Visual Verification of Safety and Liveness

Antti Valmari & Manu Setälä

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

Abstract. An exceptionally user-friendly approach to computer-aided validation / verification of concurrent and reactive systems is presented. In it, the user needs not express his verification questions formally in detail. Instead, he specifies a point of view to the system by choosing a subset of its externally observable actions. An automaton abstracts and reduces the behaviour of the system according to the choice, and shows the result graphically on a computer screen. The resulting picture represents *all* executions of the system, as seen from the chosen point of view. Thus the information in it is as comprehensive as that obtained by ordinary verification. On the other hand, like ordinary testing, the method makes it possible for a system designer to get rapid feedback with ease, to "just try the system and see how it behaves". The article concentrates on practical and philosophical issues regarding the method and contains a detailed example.

Keywords: verification, process algebra, labelled transition system, reduction

1 Introduction

Embedded software systems, communication protocols, etc. are typically *reactive* and *concurrent*. A system is reactive if it is in continuous interaction with its environment. The desired behaviour of a reactive system is usually difficult to specify and sometimes even difficult to express, because the classical model of behaviour as a partial function mapping input to output does not apply. "Concurrency" means that the system consists of several co-operating autonomous units (often called *processes*). It is extremely difficult to ensure that the processes co-operate in the intended way in all possible situations. Consequently, reactive and concurrent systems are difficult to specify, design, and validate. The designers of embedded software, parallel algorithms, and protocols are painfully well aware of these problems.

The difficulties of ensuring the correct operation of concurrent and reactive systems have led to the development of several verification methods. Many of them consist of formulating and proving theorems about (a formal model of) the system. Although much of the theorem proving stage may be automated with modern theorem prover programs, these methods require significant human assistance in formulating the theorems, developing invariants and bound functions, guiding the theorem prover through difficult proofs, etc. It is clear that these methods can be beneficially used only by mathematically talented, mathematically oriented persons. Although there are significant examples of verifications of real systems by theorem proving [CGR93], it is questionable whether theorem proving can be taken into large scale industrial use in the near future.

Another group of verification methods is based on simulating (a formal model of) the system into all states it can reach. The simulation process can be fully automated. Therefore, these *state space* methods require less mathematical skills from the user than those based on theorem proving. Lots of examples of state space -based verification tools can be found in the surveys [Boc88, InP91, Fel93], and some tools have already reached the commercial stage, e.g. [FSE94]. However, the user has still the problems of formulating the verification questions and interpreting the results. When verifying "ad-hoc" properties (such as the absence of deadlocks) this is not difficult. The situation changes when the user wants to check, for instance, that a protocol implementation conforms to an informally given service specification.

The goal of the present work is to develop a verification approach which requires less mathematical skills from its user than current state space methods. Our approach is based on state space methods, and it has become possible because of recent developments in specification methods and verification algorithms for reactive systems. Its basic idea is that the user chooses a *point of view* to the system. In practice, this means that the user lists the actions of the system he is interested in. Then an automaton abstracts the behaviour of the system according to the selected point of view and displays the result in a computer screen in a graphical form. The user gets answers to his verification questions and other useful information simply by looking at this graphical representation.

Before discussing the technicalities of visual verification, we want to give the reader a good idea of what it looks like from the user's point of view. Therefore, in Sections 2 and 3 we discuss extensively the modelling and visual verification of a small but nontrivial system, namely Peterson's mutual exclusion algorithm for two customers [Pet81]. We pay special attention to the issues arising in the verification of the "eventual access" property. A much shorter version of this example appeared in [SeV94]. Section 4 is devoted to an informal discussion of the *Chaos-Free Failures Divergences (CFFD) theory* [VaT91, VaT95] underlying our visual verification method. The CFFD theory has been developed from the failures-divergences model for CSP [BrR85, Hoa85]. We investigate why the CFFD theory is much better suited for visual verification than the CSP and CCS theories [Mil89]. In Section 5 we turn into more philosophical issues. We analyse the role of visual verification in software engineering and try to give justification for the name "visual verification", although "visual validation" might have been a technically more correct term. We also compare visual verification to testing. The conclusions, including a brief discussion of whether the method scales up into the level of "real" systems, are in Section 6.

Compared to earlier verification methods based on abstraction (e.g. [MaV89]), the novelty of our approach is the high reliance on computer-generated graphical representations of abstracted behaviour, and the emphasis on after-analysis informal interpretation of the pictures instead of before-analysis formalisation of correctness requirements. The former is much easier but, as we will argue in Section 5, not essentially less reliable than the latter. Furthermore, we will see in Section 4.2 that the well-known semantic models typically used in abstraction-based verification are ill-suited for visual verification, because they either throw too much information away, or yield

too big pictures. Visual verification of properties such as eventual access would not have been feasible without the recent development of the CFFD theory.

2 Example: Peterson's Algorithm

The CFFD theory underlying our visual verification method belongs to the large group of *process algebraic* theories. In them systems are modelled as collections of processes with synchronous communication (Ada-like rendez-vous). A process interacts with its environment by executing *visible actions* in *gates*. Each gate has a name, and a visible action occurring at a gate is denoted by the name of the gate. If two or more processes are connected to the same gate, then they have to execute the corresponding actions simultaneously. A process may also execute *invisible* or *internal* actions, which are not associated to any gate and cannot be synchronised to by other processes. Invisible actions are usually denoted by "τ". A process is often looked at as a *black-box* entity. That is, only the visible actions and their temporal orderings are considered important. This idea has made it possible to develop several powerful verification techniques and algorithms, which, in turn, are applied by visual verification.

The computer runs described in this article were performed using the *ARA* tool [VKCL93]. "ARA" is an abbreviation of "advanced reachability analysis". It can be used for analysing systems written in an extension of the specification language Basic Lotos [ISO89, BoB87]. Lotos is based on process algebras, and most of the analysis features of ARA are based on the CFFD theory. ARA was developed by VTT Electronics (VTT is a Finnish state-owned non-profit research and development organization).

2.1 First Model of Peterson's Algorithm

Peterson's algorithm is shown in pseudocode in Figure 1 almost exactly in the form it was given in [Pet81]. It consists of two processes, which communicate via three shared two-valued variables r_1, r_2 and t. Variable r_i ($1 \le i \le 2$) is used for indicating that customer i wants access to the critical section. Therefore, at line 1 r_i is set to **T** (= **true**), and again to **F** (= **false**) at line 5 when the customer exits the critical section. The purpose of t is to arbitrate between the customers if they simultaneously want access to the critical section. In line 2, each customer gives priority to the other by assigning its number to t. The notation "[*condition*] →" in line 3 denotes that a customer waits passively until the condition becomes valid. For instance, CUSTOMER_1 may proceed from line 3 to the critical section when either CUSTOMER_2 has not requested for the critical section ("r_2 = **F**"), or CUSTOMER_1 has the priority ("$t = 1$"), or both.

