Merging State-Based and Action-Based Verification

Henri Hansen, Heikki Virtanen, Antti Valmari
Tampere University of Technology, Institute of Software Systems
PO Box 553, FIN-33101 Tampere, FINLAND
{hansen,hvi,ava}@cs.tut.fi

Abstract

A formalism is presented that is intended to combine basic properties of both state-based and action-based verification. In state-based verification the behaviour of the system is described in terms of the properties of its states, whereas action-based methods concentrate on transitions between states. A typical state-based approach consists of representing requirements as temporal logic formulae, and model-checking the state space of the system against them. Action-based verification often consists of comparing systems according to some equivalence or preorder relation. We add state propositions to a typical process-algebraic action framework. Values of state propositions are propagated through process-algebraic compositions and reductions by augmenting actions with changes of proposition values. A modified parallel composition operator is used for synchronisation of processes and handling of state propositions. Efficient on-the-fly verification is obtained with four kinds of rejection conditions. The formalism is implemented in a new verification tool TVT.

1. Introduction

Formal verification of reactive and concurrent systems may help to ease and focus the reliable design and testing of systems of increasing complexity. Verification has been approached through theorem proving and state space methods. The formalisms employed in state space methods may be roughly divided into two categories, state-based and action-based formalisms. State-based methods include model checking of many temporal logics that use Kripke structures and similar models [3, 12] and action-based methods include many process algebraic methods [13, 15], where Labelled Transition Systems (LTSs) are used as a model. State-based formalisms talk about properties of the global state of the system, whereas action-based formalisms talk about how the components of a system interact. This is why some properties may be harder to express using just one approach.

It is often inconvenient to build a model and write specifications formally, if a unified approach is lacking. In this article, we define a formalism called the *labelled state transition system* or LSTS, which combines LTSs with atomic state propositions and provides a way to benefit from both state-based and action-based models.

A similar model has been used in [8], but there the state propositions have an effect on synchronisation, which we explicitly wish to avoid. The fact that propositions do not affect synchronisation makes it possible to use state-based and action-based methods separately. Process algebraic methods can be fully used for describing the components and their interaction and state-propositions preserve the information of the states of the components. This allows us to express and verify safety and liveness properties as presented in the example of Section 6.

We extend to LSTSs the methods used to compositionally construct LTS models to provide a theoretical basis for on-the-fly verification. Some effects of adding state propositions on semantic models of processes, CFFD (chaos free failures divergence) in particular, are also covered briefly. We chose CFFD because it is the weakest congruence that preserves deadlocks and all the properties that can be expressed with next-state free LTL [16].

For most of what we cover here, LSTSs and their parallel composition have already been implemented in the TVT toolset created by the Verification Algorithm Group [1] at the Institute of Software Systems of Tampere University of Technology in collaboration with Nokia and other funders.

The theoretical background of LSTSs is given in Section 2 and some semantic properties of LSTSs are defined in Section 4. Models can be constructed compositionally using a generalised parallel composition operation, which we define in Section 3. We show how state propositions may be used to express illegal behaviour that can be checked on the fly. The on-the-fly usage of propositional rules is discussed in Section 5. We use a token ring mutual exclusion protocol as an example. We introduce some errors into it, and show how they are detected in Section 6.

2. LTSs and LSTSs

Various practically equivalent definitions of labelled transition systems may be encountered in the field [17, 15, 2]. To avoid confusion, we explicitly give the one we are using.

Definition 1 (Labelled transition system) *A* labelled transition system*, abbreviated* LTS*, is a four-tuple* $(S, \Sigma, \Delta, \hat{s})$*, where* S *is the set of* states*,* Σ *is the set of* visible actions*, also known as the* alphabet*, and* invisible action $\tau \notin \Sigma$*.* $\Delta \subseteq S \times (\Sigma \cup \{\tau\}) \times S$ *is the set of* transitions*, and* $\hat{s} \in S$ *is the* initial state*.*

For convenience, the set $\Sigma \cup \{\tau\}$ is denoted by Σ_τ. The following shorthand notation has traditionally been used for sequences of transitions, or executions.

Let $u, u_1, \ldots, u_n \in \Sigma_\tau$ and $a_1, \ldots, a_n \in \Sigma$.

