):

mutual exclusion: $\forall i, j \in \{1, 2, \ldots, n\} : (i \neq j \Rightarrow \Box\neg(\mathsf{g}_i \wedge \mathsf{g}_j))$.
eventual access: $\forall i \in \{1, 2, \ldots, n\} : \Box(\mathsf{r}_i \Rightarrow \Diamond\mathsf{g}_i)$.

These formulae are difficult to check automatically in their current form, because of the use of the universal quantification "$\forall$". However, the formulae can be interpreted as $\frac{1}{2}n(n+1)$ different propositional linear temporal logic formulae: $\Box\neg(\mathsf{g}_1 \wedge \mathsf{g}_2)$, $\ldots$, $\Box\neg(\mathsf{g}_1 \wedge \mathsf{g}_n)$, $\ldots$, $\Box\neg(\mathsf{g}_{n-1} \wedge \mathsf{g}_n)$, $\Box(\mathsf{r}_1 \Rightarrow \Diamond\mathsf{g}_1)$, $\ldots$, $\Box(\mathsf{r}_n \Rightarrow \Diamond\mathsf{g}_n)$. (Here the $\frac{1}{2}n(n-1)$ formulae of the form $\Box\neg(\mathsf{g}_i \wedge \mathsf{g}_j)$ where $i > j$ were omitted, because they are equivalent to $\Box\neg(\mathsf{g}_j \wedge \mathsf{g}_i)$.) These formulae can be checked from the full LTS of the system with a suitable temporal logic *model checking* tool, provided that the numbers of the states of the clients are preserved in the LTS.

It would be tempting to appeal to the symmetry of the system to reduce further the number of formulae that have to be checked. Unfortunately, although

the structure of the system is symmetric, its initial state is not: one server differs from the others in that it has the token. Some properties are sensitive to this difference, such as "if the first request is made by Client_i, then the first permission is given to Client_i". It is true only if Client_i is the one whose server has the token initially. The mutual exclusion and eventual access properties are not sensitive to the initial asymmetry. It is, however, not trivial to see that this is the case, so appealing to it in verification is unsatisfactory at the least.

We shall thus proceed in a different direction.

3.2 Introduction to abstraction

It is easy to see that the validity of r_i and g_i in a state can be reasoned from the sequence of actions executed so far. It suffices to know which of req_i, gra_i and rel_i has been executed most recently:

- if req_i, then $\mathsf{r}_i = \mathsf{True}$ and $\mathsf{g}_i = \mathsf{False}$,
- if gra_i, then $\mathsf{r}_i = \mathsf{False}$ and $\mathsf{g}_i = \mathsf{True}$,
- if rel_i, then $\mathsf{r}_i = \mathsf{g}_i = \mathsf{False}$.

Also initially $\mathsf{r}_i = \mathsf{g}_i = \mathsf{False}$.

Another important thing is that the mutual exclusion and eventual access properties are *insensitive to stuttering*. That is, their validity depends only on the order in which various combinations of the truth values of the r_i and g_i are obtained; whether a particular combination is valid in 1, 2 or 1 000 000 successive states does not matter. This is not a coincidence. Most, perhaps all, verification researchers agree that in the context of concurrent systems, properties that are sensitive to stuttering are almost always irrelevant.

These facts make it possible to take advantage of process-algebraic *abstraction* in the verification of the mutual exclusion and eventual access properties. The idea is to recognise the actions on which the validity of a formula depends, hide the remaining actions with the hiding operator, and then construct a suitable *reduced* LTS of the system. The reduced LTS is a kind of a projection of the behaviour of the system onto the visible actions.

Before going into the details, let us have a look at what reduced LTSs corresponding to the formulae $\Box\neg(\mathsf{g}_1 \wedge \mathsf{g}_2)$ and $\Box(\mathsf{r}_1 \Rightarrow \Diamond\mathsf{g}_1)$ might look like when $n = 3$. Figure 8 shows such LTSs as produced by the ARA tool. As a matter of fact, the LTS in Figure 8(b) shows more actions than is necessary for checking eventual access, but we will soon see that also the extra actions are interesting. The numbering of states has been added when re-drawing the LTSs into this article. We will later see that precisely the same figures are obtained also with any $n > 3$.

The LTS on the left hand side is obtained after hiding all other visible actions than gra_1, gra_2, rel_1 and rel_2. That is, it is a reduced version of

$$\mathbf{hide}\ \Sigma_3^{\mathsf{Sys}} - \{gra_1, gra_2, rel_1, rel_2\}\ \mathbf{in}\ \mathsf{System}_3$$

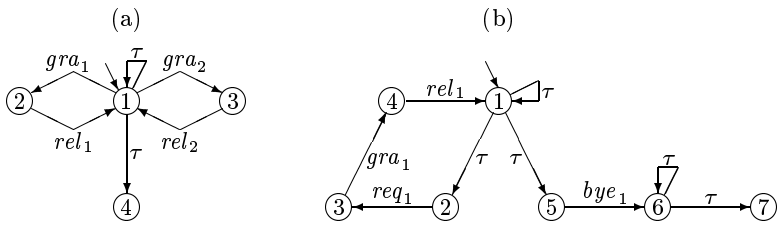

Fig. 8. (a) A mutual exclusion and (b) an eventual access view to the three-server system.

We have that g_1 holds in state 2 and only in it, while g_2 holds precisely in state 3. There is thus no state where g_1 and g_2 hold simultaneously, so $\Box\neg(\mathsf{g}_1 \wedge \mathsf{g}_2)$ holds.

The LTS on the right hand side is a reduced version of

$$\textbf{hide } \Sigma_3^{\mathsf{Sys}} - \{req_1, gra_1, rel_1, bye_1\} \textbf{ in } \mathsf{System}_3$$

That there are no output transitions from state 7 means that the system can halt or deadlock. However, this state can be reached only after Client_1 has executed bye_1, so the system cannot halt unless Client_1 chooses to never or never again request the permission. The possibility of halting is shown also by state 4 of the mutual exclusion LTS, but that LTS does not show that bye_1 is executed before halting, because bye_1 is not visible in that LTS.

The τ-loops adjacent to states 1 and 6 indicate that the system can execute forever without Client_1 being involved. It is easy to think of such executions: it suffices that the other clients request the permission again and again. What is important in the figure is that there are neither τ-loops nor deadlocks adjacent to state 3. This means that if Client_1 has executed req_1, then the system cannot halt or execute forever before Client_1 executes gra_1. In other words, if Client_1 has requested the permission, it will get the permission within a finite number of steps by the system. Therefore, $\Box(\mathsf{r}_1 \Rightarrow \Diamond\mathsf{g}_1)$ holds.

If the visible actions are req_2, gra_2, rel_2 and bye_2 or req_3, gra_3, rel_3 and bye_3, then the resulting LTS is like the LTS in Figure 8(b) with, of course, the subscripts in question. If the visible actions are gra_1, rel_1, gra_2, rel_2, gra_3 and rel_3, then the result is otherwise like the LTS in Figure 8(a), but the τ-loop has been replaced by a gra_3-rel_3-loop. Therefore, the correctness formulae hold for every value of i and j when $n = 3$.

We see that if a formula is simple enough, the corresponding reduced LTS is so small that the validity of the formula can be checked directly from the LTS without any tool. Furthermore, the reduced LTS may also contain other useful information. For instance, the LTS in Figure 8(b) shows the whole behaviour of the system as seen by Client_1. The LTS shows the possibility of termination, but shows also that termination is not possible before Client_1 executes bye_1. This kind of analysis of the behaviour of a system is known as *visual verification*. It has been discussed in more detail and compared to ordinary verification in [39].

Visual verification is not, however, the main advantage of abstraction. The main advantage is that, as will be demonstrated in Sections 3.5 to 3.7, the reduced LTS can be constructed without ever constructing the complete LTS of the system. This can be a tremendous advantage, because, as is apparent from Table 1, the size of the complete LTS may be huge.

3.3 CFFD-equivalence

Table 1 shows that System_3 has 1320 states, and Definitions 7 and 8 imply that **hide** $\Sigma_3^{\mathsf{Sys}} - \{gra_1, gra_2, rel_1, rel_2\}$ **in** System_3 has as many states as System_3. The LTS in Figure 8(a) cannot thus be the complete LTS of **hide** $\Sigma_3^{\mathsf{Sys}} - \{gra_1, gra_2, rel_1, rel_2\}$ **in** System_3. It was mentioned in Section 3.2 that it is some kind of a projection of the complete LTS onto the chosen visible actions. The goal of this section is to make precise the sense in which the LTS in the figure represents the complete LTS.

