Formal Aspects of Computing (1998) 3: 1–000
© 1998 BCS

The SH-Verification Tool — Abstraction-Based Verification of Co-operating Systems

P. Ochsenschläger[a], J. Repp[a], R. Rieke[a] and U. Nitsche[b]

[a] GMD — German National Research Centre for Computer Science, Institute for Telecooperation Technology, Rheinstr. 75, D-64295 Darmstadt, Germany
email: {ochsenschlaeger,repp,rieke}@darmstadt.gmd.de
[b] Department of Electronics and Computer Science, University of Southampton, Highfield, Southampton, SO17 1BJ, U.K.
email: un@ecs.soton.ac.uk

Keywords: Simple language homomorphisms; Asynchronous product automata; Approximate satisfaction of safety and liveness properties; Model checking; Verification tools

Abstract. The sh-verification tool comprises computing abstractions of finite-state behaviour representations as well as automata and temporal logic based verification approaches. To be suitable for the verification of so called co-operating systems, a modified type of satisfaction relation (approximate satisfaction) is considered. Regarding abstraction, alphabetic language homomorphisms are used to compute abstract behaviours. To avoid loss of important information when moving to the abstract level, abstracting homomorphisms have to satisfy a certain property called simplicity on the concrete (i.e. not abstracted) behaviour. The well known state space explosion problem is tackled by a compositional method combined with a partial order method.

1. Introduction

The aim of the sh-verification tool (sh means simple homomorphisms, which will be explained below) is to support the verification of co-operating systems. By

Correspondence and offprint requests to: Peter Ochsenschläger, GMD — German National Research Centre for Computer Science, Institute for Telecooperation Technology, Rheinstr. 75, D-64295 Darmstadt, Germany email: ochsenschlaeger@darmstadt.gmd.de and Ulrich Nitsche, Department of Electronics and Computer Science, University of Southampton, Highfield, Southampton, SO17 1BJ, U.K., email: un@ecs.soton.ac.uk co-operating systems we mean distributed systems which are characterized by freedom of decision and loose coupling of their components. This causes a high degree of nondeterminism which is handled by our methods. Typical examples of co-operating systems are telephone systems, communication protocols, smartcard systems, electronic money, contract systems, etc.

In that context verification is the proof that system components work together in a desired manner. So the dynamic behaviour of the system has to be investigated. One usual approach is to start with a formal specification of the dynamic behaviour of the system which is represented by a *labelled transition system* (LTS), and then to prove properties of such an LTS. But for real life applications the corresponding LTS are often too complex to apply this naive approach.

In contrast to the immense number of transitions of such an LTS usually only a few characteristic actions of the system are of interest with respect to verification. So it is evident to define abstractions with respect to the actions of interest and to compute a representation of such an abstract behaviour, which usually is much smaller than the LTS of the specification. For such a small representation dynamic properties can be proven more efficiently. Now, under certain conditions, properties of the system specification can be deduced from properties of the abstract behaviour.

For such an approach the following questions have to be answered:

Question 1: What does it formally mean, that a system satisfies a property (especially in the context of co-operating systems)?

Question 2: How can we formally define abstractions?

Question 3: For what kind of abstractions is there a sufficiently strong relation between system properties and properties of the abstract behaviour?

Question 4: How can we compute a representation of the abstract behaviour efficiently?

The present article is an extended and completed version of [ORRN97].

2. Approximately Satisfied Properties

As a small but typical example to illustrate our answers to these questions, we consider a system that consists of a client and a server as its main components. The client sends requests to the server, expecting the server to produce particular results. Nevertheless, for some reasons, the server may not always respond a request by sending a result, but may, as well, reject a request. The main actions that are important with respect to the client's behaviour, are sending a request and receiving a result or rejection. These actions are depicted as REQ, RES, and REJ in Figure 1. We will regard the whole system running properly, if the client, at no time, is prohibited completely from receiving a result after having sent a request.
For the moment, we regard the server as a black box; i.e. we neither consider its internal structure nor look at its internal actions. Not caring about particular actions of a specification when regarding the specification's behaviour is *behaviour*

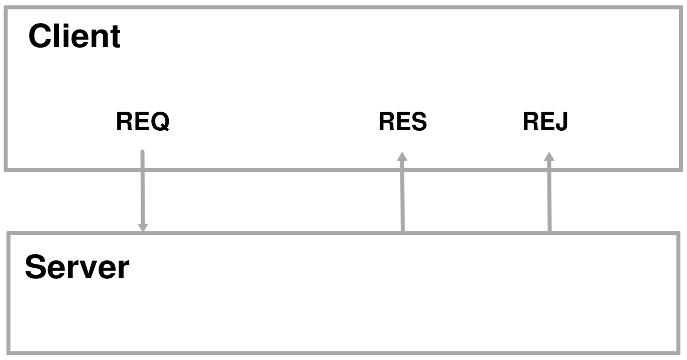

Fig. 1. Client Server Example

abstraction. If we define a suitable abstraction for the client/server system with respect to our correctness criterion, we only keep actions REQ, RES, and REJ visible, hiding all other actions.

To formalise behaviour abstraction we use terms of *formal language theory*. An LTS is completely determined by the set of its paths starting at the initial state. This set is a formal language, called the *local language* of the LTS [Eil74]. Its letters are the transitions (state, transition label, successor-state) of the LTS. Σ denotes the set of all transitions of the LTS. Consequently, there is a one-to-one correspondence between the LTS and its local language $L \subset \Sigma^*$, where Σ^* is the set of all sequences of elements of Σ including the empty sequence ϵ. Now behaviour abstraction can be formalized by *language homomorphisms*, more precisely by alphabetic language homomorphisms $h : \Sigma^* \to \Sigma'^*$ (**answer to question 2**). By these homomorphisms certain transitions are ignored and others are renamed, which may have the effect, that different transitions are identified with one another. A mapping $h : \Sigma^* \to \Sigma'^*$ is called a language homomorphism if $h(\epsilon) = \epsilon$ and $h(yz) = h(y)h(z)$ for each $y, z \in \Sigma^*$. It is called alphabetic, if $h(\Sigma) \subset \Sigma' \cup \{\epsilon\}$.

An automaton representation (*minimal automaton* [Eil74]) for the abstract behaviour of a specification (homomorphic image of the LTS's local language) can be computed by the sh-verification tool. Applying the abstraction described above to the concrete (i.e. not abstracted) behaviour of a specification of the client/server system leads to an automaton representation of abstract behaviour as presented in Figure 2. For this example $\Sigma' = \{REQ, RES, REJ\}$.

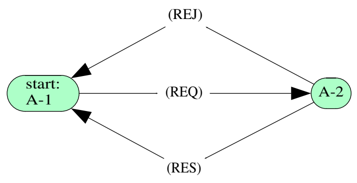

Fig. 2. Minimal Automaton.

The abstract behaviour obviously satisfies the correctness requirement mentioned above that, at no time, the client can be prohibited from receiving a result to a request. The usual concept of *linear satisfaction of properties* [AS85] (each infinite run of the system satisfies the property) is not suitable in this context since it considers also the extreme executions like "a request is always rejected". Obviously, the problem occurs because no fairness constraints are considered. We put a very abstract notion of fairness into the satisfaction relation for properties, which considers that "independent of a finitely long computation of the system, it is always possible that a request is responded by result". To formalise such "possibility properties", which are of interest when considering what we call co-operating systems, the notion of *approximate satisfaction* of properties is defined in [NO96] (**answer to question 1**):

Definition 2.1. *An automaton approximately satisfies a property if and only if each finite path of transitions of the automaton can be continued to an infinite path, which satisfies the property.*

As it is well known [AS85], system properties are divided into two types: *safety* (what happens is not wrong) and *liveness* properties (eventually something desired happens). For safety properties linear satisfaction and approximate satisfaction are equivalent [NO96]. Approximately satisfied liveness properties are liveness properties with respect to the universe of a system's behaviour. They are related to linear satisfaction of properties under strong fairness constraints for the sake of adding behaviour invariant state information [NW97].

3. Simple Homomorphisms as an Abstraction Concept

It is now the question of main interest, whether, by investigating an abstract behaviour, we may verify the correctness of the underlying concrete behaviour. We will answer this question positively, requiring a restriction of the permitted abstraction techniques. To deduce approximately satisfied properties of a specification from properties of its abstract behaviour an additional property of abstractions is required: called *simplicity of homomorphisms* on a specification [Och92, Och94b]. Simplicity of homomorphisms on specifications is a very technical condition concerning the possible continuations of finite behaviours.

Concerning abstractions $h : \Sigma^* \to \Sigma'^*$ the crucial point are the liveness properties of a Language $L \subset \Sigma^*$. To define simplicity formally we need $w^{-1}(L) = \{y \in \Sigma^* | wy \in L\}$, the *set of continuations* of a word w in a language L [Eil74]. These continuations in some sense "represent" the liveness properties of L. Generally $h(x^{-1}(L))$ is a (proper) subset of $h(x)^{-1}(h(L))$, but we want to have that $h(x^{-1}(L))$ "eventually" equals $h(x)^{-1}(h(L))$.

Definition 3.1. *A homomorphism h is called simple on L, if for each $x \in L$ there exists $w \in h(x)^{-1}(h(L))$ such that $w^{-1}(h(x^{-1}(L))) = (h(x)w)^{-1}(h(L))$.*

For regular languages simplicity of a homomorphism is a decidable property. Necessary and sufficient conditions for a homomorphism to be simple exist on the state graph level which are practically motivated and can be checked very efficiently. We will discuss this in more detail subsequently. The following theorem [NO96] shows that approximate satisfaction of properties and simplicity of homomorphisms exactly fit together for verifying co-operating systems (**answer to question 3**):

Theorem 3.2. *Simple homomorphisms define exactly the class of such Abstractions, for which holds that each property is approximately satisfied by the abstract behaviour if and only if the "corresponding" property is approximately satisfied by the concrete behaviour of the system.*

Formally, the "corresponding" property is expressed by the inverse image of the abstract property with respect to the homomorphism.

