The SH-Verification Tool - A Tutorial

Peter Ochsenschläger Jürgen Repp
Roland Rieke
SIT – Fraunhofer - Institute for Secure Telecooperation,
Rheinstr. 75, D-64295 Darmstadt, Germany
E-Mail: {ochsenschlaeger,repp,rieke}@sit.fraunhofer.de

September 30, 2002

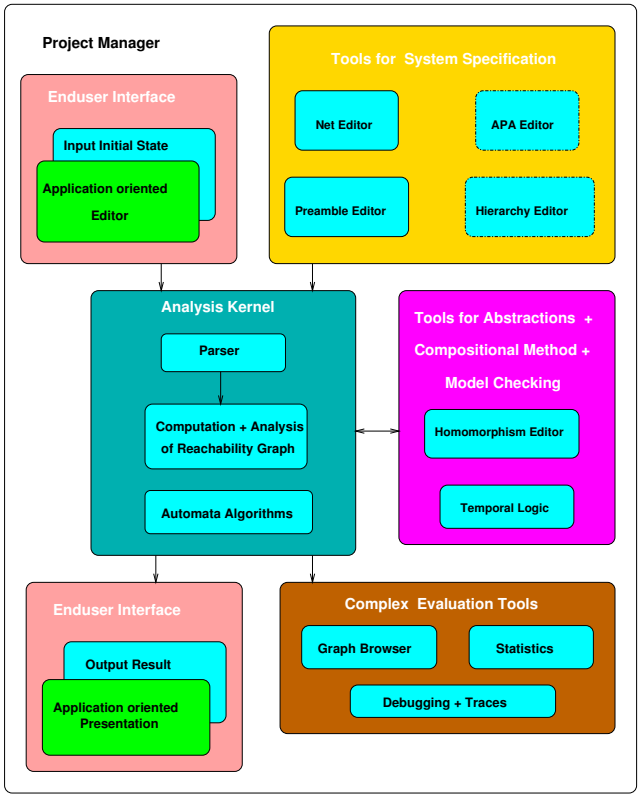

Figure 1: Components of the sh-verification tool

1 Introduction

The sh-verification tool (SHVT) [1] [ORR00b] supports formal specification, analysis and verification of cooperating systems. Figure 1 shows the structure of the tool. The main components of the system are the tools for specification, the analysis kernel, the tools for abstraction and the project manager. It is possible to extend the tool by different application oriented user interfaces. Small but typical examples shows the steps for specifying and analysing systems behaviour using the sh-verification tool.

2 A first APA Example

The presented verification method is described in [ORR00c]. The reader is referred to this paper for notations, definitions and theorems. The method does not depend on a specific formal specification technique. For practical use it has to be combined with a specification tool generating labeled transition systems (LTS [2]). The current implementation of SHVT uses asynchronous product automata (APA) [ORRN99] and product nets [3] [BOP89, OP95] as specification environments. In this tutorial we only consider APA.

APA are a universal and very flexible operational description concept for cooperating systems. It "naturally" emerges from formal language theory [ORR00a].

An APA can be seen as a family of elementary automata. The set of all possible states of the whole APA is structured as a product set; each state is divided into state components. In the following the set of all possible states is called state set. The state sets of elementary automata consist of components of the state set of the APA. Different elementary automata are "glued" by shared components of their state sets. Elementary automata can "communicate" by changing shared state components.

Figure 2 shows a graphical representation of a small asynchronous product automaton consisting of two elementary automata $Send\_A$ and $Receive\_B$ and state components $Data\_A$, $Data\_B$ and $Network$, with state sets Z_{Data_A} , Z_{Data_B} and $Z_{Network}$. The state set of the APA is the product of Z_{Data_A} , Z_{Data_B} and $Z_{Network}$. The state set of $Receive\_B$ is the product of Z_{Data_B} and $Z_{Network}$. The state set of $Send\_A$ is the product of Z_{Data_A} and $Z_{Network}$. The figure shows the structure of the automaton. The circles represent state components and a box corresponds to one elementary automaton. The full specification of the automaton includes the transition relations of the elementary automata and the initial state. The neighbourhood relation N, represented by the edges, indicates which state components are included in the state of an elementary automaton and may be changed by a state transition of the elementary automaton. A state transition of automaton $Send\_A$ may change the content of $Data\_A$ and $Network$ while $Receive\_B$ may change $Data\_B$ and $Network$.

[1] sh abbreviates simple homomorphism

[2] The semantics of formal specification techniques for distributed systems is usually based on LTS.