The main idea is to throw away information about the execution of invisible actions. For that purpose, a new notation is defined that resembles the "$-a_1 \cdots a_n\!\!\rightarrow$"-notation, but where τs are omitted from within $a_1 \cdots a_n$.

Definition 10. *Let $(S, \Sigma, \Delta, \hat{s})$ be an LTS, $s, s' \in S$, $n \geq 0$, and $a, a_1, a_2, \ldots \in \Sigma$.*

- $s \mathrel{=}\!\varepsilon\!\Rightarrow s'$ *if and only if there is an $i \geq 0$ such that $s -\!\tau^i\!\rightarrow s'$, where τ^i denotes the sequence of i copies of τ.*
- $s \mathrel{=}\!a\!\Rightarrow s'$ *if and only if* $\exists s_1, s_2 \in S : s \mathrel{=}\!\varepsilon\!\Rightarrow s_1 \wedge s_1 \mathrel{-}\!a\!\rightarrow s_2 \wedge s_2 \mathrel{=}\!\varepsilon\!\Rightarrow s'$.
- $s \mathrel{=}\!a_1a_2\cdots a_n\!\Rightarrow s'$ *if and only if*
  $\exists s_0, s_1, \ldots, s_n \in S : s_0 = s \wedge s_n = s' \wedge \forall i \in \{1, 2, \ldots, n\} : s_{i-1} \mathrel{=}\!a_i\!\Rightarrow s_i$.
- $s \mathrel{=}\!a_1a_2\cdots a_n\!\Rightarrow$ *if and only if* $\exists s' : s \mathrel{=}\!a_1a_2\cdots a_n\!\Rightarrow s'$.
- $s \mathrel{=}\!a_1a_2a_3\cdots\!\Rightarrow$ *if and only if*
  $\exists s_0, s_1, s_2, \ldots \in S : s_0 = s \wedge \forall i \in \{1, 2, 3, \ldots\} : s_{i-1} \mathrel{=}\!a_i\!\Rightarrow s_i$.

An execution of a system is *complete*, if and only if it is as long as possible, that is, it is either infinite or ends up in a state that has no output transitions. The number of visible actions in an infinite execution may be finite or infinite. In the former case, the sequence of visible actions that arises from the infinite execution is called a *divergence trace*, and in the latter case it is an *infinite trace* of the system. An execution that corresponds to a divergence trace has an infinite end part that consists of only invisible transitions. Such a situation is called a *livelock*, because then the system is busy executing, but does not make any visible progress.

If an execution ends up in a halted state, then the corresponding sequence of visible actions is finite, and is called a *deadlock trace*.

Sometimes it is useful to talk about the sequence of visible actions that arises from a finite and not necessarily complete execution of a system. The notion of *trace* is introduced for that purpose.

It has turned out that in order to derive the deadlock traces of a parallel composition, the deadlock traces of the components do not always suffice, but

their generalisation called *stable failures* is needed. A state is *stable* if and only if it has no output transitions that are labelled with τ. The *initial stability* predicate records whether the initial state is stable. A stable state *refuses* some set of visible actions, if and only if none of its output transitions is labelled with an element from the set. A stable failure is a pair that consists of a trace and a set of visible actions, such that at least one of the states where the LTS may be in after executing the trace is stable and refuses the set. If the other component of a parallel composition is simultaneously in a stable state where it can execute only actions from the set, then the parallel composition is in a deadlock, although both processes may be willing to execute more actions.

Definition 11. *Let $P = (S, \Sigma, \Delta, \hat{s})$ be an LTS.*

- *The set of the* traces *of P is*
  $Tr(P) = \{\, a_1 a_2 \cdots a_n \in \Sigma^* \mid \hat{s} =a_1 a_2 \cdots a_n\Rightarrow \,\}$.
- *The set of the* infinite traces *of P is*
  $Inftr(P) = \{\, a_1 a_2 a_3 \cdots \in \Sigma^\omega \mid \hat{s} =a_1 a_2 a_3 \cdots\Rightarrow \,\}$.
- *The set of the* divergence traces *of P is*
  $Divtr(P) = \{\, a_1 a_2 \cdots a_n \in \Sigma^* \mid \exists s \in S : \hat{s} =a_1 a_2 \cdots a_n\Rightarrow s \wedge s -\tau^\omega\rightarrow \,\}$,
  where τ^ω denotes the sequence of an infinite number of copies of τ.
- *The set of the* deadlock traces *of P is* $Dltr(P) =$
  $\{\, \sigma \in \Sigma^* \mid \exists s \in S : \hat{s} =\sigma\Rightarrow s \wedge \neg(s -\tau\rightarrow) \wedge \forall a \in \Sigma : \neg(s -a\rightarrow) \,\}$.
- *The set of the* stable failures *of P is* $Sfail(P) =$
  $\{\, (\sigma, A) \in \Sigma^* \times 2^\Sigma \mid \exists s \in S : \hat{s} =\sigma\Rightarrow s \wedge \neg(s -\tau\rightarrow) \wedge \forall a \in A : \neg(s -a\rightarrow) \,\}$.
- *The* initial stability *predicate of P is*
  $Stable(P) = \neg(\hat{s} -\tau\rightarrow)$.

Some useful laws that these sets obey are listed below.

Theorem 5. *Let $P = (S, \Sigma, \Delta, \hat{s})$ be an LTS.*

- $\varepsilon \in Tr(P)$.
- *If* $a_1 a_2 \cdots a_n \in Tr(P)$ *and* $0 \leq i \leq n$, *then* $a_1 a_2 \cdots a_i \in Tr(P)$.
- $Tr(P) = Divtr(P) \cup \{\, \sigma \mid (\sigma, \emptyset) \in Sfail(P) \,\}$.
- $Dltr(P) = \{\, \sigma \mid (\sigma, \Sigma) \in Sfail(P) \,\}$.
- *If* $(\sigma, A) \in Sfail(P)$ *and* $B \subseteq A$, *then* $(\sigma, B) \in Sfail(P)$.
- *If* $a_1 a_2 a_3 \cdots \in Inftr(P)$ *and* $0 \leq i$, *then* $a_1 a_2 \cdots a_i \in Tr(P)$.
- *If S is finite, then*
  $Inftr(P) = \{\, a_1 a_2 a_3 \cdots \in \Sigma^\omega \mid \forall i \in \{0,1,2,\ldots\} : a_1 a_2 \cdots a_i \in Tr(P) \,\}$.

The validity of the mutual exclusion property in the demand-driven token-ring system depends only on how gra_i, rel_i, gra_j and rel_j are interleaved in the executions of the system. An important feature of the mutual exclusion property is that if it is violated, then it is violated after a finite execution of the system. This execution need not be complete. Therefore, to check whether the mutual exclusion property holds, it suffices to know the traces of **hide** $\Sigma_n^{\mathsf{Sys}} - \{gra_i, gra_j, rel_i, rel_j\}$ **in** System_n for every $1 \leq i < j \leq n$. Properties that can be checked from the set of traces are called *safety properties*.

The eventual access property is not a safety property, and thus cannot be checked from the set of traces. It is a (proper) *liveness* property. Namely, that a request by a client is never granted cannot be reasoned from an incomplete execution, because it is possible that it is granted in the unknown continuation of the execution. However, if a request is not granted in a complete execution, then it is certain that the property does not hold. Therefore, the validity of the eventual access property can be reasoned from the infinite traces, divergence traces and deadlock traces of **hide** $\Sigma_n^{\mathsf{Sys}} - \{req_i, gra_i\}$ **in** System_n.

We are now ready to define formally the sense in which the LTSs in Figure 8 are equivalent to the LTSs **hide** $\Sigma_3^{\mathsf{Sys}} - \{gra_1, gra_2, rel_1, rel_2\}$ **in** System_3 and **hide** $\Sigma_3^{\mathsf{Sys}} - \{req_1, gra_1, rel_1, bye_1\}$ **in** System_3.

Definition 12. *Let* $P = (S, \Sigma, \Delta, \hat{s})$ *and* $Q = (S', \Sigma', \Delta', \hat{s}')$ *be LTSs.*

- *The* CFFD semantics *of* P *is* $(\Sigma, Sfail(P), Divtr(P), Inftr(P), Stable(P))$.
- CFFD-equivalence *is defined by* $P \simeq_{\mathsf{CFFD}} Q$ *if and only if* $\Sigma = \Sigma'$, $Sfail(P) = Sfail(Q)$, $Divtr(P) = Divtr(Q)$, $Inftr(P) = Inftr(Q)$, *and* $Stable(P) = Stable(Q)$.
- CFFD-preorder *is defined by* $P \leq_{\mathsf{CFFD}} Q$ *if and only if* $\Sigma = \Sigma'$, $Sfail(P) \subseteq Sfail(Q)$, $Divtr(P) \subseteq Divtr(Q)$, $Inftr(P) \subseteq Inftr(Q)$, *and* $Stable(P) \vee \neg Stable(Q)$.