In order to apply visual verification to the algorithm, it is necessary to express it in a formalism with gates, actions and synchronous communication. The use of ARA forced us to use Lotos in our experiments, but this choice has no fundamental significance. For simplicity, we represent the algorithm pictorially in this article. (The mapping from the pictures to Lotos is straightforward, and ARA can do the reverse mapping automatically.) Shared variables are not available. Therefore, we have to model r_1, r_2 and t by making them processes in their own right. Our first model of the algorithm thus consists of five processes, two representing the two customers, and three representing the shared variables.

$\{ r_1 = r_2 = \mathbf{F} \wedge t = 1 \}$

$CUSTOMER_1$	$CUSTOMER_2$
1: $r_1 := \mathbf{T}$	1: $r_2 := \mathbf{T}$
2: $t := 2$	2: $t := 1$
3: $[r_2 = \mathbf{F} \vee t = 1] \rightarrow$	3: $[r_1 = \mathbf{F} \vee t = 2] \rightarrow$
4: (* critical section *)	4: (* critical section *)
5: $r_1 := \mathbf{F}$	5: $r_2 := \mathbf{F}$

Fig. 1. Peterson's mutual exclusion algorithm

The statements of Peterson's algorithm are modelled by actions. For ease of reading, the names of the actions were chosen to reflect the effects of the corresponding statements. Thus the statements "$r_1 := \mathbf{T}$" and "$r_2 := \mathbf{T}$" at lines 1 of $CUSTOMER_1$ and $CUSTOMER_2$ are given the names $setr_1T$ and $setr_2T$, respectively. Similarly, *sett2* and *sett1* model line 2, and $setr_1F$ and $setr_2F$ model line 5. The tests at line 3 are represented by actions r_1isF, r_2isF, *tis1* and *tis2*. The comments on line 4 need not be modelled.

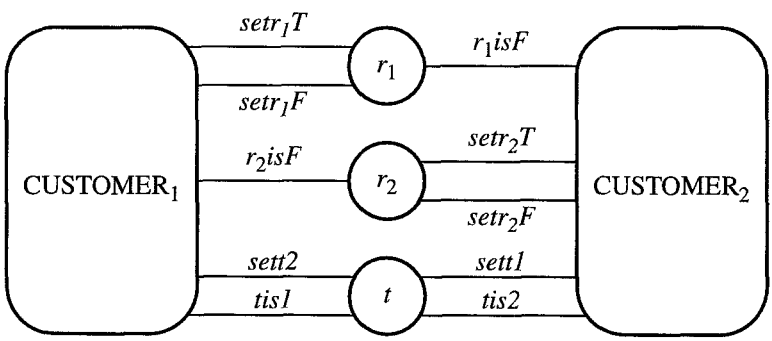

Fig. 2. Overall structure of the first model of Peterson's algorithm

The five processes are shown in Figure 2. The figure shows also which processes synchronize on each action.

Figure 3 shows the behaviours of processes $CUSTOMER_1$ and t in *labelled transition system* (*LTS*) form. An LTS is a graph-like structure with edges labelled by action names. It specifies what actions the corresponding process may perform and in which order. In Figure 3, the possible values "1" and "2" of variable t are represented by the two states of process t. (Another possibility would have been to store the value of variable t into a local variable of the corresponding process, but it would have made the presentation more complicated, because we would have had to introduce more notational conventions.) The state of t may be tested by trying actions *tis1* and *tis2*; exactly one of them is enabled at any instant of time. Action *sett1* takes t to the state corre-

sponding to the value 1 independently of the state t happened to be in when *sett1* was executed. Action *sett2* works in a similar way.

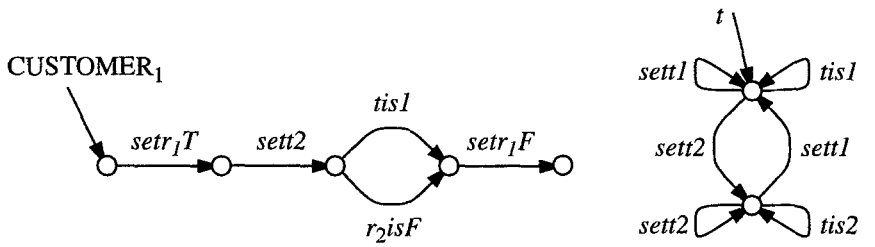

Fig. 3. LTSs of two processes of the first model

LTSs for the remaining processes can be obtained from those in Figure 3 by changing the names of actions in an obvious way.

Figures 2 and 3 model accurately Peterson's algorithm as it was given in [Pet81] and Figure 1. However, the model is in several ways unsuitable for verification. Therefore, in the next two subsections we modify it considerably.

2.2 Tuning the Model for Visual Verification

The purpose of Peterson's algorithm is to guarantee that at any instant of time, no more than one customer is in its critical section. We call this property *mutual exclusion*. Furthermore, the algorithm should guarantee the *eventual access* property: if a customer has requested access to the critical section, it should eventually be given permission to enter the section.

In order to analyse how well the model developed in the previous subsection satisfies the purpose of Peterson's algorithm, we have to recognise those actions which are important for the mutual exclusion and eventual access properties. Mutual exclusion holds if and only if CUSTOMER$_2$ is never in the state preceding action $setr_2F$ while CUSTOMER$_1$ is in the state preceding $setr_1F$. Talking about local states of processes is, however, a violation against the black-box view assumed by the verification algorithms used by visual verification. Fortunately, this problem is easy to solve: we make visible the actions preceding and succeeding the interesting states, and reason from the visible actions executed so far whether the customer is in its critical section. So, we should make actions r_1isF, r_2isF, *tis1*, *tis2*, $setr_1F$ and $setr_2F$ visible.

The name "$setr_1F$" is not, however, very informative from the verification point of view. Therefore, we give it a new name "rel_1", corresponding to the fact that with this action CUSTOMER$_1$ *rel*eases the critical section. Similarly, $setr_2F$ is re-baptized to rel_2.

According to the same logic we should next modify the names *tis1* and r_2isF to $enter_1$, but now we have a problem: two actions have to be mapped into one name without modifying the synchronisation structure of the model. Actually, Lotos has features with which this can be done, but again we prefer a solution which does not force us to introduce extra notational conventions. Namely, we introduce a new action immediately before rel_1 and call it $enter_1$, and similarly with CUSTOMER$_2$. The criti-

cal section of CUSTOMER_i is now interpreted to be the state between actions $enter_i$ and rel_i. Actions $enter_1$ and $enter_2$ will be used only for verification purposes, and they will not be synchronised to by other processes of the model than the corresponding customers. Therefore, the correctness of the algorithm does not change if they are removed. But then we get the original critical sections. In the same way, if the renaming of $setr_iF$ to rel_i would have been impractical, we could have introduced a new action immediately before $setr_iF$ and given it the name rel_i.