Our verification method, which is based on the very general notions of approximate satisfaction of properties and simple language homomorphisms, does not depend on a specific formal specification method. It can be applied to all specification techniques with an LTS-semantics.

To point out and motivate in more detail the necessity of considering approximate satisfaction of properties and of restricting suitable abstraction techniques to simple homomorphisms, we have to look more closely at the structure the server may have.

The server's answer (result or rejection) to a client's request may depend on whether a resource is available. If the resource is free, the server will respond a request by sending a result, if the resource is locked when the server is requested, the server will reject the request. Assuming that the server cannot control the resource, there is no guarantee at all that the resource is not locked all the time the client sends a request. Therefore, for some quite extreme computation scenarios, a request may always be rejected. So the best we can expect is that, in principle, there is always the possibility that a request is answered by eventually producing a result. This type of requirements is exactly captured by the definition of approximate satisfaction of properties. We revisit the correctness criterion for the client/server specification: the client is never prohibited completely from receiving a result after having sent a request.

If the resource behaves properly, i.e. it would change infinitely often from state locked to state free and vice versa in an infinite amount of time, the client/server specification will be correct with respect to the above requirement. Hence, since Figure 2 represents the abstract behaviour of the specification, the abstract as well as the concrete behaviour meet the correctness requirement. One may conjecture that the satisfaction of correctness criteria is preserved when moving from the abstract to the concrete behaviour.

Let us now consider a resource not showing a proper behaviour. A formal specification in terms of Petri nets is given in Section 7. For some reason, the resource may eventually be locked forever. Indeed, we consider now a resource that contains an error. If the resource vanishes forever, the client will never receive a result again. Thus this modified client/server specification does not meet the correctness requirement anymore. Regarding that the modified system may behave correct as well as after some time may behave incorrect, when coming to abstraction, the correct behaviour hides the incorrect one. This is, because the incorrect behaviour is a subset of the correct one, and therefore, when brought together by looking only at actions REQ, RES, and REJ, the abstract behaviour of the system is still represented in Figure 2. Here, the considered correctness requirement is not preserved when changing the viewpoint from the abstract to the concrete behaviour.

The problem of a correct subbehaviour hiding an incorrect subbehaviour under abstraction can be most easily explained when looking at the strongly connected components of the LTS that represents the concrete behaviour. For the incorrect client/server specification, Figure 3 shows the LTS representing the behaviour of the client/server specification such that the strongly connected components of the LTS are differently marked. When drawing our attention to abstraction, the strongly connected component that contains the initial state corresponds to a correct abstract subbehaviour of the specification. The second strongly connected component, which is a bottom component (no outgoing transitions from this component), corresponds to an incorrect abstract subbehaviour. When computing the minimal representation for the abstract behaviour (it is naturally the *minimal* representation of the abstract behaviour that we are interested in the abstraction framework), the two strongly connected components are “shuffled“ in such a way that the resulting LTS shows the maximal possible behaviour with respect to the shuffling. Hence the first component covers the second one and the incorrect behaviour is hidden.

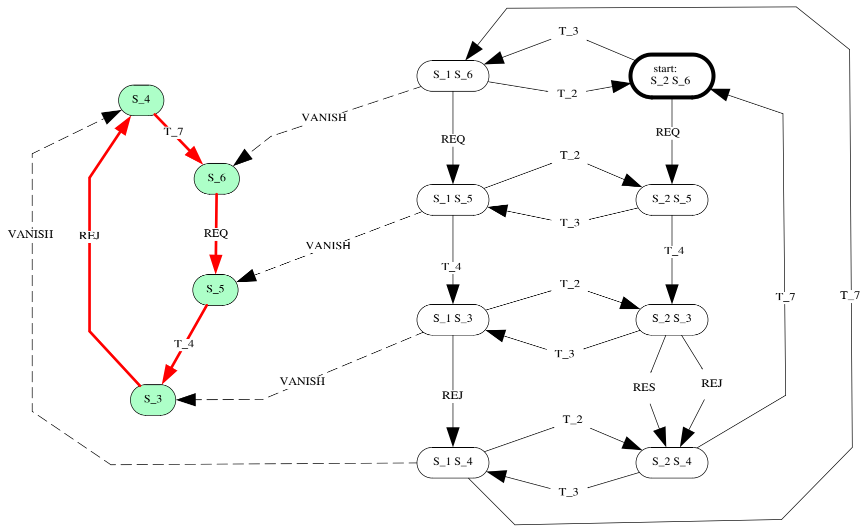

Fig. 3. LTS

In the considered example, the so far discussed problem can be detected easily when computing a graph representation of the strongly connected components of the concrete LTS, called the *component graph*. Each node in this graph representation is a strongly connected component and we have an arc from one node to another, if there exists a transition to move from one strongly connected component to the other. We label these nodes with abstract actions that can occur in the corresponding strongly connected components with respect to the defined abstraction.

The component graph of an LTS can be computed by the sh-verification tool and Figure 4 shows this graph representation for our example. Realizing that in the second node, which is a leaf because it represents a strongly connected bottom component, the action RES is missing compared to the first node. We obtain that in the strongly connected bottom component a request cannot be answered with a result anymore, which reveals exactly the violation of the requirement that we considered. There are reachable states wherefrom REQ can never be responded with RES.

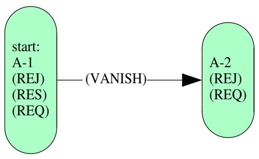

Fig. 4. Connected Components

It is rather obvious that we can only be interested in abstractions where hiding of an incorrect subbehaviour by a correct one cannot occur. For this purpose, simplicity of homomorphisms on behaviours has been defined. And, indeed, simplicity of homomorphisms is a necessary and sufficient condition for the preservation of approximately satisfied properties when changing the point of view from the abstract to the concrete behaviour.

Inspecting the strongly connected components of an LTS simplicity of an abstraction can be investigated. In [Och92, Och94b] the following sufficient condition for simplicity has been proven:

Theorem 3.3. *Let L be a Language recognized by a finite automaton $\mathcal{A}$ and let h be a homomorphism on L. If for each $x \in L$ there exists $y \in x^{-1}(L)$ leading to a dead component of $\mathcal{A}$, such that each $z \in L$ with $h(z) = h(xy)$ leads to the same dead component, then h is simple on L. This condition is satisfied for example, if each dead component contains a label a of an edge with $h(a) \neq \epsilon$, such that no edge exists outside of this component, whose label has the same image $h(a)$. If $\mathcal{A}$ is strongly connected, then each homomorphism is simple on L.*

To prove non-simplicity a necessary condition for simplicity is needed.
If $h(x^{-1}(L)) = \{\epsilon\}$ for a homomorphism $h : \Sigma^* \to \Sigma'^*, L \subset \Sigma^*$ and $x \in L$ then h is simple on L in x only if $h(x)^{-1}(h(L)) = \{\epsilon\}$. In earlier papers [Och88, Och90, Och91b] this situation has been formalized by so called deadlock languages. They consider abstract behaviours leading to states where no visible (under the abstraction) continuations exist.

Definition 3.4. *The deadlock language DL of a language L with respect to a homomorphism h is defined by $DL = \{u \in h(L) \mid \text{there exist } x \in L \text{ with } u = h(x) \text{ and } h(x^{-1}(L)) = \{\epsilon\}\}$.*

The minimal automaton of the deadlock language is called *deadlock automaton.* It can be computed by the sh-verification tool. If in our erroneous example all but action RES are hidden by a homomorphism $t : \Sigma^* \to \Sigma''^*$, with $\Sigma'' = \{RES\}$, the corresponding deadlock automaton, as well as the minimal automaton of $t(L)$, is shown in Figure 5. The deadlock automaton of the correct example is empty.
To formulate a necessary condition for simplicity using deadlock languages the notion *termination language* is needed:

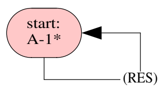

Fig. 5. Deadlock Automaton

Definition 3.5. *The termination language TL of a language L with respect to a homomorphism h is defined by* $TL = \{u \in h(L) | u^{-1}(h(L)) = \{\epsilon\}\}$.

It is easy to see that generally $TL \subset DL$ and that $TL = DL$ if the homomorphism h is simple on L [Och92, Och94b].

Concerning the homomorphism t Figure 5 shows that $TL = \emptyset \neq DL$. So t is not simple on L. To apply the necessary condition for simplicity to our homomorphism $h : \Sigma^* \to \Sigma'^*$ with $\Sigma' = \{REQ, RES, REJ\}$ we have to consider compositions of homomorphisms. Let $f : \Sigma_1^* \to \Sigma_2^*$ and $g : \Sigma_2^* \to \Sigma_3^*$ be mappings. The *composition* $g \circ f : \Sigma_1^* \to \Sigma_3^*$ is defined by $(g \circ f)(x) = g(f(x))$ for each $x \in \Sigma_1^*$. If g and f are homomorphisms then $g \circ f$ is a homomorphism too. The following theorems express the compatibility of simplicity with composition of homomorphisms [Och92, Och94b].

Theorem 3.6. *If f is simple on $L \subset \Sigma_1^*$ and g is simple on $f(L) \subset \Sigma_2^*$ then $g \circ f$ is simple on L.*

Theorem 3.7. *If $g \circ f$ is simple on $L \subset \Sigma_1^*$ then g is simple on $f(L)$.*

Considering the homomorphism $t' : \Sigma'^* \to \Sigma''^*$, defined by $t'(RES) = RES$ and $t'(X) = \epsilon$ for $X \neq RES$, we have $t = t' \circ h$. Now t' is simple on $h(L)$ because the automaton in Figure 2 is strongly connected. By the above theorem simplicity of h on L would imply simplicity of t on L, which is not true. So h is not simple on L, and the defect of our erroneous specification can be detected by simplicity investigations of appropriate homomorphisms without using the complex decision algorithm for simplicity.