"Preorder" means a reflexive and transitive relation. Obviously, $P \simeq_{\mathsf{CFFD}} Q$ if and only if $P \leq_{\mathsf{CFFD}} Q$ and $Q \leq_{\mathsf{CFFD}} P$. It follows from Theorem 5 that if $P \leq_{\mathsf{CFFD}} Q$, then $Tr(P) \subseteq Tr(Q)$. It is also true that if $P =_{\mathsf{sb}} Q$, then $P \simeq_{\mathsf{CFFD}} Q$. (We write $P \simeq_{\mathsf{CFFD}} Q$ instead of $P =_{\mathsf{CFFD}} Q$ to emphasise that unlike "=", "$=_{\mathsf{iso}}$" and "$=_{\mathsf{sb}}$", CFFD-equivalence preserves less information on τ-transitions than on visible transitions.)

The CFFD semantics preserves the validity of stuttering-insensitive linear temporal logic formulae in the following sense: if the validity of each state proposition of a formula can be reasoned from the visible actions executed so far, if Q satisfies the formula, and if $P \leq_{\mathsf{CFFD}} Q$, then also P satisfies the formula. A process P that is smaller in CFFD-preorder than Q is thus "better" than Q in the sense that P satisfies at least the same formulae as Q, and may satisfy more. It is also said that P is *more deterministic* than Q, because LTSs that are deterministic in the sense mentioned towards the end of Section 2.7 are locally minimal with respect to CFFD-preorder.

The LTSs in Figure 8 are CFFD-equivalent to **hide** $\Sigma_3^{\mathsf{Sys}} - \{gra_1, gra_2, rel_1, rel_2\}$ **in** System_3 and **hide** $\Sigma_3^{\mathsf{Sys}} - \{req_1, gra_1, rel_1, bye_1\}$ **in** System_3. Therefore, it is correct to check the validity of $\Box\neg(\mathsf{g}_1 \wedge \mathsf{g}_2)$ and $\Box(\mathsf{r}_1 \Rightarrow \Diamond\mathsf{g}_1)$ from them.

It may seem peculiar that there is no τ-loop adjacent to state 5 of the LTS in Figure 8(b), although there are τ-loops adjacent to states 1 and 6. It looks as if the rest of the system decides not to livelock (state 5), but then cancels this decision when the client executes bye_1. However, the rest of the system does not synchronise to bye_1, and cannot thus know when it is executed.

The explanation of this phenomenon is that the only thing that the CFFD semantics preserves about the history of a state is the sequence of visible actions

that have been executed. When looking at state 6, one should not reason that the possibility of livelocking was temporarily lost at state 5, because the CFFD semantics does not preserve this kind of information. What one may read is that both livelocking (state 1) and not livelocking (state 5) are possible without or before executing bye_1. One may also read that both livelocking (state 6) and not livelocking (state 7) are possible after executing bye_1. However, all these possibilities need not be available in the same execution, although Figure 8(b) seems to indicate so. In its attempt to produce as small an LTS as possible, the algorithm that reduced the LTS twisted the information that the CFFD semantics does not preserve. This might feel irritating at first, but it is the key to obtaining small LTSs.

CFFD-equivalence is a congruence with respect to action transformation, parallel composition, and many other operators found in the literature. Furthermore, CFFD-preorder is a *precongruence*, meaning that if $P \leq_{\mathsf{CFFD}} P'$ and $Q \leq_{\mathsf{CFFD}} Q'$, then $P \,||\, Q \leq_{\mathsf{CFFD}} P' \,||\, Q'$, and similarly with the action transformer and many other operators. The reason for the presence of the initial stability predicate in the definition of the CFFD semantics is to ensure the (pre)congruence property with respect to the CCS and Lotos operators known as "choice". If only action transformation and parallel composition are used, then the initial stability predicate can be omitted without violating the (pre)congruence property.

The precongruence and linear temporal logic preservation properties imply that if the system has been verified with the clients Client_i shown in Figure 6, then it works correctly also with any clients Client'_i such that $\mathsf{Client}'_i \leq_{\mathsf{CFFD}} \mathsf{Client}_i$. Consider any client whose visible actions are req_i, gra_i, rel_i and bye_i, and that has the following properties:

1. It never livelocks. However, before the first visible action, between any two visible actions, and after the last visible action it may do any finite number of invisible actions.
2. After executing bye_i it halts, and this is the only situation where it can stop trying to execute visible actions. It need not ever try to execute bye_i.
3. After executing req_i it tries to execute gra_i and then rel_i. These are the only situations where it may try to execute gra_i and rel_i. When trying to execute gra_i, it does not try alternative visible actions, and similarly when trying to execute rel_i. It need not ever try to execute req_i.

These requirements are quite reasonable. For instance, they state that bye_i is used to indicate termination and only for that; req_i, gra_i and rel_i are executed in the correct order; and the client commits to gra_i and rel_i in the sense discussed in Section 2.5. It can be shown that if Client'_i has these properties, then $\mathsf{Client}'_i \leq_{\mathsf{CFFD}} \mathsf{Client}_i$. Therefore, if the system is proven correct with Client_i, then it is correct with any clients that satisfy the properties in the above list. The clients in Figure 6 are thus very adequate.

In addition to extending the validity of affirmative verification results to clients that are CFFD-smaller than the original clients, the precongruence property of the CFFD semantics is very helpful in constructing reduced LTSs, as will become obvious in Sections 3.5 to 3.7. What is more, it was shown in [16] that

if "$\simeq$" is a congruence that preserves (1) stuttering-insensitive linear temporal logic formulae in the above-mentioned sense and (2) deadlocks, then $P \simeq Q$ implies $P \simeq_{\mathsf{CFFD}} Q$.[3] This implies that CFFD-equivalence can produce smaller reduced LTSs than any other congruence that can be used for the verification of the properties (1) and (2). CFFD-equivalence is thus optimal in a certain precise sense.

The reasons why the details of the definition of the CFFD semantics are as they are were analysed in more detail in [36].

3.4 The meaning of "CFFD"

The name "CFFD" is an abbreviation of "Chaos-Free Failures Divergences". The semantics was given this name, because it resembles the "Failures Divergences" semantics of the CSP theory, but lacks the phenomenon of the latter that divergence implies "Chaos". Namely, the CSP failures divergences semantics of an LTS P whose alphabet is Σ_P can be defined as follows (essentially from [28, Sect. 7.4.1] or [24]):

- $CSPdivtr(P) =$
  $\{\, a_1 a_2 \cdots a_n \in \Sigma_P^* \mid \exists i \in \{0, 1, \ldots, n\} : a_1 a_2 \cdots a_i \in Divtr(P) \,\}$
- $CSPfail(P) = Sfail(P) \cup (CSPdivtr(P) \times 2^{\Sigma_P})$
- The *CSP failures divergences semantics* of P is
  $(\Sigma_P, CSPfail(P), CSPdivtr(P))$.
- $P \simeq_{\mathsf{FD}} Q$ if and only if
  $\Sigma_P = \Sigma_Q$, $CSPfail(P) = CSPfail(Q)$ and $CSPdivtr(P) = CSPdivtr(Q)$.
- $P \leq_{\mathsf{FD}} Q$ if and only if
  $\Sigma_P = \Sigma_Q$, $CSPfail(P) \subseteq CSPfail(Q)$ and $CSPdivtr(P) \subseteq CSPdivtr(Q)$.[4]

The definition implies that if σ is a divergence trace of P, then all continuations of σ are CSP-divergence traces of P, and all pairs (ρ, A) where ρ is a continuation of σ and A is just any subset of Σ_P are CSP-failures of P, independently of what P can do after σ. In particular, if ε is a divergence trace of a system, then the system has every $\sigma \in \Sigma_P^*$ as its CSP-divergence trace and every $(\sigma, A) \in \Sigma_P^* \times 2^{\Sigma_P}$ as its CSP-failure. Such a system is known as *Chaos*.

[3] It is indeed the case that the presence or absence of deadlocks cannot be specified with linear temporal logic in this setting. The reason is that the logic cannot tell a deadlock apart from a livelock. Deadlock-freedom is sometimes specified as $\Box(\mathsf{en}_1 \vee \cdots \vee \mathsf{en}_n)$ where the en_i are the enabling conditions of the atomic statements of the system, but this specification assumes extensive knowledge of the inner structure of the system, and thus cannot be used simultaneously with abstraction. Abstraction does not prevent us from specifying and verifying $\Box(\mathsf{en}_1 \vee \cdots \vee \mathsf{en}_n)$, but it prevents us from knowing that it is equivalent to deadlock-freedom.