- By $s \;{-}u{\rightarrow}\; s'$ we mean $(s, u, s') \in \Delta$.
- By $s \;{-}u{\rightarrow}$ we mean $\exists s' : s \;{-}u{\rightarrow}\; s'$.
- By $s \;{-}u_1 \cdots u_n{\rightarrow}\; s'$ we mean $\exists s_0, \ldots, s_n :$ $s_0 = s \wedge s_n = s' \wedge s_0 \;{-}u_1{\rightarrow}\; s_1 \;{-}u_2{\rightarrow}\; \cdots \;{-}u_n{\rightarrow}\; s_n$.
- By $s \;{=}\varepsilon{\Rightarrow}\; s'$ we mean $\exists s_0, \ldots, s_n : s_0 = s \wedge s_n = s'$ $\wedge\, s_0 \;{-}\tau{\rightarrow}\; s_1 \;{-}\tau{\rightarrow}\; \cdots \;{-}\tau{\rightarrow}\; s_n$.
- By $s \;{=}a_1 a_2 \cdots a_n{\Rightarrow}\; s'$ we mean $\exists s_0, \ldots, s_n; s'_0, \ldots s'_n : s = s_0$ $\wedge\, s_0 \;{=}\varepsilon{\Rightarrow}\; s'_0 \;{-}a_1{\rightarrow}\; s_1 \;{=}\varepsilon{\Rightarrow}\; s'_1 \;{-}a_2{\rightarrow}\; \cdots \;{-}a_n{\rightarrow}\; s_n$ $\wedge\, s_n \;{=}\varepsilon{\Rightarrow}\; s'_n \wedge s'_n = s'$.

For infinite sequences the notations are extended in the obvious way, into $s \;{-}u_1 u_2 \cdots{\rightarrow}$ and $s \;{=}a_1 a_2 \cdots{\Rightarrow}$.

Definition 2 (Labelled state transition system) *A* labelled state transition system *or an LSTS is the tuple* $(S, \Sigma, \Delta, \hat{s}; \Pi, \Upsilon, val)$*, where LTS* $(S, \Sigma, \Delta, \hat{s})$ *is augmented with the set of* propositions Π*,* evaluation function $val : S \rightarrow 2^\Pi$*, and the set* Υ *of* permanent *propositions for which* $\Upsilon \subseteq \Pi$*.*

Figure 1 presents a simple example of an LSTS. It is a model of a client in a system that we will present in more detail in Section 6. States are usually nameless, but here they are numbered from 1 to 6 for clarity. The initial state $\hat{s}$ is the state number 1 and it is marked with a small arrow. The alphabet Σ is the set $\{req, gra, rel, bye\}$ and transitions Δ is presented with labelled arrows between states.

The propositions $\Pi = \{W, A, B\}$ and the values of the function val are shown below the state numbers. None of the propositions are permanent and so the set Υ is empty.

Many state-based verification approaches contain state variables, which may change their values when transitions are taken. There may be *guards* on transitions that define conditions under which they are taken, and *post-conditions* that allow assigning values to variables. The LSTS formalism is much weaker in this respect. It does not allow modelling of variables or guards with the state propositions. To some extent this is not a restriction. LSTSs with local variables can be compiled into LSTSs in the sense of Definition 2 with an unfolding construction similarly to coloured Petri nets [7]. Indeed, the input language of the TVT tool contains local variables, and TVT contains a compiler from it to LSTSs.

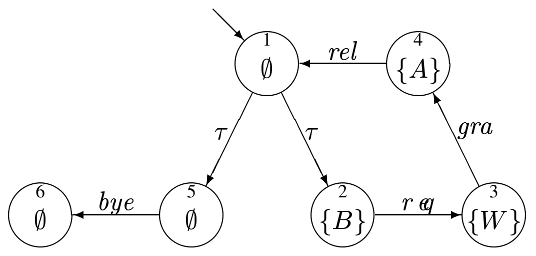

Figure 1. An example LSTS: Client in a simplified token-ring system.

What is more important is that, unlike in [8], LSTSs cannot test each others' state propositions. This implies that propositions can only be used for verification, for providing information to the designer of the system. Communication between the components of the system takes place via named actions. When shared variables are needed, they may be modelled as LSTSs in their own right. Such modelling is clumsy with traditional parallel composition operators, but not with the operator that we will define in Section 3.

To simplify the presentation of the abstract semantics of LSTSs, we encode the state propositions via an attachment to actions. This will make it possible to generalise certain process-algebraic methods of analysis and reduction from LTSs to LSTSs. In the encoding, the distinction between visible and invisible actions will be replaced by three categories: named actions, unnamed visible actions and invisible actions. We will also have to redefine the $-\cdots\rightarrow$ and $=\cdots\Rightarrow$ notations. Let the notation $A \oplus B$ denote the symmetric difference $(A - B) \cup (B - A)$. We define the following:

- If $s, s' \in S$, $u \in \Sigma_\tau$ and $P \subseteq \Pi$ then $s \;{-}\langle u, P\rangle{\rightarrow} s'$ means $(s, u, s') \in \Delta \wedge val(s') \oplus val(s) = P$.
- The elements of $\Sigma_\tau \times 2^\Pi$ are called *actions*.
- $\langle \tau, \emptyset \rangle$ is called the *invisible action*. All other actions are called *visible*.
- Actions $\langle \tau, P \rangle$, where $P \subseteq \Pi$, are called *unnamed actions*. All other actions are called *named*.