4. A Compositional Approach to Avoid State Space Explosion

Simple homomorphisms establish the coarsest, i.e. most abstract notion of system equivalence with respect to a given (abstract) requirement specification [NO96]. What still remains open is the question of how to construct an abstract behaviour to a given specification without an exhaustive construction of its state space.

To handle the well known state space explosion problem, *a compositional method* has been developed [Och94c, Och95, Och96] and implemented in the sh-verification tool. In case of well structured specifications, by applying a divide and conquer strategy this method allows to compute a representation of the abstract behaviour and to check simplicity of homomorphisms efficiently without having to compute the complex LTS of the complete specification (**answer to question 4**). This compositional method is combined with a *partial order method* based on *partially commutative languages* [Och97]. The main goal of our compositional method is to compute minimal automata of homomorphic images and to check simplicity of homomorphisms efficiently even in case of complex specifications. The fundamental idea is to embed each component of a structured system (X and Y in Figure 9) in a “simplified environment“ (Y’ and X’ in Figure 6), which shows at the interface an “equivalent behaviour“ compared to the rest of the system (shaded areas in Figure 6). This can be checked using special homomorphisms, called *boundary homomorphisms*. The complexity of this check is reduced by our partial order method.

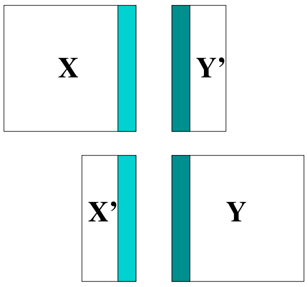

Fig. 6.

For each of these smaller systems minimal automata related to corresponding homomorphisms (which have to be finer than the boundary homomorphisms) are computed (Figure 7) and are composed (Figure 8) to obtain the desired automaton (Figure 9). This kind of composition is defined by the notion of *asynchronous product automata* and *co-operation products of formal languages*, a restricted kind of shuffle product. Simplicity of homomorphisms on co-operation products is guaranteed by a particular property of the boundary homomorphisms, which is called *co-operativity* For more details we refer to the next chapter and to [Och94c, Och95, Och96].

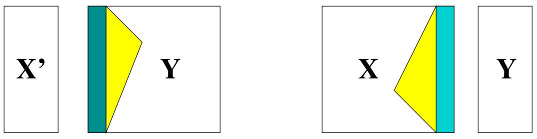

Fig. 7.

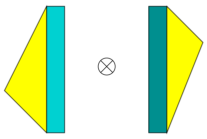

Fig. 8.

By that, compact representations of abstractions of system behaviour can be computed and simplicity of abstractions can be checked without investigating the complete behaviour of a complex system. In case of “well structured“ specifications this method causes considerable reductions of the state spaces. The

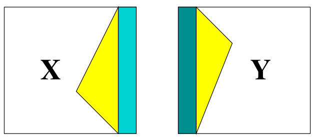

Fig. 9.

smaller systems with “simplified environments” avoid a lot of interleavings of actions (“state space explosion”), which are not relevant with respect to the considered abstraction but which are instrumental in the complex dynamics of the system.

This approach can also be used iteratively and allows induction proofs for systems with several identical components [Och96]. Using our compositional method a connection establishment and release protocol has been verified by investigating automata with about 100 states instead of 100000 states.

5. Asynchronous Product Automata

As a formal basis for our compositional approach as well as a formal specification language we now define the notion of asynchronous product automata (APA), a very general class of communicating automata. APA can be regarded as families of automata (*elementary automata*), whose sets of states are cartesian products and whose elementary automata are “glued together” by common components of these products.

Definition 5.1. An *asynchronous product automaton (APA)* consists of a family of *sets of state components* $(Z_s)_{s\in\mathcal{S}}$, a family of *elementary automata* $(\Phi_t, \Delta_t)_{t\in\mathcal{T}}$ and a *neighbourhood relation* $N : \mathcal{T} \to \wp(\mathcal{S})$ ($\wp(X)$ denotes the set of all subsets of X). For each elementary automaton (Φ_t, Δ_t)

- Φ_t is its *alphabet* and
- $\Delta_t \subset \mathsf{X}_{s\in N(t)}(Z_s) \times \Phi_t \times \mathsf{X}_{s\in N(t)}(Z_s)$ is its set of *state transition relation.*

To avoid pathological cases we assume $\mathcal{S} = \bigcup_{t\in\mathcal{T}}(N(t))$ and $N(t) \neq \emptyset$ for all $t \in \mathcal{T}$. The *states* of an APA are elements of $\mathsf{X}_{s\in\mathcal{S}}(Z_s)$ with the *initial state* $q_0 = (q_{0s})_{s\in\mathcal{S}} \in \mathsf{X}_{s\in\mathcal{S}}(Z_s)$. Formally an APA $\mathcal{A}$ is defined by a quadruple $\mathcal{A} = ((Z_s)_{s\in\mathcal{S}}, (\Phi_t, \Delta_t)_{t\in\mathcal{T}}, N, q_o)$.

“Dynamics” of APA are defined by “occurrences” of elementary automata. An elementary automaton (Φ_t, Δ_t) is *activated* in a state $p = (p_s)_{s\in\mathcal{S}} \in \mathsf{X}_{s\in\mathcal{S}}(Z_s)$ *with respect to an interpretation* $i \in \Phi_t$, if there exists $(q_s)_{s\in N(t)} \in \mathsf{X}_{s\in N(t)}(Z_s)$ with $((p_s)_{s\in N(t)}, i, (q_s)_{s\in N(t)}) \in \Delta_t$. An activated elementary automaton (Φ_t, Δ_t) may *occur* and generates a *successor state* $q = (q_r)_{r\in\mathcal{S}} \in \mathsf{X}_{s\in\mathcal{S}}$ with $q_r = p_r$ for $r \in \mathcal{S} \setminus N(t)$ and $((p_s)_{s\in N(t)}, i, (q_s)_{s\in N(t)}) \in \Delta_t$.

In this case $(p, (t, i), q)$ denotes the corresponding *occurrence step*. The occurrence of an elementary automaton changes the state components of its neighbourhood.

A sequence of the form $w = (q_1, (t_1, i_1), q_2)(q_2, (t_2, i_2), q_3)...(q_n, (t_n, i_n), q_{n+1})$ with $n \geq 1$ is called an *occurrence sequence*. If such an occurrence sequence exist, then we say that q_{n+1} is *reachable* from q_1. Additionally by definition each state is reachable from itself. $\mathcal{Q}$ (the *state space*) denotes the set of all states $q \in \times_{s \in \mathcal{S}}(Z_s)$ reachable from the initial state q_0 and Σ denotes the set of all occurrence steps, whose first component is an element of $\mathcal{Q}$. The set $L \subset \Sigma^*$ of all occurrence sequences starting with the initial state q_0 and containing the empty sequence ε is called the *occurrence language* of the corresponding APA. Σ can also be interpreted as the set of arcs of a directed graph, whose set of nodes is $\mathcal{Q}$ and whose arcs are labeled by pairs (t, i) with $t \in \mathcal{T}$ and $i \in \Phi_t$. This graph is called the *reachability graph* of the corresponding APA. By that occurrence sequences are paths in the reachability graph and the occurrence language is a regular language (local language), if the reachability graph is finite. The occurrence language as well as the reachability graph is a complete description of the dynamic behaviour of an APA.

As an example we give an APA specification of our incorrect client/server example. It is an APA representation of the Petri net in Figure 12, consisting of three elementary automata, $\mathcal{T} = \{C, S, R\}$, and four state components, $\mathcal{S} = \{CS, IS, SS, RS\}$. Figure 10 shows the neighbourhood relation N.

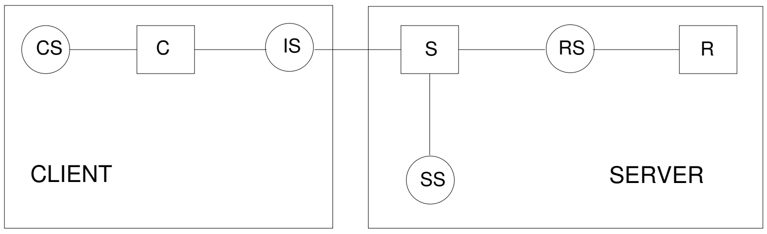

Fig. 10. Client / Server APA

State transitions of the elementary automaton C represent actions of the client. Correspondingly actions of the server and the resource manager are represented by state transitions of S and R respectively. CS and SS represent “internal” states of the client and the server. IS describes the the states of the client and server’s interface. RS represents both, internal and interface states related to the resource manager. Formally the APA is defined as follows:

state components:
$Z_{CS} = Z_{SS} = \{idle, active\}, Z_{IS} = \{emp, req, res - rej\}$,
$Z_{RS} = \{avail, navail, vanished\}$

imitial states:
$q_{0CS} = q_{0SS} = idle, q_{0IS} = emp, q_{0RS} = avail$.

alphabets:
$\Phi_C = \{REQ, T7\}$, $\Phi_S = \{RES, REJ, T4\}$, $\Phi_R = \{VANISH, T2, T3\}$.

state transition relations:
$$\Delta_C = \left\{ \begin{array}{l} ((idle, emp), REQ, (active, req)), \\ ((active, res - rej), T7, (idle, emp)) \end{array} \right\} \subset (Z_{CS} \times Z_{IS}) \times \Phi_C \times (Z_{CS} \times$$

Z_{IS}),

$$\Delta_S = \left\{ \begin{array}{l} ((idle, req, avail), T4, (active, emp, avail)), \\ ((idle, req, navail), T4, (active, emp, navail)), \\ ((idle, req, vanished), T4, (active, emp, vanished)), \\ ((active, emp, avail), RES, (idle, res-rej, avail)), \\ ((active, emp, avail), REJ, (idle, res-rej, avail)), \\ ((active, emp, navail), REJ, (idle, res-rej, navail)), \\ ((active, emp, vanished), REJ, (idle, res-rej, vanished)) \end{array} \right\} \subset (Z_{SS} \times$$

$Z_{IS} \times Z_{RS}) \times \Phi_S \times (Z_{SS} \times Z_{IS} \times Z_{RS})$,

$$\Delta_R = \{ \begin{array}{l} (avail, T3, navail), \\ (navail, T2, avail), \\ (navail, VANISH, vanished) \end{array} \} \subset Z_{RS} \times \Phi_R \times Z_{RS}.$$

State components correspond to markings of particular places of the Petri net, as for example $emp \in Z_{IS}$ denotes the empty marking of places S-4 and S-5 in Figure 12. The alphabets' elements correspond to the transitions of the Petri net. As the system is structured into three components given by the three elementary automata each alphabet represents the set of "local" actions of the corresponding component. Note that APA offer a very flexible concept for structuring specifications: decreasing the number of elementary automata increases the cardinality of the alphabets.