[4] In the CSP literature this relation is usually written as $P \sqsupseteq Q$ or $P \sqsupseteq_{\mathsf{FD}} Q$. However, the present author likes to write the preorder symbol in the same direction as the subset symbols are in the definition, so that a smaller system has less behaviour than a bigger system.

According to the CSP failures divergences semantics, any system that has executed a divergence trace becomes equivalent to Chaos. Therefore, the semantics preserves no information about the behaviour of a system that has executed a divergence trace. For instance, the CSP failures divergences semantics treats both LTSs in Figure 8 as equivalent to Chaos, and thus gives no other information about their behaviour than the possibility of livelocking initially.

The reason for this rather brutal treatment of divergence is that it gives the semantics some very nice mathematical properties. Perhaps the most important of them is that any process equation has a unique maximum solution in the semantics. As a consequence, equations such as $P = a; P \;[]\; b; \mathbf{stop}$ — or as in Figure 2, or even $P = P$ — can be immediately used to define processes. The CFFD semantics lacks this property.

Fortunately, most process equations have a unique solution with respect to strong bisimilarity. This makes it possible to define the CFFD semantics of a recursively defined process by converting the definition to an LTS with the aid of strong bisimilarity, and then extracting the semantics according to Definition 12. How to proceed from a recursive process definition to an LTS modulo strong bisimilarity is well known, and explained in [23], for instance.

3.5 Compositional LTS construction

Theorems 3 and 4 together with the congruence property of CFFD-equivalence make it possible to construct a reduced LTS for the system in the *compositional* way, without ever constructing the big complete LTS of the system. Consider $\mathbf{hide}\ \Sigma_n^{\mathsf{Sys}} - \{req_1, gra_1, rel_1, bye_1\}\ \mathbf{in}\ \mathsf{System}_n$ as an example. It is isomorphic to

$$\begin{array}{ll} \mathbf{hide}\ tkn_1, \ldots, tkn_n, dem_1, \ldots, dem_n\ \mathbf{in} & \\ \qquad (\mathsf{Server\_tkn}_1 \mid\mid \mathsf{Client}_1) & \\ \mid\mid \quad \mathbf{hide}\ req_2, gra_2, rel_2, bye_2\ \mathbf{in}\ (\mathsf{Server}_2 \mid\mid \mathsf{Client}_2) & \\ \mid\mid \quad \mathbf{hide}\ req_3, gra_3, rel_3, bye_3\ \mathbf{in}\ (\mathsf{Server}_3 \mid\mid \mathsf{Client}_3) & \\ \qquad\qquad \vdots & \\ \mid\mid \quad \mathbf{hide}\ req_n, gra_n, rel_n, bye_n\ \mathbf{in}\ (\mathsf{Server}_n \mid\mid \mathsf{Client}_n) & \end{array}$$

When $2 \leq i \leq n$, let

- $\mathsf{S}_1 = \mathbf{hide}\ req, gra, rel, bye\ \mathbf{in}\ (\mathsf{Server} \mid\mid \mathsf{Client})$,
- $\mathsf{S}_i = \mathbf{hide}\ tm, dm\ \mathbf{in}\ (\mathsf{S}_{i-1}[tm/tr, dm/dr] \mid\mid \mathsf{S}_1[tm/tl, dm/dl])$,
- $\mathsf{Station}_i = \mathsf{S}_1[tkn_i/tl, tkn_{i\oplus 1}/tr, dem_i/dl, dem_{i\oplus 1}/dr]$, and
- $\mathsf{Station}_{2\ldots i} =$
  $\mathbf{hide}\ tkn_3, \ldots, tkn_i, dem_3, \ldots, dem_i\ \mathbf{in}\ (\mathsf{Station}_2 \mid\mid \cdots \mid\mid \mathsf{Station}_i)$.

Then $\mathbf{hide}\ req_i, gra_i, rel_i, bye_i\ \mathbf{in}\ (\mathsf{Server}_i \mid\mid \mathsf{Client}_i) = \mathsf{Station}_i$, and the whole system is isomorphic to

$$\mathbf{hide}\ tkn_1, tkn_2, dem_1, dem_2\ \mathbf{in}\ (\mathsf{Server\_tkn}_1 \mid\mid \mathsf{Client}_1 \mid\mid \mathsf{Station}_{2\ldots n})$$

Induction yields that

$$\mathsf{Station}_{2\ldots i} =_{\mathsf{iso}} \mathsf{S}_{i-1}[tkn_2/tl, tkn_{i\oplus 1}/tr, dem_2/dl, dem_{i\oplus 1}/dr]$$

So the whole system is isomorphic to

$$\textbf{hide } tkn_1, tkn_2, dem_1, dem_2 \textbf{ in} \\ (\mathsf{Server\_tkn}_1 \mid\mid \mathsf{Client}_1 \mid\mid \mathsf{S}_{n-1}[tkn_2/tl, tkn_1/tr, dem_2/dl, dem_1/dr]) \quad (1)$$

Although this reasoning may seem complicated, it was just restructuring of the system, comparable to reorganisation of the terms of a finite series in mathematics. The correctness of the final isomorphism is intuitively obvious. The reasoning above was presented to make it possible to check that the correctness indeed formally follows from the definitions.

Let red be an algorithm that, given an LTS, produces a CFFD-equivalent and (hopefully) smaller LTS. That is, red is a *CFFD-preserving reduction algorithm.* Let $\mathsf{S}'_1 = \mathsf{red}(\mathsf{S}_1)$ and $\mathsf{S}'_i = \mathsf{red}(\mathsf{S}''_i)$ when $2 \leq i < n$, where S''_i is obtained from S'_{i-1} and S'_1 in the same way as S_i is obtained from S_{i-1} and S_1. Because CFFD-equivalence is a congruence, $\mathsf{S}'_i \simeq_{\mathsf{CFFD}} \mathsf{S}_i$. Table 2 shows the sizes of S''_i and S'_i for various values of i, when red is the reduction algorithm in ARA. (That algorithm has been described in [40].)

	S''_i		S'_i	
i	states	transitions	states	transitions
1	38	106	7	14
2	39	97	20	43
3	85	230	31	70
4	126	353	42	97
5	167	476	53	124
6	208	599	64	151
7	249	722	75	178

Table 2. Sizes of reduced chains of client-server pairs.

Because CFFD-equivalence is a congruence, the use of S'_{n-1} instead of S_{n-1} in (1) yields an LTS that is CFFD-equivalent to **hide** $\Sigma_n^{\mathsf{Sys}} - \{req_1, gra_1, rel_1, bye_1\}$ **in** System_n. This last computation step cannot multiply the number of states by more than 38, because the number of states in $\mathsf{Server\_tkn}_1 \mid\mid \mathsf{Client}_1$ is 38. A comparison to Table 1 shows that compositional LTS construction gives huge savings with this system, and makes it possible to verify the system with much bigger values of n than the naïve approach. Because the resulting LTS is valid for only one client, the procedure has to be repeated for each value of i in the case of $\Box(\mathsf{r}_i \Rightarrow \Diamond \mathsf{g}_i)$, and for each i-j-pair for $\Box\neg(\mathsf{g}_i \wedge \mathsf{g}_j)$. Even then the total amount of work grows very much slower than the figures in Table 1, and the S'_i need be computed only once.

3.6 Interface specifications

Although compositional LTS construction works spectacularly well in the demand-driven token-ring system, it does not always succeed. It sometimes hap-

pens that a subsystem that has been isolated from its environment has many more states than the full system. Consider a fifo queue of capacity n and two different messages. It has $2^{n+1} - 1$ states. If the fifo is within the well-known alternating bit protocol [1] and the two different messages correspond to the two values of the alternating bit, then the fifo may have at most one location in which the message is different from the message in the next location. This restricts the number of the possible contents of the fifo to $n^2 + n + 1$.

To solve this problem, *interface specifications* were presented in [9]. An interface specification is a means with which the user of a verification tool can present a guess about the behaviour of the system to the tool. The tool takes advantage of the guess to reduce the number of states, sometimes dramatically. If the guess is correct, the resulting LTS is what would have been obtained without the interface specification. Otherwise, the fact that the guess is incorrect can be seen from the LTS, in which case the LTS cannot be used for verification.