[3] a special class of high level petri nets

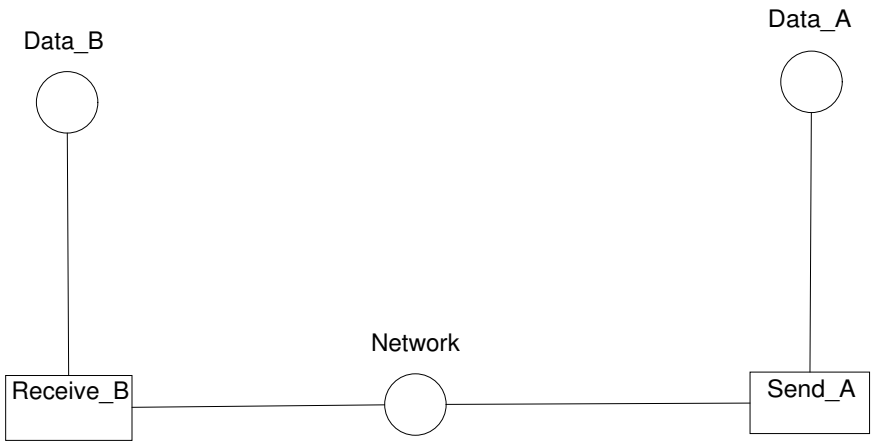

Figure 2: APA

Formally an *Asynchronous Product Automaton* consists of a family of *State Sets* $Z_S, S \in \mathbb{S}$, a family of *Elementary Automata* $(\Phi_e, \Delta_e), e \in \mathbb{E}$ and a *Neighbourhood Relation* $\mathrm{N} : \mathbb{E} \to \mathcal{P}(\mathbb{S})$; $\mathcal{P}(X)$ is the power set of X and $\mathbb{S}$ and $\mathbb{E}$ are index sets with the names of state components and elementary automata. For each Elementary Automaton (Φ_e, Δ_e)

- Φ_e is its *Alphabet* and
- $\Delta_e \subseteq \bigtimes_{S \in N(e)}(Z_S) \times \Phi_e \times \bigtimes_{S \in N(e)}(Z_S)$ is its *State Transition Relation*

For each element of Φ_e the state transition relation Δ_e defines state transitions that change only the state components in $N(e)$.
An APA's (global) *States* are elements of $\bigtimes_{S \in \mathbb{S}}(Z_S)$. To avoid pathological cases it is generally assumed that $\mathbb{S} = \bigcup_{e \in \mathbb{E}}(N(e))$ and $N(e) \neq \emptyset$ for all $e \in \mathbb{E}$. Each APA has one *Initial State* $s_0 = (q_{0S})_{S \in \mathbb{S}} \in \bigtimes_{S \in \mathbb{S}}(Z_S)$.
In total, an APA $\mathbb{A}$ is defined by $\mathbb{A} = ((Z_S)_{S \in \mathbb{S}}, (\Phi_e, \Delta_e)_{e \in \mathbb{E}}, N, s_0)$.
The behaviour of an APA is represented by all possible sequences of state transitions starting with initial state s_0.
The sequence $(s_0, (e_1, a_1), s_1)(s_1, (e_2, a_2), s_2)(s_2, (e_3, a_3), s_3) \ldots$ with $a_i \in \Phi_{e_i}$ represents one possible sequence of actions of an APA.
State transitions $(s, (e, a), \bar{s})$ may be interpreted as labeled edges of a directed graph whose nodes are the states of an APA: $(s, (e, a), \bar{s})$ is the edge leading from s to $\bar{s}$ and labeled by (e, a). The subgraph reachable from the node s_0 is called the *reachability graph* of an APA.

To illustrate how APA are represented in SHVT we consider the modelling of protocols in the following way: The actions of an protocol agent are modelled by specifying one or more elementary automata. Each automaton is connected to one or more state components. The state components that can only be accessed by the automata of one agent are used to model the memory of that agent. Agents (i.e. elementary automata) communicate via a shared compo-

nent $Network$. Sending of a message is modelled by adding the message to the content of $Network$, receiving is modelled by removing it from $Network$. Receive actions of an agent (of an elementary automaton) usually include checks that some part of the message is equal to some data stored in one of the agent's state components.

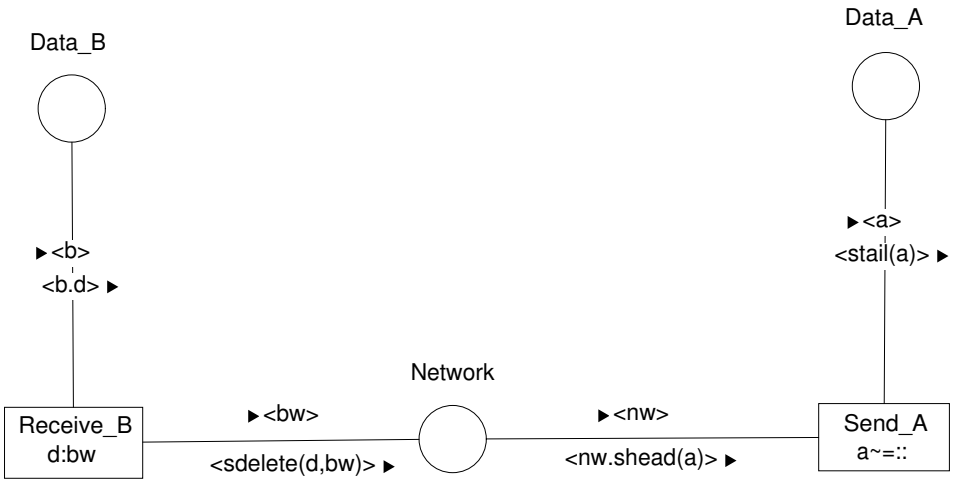

Figure 3: APA Example with Inscriptions

Figure 3 represents two agents A and B communicating via the state component $Network$ [4]. Each of the agents is equipped with a further state component $Data$ which serves as the agent's memory.