The reachability graph of our APA example is isomorphic to the LTS in Figure 3.

APA form a very general class of communicating automata. They are similar to asynchronous cellular automata introduced in [Zie89] and include different kinds of "simple" and "higher order" Petri nets as well as communicating automata as for example SDL [SSR89] or Estelle [BD87]. The above terminology is based on Petri nets: Elementary automata of APA correspond to transitions, state components correspond to places and states correspond to markings of places. By that the state transition relation is realized by the so called occurrence rule of a Petri net.

In terms of APA we now formulate our compositional method: A distributed system is an APA and the dynamical behaviour of the system is described by the occurrence language of that APA. A component (subsystem, module) of a system is defined by a subset $A \subset \mathcal{T}$. As we often consider the complement of A ("rest of the system" with respect to A) we use the abbreviation $\overline{A}$ for $\mathcal{T} \setminus A$. To consider states of an APA restricted to a subset $Y \subset \mathcal{S}$ we define $q|Y = (q_s)_{s \in Y} \in \mathsf{X}_{s \in Y}(Z_s)$ for a state $q = (q_s)_{s \in \mathcal{S}} \in \mathsf{X}_{s \in \mathcal{S}}(Z_s)$. Two special homomorphisms M_A (*module homomorphism*) and R_A (*boundary homomorphism*) on the occurrence language $L \subset \Sigma^*$ of an APA are used to express the behaviour of a component A of an APA and its behaviour at the interface to $\overline{A}$ respectively.

Notation. Let $RDA = N(A) \cap N(A')$. A homomorphism $M_A : \Sigma^* \to \Sigma^*_{MA}$ with $\Sigma_{MA} = M_A(\Sigma)$ is defined by

$$M_A((p,(t,i),q)) = \begin{cases} (p|N(A),(t,i),q|N(A)), & \text{if } t \in A \text{ and} \\ & N(t) \cap RDA \neq \emptyset, \\ (p|N(A) \setminus RDA,(t,i),q|N(A) \setminus RDA), & \text{if } t \in A \text{ and} \\ & N(t) \cap RDA = \emptyset, \\ \epsilon, & \text{if } t \in \overline{A}. \end{cases}$$

A homomorphism $R_A : \Sigma^* \to \Sigma^*_{RA}$ with $\Sigma_{RA} = R_A(\Sigma)$ is defined by

$$R_A((p,(t,i),q)) = \begin{cases} (p|RDA, q|RDA), & \text{if } t \in A \text{ and } N(t) \cap RDA \neq \emptyset, \\ \epsilon, & \text{if } t \in \overline{A} \text{ or } N(t) \cap RDA = \emptyset. \end{cases}$$

To compare homomorphisms with respect to their "degree of abstraction" we call a homomorphism $\phi : \Sigma^* \to \Delta^*$ *finer* than a homomorphism $\psi : \Sigma^* \to \Gamma^*$, if there exists a homomorphism $\nu : \Delta^* \to \Gamma^*$ with $\psi = \nu \circ \phi$. For this we use the notation $\phi\langle\psi$. In that case the homomorphic image $\phi(L)$ contains enough "information" to determine $\psi(L)$. As homomorphisms are used to describe abstractions we assume that they are alphabetic, i.e. $\phi(\Sigma) \subset \Delta \cup \{\epsilon\}$ for each homomorphism ϕ.

Notation. Two homomorphisms "acting" on disjoint components of an APA can be "combined" obtaining a new homomorphism: Let $A \subset \mathcal{T}$ and let $f : \Sigma^* \to \Phi^*$ as well as $g : \Sigma^* \to \Gamma^*$ be homomorphisms with $M_A\langle f$ as well as $M_{\overline{A}}\langle g$, then the homomorphism $f \oplus g : \Sigma^* \to (\Phi \cup \Gamma)^*$ is defined by $(f \oplus g)((p,(t,i),q)) = f((p,(t,i),q))$ if $t \in A$ and $(f \oplus g)((p,(t,i),q)) = g((p,(t,i),q))$ if $t \in \overline{A}$. $f \oplus g$ is called the *direct sum* of f and g. By the additional assumption $\Phi \cap \Gamma = \emptyset$ the direct sum of two homomorphisms is finer than both homomorphisms: There exist homomorphisms (projections) $\phi : (\Phi \cup \Gamma)^* \to \Phi$ and $\gamma : (\Phi \cup \Gamma)^* \to \Gamma^*$ with $f = \phi \circ (f \oplus g)$ and $g = \gamma \circ (f \oplus g)$.

The homomorphic image $(f \oplus g)(L)$ can be "constructed" using $f(L)$ and $g(L)$ if these two images contain enough information about the boundary behaviour of A and $\overline{A}$ respectively, i.e. that $f\langle R_A$ and $g\langle R_A$. To formulate a corresponding theorem we need some further technical notions:

Notation. If $f\langle R_A$ and $g\langle R_A$, then there exist homomorphisms $\rho_A : \Phi^\star \to \Sigma^\star_{RA}$ and $\rho_{\overline{A}} : \Gamma^\star \to \Sigma^\star_{R\overline{A}}$ with $R_A = \rho_A \circ f$ and $R_{\overline{A}} = \rho_{\overline{A}} \circ g$.
Let $\Sigma_R = \Sigma_{RA} \cup \Sigma_{R\overline{A}}$ and let $\rho : (\Phi \cup \Gamma)^* \to \Sigma^*_R$ be the homomorphism defined by: $\rho(x) = \rho_A(x)$ if $x \in \Phi$ and $\rho(x) = \rho_{\overline{A}}(x)$ if $x \in \Gamma$.
Let $RC = \{z \in \Sigma^*_R |$ if $z = (p,q)y$ with $(p,q) \in \Sigma_R$ and $y \in \Sigma^*_R$, then $p = q_0|RDA$, and if $z = x(p,q)(p',q')y$ with $(p,q),(p',q') \in \Sigma_R$ and $x,y \in \Sigma^*_R$, then $p' = q\}$.
If Σ_R is finite, then RC is a regular language (local language). The definition of RC depends on Σ_R. On account of $f\langle R_A$ and $g\langle R_{\overline{A}}$ this set can be determined using $f(L)$ and $g(L)$.

Under these assumptions the following holds:

Theorem 5.2. *Let $L \subset \Sigma^*$ be the occurrence language of an APA and $A \subset \mathcal{T}$. If $f : \Sigma^* \to \Phi^*$ and $g : \Sigma^* \to \Gamma^*$ are homomorphisms with $\Phi \cap \Gamma = \emptyset$ and $M_A\langle f\langle R_A$ as well as $M_{\overline{A}}\langle g\langle R_{\overline{A}}$, then $(f \oplus g)(L) = \phi^{-1}(f(L)) \cap \gamma^1(g(L)) \cap \rho^{-1}(RC)$.*

By this representation $(f \oplus g)(L)$ is a regular set if L is regular. In [Och96]

$\phi^{-1}(f(L)) \cap \gamma^1(g(L)) \cap \rho^{-1}(RC)$ is called the *cooperation product* of $f(L)$ and $g(L)$. It is easy to construct a finite automaton recognizing $(f \oplus g)(L)$ using corresponding automata for $f(L)$ and $g(L)$. Concerning simplicity of $f \oplus g$ we have

Theorem 5.3. *If f and g are simple on L by the same assumptions as in the above theorem, then $f \oplus g$ is simple on L too.*

The proofs of these two theorems as well as the proofs of the other theorems of this chapter can be found in [Och94c, Och96]. Essential to the statements of this chapter is "locality" of occurrence steps, i.e. that state changes only occur in the neighbourhood of the corresponding elementary automata. Therefore occurrence sequences may be "rearranged" without changing certain homomorphic images. Such "rearrangements" are the main proof techniques for these theorems.

The above theorems form one half of our compositional method. They show how abstractions of the behaviour of components of an APA can be "composed". But so far the representation of $(f \oplus g)(L)$ depends on $f(L)$ and $g(L)$. How can $f(L)$ and $g(L)$ be determined without using the (complex) occurrence language L of the complete system ? To achieve this we "embed" the components A and $\overline{A}$ in "simplified environments". Since a component of an APA can be viewed as an APA too we now have to define how two APA can be "composed".

The "gluing together" of elementary automata mentioned in the definition of APA can also be applied to arbitrary APA.