To present the interface specification method, [9] relied on the theory of partially defined processes and the so-called observation equivalence semantics [23]. Because CFFD-equivalence is rather different from observation equivalence, the theory used in [9] does not apply as such in the present framework. A theory of partially defined processes for the CFFD semantics was presented in [18, Section 5.3]. That theory could have been used as a basis for the development of the interface specification method for the CFFD semantics. We shall, however, adopt a different strategy. We modify the method a bit, and develop it in a way that is rather independent of the chosen semantics. This is advantageous, because extending a semantics to partially defined processes is often nontrivial. To our knowledge, our semantics-independent approach is new.

The use of interface specifications relies on extending LTSs with a fifth component, namely *cut states*. They are a subset of the states. Cut states have no output transitions, and are treated in a special way when computing parallel compositions. A state of the parallel composition is marked as a cut state if and only if any component process is in a cut state. No successors for cut states are constructed, even if component processes that are not in a cut state are ready to execute actions that the component processes that are in a cut state are not interested in. An LTS with cut states can thus be thought of as an incomplete LTS, where cut states mark points at which the LTS has been pruned.

If P is an LTS and Φ is an action transformer for it, then the cut states of P are also cut states of $P\Phi$, and $P\Phi$ has no other cut states.

Definition 13. *An* LTS with cut states *is a five-tuple* $(S, \Sigma, \Delta, \hat{s}, \mathit{Cut})$*, where* $(S, \Sigma, \Delta, \hat{s})$ *is an LTS,* $\mathit{Cut} \subseteq S$*, and* $\forall s \in \mathit{Cut} : \forall a \in \Sigma \cup \{\tau\} : \neg(s\ {-a}\!\rightarrow)$*. Cut is a set of* cut states*.*

Let $P = (S, \Sigma, \Delta, \hat{s}, \mathit{Cut})$ *and* $P_i = (S_i, \Sigma_i, \Delta_i, \hat{s}_i, \mathit{Cut}_i)$ *be LTSs with cut states when* $1 \leq i \leq n$*, and let* Φ *be an action transformer for* P*.*

- $\mathsf{repa}(S, \Sigma, \Delta, \hat{s}, \mathit{Cut}) = (S', \Sigma, \Delta', \hat{s}, \mathit{Cut}')$*, where*
  - $S' = \{\, s \in S \mid \exists s_0, \ldots, s_k : s_0 = \hat{s} \wedge s_k = s \wedge$
    $\forall i \in \{0, \ldots, k-1\} : s_i \notin \mathit{Cut} \wedge \exists a_i : s_i\ {-a_i}\!\rightarrow s_{i+1} \,\}$,

  - $\Delta' = \{\, (s,a,s') \mid s \in S' - \mathit{Cut} \wedge (s,a,s') \in \Delta \,\}$, *and*
  - $\mathit{Cut}' = \mathit{Cut} \cap S'$.
- $P_1 \,||\, \cdots \,||\, P_n = \mathsf{repa}(S', \Sigma', \Delta', \hat{s}', \mathit{Cut}')$, *where*
  - $(S', \Sigma', \Delta', \hat{s}') = (S_1, \Sigma_1, \Delta_1, \hat{s}_1) \times \cdots \times (S_n, \Sigma_n, \Delta_n, \hat{s}_n)$, *and*
  - $\mathit{Cut}' = \{\, \langle s_1, \ldots, s_n \rangle \in S' \mid s_1 \in \mathit{Cut}_1 \vee \cdots \vee s_n \in \mathit{Cut}_n \,\}$.
- $P\Phi = (S, \Sigma', \Delta', \hat{s}, \mathit{Cut})$, *where* $(S, \Sigma', \Delta', \hat{s}) = (S, \Sigma, \Delta, \hat{s})\Phi$.

Ordinary LTSs can be thought of as LTSs with cut states by choosing $\mathit{Cut} = \emptyset$. Strong bisimilarity of LTSs with cut states is defined like strong bisimilarity of ordinary LTSs, with the additional requirement that if $s_P \sim s_Q$, then $s_P \in \mathit{Cut}_P \Leftrightarrow s_Q \in \mathit{Cut}_Q$.

An interface specification $I = (S_I, \Sigma_I, \Delta_I, \hat{s}_I, \mathit{Cut}_I)$ is a particular kind of an LTS with cut states. It is used to reduce the size of some parallel composition $P_1 \,||\, \cdots \,||\, P_n$ that is a subsystem of a bigger system $f(P_1 \,||\, \cdots \,||\, P_n)$. It represents the guess that if σ is what $P_1 \,||\, \cdots \,||\, P_n$ sees of an execution of the system as a whole, and if ρ is obtained from σ by removing those actions that are not in Σ_I, then ρ does not lead I to a cut state. In other words, the states and transitions of I are chosen such that if the guess that the user has about the behaviour of the system is correct, then no cut state of I is reached in the full system $f(P_1 \,||\, \cdots \,||\, P_n \,||\, I)$, although it may be reached in $P_1 \,||\, \cdots \,||\, P_n \,||\, I$. (If it is not reached in $P_1 \,||\, \cdots \,||\, P_n \,||\, I$, then the interface specification fails to reduce the number of states and gives thus no benefit, but the results remain correct.)

For example, a natural guess about the behaviour of the demand-driven token-ring system is that there is at most one token in any part of the ring at any time. Figure 9 shows an interface specification that represents this guess for any S_i. The black state is a cut state. It is reached if a token enters (tl) the ring segment before the previous token leaves (tr) it.

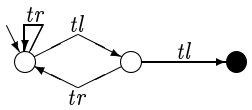

Fig. 9. An interface specification that represents the guess that a ring segment contains at most one token. The alphabet is $\{tl, tr\}$.

An interface specification $I = (S_I, \Sigma_I, \Delta_I, \hat{s}_I, \mathit{Cut}_I)$ for the parallel composition $P_1 \,||\, \cdots \,||\, P_n$ must have the following properties:

- $\Sigma_I \subseteq \Sigma_1 \cup \cdots \cup \Sigma_n$, where each Σ_i is the alphabet of P_i.
- I has no τ-transitions. (Some τ-transitions might be allowed, but stating the condition like this is simpler and does not restrict generality.)
- $\forall s \in S_I - \mathit{Cut}_I : \forall a \in \Sigma_I : s \mathrel{-a\!\!\rightarrow_I}$. That is, if a state is not a cut state, then it has an output transition for every visible action.

- $\hat{s}_I \notin Cut_I$. That is, the initial state is not a cut state. This is a meaningful requirement, because an interface specification whose initial state is a cut state would represent the certainly incorrect guess that the system cannot reach even its initial state.

These conditions imply that if each s_i, s'_i, s_I and s'_I is a state of the corresponding LTS P_i or I, then

- if $\langle s_1, \ldots, s_n, s_I \rangle -a\rightarrow \langle s'_1, \ldots, s'_n, s'_I \rangle$, then $\langle s_1, \ldots, s_n \rangle -a\rightarrow \langle s'_1, \ldots, s'_n \rangle$, and
- if $\langle s_1, \ldots, s_n \rangle -a\rightarrow \langle s'_1, \ldots, s'_n \rangle$ and $s_I \in S_I - Cut_I$, then there is s'_I such that $\langle s_1, \ldots, s_n, s_I \rangle -a\rightarrow \langle s'_1, \ldots, s'_n, s'_I \rangle$.

Let I' be I with the cut states and their adjacent transitions removed. The result is an LTS because $\hat{s}_I \notin Cut_I$. Our modified interface specification method is as follows:

- Compute a reduced version of $\mathsf{Sys}_I = f(P_1 || \cdots || P_n || I)$ using any semantics that preserves the presence of cut states. If the cut states are marked by, for instance, introducing a new action *cut* and adding a *cut*-transition from each cut state to a deadlock state, then the CFFD semantics can be used. Also many other semantic models can be used here, including the *trace semantics* which is defined by $P \simeq_{\mathsf{tr}} Q$ if and only if $\Sigma_P = \Sigma_Q$ and $Tr(P) = Tr(Q)$, where Σ_P and Σ_Q are the alphabets of P and Q. The trace semantics is beneficial, because it gives very good LTS reduction results. Furthermore, it is correct to cut the LTSs at cut states during the computation of parallel compositions as was shown in Definition 13 (assuming that the *cut*-transitions are not removed), because the behaviour after a cut state does not affect the reachability of cut states.
- If Sys_I contains any cut states, then the guess that yielded I was incorrect. Then Sys_I is useless and one should try another I.
- If Sys_I contains no cut states, then the guess was correct. In this case, $f(P_1 || \cdots || P_n) =_{\mathsf{sb}} f(P_1 || \cdots || P_n || I')$. Therefore, the original system can be analysed by computing a reduced version of $f(P_1 || \cdots || P_n || I')$ and analysing it. This is correct for any semantics x such that $P =_{\mathsf{sb}} Q \Rightarrow P \simeq_{\mathsf{x}} Q$. Almost every semantics presented in the literature has this property. Because I' contains no cut states, $f(P_1 \,||\, \cdots \,||\, P_n \,||\, I')$ can be computed using ordinary methods — no theory or algorithms for partially defined processes are needed.