In order to make talking about the eventual access property more pleasant, we change the names "$setr_iT$" to "req_i" denoting "*req*uest". The current version of CUSTOMER_1 is shown in Figure 4.

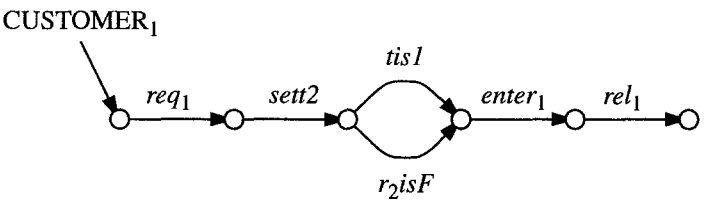

Fig. 4. Second version of CUSTOMER_1

The interesting actions regarding the mutual exclusion and eventual access properties are now called req_1, req_2, $enter_1$, $enter_2$, rel_1 and rel_2. We declare them visible and the others invisible.

Our model of Peterson's algorithm is now technically in a suitable form for visual verification. Unfortunately, it has some subtle but serious problems: it implicitly contains some unjustified assumptions. Therefore, an attempt to verify it using *any* verification method (visual verification or others) would yield incorrect results. In the next subsection we fix these problems.

2.3 Fixing Incorrect Implicit Assumptions

The model developed in Section 2.2 represents Figure 1 accurately, and is technically suitable for visual verification. However, it contains an implicit assumption which should not be made. Namely, in it customers try to get access to their critical sections only once. A verification conducted using it reveals whether the mutual exclusion property holds when the customers try to enter the critical sections for the first time, but it does not tell anything about what happens in subsequent times.

To fix this problem, we redirect the rel_i-actions to lead to the initial states. It is important to notice that the algorithm in [Pet81] did not contain this kind of return to initial state. Instead, its correctness proof took into account the possibility that it is repeated any number of times. That is, Peterson did not represent repetition formally in the algorithm, but informally in its correctness proof. This episode is an example of the fact that the use of a formal verification method forces the formalisation of details which may be difficult to recognise from the original problem description.

After this modification, the model can be used for the verification of the mutual exclusion property. Regarding the eventual access property, one problem still remains.

Namely, we have not specified what actions the customers *have to do* eventually, and what they *may choose to do not* if they don't want to. It is obvious that if one customer stays in its critical section forever, then the other customer cannot ever be granted access to its critical section without violating the mutual exclusion property. So, we have to require that if CUSTOMER_i has just executed $enter_i$, then it will execute rel_i sooner or later. In general, if a customer has started to execute its part of the mutual exclusion algorithm, it should execute it to completion (if it can).

On the other hand, a customer needs not ever request for access to the critical section if it does not want to. Therefore, the actions req_i are in a special position: the customers may legally forever refuse executing them, even if they are the only executable actions in the whole system. This difference between req_i and the other actions has not been encoded into our model. The CFFD theory includes the general assumption that if something can happen, then something will eventually happen. As a consequence, our current model of the customers specifies that a customer will eventually request for access to the critical section, if nothing else can happen. But under this assumption eventual access would be guaranteed even by an algorithm which first grants access to CUSTOMER_1, then to CUSTOMER_2, then again to CUSTOMER_1, and so on independently of who has requested for access.

Representing the difference between actions (or statements) which need and need not be performed is tricky. In temporal logics various *fairness* assumptions are used for this task. Fairness assumptions do not, however, work in the context of the CFFD theory (or almost any other process algebraic theory, for that matter), because they are not *compositional* in the sense required by the algorithms (see Section 4.2). Therefore, we model the difference using extra actions as follows.

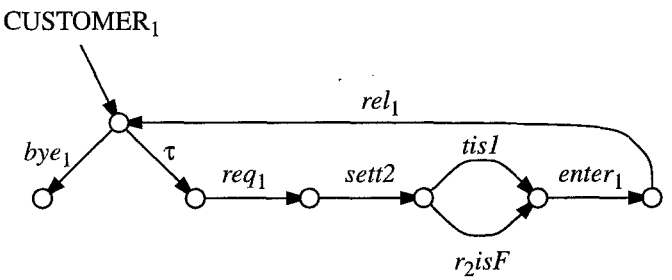

Fig. 5. Final version of CUSTOMER_1

When a customer is in its initial state, there are two possibilities: it will eventually request for access to the critical section, or it will not. This decision can be modelled by two actions starting at the initial state, one leading to the state immediately preceding action req_i, and the other leading to a deadlock state. These actions should be inaccessible to the other processes of the model, because otherwise they might force the decision by blocking one of the actions, which would be against the assumption that the decision is the customer's. So, it would be natural to make them invisible. However, the action leading to the deadlock is necessary for interpreting the verification

results. Therefore, we give it the name bye_i, and require that only CUSTOMER_i is connected to it. The resulting model of CUSTOMER_1 is in Figure 5.

The overall structure of the final model is shown in Figure 6. In the figure, visible actions are shown as actions accessible outside the system.

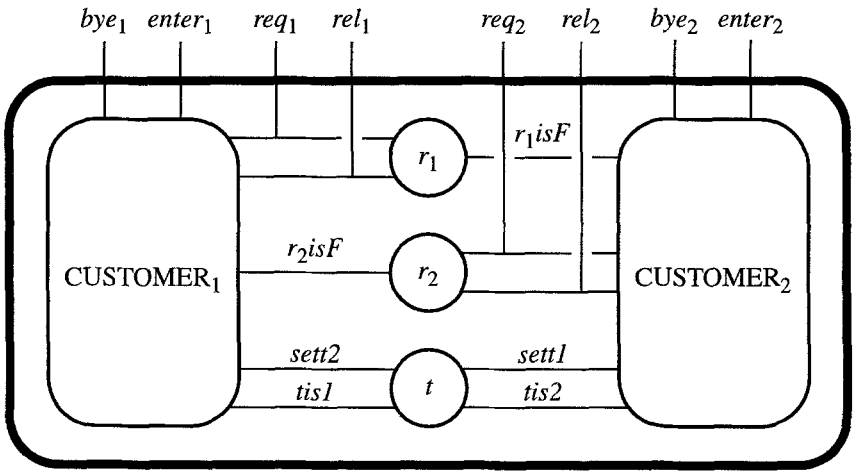

Fig. 6. Overall structure of the final model

At this point we would like to emphasize that it was not our visual verification method which forced us to make the modifications described in this subsection. The previous model was an inaccurate representation of the system we wanted to verify — it simply did not contain all the information the correctness of the algorithm depends on. (We will return to this issue in Section 3.2.) Therefore, no matter what verification method were used, the model would have been inappropriate.