Definition 5.4. Let therefore $\mathcal{A}k = ((Zk_s)_{s \in \mathcal{S}k}, (\Phi k_t, \Delta k_t)_{t \in \mathcal{T}k}, Nk, qk_0)$ with $k \in \{1, 2\}$ be two APA with $\mathcal{T}1 \cap \mathcal{T}2 = \emptyset$ and $Z1_s = Z2_s$ as well as $q1_{0s} = q2_{0s}$ for all $s \in \mathcal{S}1 \cap \mathcal{S}2$. Now the *asynchronous product* $\mathcal{A}1 \otimes \mathcal{A}2$ is defined by
$\mathcal{A}1 \otimes \mathcal{A}2 = ((Z_s)_{s \in \mathcal{S}}, (\Phi k_t, \Delta k_t)_{t \in \mathcal{T}}, N, q_0)$ with $\mathcal{S} = \mathcal{S}1 \cup \mathcal{S}2, \mathcal{T} = \mathcal{T}1 \cup \mathcal{T}2$,
$Z_s = Zk_s$ and $q_{0s} = qk_{0s}$ for all $s \in \mathcal{S}k$ and $(\Phi_t, \Delta_t) = (\Phi k_t, \Delta k_t)$ for all $t \in \mathcal{T}k$ and $N(t) = Nk(t)$, where $k \in \{1, 2\}$. We also say that $\mathcal{A}1 \otimes \mathcal{A}2$ is constructed from $\mathcal{A}1$ and $\mathcal{A}2$ by *gluing together at the common state components* $\mathcal{S}1 \cap \mathcal{S}2$.

If A and $\overline{A}$ are complementary components of an APA, then this APA is the asynchronous product of A and $\overline{A}$. In terms of boundary behaviour the following theorem gives a sufficient condition to "embed" a component of an APA in different "environments" without changing its behaviour.

Let $\mathcal{X}$, $\mathcal{Y}'$, $\mathcal{X}'$ and $\mathcal{Y}$ be four APA, for which the asynchronous products $\mathcal{X} \otimes \mathcal{Y}'$, $\mathcal{X}' \otimes \mathcal{Y}$ and $\mathcal{X} \otimes \mathcal{Y}$ are defined. Let X, Y', X' and Y are the corresponding index sets of their elementary automata and SX, SY', SX' and SY the index sets of their state components. Additionally we assume that $SX \cap SY' = SX' \cap SY = SX \cap SY$. LXY', $LX'Y$ as well as LXY may denote the occurrence languages of $\mathcal{X} \otimes \mathcal{Y}'$, $\mathcal{X}' \otimes \mathcal{Y}$ and $\mathcal{X} \otimes \mathcal{Y}$ respectively.

Theorem 5.5. *If $R_X(LXY') = R_{X'}(LX'Y)$ and $R_{y'}(LXY') = R_Y(LX'Y)$, then $M_X(LXY) = M_X(LXY')$ and $M_Y(LXY) = M_Y(LX'Y)$.*

Let $\mathcal{X} \otimes \mathcal{Y}$ be a representation of the APA considered in the first two theorems, then $Y = \overline{X}$ and $L = LXY$. If Y' and X' are "simplified versions" of Y and X respectively then LXY' and $LX'Y$ can be "less complex" (with an essentially smaller state space) than L. Now applying the above theorem $f(L)$ and $g(L)$ can be determined using LXY' and $LX'Y$ instead of L because $M_X\langle f$ and $M_{\overline{X}}\langle g$.

To derive simplicity of homomorphisms on L from investigations on LXY' and $LX'Y$ we need a “cooperating property” of APA [Och96]:

Definition 5.6. Let $LXY' \subset \Xi^*$ be the occurrence language of $\mathcal{X} \otimes \mathcal{Y}'$ and let $f : \Xi^* \to \Phi^*$ be a homomorphism. $\mathcal{X}$ is called *cooperative in* $\mathcal{X} \otimes \mathcal{Y}'$ *with respect to* f, if $M_X\langle f$, $\Phi \cap M_{Y'}(\Xi) = \emptyset$ and if for each $x \in LXY'$ there exists a finite subset $H \subset (f \oplus M_{Y'})(x)^{-1}((f \oplus M_{Y'})(LXY'))$ with $\epsilon \in H$ such that for each $u \in H$ either
$u^{-1}((f \oplus M_{Y'})(x^{-1}(LXY'))) = ((f \oplus M_{Y'})(x)u)^{-1}((f \oplus M_{Y'})(LXY'))$
or
$u^{-1}(H) \cap M_{Y'}(\Xi) = ((f \oplus M_{Y'})(x)u)^{-1}((f \oplus M_{Y'})(LXY')) \cap M_{Y'}(\Xi)$ and
$u^{-1}(H) \cap \Phi \neq \emptyset$ if $((f \oplus M_{Y'})(x)u)^{-1}((f \oplus M_{Y'})(LXY')) \cap \Phi \neq \emptyset$.

In combination with the previous theorem the following two theorems [Och96] allow to derive simplicity of f and g on $L = LXY$ from investigations on LXY' and $LX'Y$.

Theorem 5.7. *Let r and s be homomorphisms with $M_X\langle r\langle R_X$, $M_Y\langle s\langle R_Y$, $r(LXY') = M_{X'}(LX'Y)$ and $s(LX'Y) = M_{Y'}(LXY')$. If $\mathcal{X}$ is cooperative in $\mathcal{X} \otimes \mathcal{Y}'$ with respect to r and if $\mathcal{Y}$ is cooperative in $\mathcal{X}' \otimes \mathcal{Y}$ with respect to s, then $r \oplus s$ is simple on LXY.*

Theorem 5.8. *Let r and s be homomorphisms with $M_X\langle r\langle R_X$, $M_Y\langle s$. If $r \oplus s$ is simple on LXY then $M_X \oplus s$ is simple on LXY.*

Using the partial order method developed in [Och97] $\mathcal{X}'$ and $\mathcal{Y}'$ can be computed efficiently on the basis of $\mathcal{X} \otimes \mathcal{Y}$ and simplicity of $f \oplus g$ on L can be checked directly.

6. Temporal Logic and Abstraction

Our verification approach can also be combined with *temporal logic* [Nit94a, Nit94c, Nit94d, Nit94b, Nit95, Nit98]. In terms of temporal logic, the automaton of Figure 2 approximately satisfies the formula $\mathcal{G}(\mathcal{F}(RES))$ ($\mathcal{G}$: always-operator, $\mathcal{F}$: eventually-operator; thus $\mathcal{G}(\mathcal{F}(RES))$ means ”infinitely often result”), but the system in Figure 3 does not. This is indeed the case because the abstracting homomorphism is not simple. Using an appropriate type of *model checking*, approximate satisfaction of temporal logic formulae can be checked by the sh-verification tool.

The Algorithm for checking approximate satisfaction of propositional linear-time temporal logic formulae (PLTL- formulae) is based on the algorithm for linear satisfaction of PLTL-formulae by Gerth, Peled, Vardi, and Wolper [GPVW96]. The key construction of the algorithm is the construction of a Büchi-automaton BA to a PLTL-formula η.

The *temporal logic formulae* (TL-formulae) we use are constructed as follows:

- *True* and *False* are atomic TL-formulae.

- The edge-labels of the automaton representing the concrete or abstracted behaviour of the system we are checking are atomic TL-formulae. In addition, ε is an atomic TL-formula (atomic proposition). ε is satisfied for a concrete action if and only if the action is mapped to the empty word by the abstraction.
- Formulae can be combined using the usual Boolean operators $\wedge$, $\vee$, $\neg$ and combinations thereof $\Rightarrow$ and $\Leftrightarrow$.
- Formulae can be combined using the usual temporal logic operators $\mathcal{G}$ (always), $\mathcal{F}$ (eventually), $\mathcal{U}$ (until), $\mathcal{B}$ (before) and $\mathcal{X}$ (next).
- Internally we use a temporal operator $\mathcal{V}$ which is the dual of the until-operator $\mathcal{U}$ ($\phi\mathcal{V}\psi$ is equivalent to $\neg((\neg\phi)\mathcal{U}(\neg\psi))$).
- In the automata that are generated during the model-checking optionally the before-operator $\mathcal{B}$ ($\phi\mathcal{B}\psi$ is equivalent $\neg((\neg\phi)\mathcal{U}\psi)$) can be used instead of $\mathcal{V}$ for better readability.

Algorithm. To check whether a behaviour B satisfies the property P_η represented by a PLTL-formula η approximately, one has to check whether $\mathrm{pre}(B) = \mathrm{pre}(B \cap \mathrm{P}_\eta)$. Herein, "pre(...)" designates the set of all finitely long prefixes of ω-words in "...". Since $\mathrm{pre}(B) \supseteq \mathrm{pre}(B \cap \mathrm{P}_\eta)$ always holds, this can be reduced to $\mathrm{pre}(B) \subseteq \mathrm{pre}(B \cap \mathrm{P}_\eta)$. Algorithmically, we have to check whether $\mathrm{pre}(B) \cap \mathrm{C}(B \cap \mathrm{P}_\eta)$ is the empty set. "C(...)" denotes the complement of "..." with respect to Σ^* (Σ is the set of all actions of the system, i.e. the alphabet of the ω-languages B and P_η).

An example for the automata used in the above construction as implemented in our tool is given in the appendix.

Our experience in practical examples shows that the combination of computing a minimal automaton of an LTS and model checking on this abstraction is significantly faster than direct model checking on the LTS.

The preservation result for approximately satisfied properties (Theorem 3.2) can be formulated in terms of PLTL using a syntactic transformation on PLTL-formulae [Nit94a, Nit98]. An example is given in section 8.