In the above example data stored on state component $Data\_A$ can be sent by the elementary automaton $Send\_A$ via $Network$. The elementary automaton $Receive\_B$ can read data from $Network$ and store it in the state component $Data\_B$. Every edge of an APA is labeled with two inscriptions describing the state transition relation of the corresponding elementary automaton. One denotes the match for the data read from the state component (read-inscription, denoted by a black arrow ▶ at the beginning of the line), the other denotes the value that will be returned to the state component (write-inscription, denoted by a black arrow ▶ at the end of the line). Elementary automata may contain two different kinds of inscriptions: *predicates* as in $Send\_A$ and assignments to *interpretation variables* (see below) as in $Receive\_B$. A transition of an elementary automaton may occur if:

- read-inscriptions of all edges connecting that automaton with a state component match the data stored in the respective state component
- the predicate in the elementary automaton inscription is true (optional)
- every value denoted by the write-inscriptions can be computed.

Let us assume that the state components of our example have the following initial values:

[4] You find this APA in "[install-dir]/demo/apas/SimpleAPA/simple.apa"

```
Data_A   <('t1','data').('t2','data')>
Data_B   <::>
Network  <::>
```

:: denotes the empty sequence. Going out from this initial state the following reachability graph is computed by the tool:

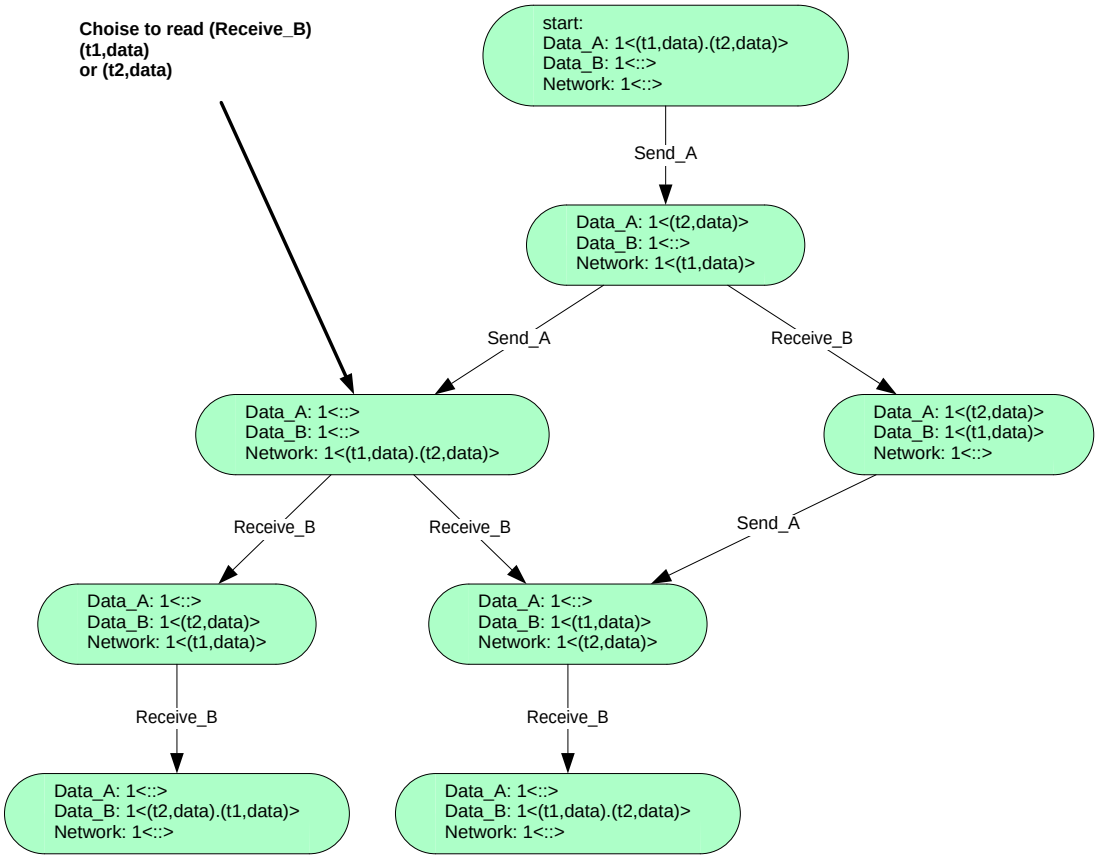

Figure 4: Reachability Graph

For simplicity edges are only labeled by the name of the corresponding elementary automaton e (instead of (e, a)). In the initial state the elementary automaton $Send_A$ can perform a state transition: It assigns the value of $Data_A$ (i.e. `('t1','data').('t2','data')`) to the variable a and checks that this value is not equal to the empty sequence. It then reads (removes) `('t1','data').('t2','data')` from $Data_A$ and returns (adds) the tail of `('t1','data').('t2','data')` (i.e. `('t2','data')`) to $Data_A$ (using the function *stail*). In the same state transition it reads the content of the state component $Network$, which is the empty sequence in the initial state, and assigns it to the variable nw. It finally writes the value of nw concatenated with the head of the value of a (i.e. `('t1','data')`) to $Network$.