3 Visual Verification of Peterson's Algorithm

3.1 Verification of the Model

In the previous section we developed a verification model for Peterson's algorithm. The ARA tool can be used to construct an LTS representing its behaviour, i.e. the joint behaviour of the processes of the system. It has 74 states and 140 transitions. So it is too big for manual inspection or to be shown here.

Like many other tools based on process algebraic theories, ARA contains a *reduction* algorithm which takes an LTS as input and produces a smaller LTS which behaves in the same way as the input LTS as far as the visible actions are concerned. When this algorithm is applied to the LTS obtained from our Lotos verification model of Peterson's algorithm, the result has 33 states and 51 transitions. It is thus much smaller than the input LTS, but still far too big for manual inspection.

With typical state space verification techniques, the next step would be an investigation of the properties of the LTS with some query or model checking tool; or a repe-

tition of the LTS construction after adding assertions and other "on-the-fly" error-catching devices to the verification model; or the (automated) comparison of the LTS with another LTS representing the requirement specification of the system. (Actually, ARA was originally designed for the latter two kinds of verification.) The use of a query tool requires that the user can find the right questions and express them in the input language of the tool. Except for some simple properties such as mutual exclusion, the design of error-catching devices is difficult and error prone. For instance, the eventual access property has to be formulated using devices such as states which should or should not be visited infinitely many times during an execution. And, finally, LTS comparison requires that someone designs the LTS representing the requirements. In conclusion, all these techniques require that the user has a good idea of what he wants to know about the system, and that he can express it in a formal way.

With visual verification the user needs not know more at this stage than what actions are important regarding the properties he wants to check. For instance, the validity of the mutual exclusion property depends only on the relative ordering of entries to and exits from the critical sections. In terms of our verification model, this means that only $enter_1$, $enter_2$, rel_1 and rel_2 are important. Therefore, we continue by declaring them visible and the other actions invisible. Then we use ARA to construct and reduce the LTS of the model. The full LTS of the new model differs from the full LTS of the previous model only in that all bye_i- and req_i-transitions have been changed to invisible transitions, so the numbers of states and transitions are still 74 and 140. However, the reduced LTS has become much smaller, because more actions are invisible. It contains only 4 states and 5 transitions, and it is shown in Figure 7.[1]

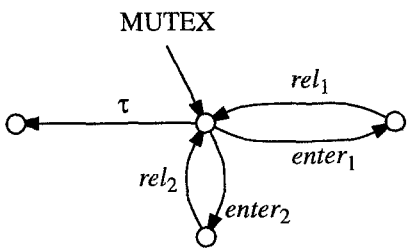

Fig. 7. A reduced LTS for verifying mutual exclusion

It is easy to see from Figure 7 that CUSTOMER_1 is in its critical section in the rightmost state and only in it, and CUSTOMER_2 is in its critical section in the bottommost state and only in it. The two customers are thus never in their critical sections simultaneously, and the mutual exclusion property is not violated.

However, if we investigate Figure 7 further, it may start to seem surprising. For instance, why is there a τ-transition leading to a deadlock? It is because we wanted that

1 ARA has a tool for showing small LTSs on computer screen. Figure 7 was designed with it. We had to redraw the figure because, to save space, ARA does not write action names next to the transitions; instead, it represents them by different colours. All LTS figures in this article were drawn according to the lay-out designed by ARA.

in our model, customers need not ever (or ever again) request for access to the critical sections if they do not want to. If both customers choose not to request for access, then nothing happens in the system, so it is in a deadlock. The reader may also miss a τ-transition before action $enter_1$, or a state where an $enter_1$-rel_1 cycle is possible but an $enter_2$-rel_2 cycle is not. The absence of such states and transitions will be explained after the introduction of CFFD-semantics in Section 4.1. We will explain there also why it is good and important that they are absent.

To verify that CUSTOMER_1 eventually gets access to the critical section if it requests for it, it is sufficient to leave visible the actions req_1 and $enter_1$. The resulting reduced LTS is shown in Figure 8. From it we see that the system is bound to execute $enter_1$ if it has executed req_1, so the eventual access property holds.

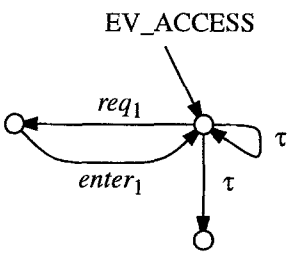

Fig. 8. A reduced LTS for verifying eventual access

However, there seems to be something peculiar also in Figure 8. According to it, the system may deadlock or livelock in its initial state. We already know that the system terminates if both customers decide not to request for access to the critical sections. Unfortunately, we do not know whether this is the *only* reason for the deadlock in Figure 8. Similarly, the livelock (τ-loop) has an acceptable explanation. If CUSTOMER_1 requests never but CUSTOMER_2 requests repeatedly for access to the critical section, then the system runs around forever, but none of the corresponding actions is visible in our model, because we left visible only req_1 and $enter_1$. So the infinite execution is represented by a τ-loop. But, again, we do not know whether there are also other reasons for the livelock. According to Figure 8, it is at least in theory possible that CUSTOMER_1 wants to request for access to the critical section, but cannot do that, because the system has deadlocked or livelocked.

In order to ensure that the deadlock and livelock in Figure 8 have only legal reasons, we analyse one more model. Its purpose is to check whether CUSTOMER_1 can request for access to the critical section if it wants to. We call this property *unprevented request*. Action bye_1 denotes that CUSTOMER_1 does not any more want to get access to the critical section. Therefore, we leave actions req_1 and bye_1 visible. Furthermore, in order to get rid of the τ-loop, we have to leave visible at least one of the actions of CUSTOMER_2 other than bye_2. Without knowing what would be the best one, we arbitrarily choose req_2. The resulting reduced LTS is shown in Figure 9.

From Figure 9 we see that deadlocks and livelocks cannot prevent CUSTOMER_1 from executing req_1, so the unprevented request property holds.

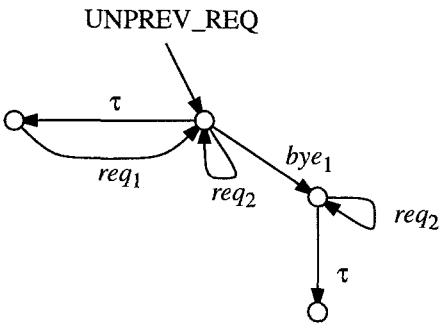

Fig. 9. A reduced LTS for verifying that CUSTOMER_1 may perform req_1 if it wants to

To complete our verification, we have to repeat the analysis of the eventual access and unprevented request properties with the roles of the customers reversed. As the result, we get figures which are otherwise the same as Figures 8 and 9, but all "1"s have been replaced by "2"s and vice versa.