7. The Tool

As mentioned above, our verification method does not depend on a specific formal specification technique. For practical use the sh-verification tool has to be combined with a specification tool generating labeled transition systems. We have done this using the product net machine, and we are now implementing a specification environment based on asynchronous product automata (APA). Figure 11 shows the structure of the tool.
The product net machine is a tool for the design and analysis of product nets [Och91a]. Product nets [BOP89, OP95] are high level Petri nets with individual tokens. Figure 12 shows a product net specification of our client/server example, where the resource may eventually be locked forever. The shaded places are initially marked. In Figure 12 we do not use most of product nets' possible features. Indeed it is just a product net representation of a Petri net. The LTS

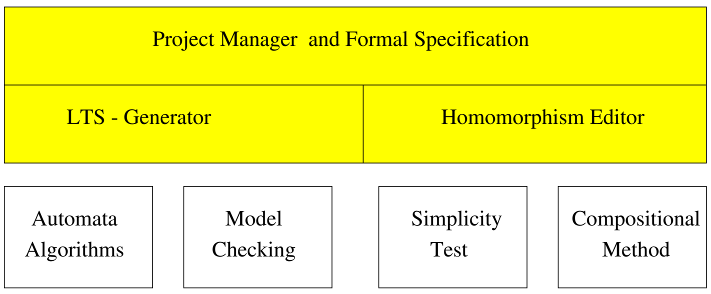

Fig. 11. The sh-verification tool

of Figure 3 is computed by the product net machine; it is the reachability graph of the product net in Figure 12.

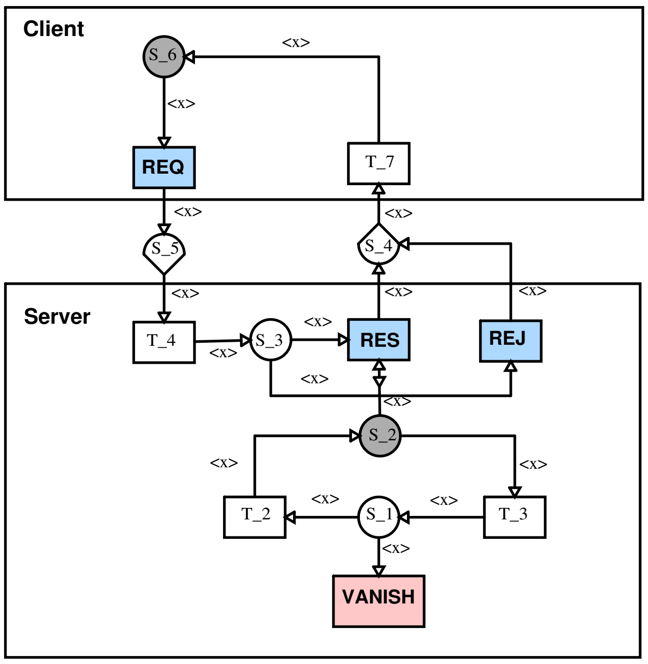

Fig. 12. Client Server Example

Practical experiences have been gained with large specifications, for example with ISDN-, XTP-, and smartcard protocols and by investigating service interactions in intelligent telecommunication systems [Klu92, Sch92, Gie93, Och93, OP93, Neb94, Och94a, OP95, CDGE$^+$96, CDF$^+$96]. Now our interest is focused on the verification of binding co-operations including electronic money and contract systems.

Technical Requirements. The sh-verification tool and the product net machine are both implemented in Allegro Common Lisp. The software is freely available (currently for Solaris and Windows NT) for non commercial purposes, but cannot be distributed via anonymous FTP, because of restrictions in the license agreement for the runtime library of the lisp system. For more information please contact the authors.

8. A Case Study

To demonstrate our method on a more realistic example we consider a model of the basic call process of an intelligent telephone network (IN). This call process, named the *basic call state model*(BCSM), is currently being standardized. This standardization process is structured in eight steps. Each step leads to a more detailed BCSM. These differently detailed standardization steps are called *capability sets* (CS). The currently standardized capability set is CS-1 (see [Q.1b] and especially [Q.1a]).

In this section, we verify a product net specification of the BCSM; throughout this section, we assume capability set CS-1 when briefly talking of BCSM [Q.1b, Q.1a]. We do not present the specification itself, but relate finite-state systems that we computed as abstractions of the BCSM specification's behaviour to automata descriptions in the standardization paper [Q.1a]. The complete specification can be found in [DFGE$^+$95]. Before starting with the verification steps, we give a brief introduction to the BCSM.

The BCSM handles the basic call process of an IN. This call process is internally structured, distinguishing caller and callee. The part of the BCSM related to a caller is named *originating* BCSM; abbreviated: O-BCSM. The callee oriented part is named *terminating* BCSM, or T-BCSM. Services in the IN, as for example call forwarding, are add-on features. The interface between BCSM and services is the *service logic*. The general structure of an IN, including caller and callee, is depicted in Figure 13. Dashes represent communication channels.

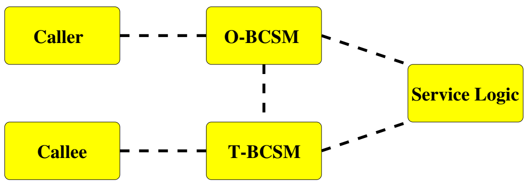

Fig. 13. The basic structure of an IN.

The BCSM is some kind of a finite-state system. The states are called *points in call* (PIC). Added to PIC are *detection points* (DP) from where the service logic may be invoked. The finite-state system that represents the BCSM is described graphically as well as textually in the standard [Q.1a]. A product net specification of the originating as well as the terminating BCSM was established in the SERVINT-project [NO95, DFGE$^+$95, DFGE$^+$96]. There, ambiguities in the standard, and contradictions between the textual and graphical description of the BCSM are resolved.

Two abstractions of the BCSM specification's behaviour leaving visible only the actions related to the O-BCSM and T-BCSM respectively, lead to exactly the resolved finite-state systems of the standard [Q.1a] representing O-BCSM and T-BCSM. Since all abstractions mentioned are obtained by applying an abstracting homomorphism that is simple on the concrete behaviour, approximately satisfied properties of the abstract behaviour represent corresponding approximately satisfied properties of the concrete behaviour. This observation, in principle, is sufficient to verify the correctness of the specification in comparison to [Q.1a]: the components of the BCSM behave in their concrete environment in the same way as they would behave in an idealized environment. In the subsequent paragraphs, we look more closely at the T-BCSM's behaviour, checking explicitly some properties.

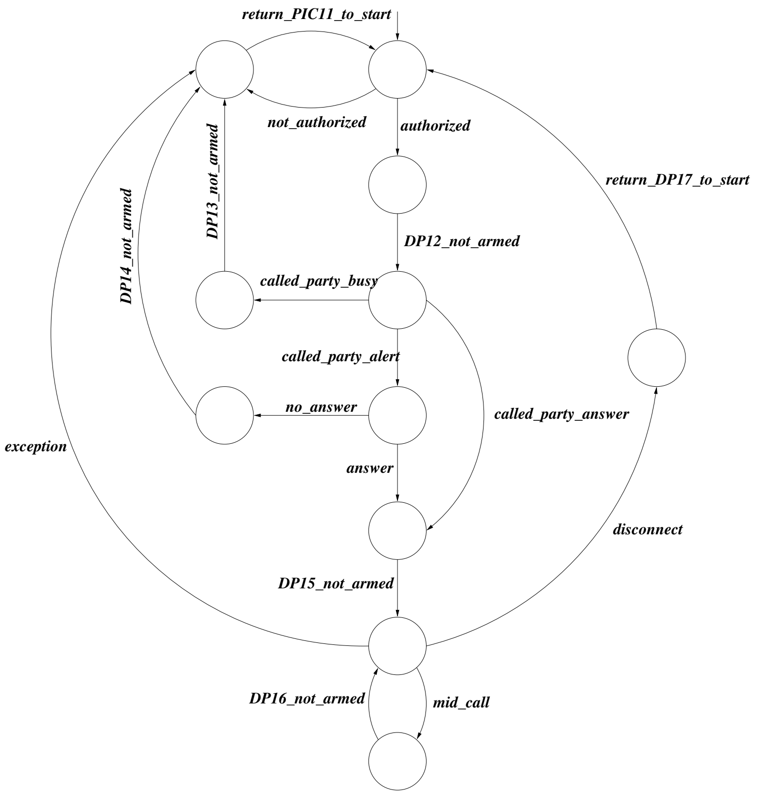

Fig. 14. T-BCSM.

After applying a homomorphism that is the identity function on actions relevant to the T-BCSM and that takes all other actions to the empty word, we obtain the finite-state system in Figure 14 that represents the abstract behaviour of the whole BCSM specification that is related to the T-part. As it is verified by our tool the abstracting homomorphism is simple on the concrete behaviour. It extracts the T-BCSM's behaviour when the T-BCSM is embedded in the environment that is the O-BCSM. We obtain exactly the behaviour as presented in the standard [Q.1a]. Consequently all properties that the standard demands the T-BCSM to satisfy are indeed satisfied for the T-BCSM in the BCSM specification. Nevertheless we check explicitly a property of the T-BCSM by firstly applying another abstraction step.

Figure 14 contains some actions that are interruptions to the straightforward calling process. These actions are *not_authorized*, *called_party_busy*, *no_answer*, and *exception*. Whenever one of these actions occurs, an exception handling is necessary. The exception handling is performed by PIC 11. Occurrence of the action *return_PIC11_to_start* indicates that a successful exception handling has taken place. To check the property "whenever an interruption of the calling process occurs, an exception handling takes place", we can define a suitable abstraction on the behaviour presented in Figure 14 that keeps visible the interruption and the exception handling, and check a suitable temporal logic formula on the abstract behaviour.

A suitable abstraction on T-BCSM's behaviour is defined by the homomorphism that maps *not_authorized*, *called_party_busy*, *no_answer*, and *exception* on the abstract action *interruption*, that maps action *return_PIC11_to_start* on the abstract action *exception_handling*, and that takes all other actions to the empty word. The resulting abstract behaviour is depicted in Figure 15.

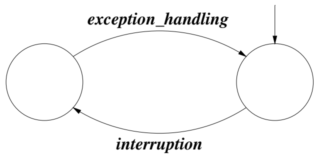

Fig. 15. An abstraction of the T-BCSM.