- $s = \varepsilon \Rightarrow s'$ means $\exists s_0, \ldots, s_n : s_0 = s \wedge s_n = s' \wedge s_0 -\langle \tau, \emptyset \rangle \rightarrow s_1 -\langle \tau, \emptyset \rangle \rightarrow \cdots -\langle \tau, \emptyset \rangle \rightarrow s_n$.
- $s = \langle a, P \rangle \Rightarrow s'$ means $(a \neq \tau \vee P \neq \emptyset) \wedge \exists s_1, s_2 : s = \varepsilon \Rightarrow s_1 \wedge s_1 - \langle a, P \rangle \rightarrow s_2 \wedge s_2 = \varepsilon \Rightarrow s'$. This notation thus assumes that the action is visible.
- The case of $= \langle a_1, P_1 \rangle \cdots \langle a_n, P_n \rangle \Rightarrow$ is defined in the same way as for LTSs for both finite and infinite sequences.

3. Parallel composition

Practically all compositional methods of putting a system together rely on some concept of synchronisation. Making actions invisible or renaming them for synchronisation is achieved using the basic process algebraic operators of *hiding*, *renaming* and *multiple renaming* [15]. The parallel composition operator of this section handles synchronisation of named actions while unnamed actions are considered internal and do not synchronise and proposition are handled separately from actions.

The generalised parallel composition operator uses *synchronisation rules* to do not only synchronisation, but hiding and (multiple)renaming of actions as well. It is also possible to implement *restriction* [13] using it. Synchronisation rules are similar to synchronisation vectors in [2]. The marked difference is that a synchronisation rule also contains a label for the resulting transition. For the purposes of this article, the generalised parallel composition is the only type of operator we need. Its LTS version was defined in [10].

Definition 3 (Synchronisation rule) *Let $L_1, \ldots, L_n$ be LSTSs, with alphabets $\Sigma_1, \ldots, \Sigma_n$. We assume that "$-$" $\notin \Sigma_i$. Let Ψ be any set of symbols with $\tau \notin \Psi$. A* synchronisation rule *for $L_1, \ldots, L_n$ and Ψ is a $(n+1)$-dimensional vector, written as $\langle a_1, \ldots, a_n; a \rangle$, for which*

- $\forall i \in \{1, \ldots, n\} : a_i \in \Sigma_i \cup \{\text{"}-\text{"}\}$
- $\exists i \in \{1, \ldots, n\} : a_i \neq \text{"}-\text{"}$
- $a \in \Psi \cup \{\tau\}$

In a synchronisation rule, each a_i that is not "$-$" specifies an action of L_i that synchronises to produce the action a. If $a_i = \text{"}-\text{"}$ it means that L_i does not participate in the execution of a.

Synchronisation rules are not affected in any way by the state propositions of the components and vice versa. To talk about how the state propositions behave, we need *proposition rules*.

Definition 4 (Proposition rule) *Let $L_1, \ldots, L_n$ be LSTSs with proposition sets $\Pi_1, \ldots, \Pi_n$. Then a* proposition rule *for them is a pair (ϕ, π), where π is a name, and ϕ is a boolean expression formed with boolean variables of the form (p, i), where $p \in \Pi_i$, connectives $\{\vee, \wedge, \neg\}$ and constants* True *and* False*. In a proposition rule ϕ is called the* expression part *and π the* result.

For a collection of states $(s_1, \ldots, s_n)$ we assign truth values to each (p, i) as True *iff $p \in val_i(s_i)$ and* False *iff $p \notin val_i(s_i)$. If ϕ then evaluates as* True*, we write $(s_1, \ldots, s_n) \models \phi$.*

In an LSTS, there may be isolated states that can not be reached from the initial state by any execution. To restrict our investigations to states that are of interest, we define the following:

Definition 5 (Reachable part)
Let $L = (S, \Sigma, \Delta, \hat{s}; \Pi, \Upsilon, val)$ be an LSTS. The reachable part *of L is $repa(L) = (S', \Sigma, \Delta', \hat{s}; \Pi, \Upsilon, val')$, where:*

- $S' = \{ s \in S \mid \exists \sigma \in (\Sigma_\tau \times 2^\Pi)^\star : \hat{s} -\sigma\rightarrow s \}$,
- $\Delta' = \Delta \cap (S' \times \Sigma_\tau \times S')$,
- $val'(s) = val(s)$ *whenever* $s \in S'$.

And by definition, all other components remain the same.