To justify the correctness of the method we introduce two precongruence relations that are related to strong bisimilarity.

Definition 14. *Let* $P = (S_P, \Sigma_P, \Delta_P, \hat{s}_P, Cut_P)$ *and* $Q = (S_Q, \Sigma_Q, \Delta_Q, \hat{s}_Q, Cut_Q)$ *be LTSs with cut states.*

- $P \leq_{\mathsf{CB1}} Q$ *if and only if* $\Sigma_P = \Sigma_Q$ *and there is* "$\sim$" $\subseteq S_P \times S_Q$ *such that* $\hat{s}_P \sim \hat{s}_Q$*, and for every* $s_P, s'_P \in S_P$ *and* $s_Q, s'_Q \in S_Q$ *such that* $s_P \sim s_Q$*:*
  - *If* $s_P \in Cut_P$*, then* $s_Q \in Cut_Q$*.*

  - *If* $s_P \mathrel{-a\!\rightarrow_P} s'_P$, *then* $s_Q \in Cut_Q$ *or* S_Q *contains an* s *such that* $s_Q \mathrel{-a\!\rightarrow_Q} s$ *and* $s'_P \sim s$.
  - *If* $s_Q \mathrel{-a\!\rightarrow_Q} s'_Q$, *then* S_P *contains an* s *such that* $s_P \mathrel{-a\!\rightarrow_P} s$ *and* $s \sim s'_Q$.
- $P \leq_{\mathsf{CB2}} Q$ *if and only if* $\Sigma_P = \Sigma_Q$ *and there is* "$\sim$" $\subseteq S_P \times S_Q$ *such that* $\hat{s}_P \sim \hat{s}_Q$, *and for every* $s_P, s'_P \in S_P$ *and* $s_Q, s'_Q \in S_Q$ *such that* $s_P \sim s_Q$:
  - $s_P \in Cut_P \Leftrightarrow s_Q \in Cut_Q$.
  - *If* $s_P \mathrel{-a\!\rightarrow_P} s'_P$, *then* S_Q *contains an* s *such that* $s_Q \mathrel{-a\!\rightarrow_Q} s$ *and* $s'_P \sim s$.
  - *If* $s_Q \mathrel{-a\!\rightarrow_Q} s'_Q$, *then* $s'_Q \in Cut_Q$ *(yes,* s'_Q*!) or* S_P *contains an* s *such that* $s_P \mathrel{-a\!\rightarrow_P} s$ *and* $s \sim s'_Q$.

It follows from the definition that "$\leq_{\mathsf{CB1}}$" and "$\leq_{\mathsf{CB2}}$" are precongruences. Furthermore, if $P \leq_{\mathsf{CB1}} Q$ and $Cut_Q = \emptyset$, then $P =_{\mathsf{sb}} Q$; and if $P \leq_{\mathsf{CB2}} Q$ and $Cut_Q = \emptyset$, then $P =_{\mathsf{sb}} Q$. Also, $P \leq_{\mathsf{CB1}} P \,||\, I$ and $I' \leq_{\mathsf{CB2}} I$. From these it follows that $f(P) \leq_{\mathsf{CB1}} f(P \,||\, I)$ and $f(P \,||\, I') \leq_{\mathsf{CB2}} f(P \,||\, I)$. Therefore, either $f(P \,||\, I)$ has cut states, or $f(P) =_{\mathsf{sb}} f(P \,||\, I) =_{\mathsf{sb}} f(P \,||\, I')$.

Table 3 shows the sizes of the reduced versions of S_i obtained with repeated use of the interface specification I^9 in Figure 9. That is, it shows the sizes of S^I_i, where $\mathsf{S}^I_1 = \mathsf{S}'_1$, and

$$\mathsf{S}^I_i = \mathsf{red}\big(\, \mathbf{hide}\, tm, dm\, \mathbf{in}\, (\mathsf{S}^I_{i-1}[tm/tr, dm/dr] \,||\, \mathsf{S}'_1[tm/tl, dm/dl] \,||\, I^9)\,\big)$$

when $i > 1$. The sizes include the above-mentioned *cut*-transitions and their end states. No other transitions have been constructed from the start states of the *cut*-transitions.

i	states	transitions
1	7	14
2	16	34
3	16	34
4	16	34

Table 3. Sizes of chains of client-server-pairs with interface specifications.

In the previous section we obtained the result that $\mathbf{hide}\ \Sigma^{\mathsf{Sys}}_n - \{req_1, gra_1, rel_1, bye_1\}\ \mathbf{in}\ \mathsf{System}_n$ is isomorphic to

$$\begin{array}{l}\mathbf{hide}\ tkn_1, tkn_2, dem_1, dem_2\ \mathbf{in}\\ \qquad (\mathsf{Server\_tkn}_1 \,||\, \mathsf{Client}_1 \,||\, \mathsf{S}_{n-1}[tkn_2/tl, tkn_1/tr, dem_2/dl, dem_1/dr])\end{array}$$

In Section 3.2 we saw that its reduced behaviour when $n = 3$ is as in Figure 8(b). If S^I_{n-1} is used instead of S_{n-1} when $2 \leq n \leq 5$, then the resulting LTSs have no cut states. They are thus CFFD-equivalent to $\mathbf{hide}\ \Sigma^{\mathsf{Sys}}_n - \{req_1, gra_1, rel_1, bye_1\}$ $\mathbf{in}\ \mathsf{System}_n$, but they are obtained with less effort than with the ordinary compositional method, as comparison to Table 2 reveals. The savings are not significant. However, Table 3 drops a hint that interface specifications make a much better result possible. This result will be discussed in the next section.

3.7 Induction

An interesting aspect of the figures in Table 3 is that the LTS stops growing when $n = 2$. This makes it natural to ask: is S_3^I different from S_2^I?

This question can be answered with the ARA LTS comparison tool. The result is that $S_3^I \simeq_{\mathsf{CFFD}} S_2^I$.

This result has the consequence that $S_i^I \simeq_{\mathsf{CFFD}} S_2^I$ for any $i \geq 2$. Namely, if $S_i^I \simeq_{\mathsf{CFFD}} S_{i-1}^I$, then due to the facts that $\mathsf{red}(P) \simeq_{\mathsf{CFFD}} P$ and CFFD-equivalence is a congruence with respect to action transformation and parallel composition,

$$\begin{array}{rcl} S_{i+1}^I & = & \mathsf{red}\big(\,\mathbf{hide}\; tm, dm\; \mathbf{in}\; (\mathsf{S}_i^I[tm/tr, dm/dr] \;||\; \mathsf{S}_1[tm/tl, dm/dl] \;||\; I^9)\,\big) \\ & \simeq_{\mathsf{CFFD}} & \mathsf{red}\big(\,\mathbf{hide}\; tm, dm\; \mathbf{in}\; (\mathsf{S}_{i-1}^I[tm/tr, dm/dr] \;||\; \mathsf{S}_1[tm/tl, dm/dl] \;||\; I^9)\,\big) \\ & = & S_i^I \end{array}$$

from which the claim follows by induction. Furthermore, because of the congruence property of CFFD-equivalence,

$$\begin{array}{ll} & \mathbf{hide}\; \Sigma_n^{\mathsf{Sys}} - \{req_1, gra_1, rel_1, bye_1\}\; \mathbf{in}\; \mathsf{System}_n \\ =_{\mathsf{iso}} & \mathbf{hide}\; tkn_1, tkn_2, dem_1, dem_2\; \mathbf{in} \\ & \quad (\mathsf{Server\_tkn}_1 \;||\; \mathsf{Client}_1 \;||\; \mathsf{S}_{n-1}[tkn_2/tl, tkn_1/tr, dem_2/dl, dem_1/dr]) \\ \simeq_{\mathsf{CFFD}} & \mathbf{hide}\; tkn_1, tkn_2, dem_1, dem_2\; \mathbf{in} \\ & \quad (\mathsf{Server\_tkn}_1 \;||\; \mathsf{Client}_1 \;||\; \mathsf{S}_{n-1}^I[tkn_2/tl, tkn_1/tr, dem_2/dl, dem_1/dr]) \\ \simeq_{\mathsf{CFFD}} & \mathbf{hide}\; tkn_1, tkn_2, dem_1, dem_2\; \mathbf{in} \\ & \quad (\mathsf{Server\_tkn}_1 \;||\; \mathsf{Client}_1 \;||\; \mathsf{S}_2^I[tkn_2/tl, tkn_1/tr, dem_2/dl, dem_1/dr]) \\ \simeq_{\mathsf{CFFD}} & \text{the LTS in Figure 8(b)} \end{array}$$

whenever $n \geq 3$. This reasoning relies on the assumption that the final LTSs obtained with the interface specifications do not contain cut states. This is true because they did not when $n = 3$, and bigger systems are constructed in the same way except that S_2^I is replaced by the CFFD-equivalent LTS S_{n-1}^I.