3.2 Additional Checks

We have verified that our model has the mutual exclusion, eventual access and unprevented request properties. The development of our verification model was, however, rather complicated; we motivated two modifications to Peterson's original version by saying that otherwise certain kinds of errors cannot be detected. This raises the question: can we rely on the ability of even our final verification model to reveal errors? To test this, we do now some analysis runs with some errors introduced to Peterson's algorithm, and check that the errors manifest themselves in the reduced LTSs.

In our first check we remove the statements at line 2 of Figure 1. This should fool CUSTOMER_1 to think that it is its turn to enter the critical section although in reality it is not, leading to a violation of the mutual exclusion property. And it does, as can be seen from the leftmost state in the bottom row in Figure 10.

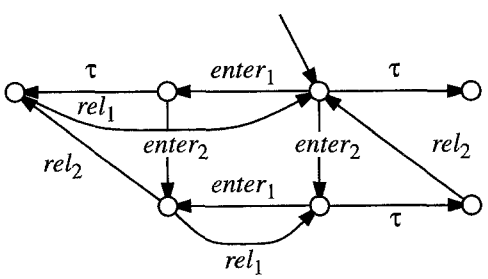

Fig. 10. A mutual exclusion verification picture of an incorrect algorithm

In the next check we remove the tests "$r_1 = \mathbf{F}$" and "$r_2 = \mathbf{F}$" from line 3 of Figure 1. After this modification, the algorithm forces the customers to enter their critical sec-

tions by turns. This error emerges in Figure 11 (a) as a deadlock after req_1, causing violation of eventual access. The error could not have been detected with a verification model not representing the assumption that a customer needs not request for access to its critical section if it does not want to. Indeed, if we remove also the bye_i-actions, then we get Figure 11 (b), where eventual access holds.

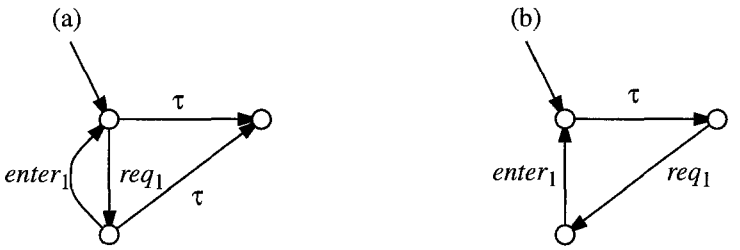

Fig. 11. Eventual access verification pictures of an incorrect algorithm when the customers (a) may (b) may not decide *not* to request for access to the critical section

4 Theoretical Basis of Visual Verification

It is obvious from Section 3 that the ability to reduce LTSs without modifying their visible-action-related (often called *externally observable*) properties is crucial to visual verification. Nothing can be modified without changing at least some of its properties — at least the number of states changes in a reduction (or otherwise we get no reduction). Of course, on the basis of reduced LTSs no conclusions should be made about properties which are not preserved in the reduction. Therefore, it is important to know exactly what properties are preserved.

The reduction algorithm in the ARA tool preserves the so-called *Chaos-Free Failures Divergences (CFFD) semantics* of systems. CFFD-semantics were first described in [VaT91], and their theory has been extensively studied in [VaT95]. We do not repeat the formal theory here but try only to give an intuitive picture. We restrict our discussion to finite-state systems; for the infinite-state case see [VaT95].

4.1 CFFD-Semantics

The CFFD-semantics of a finite-state system consist of its *traces*, *stable failures*, *divergence traces* and *initial stability*. We consider each of these in turn. (Actually, the traces are not explicitly present in the formal definition of CFFD-semantics, because they can be derived from the other components.)

A *trace* of a system is any sequence of visible actions the system may execute starting from its initial state. The system is allowed to execute invisible actions before, after, and between the visible actions, but they are not included into the trace. The execution need not be complete; even the execution consisting of doing nothing generates a trace, namely the *empty* trace containing zero visible actions. The empty trace is usually denoted by "ε". The traces of the model in Figure 8 are

ε,
req_1,
req_1 $enter_1$,
req_1 $enter_1$ req_1,
req_1 $enter_1$ req_1 $enter_1$,

and so on. The significance of the traces for verification should be apparent. For instance, mutual exclusion is violated if and only if the system has a trace where $enter_1$ and $enter_2$ occur without rel_1 in between, or $enter_2$ and $enter_1$ occur without rel_2 in between.

Stable failures carry information about deadlocks and the choices made or allowed by the system. A stable failure consists of two components. The first component is a trace of the system, and the second component is a set of actions. The interpretation of a stable failure is that it is possible for the system to execute the trace in such a way that the system ends up in a state where it can execute neither actions from the set nor invisible actions. For instance,

$$(req_1\ enter_1\ req_1,\ \{req_1\})$$

is a stable failure of the system in Figure 8, because it cannot execute req_1 or τ after cycling the req_1-$enter_1$-cycle $1^1/_2$ times; and

$$(enter_1\ rel_1\ enter_1\ rel_1\ enter_2\ rel_2,\ \{enter_1, enter_2\})$$

is a stable failure of the system in Figure 7, because it can make the τ-move and end up in a deadlock any time it has returned into its initial state. However, $(\varepsilon, \{req_1\})$ is not a stable failure of the system in Figure 5, because in the initial state it can execute the invisible action τ, and after τ it can perform req_1. On the other hand, $(\varepsilon, \{bye_1\})$ is its stable failure.

A system can end up in a deadlock after executing the trace $a_1a_2...a_n$ if and only if $(a_1a_2...a_n, \Sigma)$ is its stable failure, where Σ is the set of all visible actions of the system. In this way stable failures record the deadlocks of the system.

A *divergence trace* of a system is a trace such that, after executing it, the system may be able to execute invisible actions without limit. For instance, due to the τ-loop adjacent to the initial state of the LTS in Figure 8, ε and req_1 $enter_1$ are its divergence traces but req_1 is not. Divergence traces carry information about the livelock and progress properties of systems. For instance, the system in Figure 8 is guaranteed to perform $enter_1$ after req_1, but not the other way round, because it may run around forever in the τ-loop instead (and because it may deadlock).

The *initial stability* consists of only one bit of information. A system is initially stable if and only if its first action cannot be invisible. Initial stability is included into CFFD-semantics for technical reasons we cannot go into here (see [VaT91 or VaT95]). In many cases it can be omitted. Its only effect is that sometimes the reduced LTS may contain an apparently unnecessary initial τ-move, such as the one in Figure 11 (b).