Definition 6 (Generalised parallel composition)
Let $L_1, \ldots, L_n$ be LSTSs, with $L_i = (S_i, \Sigma_i, \Delta_i, \hat{s}_i; \Pi_i, \Upsilon_i, val_i)$. Let C be a set of synchronisation rules for these and a given set Ψ. Let Φ be a set of proposition rules for the LSTSs. Then the generalised parallel composition *$par_{C,\Phi}(L_1, \ldots, L_n)$ is the reachable part of the LSTS $L = (S, \Sigma, \Delta, \hat{s}; \Pi, \Upsilon, val)$, for which*

- $S = S_1 \times \cdots \times S_n$
- $\Sigma = \Psi$
- $\Delta_\tau = \{ ((s_1, \ldots, s_n), \tau, (s'_1, \ldots, s'_n)) \mid \exists i : (s_i, \tau, s'_i) \in \Delta_i \wedge \forall j \in \{1, \ldots, n\} : (j = i \vee s_j = s'_j) \}$
- $\Delta_\Sigma = \{ ((s_1, \ldots, s_n), a, (s'_1, \ldots, s'_n)) \mid \exists a_1, \ldots, a_n : \langle a_1, \ldots, a_n; a \rangle \in C \wedge \forall i \in \{1, \ldots, n\} : ((a_i \neq \text{"}-\text{"} \wedge (s_i, a_i, s'_i) \in \Delta_i) \vee (a_i = \text{"}-\text{"} \wedge s_i = s'_i)) \}$
- $\Delta = \Delta_\tau \cup \Delta_\Sigma$
- $\hat{s} = (\hat{s}_1, \ldots, \hat{s}_n)$
- $\Pi = \{ \pi \mid \exists \phi : (\phi, \pi) \in \Phi \} \cup \Upsilon_1 \cup \cdots \cup \Upsilon_n$
- $\Upsilon = \Upsilon_1 \cup \cdots \cup \Upsilon_n$

- $val((s_1,\ldots,s_n)) = \{\, \pi \in \Pi \mid \exists \phi : (\phi,\pi) \in \Phi \wedge (s_1,\ldots,s_n) \models \phi \,\} \cup (\Upsilon_1 \cap val_1(s_1)) \cup \cdots \cup (\Upsilon_n \cap val_n(s_n))$.

The synchronisation rules define which named actions synchronise and which actions they produce. Synchronisation may result in any action, even invisible. Thus, it implements also hiding and renaming. In fact, it can be shown that this parallel composition can do exactly the same transformations as the traditional parallel composition, hiding and multiple renaming combined, when applied to LTSs [11].

The values of state propositions are evaluated according to given proposition rules and they have no effect on synchronisation. Furthermore, the values of permanent propositions propagate to the result even in the absence of proposition rules. The same proposition may appear as the result of several rules and be an element of several Υ_i. In this case, the disjunction of the values is taken. If there is any disagreement on the kind of proposition between several definitions, persistence wins non-persistence.

The traditional parallel composition can be implemented by choosing $\Psi = \Sigma_1 \cup \cdots \cup \Sigma_n$ and generating synchronisation rules $\langle a_1,\ldots,a_n; a\rangle$ for every action $a \in \Psi$ by setting $a_i = a$ if $a \in \Sigma_i$ and $a_i =$ "$-$" otherwise.

4. CFFD-semantics of LSTSs

To talk about behaviours of systems in an abstract way, we use the concept of semantics. A semantic model induces an equivalence of systems that equates systems that we deem to have the same kind of behaviour. The CFFD semantics model we present is known to be the weakest compositional semantics to preserve all the failures and divergences of a system as well as properties that can be expressed in linear temporal logic without the next-state operator [16].

The sets of traces (finite and infinite), divergence traces, stable failures, stability of given states and stability of the whole system are properties of given LSTSs. Many different semantic models can be defined using these sets, e.g. CSP [6], NDFD [9] or CFFD. Let L be an LSTS and $s \in S$. Then we define the following:

- The set of (finite) *traces*,
  $Tr(L) = \{\, \sigma \in (\Sigma_\tau \times 2^\Pi)^\star \mid \exists s : \hat{s} =\sigma\Rightarrow s \,\}$
- The set of *infinite traces*,
  $Inftr(L) = \{\, \sigma \in (\Sigma_\tau \times 2^\Pi)^\omega \mid \hat{s} =\sigma\Rightarrow \,\}$
- A state *s diverges*, $diverges(s) \Leftrightarrow s -\langle \tau,\emptyset\rangle^\omega\rightarrow$
- The set of *divergence traces*, $Divtr(L) =$
  $\{\, \sigma \in (\Sigma_\tau \times 2^\Pi)^\star \mid \exists s : \hat{s} =\sigma\Rightarrow s \wedge diverges(s) \,\}$
- A state is *stable*, $Stable(s) \Leftrightarrow \forall s' \in S : (s,\tau,s') \notin \Delta$
- The LSTS is *stable*, $Stable(L) \Leftrightarrow Stable(\hat{s})$
- The set of *stable failures*,
  $Sfail(L) = \{\, (\sigma, A) \in (\Sigma_\tau \times 2^\Pi)^\star \times 2^\Sigma \mid \exists s : Stable(s) \wedge \hat{s} =\sigma\Rightarrow s \wedge \forall a \in A \cup \{\tau\}, s' \in S : (s,a,s') \notin \Delta \,\}$
- The set of *initial propositions* $\hat{\Pi}(L) = val(\hat{s})$