Obviously, $\mathcal{G}(\mathit{interruption} \Rightarrow \mathcal{X}\,\mathit{exception\_handling})$ represents an approximately satisfied property of the behaviour presented in Figure 15. Let h be the abstracting homomorphism on the concrete behaviour that generates the abstract behaviour in Figure 14 and let h' be the abstracting homomorphism on this abstract behaviour that generates the more abstract behaviour presented in Figure 15. Because h is simple on the concrete behaviour and h' is simple on the behaviour presented in Figure 14 (both behaviours have a strongly connected finite-state representation), $h \circ h'$ is simple on the BCSM specification's behaviour (Theorem 3.6). According to the syntactic transformation of PLTL-formulae [Nit98] we obtain, that

$$\mathcal{G}(\varepsilon \vee (\mathit{interruption} \Rightarrow (\varepsilon\,\mathcal{U}(\neg\varepsilon \wedge \mathcal{X}(\varepsilon\,\mathcal{U}\,\mathit{exception\_handling})))))$$

is an approximately satisfied property of the BCSM specification's behaviour. Simplification of this formula leads to

$$\mathcal{G}(\mathit{interruption} \Rightarrow \varepsilon\,\mathcal{U}\,\mathit{exception\_handling}).$$

If we interpret this result, we find that the reasonable computations of the BCSM specification (in this context the reasonable computations are once again the fair ones) satisfy the *correct exception handling property*. This illustrates how stepwise abstraction can be performed which can be regarded as an inverse stepwise refinement.

9. Conclusions

We have presented the basic functionality of the sh-verification tool in this article. The tool is equipped with the main features necessary to verify specifications of co-operating systems of industrial size. It comprises a satisfaction relation with an inherent fairness assumption and an abstraction concept adequate for the particular, practically useful satisfaction relation. Our verification method, which is based on the very general notions of approximate satisfaction of properties and simple language homomorphisms, does not depend on a specific formal specification method. It can be applied to all those specification techniques having an LTS-semantics.

Summarizing, using simple abstractions and approximate satisfaction verification can be done in two ways and is supported by our tool:

- System properties are explicitly given by temporal logic formulae or Büchi-automata. They can be checked on the abstract behaviour (under a simple homomorphism).
- Specifications of different abstraction levels are compared by corresponding simple homomorphisms. In that case system properties are given implicitly.

There exists a variety of verification tools which can be found in the literature. Some are model-checking based, others are proof system based. We consider COSPAN [Kur94] to be closest to the sh-verification tool. COSPAN is automata based and contains a homomorphism based abstraction concept. Since the transition labels of automata in COSPAN are in a Boolean algebra notation, the abstraction homomorphisms are Boolean algebra homomorphisms which correspond to non-erasing alphabetic language homomorphisms on the automata level. The sh-verification tool, in addition, offers erasing homomorphisms as an abstraction concept. COSPAN also considers only linear satisfaction of properties. Thus fairness assumptions need to be made explicitly in this tool. Besides many other tools we want to name only two more. Since it was one of the first verification tools, CESAR should be mentioned [QS82]. A tool which uses the modal μ-calculus as a specification language for properties [Sti89] is the concurrency workbench [CPS93].

We consider the main strength of our tool to be the combination of an inherent fairness assumption in the satisfaction relation, an abstraction technique compatible with approximate satisfaction, and a suitable compositional and partial order method for the construction of only a partial state space. The sh-verification tool's user interface and general handling has reached a level of maturity that enabled its successful application in the industrial area [NO95, DFGE$^+$95, DFGE$^+$96].

References

[AS85] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, October 1985.

[BD87] S. Budkowski and P. Dembinski. An introduction to estelle. *Computer Networks and ISDN-Systems*, 14:3–23, 1987.

[BOP89] H. J. Burkhardt, P. Ochsenschläger, and R. Prinoth. Product nets — a formal description technique for cooperating systems. GMD-Studien 165, Gesellschaft für Mathematik und Datenverarbeitung (GMD), Darmstadt, September 1989.

[CDF+96] C. Capellmann, R. Demant, F. Fatahi, R. Galvez-Estrada, U. Nitsche, and P. Ochsenschläger. Verification by behavior abstraction: A case study of service interaction detection in intelligent telephone networks. In *Computer Aided Verification (CAV) '96*, volume 1102 of *Lecture Notes in Computer Science*, pages 466–469, New Brunswick, 1996.

[CDGE+96] C. Capellmann, R. Demant, R. Galvez-Estrada, U. Nitsche, and P. Ochsenschläger. Case study: Service interaction detection by formal verification under behaviour abstraction. In Tiziana Margaria, editor, *Proceedings of International Workshop on Advanced Intelligent Networks'96*, pages 71–90, Passau, March 1996.

[CPS93] R. Cleaveland, J. Parrow, and B. Steffen. The concurrency workbench: A semantics-based tool for the verification of finite-state systems. In *TOPLAS 15*, pages 36–72, 1993.

[DFGE+95] R. Demant, F. Fatahi, R. Galvez-Estrada, U. Nitsche, and P. Ochsenschläger. Abschlußbericht des GMD-/Telekom-Projekts Formale Spezifikations- und Verifikationsmethoden zur Behandlung der Service-Interaction-Problematik – SERVINT. Abschluß bericht, GMD, Dezember 1995.

[DFGE+96] R. Demant, F. Fatahi, R. Galvez-Estrada, U. Nitsche, and P. Ochsenschläger. Zwischenbericht des GMD-/Telekom-Projekts Formale Spezifikations- und Verifikationsmethoden zur Behandlung der Service-Interaction-Problematik – SERVINT2. Zwischenbericht, GMD, Juli 1996.

[Eil74] S. Eilenberg. *Automata, Languages and Machines*, volume A. Academic Press, New York, 1974.

[Gie93] H. Giehl. Verifikation von Smartcard-Anwendungen mittels Produktnetzen. GMD-Studien 225, Gesellschaft für Mathematik und Datenverarbeitung (GMD), Darmstadt, 1993.

[GPVW96] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In P. Dembinski and M. Sredniawa, editors, *Protocol Specification, Testing, and Verification XV '95*, pages 3–18. Chapman & Hall, 1996.

[Klu92] W. Klug. OSI-Vermittlungsdienst und sein Verhältnis zum ISDN-D-Kanalprotokoll. Spezifikation und Analyse mit Produktnetzen. Arbeitspapiere der GMD 676, Gesellschaft für Mathematik und Datenverarbeitung (GMD), Darmstadt, 1992.

[Kur94] R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes.* Princeton University Press, Princeton, New Jersey, first edition, 1994.

[Neb94] M. Nebel. Ein Produktnetz zur Verifikation von Smartcard-Anwendungen in der STARCOS-Umgebung. GMD-Studien 234, Gesellschaft für Mathematik und Datenverarbeitung (GMD), Darmstadt, 1994.

[Nit94a] U. Nitsche. Propositional linear temporal logic and language homomorphisms. In Anil Nerode and Yuri V. Matiyasevich, editors, *Logical Foundations of Computer Science '94, St. Petersburg*, volume 813 of *Lecture Notes in Computer Science*, pages 265–277. Springer Verlag, 1994.

[Nit94b] U. Nitsche. Simple homomorphisms and linear temporal logic. Arbeitspapiere der GMD 889, GMD – Forschungszentrum Informationstechnik, Darmstadt, December 1994.

[Nit94c] U. Nitsche. A verification method based on homomorphic model abstraction. In *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing*, page 393, Los Angeles, 1994. ACM Press.

[Nit94d] U. Nitsche. Verifying temporal logic formulas in abstractions of large reachability graphs. In J. Desel, A. Oberweis, and W. Reisig, editors, *Workshop: Algorithmen und Werkzeuge für Petrinetze.* Humboldt Universität Berlin, 1994.

[Nit95] U. Nitsche. A finitary language semantics for propositional linear temporal logic (abstract). In *Preproceedings of the 2nd International Conference on Developments in Language Theory.* University of Magdeburg, 1995.

[Nit98] U. Nitsche. *Verification of Co-Operating Systems and Behaviour Abstraction.* PhD thesis, University of Frankfurt, Germany, 1998.

[NO95] U. Nitsche and P. Ochsenschläger. Zwischenbericht des GMD-/Telekom-Projekts Formale Spezifikations- und Verifikationsmethoden zur Behandlung der Service-Interaction-Problematik – SERVINT. Zwischenbericht, GMD, Juli 1995.

[NO96] U. Nitsche and P. Ochsenschläger. Approximately satisfied properties of systems and simple language homomorphisms. *Information Processing Letters*, 60:201–206, 1996.

[NW97] U. Nitsche and P. Wolper. Relative liveness and behavior abstraction (extended abstract). In *Proceedings of the 16th ACM Symposium on Principles of Distributed Computing (PODC'97)*, Santa Barbara, CA, 1997.

[Och88] P. Ochsenschläger. Projektionen und reduzierte Erreichbarkeitsgraphen. Arbeitspapiere der GMD 349, Gesellschaft für Mathematik und Datenverarbeitung (GMD), Darmstadt, Dezember 1988.

[Och90] P. Ochsenschläger. Modulhomomorphismen. Arbeitspapiere der GMD 494, Gesellschaft für Mathematik und Datenverarbeitung (GMD), Darmstadt, Dezember 1990.

[Och91a] P. Ochsenschläger. Die Produktnetzmaschine. *Petri Net Newsletter*, 39:11–31, August 1991. Also appeared as a GMD Arbeitspapier Nr. 505, 1991.

[Och91b] P. Ochsenschläger. Modulhomomorphismen II. Arbeitspapiere der GMD 597, Gesellschaft für Mathematik und Datenverarbeitung (GMD), Darmstadt, November 1991.

[Och92] P. Ochsenschläger. Verifikation kooperierender Systeme mittels schlichter Homomorphismen. Arbeitspapiere der GMD 688, Gesellschaft für Mathematik und Datenverarbeitung (GMD), Darmstadt, Oktober 1992.