In this simple case, variables as inscription of edges (e.g. a in the above

example) are bound to the complete content of the state component and are processed completely (there are other types of bindings as well). In contrast, so-called interpretation variables that occur at the left side of : in inscriptions of elementary automata are used to express different state transitions with respect to different values of the interpretation variables. For an elementary automaton e the different values of the interpretation variables correspond to the elements of its alphabet Φ_e.

In our example, d is such an interpretation variable as it occurs as the inscription of the elementary automaton $Receive_B$. First the value of the state component $Network$ is bound to the standard variable bw. Then each of the components of the sequence bw is non-deterministically bound to the interpretation variable d (indicated by the expression $d : bw$ inside $Receive_B$). The automaton now performs one state transition for each of the components of bw. This is why $Receive_B$ can perform no state transition in the initial state: The initial value of $Network$ is the empty sequence which does not contain any component.

However, after the first state transition $Receive_B$ can perform a state transition. Now $Network$ contains `('t1','data')`, which is assigned first to bw and then to d. At the same time, `('t1','data')` is removed from and the result of `sdelete(d,bw)` (the sequence bw without the component d) is added to $Network$. In the same state transition, $Receive_B$ reads the content of the state component $Data_B$, assigns it to b and writes b, concatenated with whatever component of d it chose to process. In the first state, there is only one component to choose, namely `('t1','data')`. In the state transition sequence where the first and the second state transitions are performed by $Send_A$, the content of $Network$ is `('t1','data').('t2','data')`. Thus in this state $Receive_B$ can continue either with `d = ('t1','data')` or with `d = ('t2','data')`. See figure 4 for the complete reachability graph.

In our example every state component has the same state set *sequence*. The following preamble defines the data type and the function *shead* used in our example:

```
defset tag = { 't1', 't2' };
defset data = { 'data'};
defset message = pro ( tag, data );
defset sequence = seq (message);
defcase shead : sequence>> sequence
     shead(x) = if l(x) > 0 then seg(x,1,1)
                            else ::;
```

tag and *data* are finite sets defined by listing their elements, *message* is the cartesian product of the sets *tag* and *data*. *sequence* is the set of finite sequences of elements of the set *message* including the empty sequence denoted by ::. The function *shead* provides the first element of a sequence. The other functions used in the APA, *stail* (tail of a sequence) and *sdelete* (deletes one element in a sequence) are predefined functions. For more elaborated examples and the description of the preamble language see [FhG].

You can read this APA into the APA editor from "[install-dir]/demo/apas/SimpleAPA/simple.apa".
Compile the Preamble with the command "File>Compile Buffer". After "Parser>Load Preamble" in the APA Editor perform the command "Parser>Analysis". The analysis window will be opened an you can compute the reachability graph with "Start Exhaustive Analysis". The graph can be drawn with "mouse-r Draw Graph" on object "Reachability Graph simple". The values of the state components will not be displayed. You can inspect these values for one node with "mouse-r Show Object".