Definition 7 (CFFD-equivalence) *Let L be a LSTS. The* CFFD-semantics *of L is*

$$(\Sigma, \Pi, \Upsilon, Sfail(L), Divtr(L), Inftr(L), \hat{\Pi}(L))$$

Two LSTSs, L_1 and L_2 are CFFD-equivalent*, denoted $L_1 \simeq_{\mathsf{CFFD}} L_2$ iff $\Sigma_1 = \Sigma_2 \wedge \Pi_1 = \Pi_2$
$\wedge\, Sfail(L_1) = Sfail(L_2) \wedge Divtr(L_1) = Divtr(L_2)$
$\wedge\, Inftr(L_1) = Inftr(L_2) \wedge \hat{\Pi}(L_1) = \hat{\Pi}(L_2)$.*

Definition 7 differs from the CFFD-semantics of LTSs in several respects. First, Π and Υ are added to the "static" part of the semantics. Second, traces now contain information on values of state propositions. If $\hat{s} =\sigma\Rightarrow s$, then $val(s)$ can be reconstructed from $\hat{\Pi}(L)$ and σ. In other words, changes of state propositions are visible. This suffices for verifying stuttering-insensitive linear temporal logic properties [12].

However, state propositions do not affect the refusal sets (the A) in stable failures. Furthermore states that have outgoing τ-transitions are not considered stable, even if the transitions affect values of propositions. These are because propositions do not affect synchronisation. As a consequence, acceptance-graph-based algorithms such as [14, 18] have less information to process and yield smaller results.

It is also worth mentioning that when dealing with finite LTSs, the set of traces can be deduced from the sets of divergence traces and stable failures and infinite traces are needed only when dealing with infinite LTSs. With LSTSs this is not the case, because an infinite behaviour may occur that leaves no information in either stable failures or divergence traces. For example, in Figure 2, states are labelled with propositions P and Q. The actions after $\langle a,\emptyset\rangle$ can not be deduced from stable failures and divergence traces alone, since there are no divergence traces and neither of the two rightmost states are stable. To solve this, we have to include the infinite traces. If we wished to restrict ourselves to finite

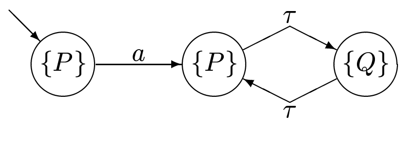

Figure 2. An example of need for infinite traces

LSTSs, the set of finite traces could be used instead of the set of infinite traces.

In process algebras, it is imperative, that the equivalence defined by the semantic model goes together well with the process-algebraic operators we wish to use, i.e. it should be a *congruence*.

Definition 8 (Congruence) *Let* $L_1, \ldots, L_n$ *and* $M_1, \ldots, M_n$ *be LSTSs. An equivalence relation* $\simeq_{\mathsf{X}}$ *is a* congruence *with respect to an operator* f*, iff*

$$\begin{aligned} & L_1 \simeq_{\mathsf{X}} M_1 \wedge \cdots \wedge L_n \simeq_{\mathsf{X}} M_n \\ \Rightarrow \quad & f(L_1, \ldots, L_n) \simeq_{\mathsf{X}} f(M_1, \ldots, M_n) \end{aligned}$$

If our semantic model is a congruence with respect to the operators we use, we may replace any part of the system with an equivalent one without disturbing the equivalence of the whole system. This is imperative if we analyse our system compositionally, as we will in the example later on.

Theorem 1 *CFFD-equivalence is a congruence with respect to the generalised parallel composition.*

The proof of congruence when dealing with LTSs is presented in [11]. For LSTS there is no real difference, since the propositions have no effect on synchronisation and the values of propositions can be deduced from the traces.

5. On-the-fly verification with proposition rules

In addition to ordinary propositions, four different special names *Rej*, *Llrej*, *Dlrej* and *Infrej* can appear as the result of a proposition rule. The states, where expressions associated with *Rej*, *Llrej*, *Dlrej* or *Infrej* are true, are called *rejection states*.