The fact that CFFD-semantics preserve only the traces, stable failures, divergence traces and initial stability has sometimes surprising consequences. For instance, in Figure 7 the transition labelled $enter_1$ starts at the initial state, although $CUSTOMER_1$ performs four other actions before $enter_1$. This is because the four actions were declared invisible, and the reduction algorithm in ARA introduces invisible actions to

reduced LTSs only where necessary for representing the stable failures, divergence traces and initial instability. Action $enter_1$ does not need a preceding τ-transition, because the only effect of such a transition would be the addition of stable failures such as (ε, $\{enter_2\}$) to the semantics, but they are already in the semantics because of the τ-transition leading to a deadlock.

An attempt to locate the execution of bye_2 in Figure 9 will yield another surprise. After bye_2, CUSTOMER_2 cannot ever perform req_2; before bye_2, nothing can prevent CUSTOMER_2 from executing req_2. In the leftmost state of Figure 9 bye_2 thus appears to have been executed, while in the initial state it appears not. But how can there then be a transition from the leftmost state to the initial state? The answer is that because bye_2 was declared invisible, the figure does not even try to show when it is executed. Instead, it shows whether it is possible for the system to refuse visible actions. At any state the system may invisibly refuse req_2 by choosing bye_2 instead. This is shown by τ-transitions leading to states where req_2 is not executable. The system refuses req_2 also if it has performed bye_2 some earlier time. CFFD-semantics do not reveal whether the decision between executing and refusing req_2 was made now or some earlier time. It shows only that after the visible actions shown, the system may be in a state where req_2 is executable, but it may also be in a state where it is not. Note that before executing bye_1, the system cannot enter a state where req_1 cannot be executed immediately or after some invisible actions. This is true about the model, preserved by CFFD-semantics, and apparent in Figure 9.

Similar reasoning explains why Figure 7 does not have a state where an $enter_1$-rel_1 cycle is possible but an $enter_2$-rel_2 cycle is not.

The user has also to be careful with the interpretation of divergences. CFFD-semantics do not distinguish τ-loops which can be exited from τ-loops which cannot. For instance, CFFD-semantics consider equivalent the two LTSs in Figure 12. The reader may dislike this feature of CFFD-semantics (so do we, because it makes more difficult the analysis of progress properties), but so far nobody has been able to find a compositional CFFD-like semantic model not suffering from this problem.

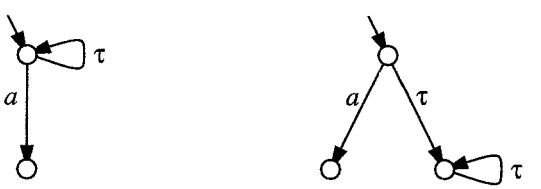

Fig. 12. Two CFFD-equivalent processes

In conclusion, difficulties in understanding reduced LTSs are often due to forgetting that CFFD-semantics allow throwing lots of information away. It is, however, crucial for visual verification that the LTS figures are as small as possible. Therefore, we have to make a compromise; we cannot afford to preserve information which is not really needed. In the next subsection we explain why we think that CFFD-semantics are a good compromise.

4.2 Why CFFD-Semantics?

Why did we choose CFFD-semantics for the basis of visual verification, and not one of the well-established semantic models, such as the *trace semantics*, *CSP-semantics* [BrR85, Hoa85], or Milner's *observation equivalence* [Mil89]?

The *trace semantics* of a system consist simply of its traces. They can be used for the verification of so-called *safety properties*, such as mutual exclusion. Actually, they are the best possible semantic model for verifying general safety properties. However, because they do not preserve deadlock and divergence information, they do not consider the system in Figure 8 any different from the system in Figure 13 (a), although the eventual access property holds in the former but not in the latter. Hence trace semantics do not preserve many of the properties we want to verify.

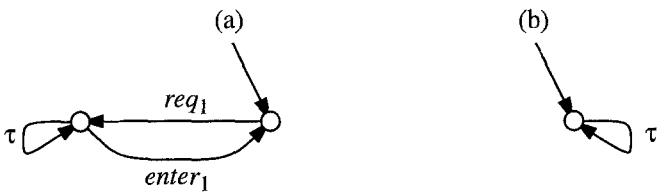

Fig. 13. A (a) trace equivalent (b) CSP-equivalent LTS to the LTS in Figure 8

CSP-semantics are very similar to CFFD-semantics; actually, the latter were developed from them. Because of reasons we cannot go into here, CSP-semantics do not preserve any information of a process after it has executed a divergence trace. Therefore, if CSP-semantics were used, the analysis of the eventual access property would have produced the LTS in Figure 13 (b) instead of Figure 8. Figure 13 (b) does not tell us anything about the eventual access property. CSP-semantics are thus not very useful if we may have livelocks in our systems. Because visual verification is experimental, intermediate models often contain livelocks even if we want the final models not to contain them. Furthermore, visual verification requires that only few actions are left visible, which causes "view-dependent" livelocks such as the one in Figure 8. In the absence of livelocks, CSP- and CFFD-semantics are almost the same, so it does not make any difference to use one instead of the other. ("Almost", because CSP-semantics do not contain the "initial stability" component.)

The "standard" version of Milner's *observation equivalence* does not preserve divergence information, so it, too, would be useless in the verification of eventual access. It is possible to add divergence preservation to observation equivalence. The result was investigated in [Elo94]. It was proven in [Elo94] that divergence-preserving observation equivalence preserves strictly more information than CFFD-semantics. Hence it can be used for the verification of all the properties which may be verified using CFFD-semantics, and more. However, the extra strength does not come for free. The more properties are preserved, the less an LTS can be reduced. If the semantics preserve properties we are not interested in, then the reduced LTSs are often unnecessarily big. For instance, with divergence-preserving observation equivalence the LTS in Figure 7 would have contained 33 states and 64 transitions, making it virtually

unreadable. Similarly, Figures 8 and 9 would have grown to 10 and 10 states and 16 and 22 transitions. In the case of Figures 7 and 9 the numbers would have been the same with ordinary observation equivalence, because the LTSs shown in them do not contain divergences. So, divergence-preserving and ordinary observation equivalence do not seem to produce small enough reduced LTSs for visual verification.

One important reason for choosing CFFD-semantics was that they (like all the above-mentioned established semantic models) are *compositional.* That is, if we replace a component process in a system by a CFFD-equivalent process, then the resulting system is CFFD-equivalent to the original one. Compositionality gives us powerful weapons for fighting the *state explosion* problem (see e.g. [ChK93, GrS90, MaV89, Val93]), which all state space verification methods suffer from. Although it may sound surprising, compositional semantic models preserving required properties and not much more are very difficult to find. For instance, the Lotos testing equivalence described in [ISO89, Annex B] is not compositional.