3 Client Server APA Example

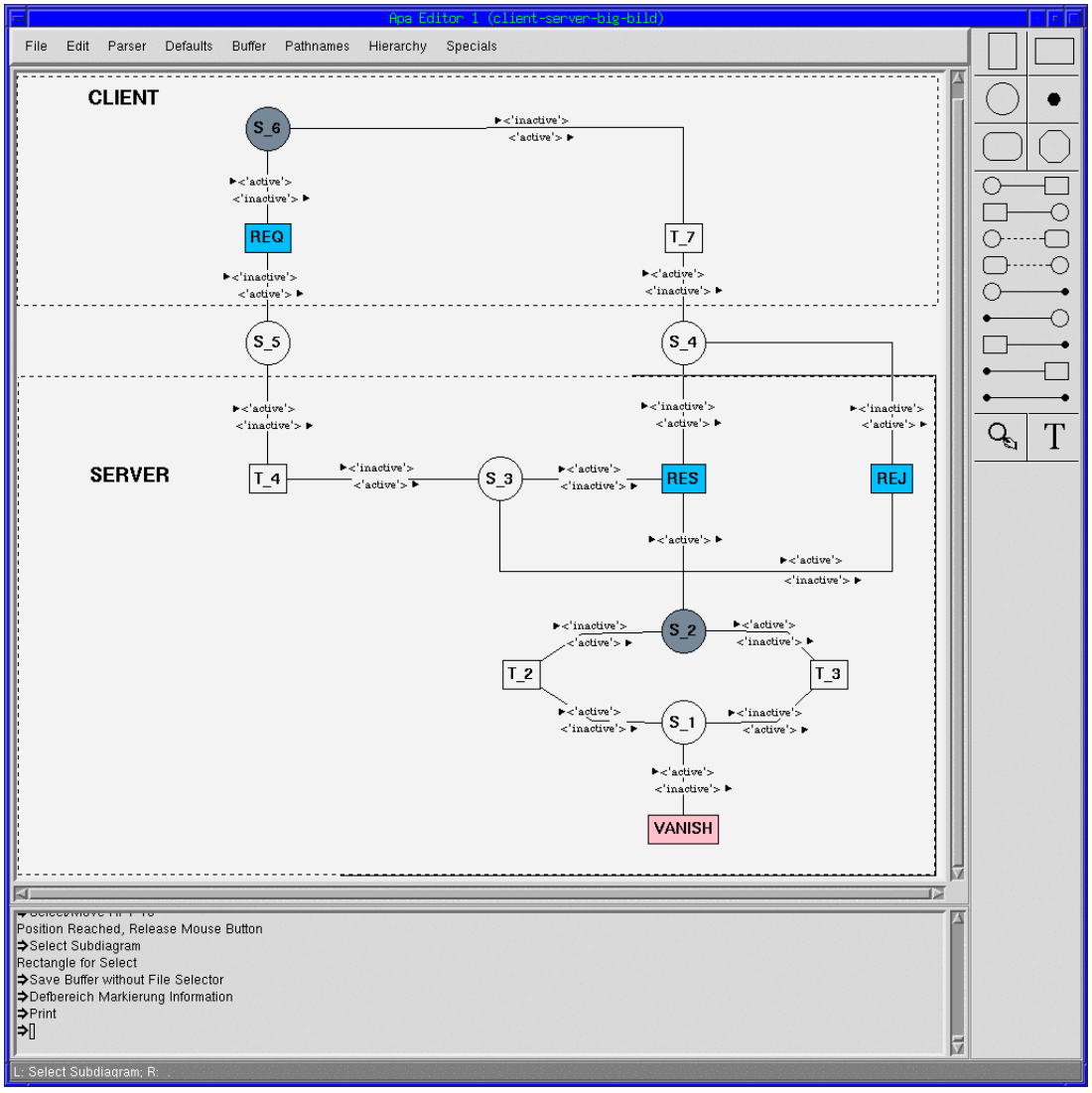

Figure 5: Client Server Example

To illustrate the usage of the verification method described in [ORR00c] we consider an example of a system that consists of a client and a server as its main components. The client sends requests REQ to the server, expecting the server to produce particular results. Nevertheless, for some reasons, the server may not always respond to a request by sending a result RES, but may, as well, reject a request REJ (Figure 7).

Figure 5 shows a APA specification of this example. It is a global model for the systems behaviour. In this model an elementary automaton can perform a transition if every of its state components has the value $'active'$. The initial value of the shaded state components S_6 and S_2 is *active*, the others are *inactive*. So initially only the elementary automata T_3 and REQ can act. Note that the resource may eventually be locked forever.

Usually complex systems are specified hierarchically. This is supported by the *project manager* of the tool. (In our simple example the specification is flat.)

The LTS in Figure 6, which is the reachability graph of the APA in Figure 5, is computed by the tool. For better readability we have inserted the labels of the *active* state components into the nodes.

This LTS consists of two strongly connected components (marked by different colors). Usually the LTS of a specification is too complex for a complete graphical presentation; there are several features to inspect the LTS.

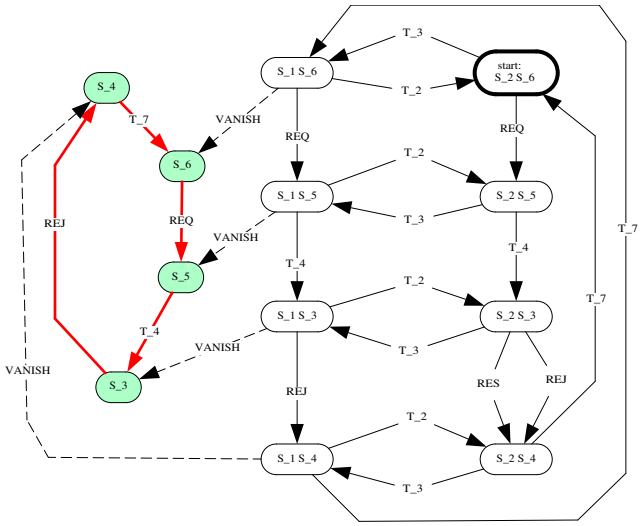

Figure 6: LTS

4 Abstraction

In the example the important actions with respect to the client's behaviour, are sending a request and receiving a result or rejection.

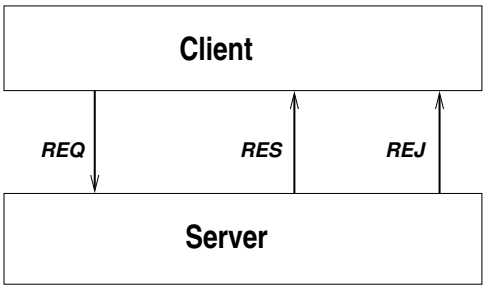

Figure 7: Client Server Abstract View

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* (correctness criterion).

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 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. This is supported by the homomorphism editor of the tool (Figure 8).