Rejection states are used to define behaviours that are considered erroneous. The conditions in the rules that introduce them are checked on-the-fly while the parallel composition is being generated.

Let $L_1, \ldots, L_n$ be LSTSs, and their parallel composition $par_{C,\Phi}(L_1, \ldots, L_n)$. The resulting LSTS is denoted by $L = (S, \Sigma, \Delta, \hat{s}; \Pi, \Upsilon, val)$. Then the following checks are made:

- If $\exists s \in S : \exists \phi : (\phi, Rej) \in \Phi \wedge (s \models \phi) \wedge \exists \sigma \in (\Sigma_\tau \times 2^{\Pi})^* : \hat{s} =\sigma\Rightarrow s$, the system is considered erroneous, and σ is an illegal trace.

- If $\exists s \in S : \exists \phi : (\phi, Dlrej) \in \Phi \wedge (s \models \phi) \wedge \exists \sigma \in (\Sigma_\tau \times 2^{\Pi})^* : \hat{s} =\sigma\Rightarrow s \wedge \forall \langle a, P \rangle \in \Sigma_\tau \times 2^{\Pi} : \neg(s -\langle a, P \rangle \rightarrow)$, the system is erroneous and σ is an illegal deadlock trace.

- A rule for *Llrej* can only be of the form $((\pi, i), Llrej)$, where $\pi \in \Pi_i$. That is, it is only a proposition of one component. The system has σ as an illegal livelock trace, if and only if there is a cycle $s_1 -\langle a_2, P_2 \rangle \rightarrow s_2 -\langle a_3, P_3 \rangle \rightarrow \cdots -\langle a_n, P_n \rangle \rightarrow s_n -\langle a_1, P_1 \rangle \rightarrow s_1$ such that $\hat{s} =\sigma\Rightarrow s_1$, (π, i) evaluates to True in s_1, and the ith component process does not participate in the cycle.

- With *Infrej*, the requirements are is otherwise the same as for *Llrej*, but the ith component process must participate in the cycle. In this case the result of removing all invisible ($\langle \tau, \emptyset \rangle$) actions from $\sigma(\langle a_2, P_2 \rangle \cdots \langle a_n, P_n \rangle \langle a_1, P_1 \rangle)^\omega$ is an illegal infinite trace.

The conditions mean that whenever *Rej* is activated, some safety property has been violated, i.e., there is an illegal trace. Whenever *Dlrej* is activated, we check if the system is in a deadlock. If the left-hand side of a condition on *Llrej* holds for a component process, then the rest of the system is checked for infinite behaviour in which the activating process does not participate. *Infrej* is also activated by a single process, but it is used to check for infinite behaviour in which the activating process does participate infinitely often and which takes the component to such a state infinitely many times. This is essentially the same thing as acceptance by a Büchi automaton [5]. However, because of the way processes synchronise, only stuttering-insensitive properties can be checked in this way.

Algorithmically the detection of *Rej* and *Dlrej* is quite straight-forward: Check for reachability of the state. If it is a *Rej*-state, error is detected. For *Dlrej* one also has to check whether the state is a deadlock. Several relatively efficient methods exist for the detection of errors of the *Llrej* [5] and *Infrej* [4] type.

If no errors are found in the parallel composition, then these conditions will have no effect whatsoever on the resulting LSTS. If an error is found, the construction of the parallel composition is aborted and the error is reported.

6. Example of on-the-fly verification

To explain the different rejection states and usage of state propositions in the construction of a whole system, we give a simple example, a demand-driven token-ring system for mutual exclusion. To illustrate how errors are detected on the fly, we introduce some — admittedly artificial — errors to the system.

The idea of the token ring is that one single token is passed clockwise (*tr* for sending and *tl* for receiving) and demands for the token are passed counterclockwise (*dl* and *dr* for sending and receiving) among servers. Each server serves one client. If the client has requested access, the

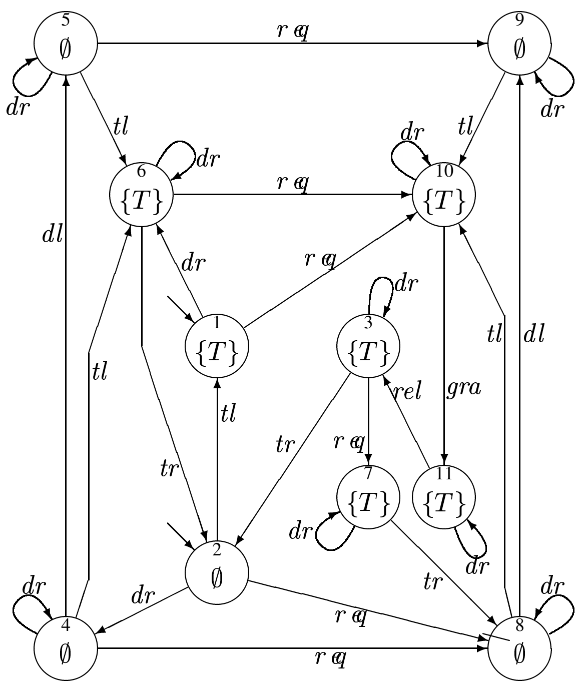

Figure 3. The LSTS presentation of a token ring server

server grants this, provided it has the token. Once the client has been served, the server immediately passes the token on. This ensures that the other clients get a chance of being served before the present client is served again. The token is also passed on, if there is no request and a demand for the token has been received.