It was shown in [KaV92] that CFFD-semantics are the weakest possible compositional semantic model which preserves all deadlocks, and all properties expressible in classic state-based linear temporal logic excluding the "next state" operator. This logic is often used for specification and verification. This is one more reason why we believe CFFD-semantics to be a good compromise between expressibility and reduction power.

5 Visual Verification in System Development

The basic difference between visual verification and classic state space verification is that the user of the former needs not specify his verification questions in great detail. Instead, he only lists the actions which are interesting regarding the property, and lets an automaton abstract away the uninteresting actions and represent the result in a graphical form. Then the user looks at the resulting picture and tries to decide whether the behaviour shown in it is acceptable.

Of course, if the user does not read the picture carefully, he may ignore some errors in the system. But the same may happen with classic verification methods if the user does not design and formalise verification questions carefully. With visual verification, the picture may warn him about problems he might not otherwise have even thought about. For instance, Figure 8 raised our concern about whether CUSTOMER_1 may be prevented from ever indicating that it wants access to the critical section. Similar feedback is more difficult to obtain with classic verification.

An ordinary test may be performed on software in two different ways. The user may compute the answer the program should produce before starting the test run, or he may just start the program and check the correctness of the output only after seeing it. The former approach is more systematic and the latter is more flexible. The former approach is often used in serious final testing, but it is almost certain that most programmers do test runs of the latter kind while developing programs.

Similarly, ordinary verification is good for checking the final result. However, when we are developing a system, it is useful to get some feedback about its behaviour without consuming too much effort for the designing of verification questions. Visual verification makes it possible to just "try the system and see what it does". On the other

hand, the results of visual verification are different from the results of ordinary test runs in that they are comprehensive. No number of test runs can guarantee that the mutual exclusion property holds, but a single visual verification run may suffice.

The word "verification" is usually used to denote the act of checking that a system satisfies some specification. By "validation" people usually mean the act of checking that a system behaves as its designers or users want. Validation is by necessity informal, because at some stage the behaviour of the system has to be compared to the intuitive expectations living in the user's head. The user may write a formal specification expressing his expectations and verify that the system satisfies it, but then he has the problem of checking whether his specification agrees with his intuition. Therefore, no matter how we validate a system, there is always at least one informal step. With verification this step may be made easier, however, because the behaviour of the specification may be much easier to understand than the behaviour of the system, and verification guarantees that if the specification is acceptable, then also the system is.

In our visual verification method, the user does not write any document, formula or set of assertion statements expressing how he wants the system to behave. Instead, he asks an automaton to show the behaviour of the system from some point of view, and then accepts or rejects the result. Therefore, visual verification is not "verification" in the above sense of the word; "visual validation" would have been a technically more correct term.

The goal of validation is to get as much confidence as possible to the behaviour of the system. The obtained confidence level depends on many things, including the reliability of the formal steps (programmers of verification tools and even mathematicians make errors), and the difficulty of the unavoidable informal step. With ordinary verification the main concern is typically whether the user has found the right questions (and enough of them), and whether the representations of the questions as assertions, temporal logic formulas, etc. are accurate. With visual verification the main concern is typically whether the user interprets correctly the final LTS pictures. We believe that interpreting the pictures is no more difficult than writing temporal logic formulas, for instance. Therefore, we claim that the confidence level obtained by visual verification is not inferior to the confidence level obtainable by ordinary verification, and is far above the confidence level obtainable by performing test runs on the verification model. Quite the contrary, visual verification is sometimes better than ordinary verification, because a weird-looking picture may cause the user to realize problems he would not have otherwise even thought about.

So we believe that the confidence level obtained by visual verification is comparable to the confidence level obtained by ordinary verification. Furthermore, the reduction stage in visual verification is certainly formal. The term "visual validation" would have hidden the fact that the method is based on formal theory and automatic tools. Because of these reasons we chose the name "visual verification".

6 Conclusions

Visual verification makes it possible for system developers to get comprehensive information about the behaviours of their systems without having to pay much effort to the design of verification questions. In essence, ordinary verification answers the question

"Does the system behave in this way?", while visual verification answers the question "How does the system behave, as seen from this point of view?". Therefore, visual verification is much better suited for experimental use than ordinary verification. With it the system developer can play with his design ideas and get almost immediate feedback. If some picture looks peculiar, more information may be obtained by repeating the analysis with a different set of visible actions. We saw an example of this in Section 3.1, where we did an extra analysis to find out the reasons for the deadlock and livelock in Figure 8. We believe that due to these features, visual verification is easier to adopt by industrial system designers than ordinary verification.

Visual verification requires that the user is able to interpret the LTS pictures correctly. Ordinary verification requires that the user is able to recognise the essential features of the correct behaviour of the system and express them formally. Therefore, both approaches to verification require some skills from the user. In our opinion, mis-interpretation of LTS pictures is no more likely than insufficient or incorrect encoding of the expected correct behaviour. Furthermore, *all* formal verification methods require that all essential assumptions regarding the behaviour of the system are represented in its verification model in one way or another. We saw in Section 2.3 that important assumptions may be hidden in original system descriptions, so they may be easily missed. Because of these (and other) reasons, all verification methods are, in the end, unreliable. We believe that the results obtained by visual verification are in practice as reliable as those obtained by ordinary verification.

We used the recently developed CFFD semantic model in our visual verification method. This was not an arbitrary choice. We demonstrated in Section 4.2 that observation equivalence would have yielded too big pictures, and CSP-equivalence would have made impossible the analysis of systems with livelocks. So visual verification would not have been feasible with either of them. Visual verification of safety properties is possible with trace semantics, but if one wants to go beyond that, something like CFFD-equivalence is necessary.

An important question regarding the practicality of visual verification is whether it scales up into the level of "real" systems. Two issues may seem problematic:

- Will the reduced LTSs be small enough to be understandable?
- Will the LTS construction and reduction algorithms be able to process big enough systems?

In the case of Peterson's algorithm the reduced LTSs were certainly small enough. It is clear that to keep them small, the set of visible actions should be small and carefully chosen. It is therefore essential to concentrate on one aspect of the system at a time. For instance, in Section 3.1 we constructed one LTS picture for checking mutual exclusion, another one for eventual access, and yet another to ensure that the customer may not be prevented from requesting access to the critical section. We believe that this approach is applicable also with larger systems than Peterson's algorithm. Well-designed large systems contain several interfaces between system components, and they are likely to be amenable for visual verification. However, more experience has to be gathered before it can be said for certain how often in practice it is possible to obtain small enough LTSs.