[Och93] P. Ochsenschläger. Verifikation verteilter Systeme mit Produktnetzen. *PIK*, 16:42–43, 1993.

[Och94a] P. Ochsenschläger. Kompositionelle Verifikation kooperierender Systeme. Arbeitspapiere der GMD 885, GMD – Forschungszentrum Informationstechnik, Darmstadt, Dezember 1994.

[Och94b] P. Ochsenschläger. Verification of cooperating systems by simple homomorphisms using the product net machine. In J. Desel, A. Oberweis, and W. Reisig, editors, *Workshop: Algorithmen und Werkzeuge für Petrinetze*, pages 48–53. Humboldt Universität Berlin, 1994.

[Och94c] P. Ochsenschläger. Verifikation von Smartcard-Anwendungen mit Produktnetzen. In *Tagungsband des 4. SmartCard Workshops*, Darmstadt, 1994.

[Och95] P. Ochsenschläger. Compositional verification of cooperating systems using simple homomorphisms. In J. Desel, H. Fleischhack, A. Oberweis, and M. Sonnenschein, editors, *Workshop: Algorithmen und Werkzeuge für Petrinetze*, pages 8–13. Universität Oldenburg, 1995.

[Och96] P. Ochsenschläger. Kooperationsprodukte formaler Sprachen und schlichte Homomorphismen. Arbeitspapiere der GMD 1029, GMD – Forschungszentrum Informationstechnik, Darmstadt, 1996.

[Och97] P. Ochsenschläger. Schlichte Homomorphismen auf präfixstabilen partiell kommutativen Sprachen. Arbeitspapiere der GMD 1106, GMD – Forschungszentrum Informationstechnik, Darmstadt, 1997.

[OP93] P. Ochsenschläger and R. Prinoth. Formale Spezifikation und dynamische Analyse verteilter Systeme mit Produktnetzen. In *Informatik aktuell Kommunikation in verteilten Systemen*, pages 456–470, München, 1993. Springer Verlag.

[OP95] P. Ochsenschläger and R. Prinoth. *Modellierung verteilter Systeme – Konzeption, Formale Spezifikation und Verifikation mit Produktnetzen.* Vieweg, Wiesbaden, 1995.

[ORRN97] P. Ochsenschläger, J. Repp, R. Rieke, and U. Nitsche. The SH-verification tool. In *Proceedings of the 2nd International Workshop on Formal Methods for Industrial Critical Systems (FMICS'97)*, Cesena, Italy, 1997.

[Q.1a] Draft Revised ITU-T Recommendation Q.1214: *Distributed Functional Plane for Intelligent Network CS-1.* March 1995.

[Q.1b] ITU-T Recommendations Q.12xx – Q series: *Intelligent Network Recommendation.* 1992.

[QS82] J.P. Queille and J. Sifakis. Specification and verification of concurrent systems in cesar. volume 137 of *Lecture Notes in Computer Science*, pages 337–351, 1982.

[Sch92] S. Schremmer. ISDN-D-Kanalprotokoll der Schicht 3. Spezifikation und Analyse mit Produktnetzen. Arbeitspapiere der GMD 640, Gesellschaft für Mathematik und Datenverarbeitung (GMD), Darmstadt, 1992.

[SSR89] R. Saracco, J. R. W. Smith, and R. Reed. *Telecommunication Systems' Engineering using SDL.* North Holland, 1989.

[Sti89] C. Stirling. An introduction to modal and temporal logics for CCS. In A. Yonezawa and T. Ito, editors, *Concurrency: Theory, Language, and Architecture*, volume 391 of *Lecture Notes in Computer Science*. Springer Verlag, 1989.

[Zie89] W. Zielonka. Safe executions of recognizable trace languages by asynchronous automata. In *LNCS 363*. Springer Verlag, 1989.

Appendix

As an example for the temporal logic algorithms described in chapter 6 we describe the steps performed by our tool to check whether the property $\mathcal{G}(\mathcal{F}(RES))$ is satisfied approximately by the "behaviour-automaton" of Figure 3.

On the automaton level, we have to perform 7 Steps:

1. Compute a Büchi-Automaton representing the property given by a PLTL-formula according to [GPVW96]. For the formula $\mathcal{G}(\mathcal{F}(RES))$ which represents the property that "a result RES is infinitely often produced" the automaton construction is represented in two steps in Figure 16 and Figure 17.

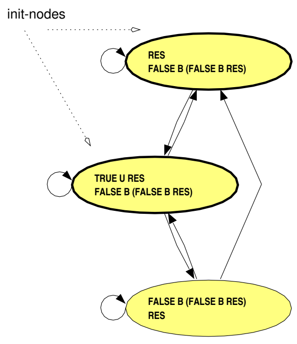

Fig. 16. Graph for $\mathcal{G}(\mathcal{F}(RES))$

2. Construct the synchronous product of the automaton constructed so far and the automaton representing the behaviour of the considered system. The synchronous product is the construction of the intersection of languages on the automaton level. For the "property- automaton" of Figure 17 and the "behaviour-automaton" of Figure 3, the "product-automaton" is represented in Figure 18.
3. Reduce the resulting Büchi-Automaton (remove all states which are not reachable from the initial state or from which no cycle containing an accepting state is reachable).
4. Ignore acceptance conditions (make all states accepting) and do not interpret the automaton anymore as an automaton on infinite (ω-) words but as one on finitely long words (this corresponds to the prefix construction ("pre(...)").
5. Construct the complement automaton (for a finite-word automaton, not an ω-automaton).
6. Construct the intersection with the automaton representing the behaviour.
7. Check whether the resulting automaton is empty (does not accept words).

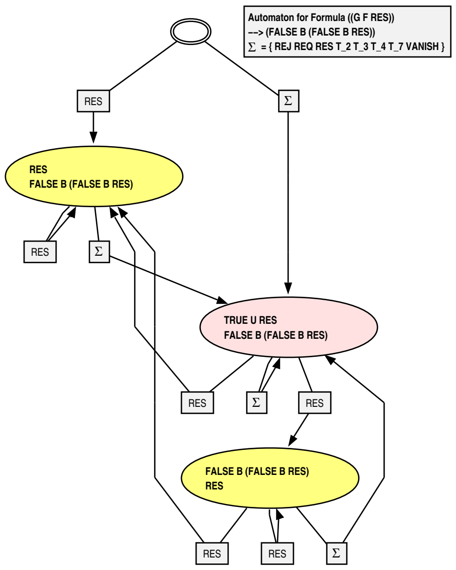
Automaton for Formula ((G F RES))
---> (FALSE B (FALSE B RES))
Σ = { REJ REQ RES T_2 T_3 T_4 T_7 VANISH }
RES
Σ
RES
FALSE B (FALSE B RES)
RES
Σ
TRUE U RES
FALSE B (FALSE B RES)
RES
Σ
RES
FALSE B (FALSE B RES)
RES
RES
RES
Σ

Fig. 17. Automaton for $\mathcal{G}(\mathcal{F}(RES))$

For the considered example of the behaviour in Figure 3 and the property given by the PLTL-formula $\mathcal{G}(\mathcal{F}(RES))$, the behaviour satisfies the property approximately.

Besides the algorithm described above that checks for approximate satisfaction we also have implemented algorithms for other kinds of satisfaction relations (PLTL and AGEF [Nit98]).

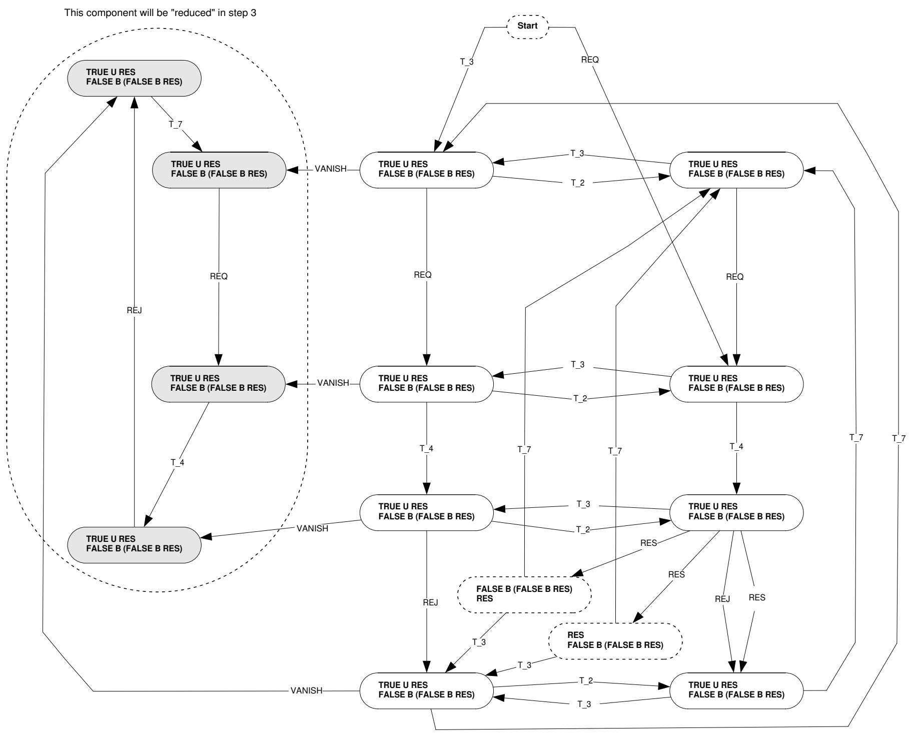
This component will be "reduced" in step 3
Start
TRUE U RES FALSE B (FALSE B RES)
FALSE B (FALSE B RES) RES
RES FALSE B (FALSE B RES)
T_3
T_2
T_4
T_7
REQ
REJ
RES
VANISH

Fig. 18. The synchronous product automaton of Figure 3 and Figure 17