Figure 3 shows the CFFD minimised LSTS model of a single server without errors. The state 1 represents the initial state of the server that originally has the token, and 2 the initial state of the other servers that do not have the token initially. The states in which a server has the token are labelled with a permanent proposition T (for token). It is possible to check that the token never disappears by checking that the permanent proposition T remains true in the every global state of the system. With pure action-based methods, this check requires complicated bookkeeping of the movements of the token.

In Figure 1, there is LSTS model of the simplified client. It repeatedly requests access or decides that it will not need the common resource anymore.

Let a token ring system have n client–server pairs making a total of $2n$ components in parallel composition. Let the servers be numbered with odd numbers, $2i-1$ for server L_i, and clients with even numbers, $2i$ for client M_i. A synchronisation rule is a $2n+1$ component vector of the form $\langle a_1, a_2, \ldots, a_{2n-1}, a_{2n}; x\rangle$, where x is an action of the resulting system, a_i is an action of the server or "–", if i is odd, and a_i is an action of the client or "–", if i is even. The actions of the result are not needed here, so for every rule $x = \tau$.

For controlled shutdown, each client M_i needs the rule $\langle$"–", …, "–", *bye*, "–", …, "–"; $\tau\rangle$ where a_j=*bye* if j=$2i$, or a_j="–" otherwise.

For each server–client pair L_i, M_i three synchronisation rules are needed:

1° For requests the rule $\langle$"–", …, *r eq*, *r eq*, "–", …; $\tau\rangle$, where a_j=*r eq*, if $j=2i-1 \vee j=2i$, or a_j="–" otherwise.

2° For granting access the rule $\langle$"–", …, *gra*, *gra*, …; $\tau\rangle$, where a_j=*gra*, if $j=2i-1 \vee j=2i$, or a_j="–" otherwise.

3° For releasing the rule $\langle$"–", …, *rel*, *r el*, "–", …; $\tau\rangle$, where a_j=*r el*, if $j=2i-1 \vee j=2i$, or a_j="–" otherwise.

For each pair of adjacent servers in the closed ring, L_i, L_j, where $j = (i \bmod n) + 1$ two rules are needed:

1° $\langle$"–", …, *tl*, "–", *tr*, "–", …; $\tau\rangle$, where a_k=*tl*, if k=$2i-1$, or a_k=*tr*, if k=$2j-1$, or a_k="–" otherwise, for passing the token, and

2° $\langle$"–", …, *dl*, "–", *dr*, "–", …; $\tau\rangle$, where a_k=*dl*, if k=$2i-1$, or a_k=*dr*, if $k = 2j-1$, or $a_j k$="–" otherwise, for transmitting the demands.

We are mainly interested in how the system as a whole behaves from the point of view of a client in two aspects. First, that only one client has access at the same time, and second, that the client eventually gets access if it has requested it. In order to check these, the client is augmented with propositions $\Pi = \{W, A, B\}$ for marking states where the client is waiting access (W), has access (A), or is about to request access (B).

Violations of the mutual exclusion property are detected by adding a proposition rule $((A, 2i) \wedge (A, 2j), Rej)$ for each pair M_i, M_j of different clients. The required liveness property is checked with two proposition rules for each single client M_i. The rule $((W, 2i), Llrej)$ ensures that the client does not starve, and the rule $((W, 2i), Dlrej)$ ensures that the system does not halt while client i has requested service.

This original model is correct, meaning that it does not activate any rejections. So, to demonstrate the detection of errors we add some. From the original system we device three different faulty ones by modifying some of the components. The added faults are shown in Figure 4.

In the first experiment we add an invisible transition from state 2 to 1 in one server. The faulty server may jump from 2 to 1 without the environment being able to prevent it. If the client served by the faulty server now makes a request, the server acts as if it has the token and grants access to its client. At the same time, a correctly working server has the real token and it may be serving its client. This error was detected, as it activated one of the rules of the form $((A, 2i) \wedge (A, 2j), Rej)$.