The problem of too big systems hampers all state space verification methods, so here visual verification is not better or worse off than ordinary state space verification. The problem has been extensively studied and interesting results have been obtained. A proper discussion of them would be beyond the scope of this article, but let us mention that many of them are applicable in our context, such as compositional LTS construction [ChK93, GrS90, MaV89, Val93] and stubborn sets [WoG93, Val94]. See also [Val95] for a discussion of the state explosion problem in the context of CFFD-semantics.

In addition to Peterson's algorithm, we have applied visual verification to various versions of the alternating bit protocol [VKS96] and to a collision-avoidance protocol for the Ethernet, with good results. Capacity problems with the ARA tool have prevented us from analysing larger examples, but we are currently developing more powerful tools. One thing can be said for certain: a typical system designer can verify bigger systems and obtain more reliable results by visual verification than by just looking at his design.

Acknowledgements

The work of A. Valmari was partly funded by the Technical Research Centre of Finland (VTT) and the Technology Development Centre of Finland (TEKES) as part of the European Union ESPRIT BRA Project REACT (6021). The work of M. Setälä was funded by The Academy of Finland, project REFORM.

References

[Boc88] Bochmann, G. v.: *Usage of Protocol Development Tools: The Results of a Survey.* Proceedings of the 7th International Symposium on Protocol Specification, Testing and Verification (1987), North-Holland 1988.

[BoB87] Bolognesi, T. & Brinksma, E.: *Introduction to the ISO Specification Language LOTOS.* Computer Networks and ISDN Systems 14 1987 pp. 25–59. Also in: The Formal Description Technique LOTOS, North-Holland 1989, pp. 23–73.

[BrR85] Brookes, S. D. & Roscoe, A. W.: *An Improved Failures Model for Communicating Processes.* Proceedings of the NSF-SERC Seminar on Concurrency, Lecture Notes in Computer Science 197, Springer-Verlag, 1985, pp. 281–305.

[ChK93] Cheung, S. C. & Kramer, J.: *Enhancing Compositional Reachability Analysis with Context Constraints.* Proceedings of the first ACM SIGSOFT Symposium on the Foundations of Software Engineering, ACM Software Engineering Notes, 18(5) 1993, pp. 115–125.

[CGR93] Craigen, D., Gerhart, S. & Ralston, T.: *Formal Methods Reality Check: Industrial Usage.* Proceedings of Formal Methods Europe '93, Lecture Notes in Computer Science 670, Springer-Verlag 1993, pp. 250–267.

[Elo94] Eloranta, J.: *Minimal Transition Systems with Respect to Divergence Preserving Behavioural Equivalences.* Doctoral thesis, University of Helsinki, Department of Computer Science, Report A-1994-1, Helsinki, Finland 1994, 162 p.

[Fel93] Feldbrugge, F.: *Petri Net Tool Overview 1992.* Advances in Petri Nets 1993, Lecture Notes in Computer Science 674, Springer-Verlag 1993, pp. 169–209.

[FSE94] Formal Systems (Europe) Ltd.: *Failures Divergence Refinement User Manual and Tutorial*, version 1.4 1994.

[GrS90] Graf, S. & Steffen, B.: *Compositional Minimization of Finite State Processes.* Computer-Aided Verification '90 (Proceedings of a workshop), AMS-ACM DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Vol. 3, American Mathematical Society 1991, pp. 57–73.

[Hoa85] Hoare, C. A. R.: *Communicating Sequential Processes.* Prentice-Hall 1985, 256 p.

[InP91] Inverardi, P. & Priami, C.: *Evaluation of Tools for the Analysis of Communicating Systems.* EATCS Bulletin 45, October 1991, pp. 158–185.

[ISO89] 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.

[KaV92] Kaivola, R. & Valmari, A.: *The Weakest Compositional Semantic Equivalence Preserving Nexttime-less Linear Temporal Logic.* Proceedings of CONCUR '92, Lecture Notes in Computer Science 630, Springer-Verlag 1992, pp. 207–221.

[MaV89] Madelaine, E. & Vergamini, D.: *AUTO: A Verification Tool for Distributed Systems Using Reduction of Finite Automata Networks.* Formal Description Techniques II (Proceedings of FORTE '89), North-Holland 1990, pp. 61–66.

[Mil89] Milner, R.: *Communication and Concurrency.* Prentice-Hall 1989, 260 p.

[Pet81] Peterson, G. L.: *Myths about the Mutual Exclusion Problem.* Information Processing Letters 12 (3) 1981, pp. 115–116.

[SeV94] Setälä, M. & Valmari, A.: *Validation and Verification with Weak Process Semantics.* Proceedings of Nordic Seminar on Dependable Computing Systems 1994, Lyngby, Denmark, August 1994, pp. 15–26.

[VaT91] Valmari, A. & Tienari, M.: *An Improved Failures Equivalence for Finite-State Systems with a Reduction Algorithm.* Protocol Specification, Testing and Verification XI (Proceedings of PSTV '91), North-Holland 1991, pp. 3–18.

[Val93] Valmari, A.: *Compositional State Space Generation.* Advances in Petri Nets 1993, Lecture Notes in Computer Science 674, Springer-Verlag 1993, pp. 427–457. (Earlier version in Proceedings of the 11th International Conference on Application and Theory of Petri Nets, Paris, France 1990, pp. 43–62.)

[VKCL93] Valmari, A., Kemppainen, J., Clegg, M. & Levanto, M.: *Putting Advanced Reachability Analysis Techniques Together: the "ARA" Tool.* Proceedings of Formal Methods Europe '93, Lecture Notes in Computer Science 670, Springer-Verlag 1993, pp. 597–616.

[Val94] Valmari, A.: *State of the Art Report: Stubborn Sets.* Petri Net Newsletter 46, April 1994, pp. 6-14.

[Val95] Valmari, A.: *Failure-based Equivalences Are Faster Than Many Believe.* Structures in Concurrency Theory, Proceedings, Berlin, Germany, May 1995, Springer-Verlag "Workshops in Computing" series 1995, pp. 326–340.

[VaT95] Valmari, A. & Tienari, M.: *Compositional Failure-based Semantic Models for Basic LOTOS.* Formal Aspects of Computing (1995) 7: 440–468.

[VKS96] Valmari, A., Karsisto, K. & Setälä, M.: *Visualisation of Reduced Abstracted Behaviour as a Design Tool.* To appear in Proceedings of Fourth Euromicro Workshop on Parallel and Distributed Processing, Braga, Portugal, Jan. 1996, IEEE publ., 8 p.

[WoG93] Wolper, P. & Godefroid, P.: *Partial-Order Methods for Temporal Verification.* Proceedings of CONCUR '93, Lecture Notes in Computer Science 715, Springer-Verlag 1993, pp. 233–246.