Figure 8(b) is thus valid for any $n \geq 3$. A separate analysis shows that it is valid also when $n = 2$. Whatever useful information Figure 8(b) provides is thus valid for any $n \geq 2$. In particular, that the eventual access property holds for Client_1 has now been shown for all ring sizes of at least two.

A lot of work remains still to be done. The eventual access property must be proven for Client_i when $2 \leq i \leq n$, and the mutual exclusion property must be proven for every $1 \leq i < j \leq n$. Because $S_i^I \simeq_{\mathsf{CFFD}} S_2^I$ whenever $i \geq 2$, more than two clients between Client_1 and Client_i is equivalent to precisely two clients in the same interval. Thus the nine cases where i ranges from 2 to 4 and n ranges from i to $i + 2$ cover all the missing eventual access proofs, assuming, of course, that the final LTSs never contain cut states. But we can reason that this will hold: If the final LTS P contains cut states, then also $\mathbf{hide}\; \Sigma_n^{\mathsf{Sys}}\; \mathbf{in}\; P$ contains cut states. However, $\mathbf{hide}\; \Sigma_n^{\mathsf{Sys}}\; \mathbf{in}\; P$ is equivalent to $\mathbf{hide}\; req_1, gra_1, rel_1, bye_1\; \mathbf{in}$ (the LTS in Figure 8(b)), which does not contain cut states.

The mutual exclusion property can be fully verified by considering 36 cases: i ranges from 1 to 4, j ranges from $i + 1$ to $i + 3$, and n ranges from j to $j + 2$.

In these cases, nothing, S_1, or S_2^I is used in the intervals between Client_1, Client_i, Client_j, and Client_1 again, depending on the width of the interval.

Thus 47 analyses suffice for verifying the correctness of the demand-driven token-ring system of any size $n \geq 2$. Although 47 is not too much for an automatic tool, there is still one trick left with which the number can be made smaller. Namely, now that it is known that cut states are never reached in a full system, the cut state and its incoming transition can be removed from the interface specification in Figure 9. Let $\mathsf{S}_i'^I$ be obtained like S_i^I, but with the modified interface specification. One can verify with ARA that $\mathsf{S}_1'^I \leq_{\mathsf{CFFD}} \mathsf{S}_2'^I$. As was discussed in Section 3.3, together with the precongruence and logic-preservation properties of "$\leq_{\mathsf{CFFD}}$" this implies that although the final LTSs obtained with $\mathsf{S}_1'^I$ are not necessarily the same nor even CFFD-equivalent to the LTSs obtained with $\mathsf{S}_2'^I$, if any of the former violates the mutual exclusion or eventual access property, then also the corresponding latter LTS does.

As a consequence, the cases where $i = 3$, $j = i + 2$ or $n = j + 1$ for mutual exclusion and $i = 3$ or $n = i + 1$ for eventual access can be skipped. This leaves 17 cases.

In conclusion, 17 finite computer runs suffice for verifying the mutual exclusion property for every pair of clients and the eventual access property for every client of the demand-driven token-ring systems of all sizes ≥ 2.

As was mentioned in the introduction, induction-type of arguments have been utilised in automatic verification of concurrent systems at least in [44, 20, 5]. Induction has been applied before this article also in the context of the CFFD semantics. The applications presented in [15, 38] are particularly interesting.

In [15], the sliding window protocol [30] was verified independently of the capacities of the channels through which the protocol sender and receiver communicate. A data source or sending client was added to the system, the protocol sender was approximated from above using the variant of CFFD-preorder presented in [16], induction was used to show that the behaviour is independent of the number of channel positions, and so-called *data independency* [43] was appealed to in order to prove that the protocol is correct for arbitrary data. Data-independency means that the protocol does not look at the data it delivers, it only moves it around. It can be used to justify that if the protocol makes errors, then the errors can be found with certain fixed data sources that send only a small number of different messages, and send them in a specific order.

In [38], an integer counter that counts from some fixed value k towards 0 was represented as a chain of k processes. The counter was used as the retransmission counter of a modified version of the alternating bit protocol, where the transmission of any message is tried at most $k + 1$ times, while in the ordinary alternating bit protocol there is no upper limit. Induction was used to prove that the behaviour of the system is independent of the value of k. The same technique was used to show that if there is a finite upper limit to the number of messages that channels can lose in a row, then the ordinary alternating bit protocol behaves correctly independently of the value of the limit.

4 Discussion

We modelled the n-server demand-driven token-ring system as a parallel composition of labelled transition systems. Then we stated two requirements for the system: mutual exclusion and eventual access.

We used first the full system for verifying that the requirements hold, but this was possible only for small values of n because of the state explosion problem. Fortunately, we had divided mutual exclusion and eventual access to a number of "small" properties that refer to at most two component processes at a time. This made it possible to verify the small properties one by one by hiding most actions of the system, and then constructing a reduced LTS for the system with the compositional method. Already the basic compositional method reduced dramatically the number of states that had to be constructed, and made it possible to verify the system with much larger values of n. When it was used together with interface specifications and induction, it became possible to verify the system for *all* values of n ($n \geq 2$) by conducting 17 verification runs. The constructed reduced LTSs were either CFFD-equivalent to or CFFD-larger than the corresponding full LTSs.

The CFFD semantics is definitely not the only semantics that can be used or has been used in compositional LTS construction. Perhaps the three most commonly used semantics are *observation equivalence* [23] (also known as *weak bisimilarity*), the failures divergences model of the CSP theory [28, 4, 11], and the trace semantics.

The trace semantics was already mentioned in Section 3.6. It consists of the alphabet and the set of the traces of the LTS. It is thus essentially the same thing as a "language" in the classical theory of automata and formal languages. Because it throws almost all information on deadlocks and livelocks away, it cannot be used in the verification of the eventual access property. However, it can be used for mutual exclusion and other stuttering-insensitive safety properties. As a matter of fact, it is optimal for them in the sense that it gives at least as good reduction results as any other congruence that can be used for all stuttering-insensitive safety properties. The CFFD semantics was seen optimal in the same sense for a much larger set of properties towards the end of Section 3.3. Different sets of properties have different optimal equivalences, because the larger the set is, the more information the equivalence must preserve in order to not change the truth value of any property in the set.

The CFFD semantics was compared to the failures divergences semantics of CSP in Section 3.4. It was pointed out that they resemble each other a lot, but the CSP semantics could not have been used in this case study because of the way it treats divergence. A more detailed discussion on the relationship between the CFFD semantics, the CSP semantics and some other semantics that are based on stable failures or related notions can be found in [36]. Some of these semantics are optimal for the verification of linear temporal logic [16], deadlocks [32] or livelocks [26] in the above-mentioned sense.

Like the definition of strong bisimilarity, the definition of observation equivalence states the existence of a certain kind of a simulation relation between

the sets of the states of the LTSs that are compared. In the relation, instead of individual transitions $(s_1, a, s_2) \in \Delta$, abstracted transitions $s_1 =a\Rightarrow s_2$ for $a \in \Sigma \cup \{\varepsilon\}$ are simulated by $s'_1 =a\Rightarrow s'_2$, and similarly in the opposite direction. Because $s =\varepsilon\Rightarrow s$ is always true, observation equivalence allows the simulation of a local τ-loop $s -\tau\rightarrow s$ by nothing. Therefore, it does not preserve divergence information, and cannot be used for the verification of the eventual access property.

Observation equivalence can, however, be used for the verification of the property "in every reachable state, if a request from a client is pending, then there is at least one possible future in which the client is eventually served". This is a strictly weaker property than eventual access (which, in essence, promises service in *every* possible future instead of at least one), but is definitely better than nothing, and may well be sufficient in practice. This property talks about alternative futures in the middle of an execution, and is thus a *branching time* property. It is possible to demonstrate with an example that the CFFD semantics does not preserve this property. The CFFD semantics preserves only *linear time* properties. We have already seen an example of the CFFD semantics failing to preserve a branching time property, when we discussed in Section 3.3 the absence of a τ-loop adjacent to state 5 in Figure 8(b).