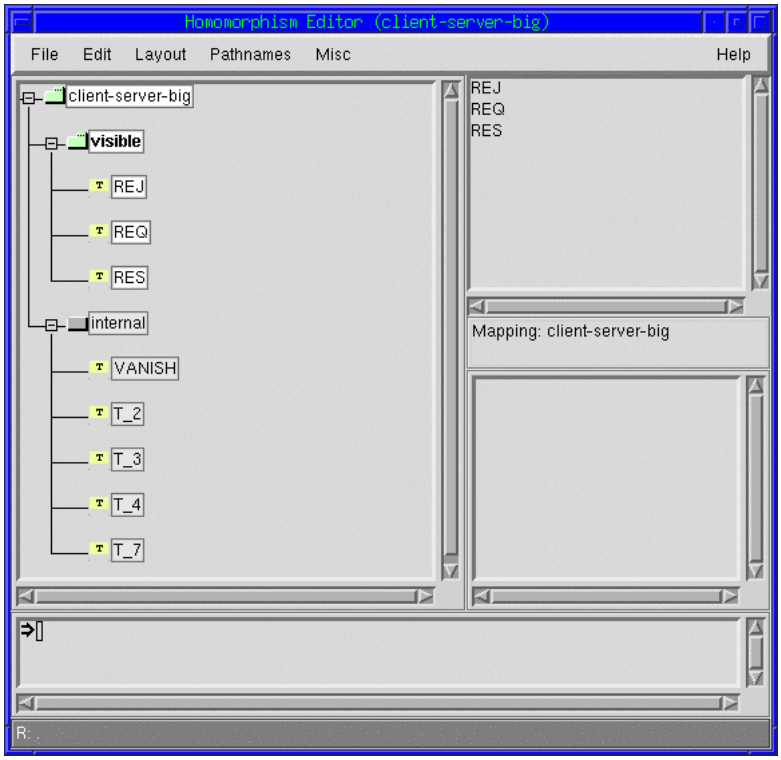

Figure 8: Homomorphisms Editor

An automaton [5] representing the abstract behaviour of the specification can be computed by the sh-verification tool (Figure 9). It obviously satisfies the required property. The next step is to check whether the concrete behaviour also satisfies the correctness requirement mentioned above. For that purpose we have to prove simplicity of the defined homomorphism.

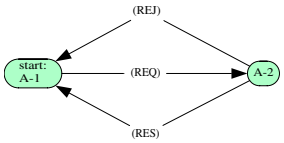

Figure 9:
Minimal Automaton.

Simplicity of an abstraction can be investigated inspecting the strongly connected components of the LTS by a sufficient condition [ORR00c]. The component graph in Figure 10 (combined with the homomorphic images of the arc labels of the corresponding graph components) does not satisfy this condition,

[5] *the minimal automaton*

so nothing can be said about simplicity.

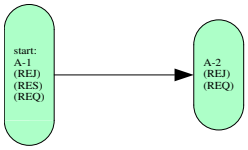

Figure 10: Component Graph

We now try to refine the homomorphism such that the sufficient condition for simplicity can be proven. Inspecting the edge between the two nodes of the component graph shows that the action $VANISH$ causes the transitions between this two components (Figure 11). The refined homomorphism, which additionally keeps $VANISH$ visible, satisfies the sufficient condition for simplicity. Figure 12 shows the corresponding automaton. This automaton obviously violates the required property, so the systems behaviour does not satisfy this property.

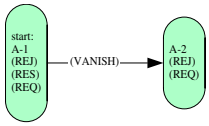

Figure 11: Component Graph

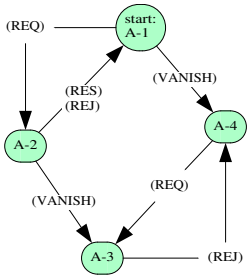

Figure 12: Minimal Automaton (with VANISH)

These simplicity investigations, which are supported by the tool, detect the error in the specification. In [Och92, Och94a] a necessary condition for simplicity is given. It is based on so called deadlock languages and shows non-simplicity of our *REQ-RES-REJ*-homomorphism [ORRN99].

To handle the well known state space explosion problem *a compositional method* [Och96] [ORR00a] is implemented in the sh-verification tool. This approach can also be used iteratively and provides a basis for induction proofs in case of 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 Temporal Logic

Our verification approach can also be combined with *temporal logic* [ORRN99]. In terms of temporal logic, the automaton of Figure 9 approximately satisfies [ORRN99] 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 6 does not.