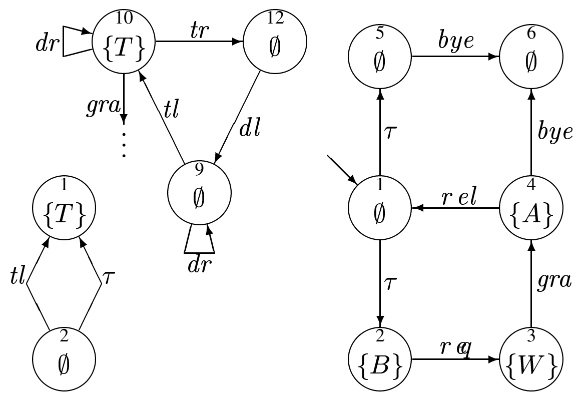

Figure 4. The erroneous parts of the broken components

In the second experiment we add a new state 12 and two transitions: tr from 10 to 12 and dl from 12 to 9 in one of the servers. Suppose the client served by the faulty server has made a request and the server has the token. The client is in the state where W holds, the server in state 10. Now the server may pass the token on by executing "tr", if its rightmost neighbour is ready to receive it. Then the server makes a demand for the token dl, and waits for the token to arrive. As the rest of the system is working correctly, the token will eventually arrive, and the server will return to state 10. This loop may be executed *ad infinitum* while the client waits in the very same state. The error is detected by the rule $((W, 2i), \mathit{Llrej})$.

In the third experiment we add a transition *bye* from the state of a client where A holds. The faulty client makes a request, which is eventually granted, as the rest of the system is working correctly. When in state A, the client is able to execute *bye* on its own. The server will never receive the release, so it will get stuck in state 11. Suppose for example that all the other clients have made requests. None of their servers have the token, so they will eventually all end up waiting for it in state 9. The system now deadlocks, and one of the clients will have the rule $((W, 2i), \mathit{Dlrej})$ detect this error.

We did the examples on the TVT-tool using four servers and clients. TVT has been published in open source under NOKOS-licence and it its available for download at [1].

Acknowledgements

The authors would like to thank Antti Kervinen for checking the examples. The TVT project has been funded in part by Nokia.

References

[1] http://www.cs.tut.fi/ohj/VARG/. Verification Algorithm Research Group home page.

[2] A. Arnold. *Finite Transition systems*. Prentice-Hall, 1994. 177 p.

[3] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999. 314 p.

[4] C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory-efficient algorithms for the verification of temporal properties. In *Formal Methods in System Design vol. 1*, pages 275–288, 1992.

[5] H. Hansen, W. Penczek, and A. Valmari. Stuttering insensitive automata for on-the-fly detection of livelock properties. In *FMICS'02*, 2002.

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

[7] K. Jensen. *Coloured Petri Nets: Basic Concepts*, volume 1. Springer-Verlag, 1992.

[8] R. Kaivola. *Equivalences, Preorders and Compositional Verification for Linear Time Temporal Logic and Concurrent Systems*. PhD thesis, Universty of Helsinki, 1996.

[9] R. Kaivola and A. Valmari. The weakest compositional semantic equivalence preserving nexttime-less linear temporal logic. In *Proceedings of CONCUR '92, Third International Conference on Concurrency Theory*, number 630 in Lecture Notes in Computer Science, pages 207–221. Springer-Verlag, 1992.

[10] K. Karsisto. A more flexible parallel composition operator. In *Proceedings of the Fifth Symposium on Programming Languages and Software Tools*, Series of Publications C, pages 13–23, Jyväskylä, June 1997. University of Helsinki, Department of Computer Science. Report C-1997-37.

[11] K. Karsisto. A new parallel composition operator for verification tools. PhD thesis manuscript, 116 p, 2002.

[12] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1992. 427 p.

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

[14] A. W. Roscoe. Model checkin CSP. In *A Classical Mind: Essays in Honour of C. A. R. Hoare*, pages 353–378. Prentice-Hall, 1994.

[15] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1998. 565 p.

[16] A. Valmari. A chaos-free failures-divergences semantics with applications to verification. In J. Davies, B. Roscoe, and J. Woodcock, editors, *Millenial Perspectives in Computer Science*, Cornerstones of Computing, pages 365–382. PALGRAVE, 2000.

[17] A. Valmari. Composition and abstraction. In F. Cassez, C. Jard., B. Rozoy, and M. Ryan, editors, *Modelling and Verification of Parallel Processes*, number 2067 in LNCS Tutorials, Lecture Notes in Computer Science, pages 58–99. Springer-Verlag, 2001.

[18] A. Valmari and M. Tienari. An improved failures equivalence for finite-state systems with a reduction algorithm. In *Proc. Protocol Specification, Testing and Verification XI*, pages 3–18. North Holland, 1991.