Observation equivalence can be made divergence-preserving by adding the extra requirement that a diverging and non-diverging state cannot simulate each other, where s is diverging if and only if $s -\tau^\omega\rightarrow$. Excluding the initial stability predicate, which is often not needed and is easy to take into account separately when needed, this variant of observation equivalence implies CFFD-equivalence. It can, therefore, be used in the verification of whatever can be verified with CFFD-equivalence, including eventual access. However, because it is strictly stronger than CFFD-equivalence, it does not allow reduction algorithms to throw as much information away as CFFD-equivalence allows, and therefore it does not give as good reduction results as CFFD-equivalence gives.[5]

In addition to other process-algebraic semantic models, there are many approaches to verification that are not based on process algebras. In addition to compositional LTS construction, interface specifications and induction, there are numerous other methods for alleviating the state explosion problem. We already mentioned the use of symmetries in Section 3.1 (and pointed out why it does not work in this case study), and data-independency in Section 3.7.

Two other methods that suit particularly well the framework in this article are *stubborn sets* [34, 35] and *on-the-fly verification* [42, 14, 12, 6, 27, 8, 10, and others], which can be used also simultaneously [31].

Stubborn sets are a group of methods for reducing the number of states during the computation of a parallel composition or a more complicated expression such that certain properties of the system are preserved. They can be fully

[5] It has been claimed that observation equivalence and its variants are more efficient in verification because they have faster algorithms than equivalences in the same family as CFFD. This claim is based on a misconception, confusing reduction with minimisation, as was pointed out in [33].

hidden in a tool, so that no special effort is required from the user. A natural application area for the CFFD-preserving and other process-algebraic stubborn set methods is expressions of the form **hide** A **in** $(P_1 \,||\, \cdots \,||\, P_n)$. The more actions are hidden, the better reduction results are obtained.

On-the-fly verification means the addition to the system of an observer that raises an alarm immediately when the system does something wrong. It can reduce states by

- stopping the construction of an LTS immediately when an error is found (this is more helpful than it might sound, because it prunes most of the usually very numerous extra states that an erroneous system has because of the error),
- preventing the system from entering parts of its LTS that are irrelevant for the property (one way to implement this is to remove from the observers of [10] the states and transitions from which no alarm states can be reached), and
- directing other methods, such as stubborn sets, to first investigate those parts of an LTS where an error is most likely found [31, 29].

Perhaps surprisingly, with some support from the parallel composition tool, on-the-fly verification can be used for both safety and liveness properties.

The basic ideas of the above-mentioned and many other verification algorithms and methods are presented in the survey [35].

Acknowledgements

Mikko Tiusanen has given valuable comments on a draft of this article.

References

1. Bartlett, K. A., Scantlebury, R. A. & Wilkinson, P. T.: "A Note on Reliable Full-Duplex Transmission over Half-Duplex Links". *Communications of the ACM* 12(5) 1969, pp. 260–261.
2. Bolognesi, T. & Brinksma, E.: "Introduction to the ISO Specification Language LOTOS". *Computer Networks and ISDN Systems* 14 (1987), pp. 25–59.
3. 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.
4. Brookes, S. D. & Roscoe, A. W.: "An Improved Failures Model for Communicating Sequential Processes". *Proc. NSF-SERC Seminar on Concurrency*, Lecture Notes in Computer Science 197, Springer-Verlag 1985, pp. 281–305.
5. 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.
6. 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.

7. Fernandez, J.-C.: "An Implementation of an Efficient Algorithm for Bisimulation Equivalence". *Science of Computer Programming* 13 (1989/90) pp. 219–236.
8. 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.
9. 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.
10. Helovuo, J. & Valmari, A.: "Checking for CFFD-Preorder with Tester Processes". *Proc. TACAS 2000, Tools and Algorithms for the Construction and Analysis of Systems, 6th International Conference*, Lecture Notes in Computer Science 1785, Springer-Verlag 2000, pp. 283–298.
11. Hoare, C. A. R.: *Communicating Sequential Processes.* Prentice-Hall 1985, 256 p.
12. Holzmann, G. J.: *Design and Validation of Computer Protocols.* Prentice-Hall 1991, 500 p.
13. *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.
14. Jard, C. & Jeron, T.: "On-Line Model Checking for Finite Linear Temporal Logic Specifications". *Proc. International Workshop on Automatic Verification Methods for Finite State Systems*, Lecture Notes in Computer Science 407, Springer-Verlag 1990, pp. 189–196.
15. Kaivola, R.: "Using Compositional Preorders in the Verification of Sliding Window Protocol". *Proc. Computer Aided Verification 1997*, Lecture Notes in Computer Science 1254, Springer-Verlag 1997, pp. 48–59.
16. 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.
17. Kanellakis, P. C. & Smolka, S. A.: "CCS Expressions, Finite State Processes, and Three Problems of Equivalence". *Information and Computation* 86 (1) 1990 pp. 43–68.
18. Karvi, T.: *Partially Defined Lotos Specifications and their Refinement Relations.* Ph.D. Thesis, Report A-2000-5, Department of Computer Science, University of Helsinki, Finland, Helsinki University Printing House 2000, 157 p.
19. Kokkarinen, I.: *Reduction of Parallel Labelled Transition Systems with Stubborn Sets.* M. Sc. (Eng.) Thesis (in Finnish), Tampere University of Technology, Finland, 1995, 49 p.
20. 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.
21. 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.
22. Manna, Z. & Pnueli, A.: *The Temporal Logic of Reactive and Concurrent Systems, Volume I: Specification.* Springer-Verlag 1992, 427 p.
23. Milner, R.: *Communication and Concurrency.* Prentice-Hall 1989, 260 p.
24. Olderog, E.-R. & Hoare, C. A. R.: "Specification-Oriented Semantics for Communicating Processes". *Acta Informatica* 23 (1986) 9–66.

25. 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.
26. Puhakka, A. & Valmari, A.: "Weakest-Congruence Results for Livelock-Preserving Equivalences". *Proc. CONCUR '99 (Concurrency Theory)*, Lecture Notes in Computer Science 1664, Springer-Verlag 1999, pp. 510–524.
27. Roscoe, A. W.: "Model-Checking CSP". *A Classical Mind: Essays in Honour of C. A. R. Hoare*, Prentice-Hall 1994, pp. 353–378.
28. Roscoe, A. W.: *The Theory and Practice of Concurrency*. Prentice-Hall 1998, 565 p.
29. Schmidt, K.: "Stubborn Sets for Standard Properties". *Proc. Application and Theory of Petri Nets 1999*, Lecture Notes in Computer Science 1639, Springer-Verlag 1999, pp. 46–65.
30. Stenning, N. V.: "A Data Transfer Protocol". *Computer Networks*, vol. 11, 1976, pp. 99–110.
31. 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.
32. Valmari, A.: "The Weakest Deadlock-Preserving Congruence". *Information Processing Letters* 53 (1995) 341–346.
33. 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.
34. 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.
35. Valmari, A.: "The State Explosion Problem". *Lectures on Petri Nets I: Basic Models*, Lecture Notes in Computer Science 1491, Springer-Verlag 1998, pp. 429–528.
36. Valmari, A.: "A Chaos-Free Failures Divergences Semantics with Applications to Verification". *Millennial Perspectives in Computer Science, Proceedings of the 1999 Oxford–Microsoft Symposium in Honour of sir Tony Hoare*, Palgrave 2000, pp. 365–382.
37. Valmari, A., Kemppainen, J., Clegg, M. & Levanto, M.: "Putting Advanced Reachability Analysis Techniques Together: the 'ARA' Tool". *Proc. Formal Methods Europe '93: Industrial-Strength Formal Methods*, Lecture Notes in Computer Science 670, Springer-Verlag 1993, pp. 597–616.
38. 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.
39. Valmari, A. & Setälä, M.: "Visual Verification of Safety and Liveness". *Proc. Formal Methods Europe '96: Industrial Benefit and Advances in Formal Methods*, Lecture Notes in Computer Science 1051, Springer-Verlag 1996, pp. 228–247.
40. 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.
41. Valmari, A. & Tienari, M.: "Compositional Failure-Based Semantic Models for Basic LOTOS". *Formal Aspects of Computing* (1995) 7: 440–468.
42. Vardi, M. Y. & Wolper, P.: "An Automata-Theoretic Approach to Automatic Program Verification". *Proc. 1st Annual IEEE Symposium on Logic in Computer Science*, IEEE Computer Society Press 1986, pp. 332–344.

43. Wolper, P.: "Expressing Interesting Properties of Programs in Propositional Temporal Logic". *Proc. 13th ACM Symposium on Principles of Programming Languages*, 1986, pp. 184–193.
44. 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.