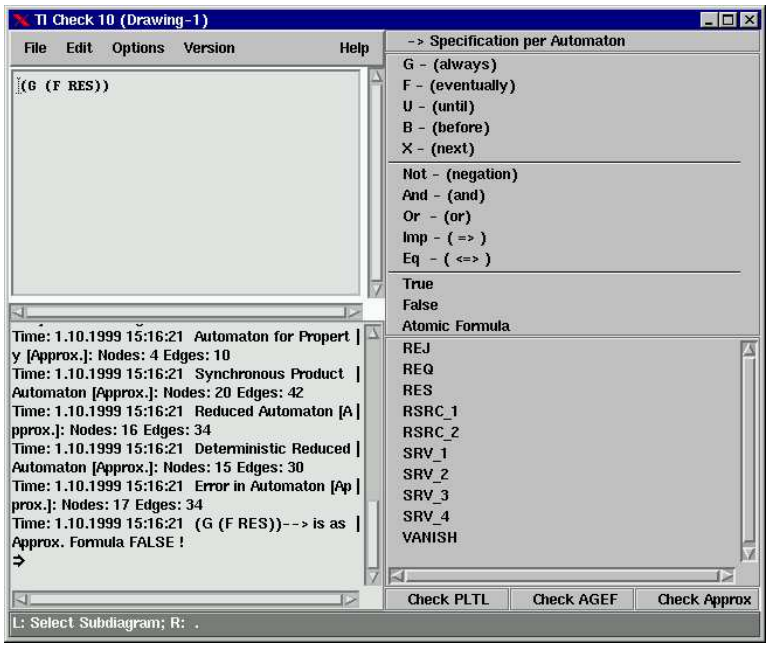

Figure 13: Temporal Logic Formula Editor

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.

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.

6 Applications

Practical experiences have been gained with large specifications:

- ISDN and XTP protocols [Klu92, Sch92, OP93]
- Smartcard systems [Neb94, Och94b]
- Service interactions in intelligent telecommunication systems [CDGE+96, CDF+96].

- The tool has also been applied to the analysis of cryptographic protocols [Bas99, Rud98, Rud99]. In this context an application oriented user-interface has been developed for input of cryptographic formulae and presentation of results in this syntax.
- Currently our interest is focused on the verification of binding cooperations including electronic money and contract systems. Recently some examples in that context have been investigated with our tool [Fox98, Roß98, Kap02].

7 Technical Requirements

The sh-verification tool is implemented in Allegro Common Lisp (currently for Linux and Windows NT). For more information please contact the authors.

8 Replay the Client Server Example

If you want to replay this example with the SHVT perform the following instructions:

1. Start the project manager.
2. Read file "[install-dir]/demo/demo.prj". The project tree with all demo examples will be shown.
3. Select node "demo APA > Client_Server_Example >" in the project tree (mouse-l).
4. Apply "mouse-r-Edit" to the file "clsrv-err.apa", which now is displayed in the second pane. The APA will be drawn in a new pane.
5. Command: "Parser>Load Preamble"
6. Command: "Parser > Analysis" will open the analysis window.
7. Command: "Start Exhaustive Analysis" - the rechability graph will be computed (Figure 6).
8. Command: "Homomorphism Editor" opens the editor for defining abstractions.
9. Read a predefined mapping from "clsrv1.map".
10. Command: "File > Compute Minimal Automaton" will compute the minimal automaton, which can be drawn with "mouse-r Draw Graph" on object "Minimal Automaton clsrc-err" (Figure 9).
11. "mouse-r Check Simplicity" on object "Minimal Automaton clsrc-err" will show: "no decision about simplicity can be made".

12. "mouse-r Determine Connected Components" on object "Reachbility Graph clsrv-err" will compute the connected components for further investigation.

13. "mouse-r Draw Graph" on object "Connected Components of Reachability Graph clsrv-err" will show the graph from Figure 10.

14. "mouse-r Show Objects" on the edge of this graph followed by "mouse-r Show Number of Origin Edges", "mouse-r Show Objects" on the current objects will display the origin edges with the transition vanish.

15. Use command: "Homomorphism Editor" from the analysis window to refine the abstraction.

16. Read a predefined mapping from "clsrv2.map".

17. Command: "File > Compute Minimal Automaton" will compute the minimal automaton, which can be drawn with "mouse-r Draw Graph" on object "Minimal Automaton clsrc-err" (Figure 12). The minimal automaton indicating the error is computed.

9 An alternative model of the client/server example

Here we show an alternative APA representation of the client/server example in Figure 5, consisting of three elementary automata, $\mathbb{E} = \{C, S, R\}$, and four state components, $\mathbb{S} = \{CS, IS, SS, RS\}$. Figure 14 shows the neighbourhood relation N.

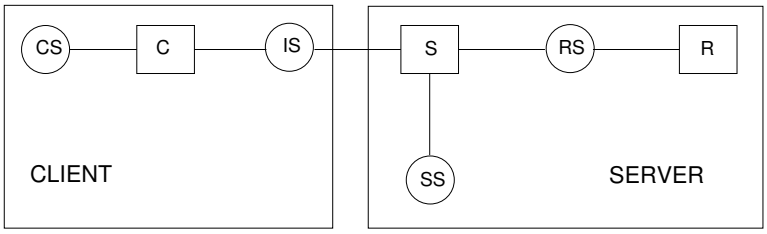

Figure 14: Client / Server APA

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:

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

This is represented in the tool in the preamble as:

```
defset Z_CS = { idle, active };
defset Z_SS = { idle, active };
defset Z_IS = { emp, req, res_rej };
defset Z_RS = { avail, navail, vanished };

defset Z_C = pro(Z_CS,Z_IS);
defset Z_S = pro(Z_SS,Z_IS,Z_RS);
defset Z_R = Z_RS;
```

imitial states:

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

alphabets:

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

This is represented in the tool in the preamble as:

```
defset PHI_C = {REQ, T7};
defset PHI_S = {RES, REJ, T4};
defset PHI_R = {VANISH, T2, T3};
```

state transition relations:

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

The corresponding function in the tool is:

```
defcase delta_C: pro (Z_C,PHI_C) >> Z_C
        delta_C (state,action) =
                if state = (idle,emp)       & action = REQ then (active,req),
                if state = (active,res_rej) & action = T7  then (idle,emp)
                else state;
```

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

The corresponding function in the tool is:

```
defcase delta_S: pro (Z_S,PHI_S) >> Z_S
    delta_S (state,action) =
        if state = (idle,req,avail)      & action = T4
           then (active,emp,avail),
        if state = (idle,req,navail)     & action = T4
           then (active,emp,navail),
        if state = (idle,req,vanished)   & action = T4
           then (active,emp,vanished),
        if state = (active,emp,avail)    & action = RES
           then (idle,res_rej,avail),
        if state = (active,emp,avail)    & action = REJ
           then (idle,res_rej,avail),
        if state = (active,emp,navail)   & action = REJ
           then (idle,res_rej,navail),
        if state = (active,emp,vanished) & action = REJ
           then (idle,res_rej,vanished)
        else state;
```

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

The corresponding function in the tool is:

```
defcase delta_R: pro (Z_R,PHI_R) >> Z_R
        delta_R (state,action) =
                if state = avail  & action = T3 then navail,
                if state = navail & action = T2 then avail,
                if state = navail & action = VANISH then vanished
                else state;
```

State components correspond to sets of state components in figure 5, as for example $emp \in Z_{IS}$ corresponds to $inactive \in Z_{S-4}$ and $inactive \in Z_{S-5}$. The alphabets' elements correspond to the elementary automata in figure 5. 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.

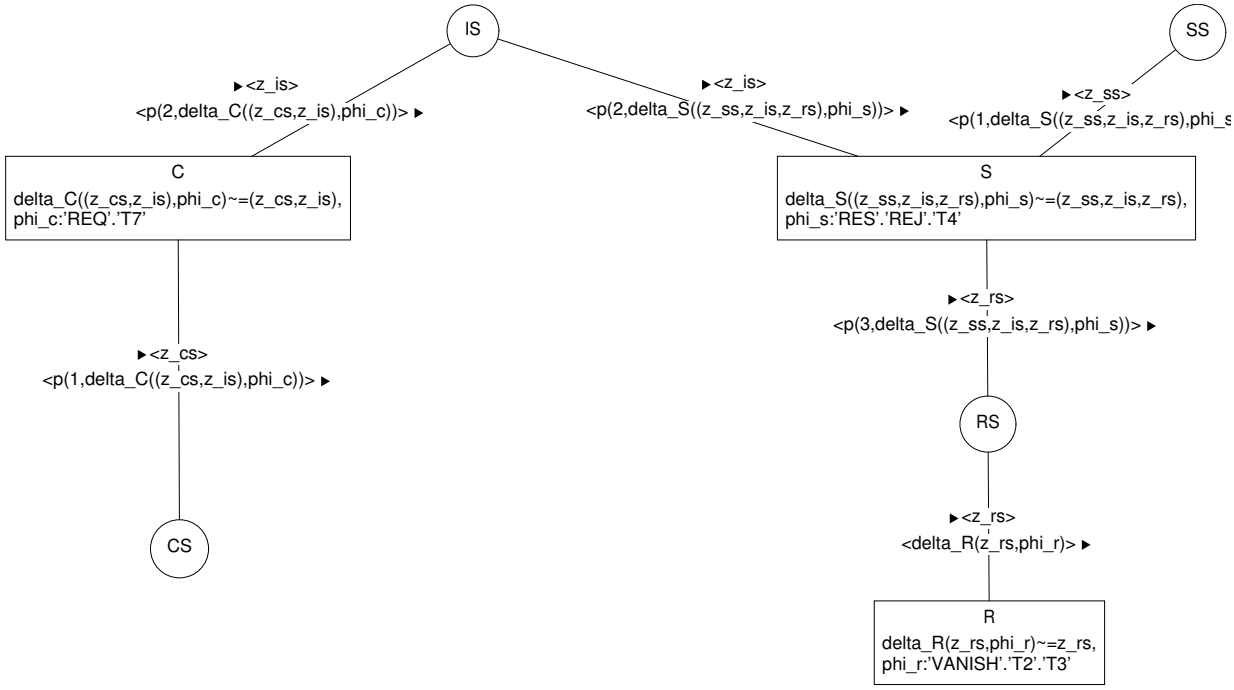

Figure 15: Client / Server APA

Figure 15 shows the Client/Server APA represented in the notation of the tool. The reachability graph of this APA is isomorphic to the LTS in Figure 6.

If you replay this example analog to the hints in section 8 (select node "demo APA > Simple_Client_Server_Example2 >") please note that the homomorphism used here (see figure 16) looks quite different. You have to use predicates to get a mapping corresponding to the one in figure 8.

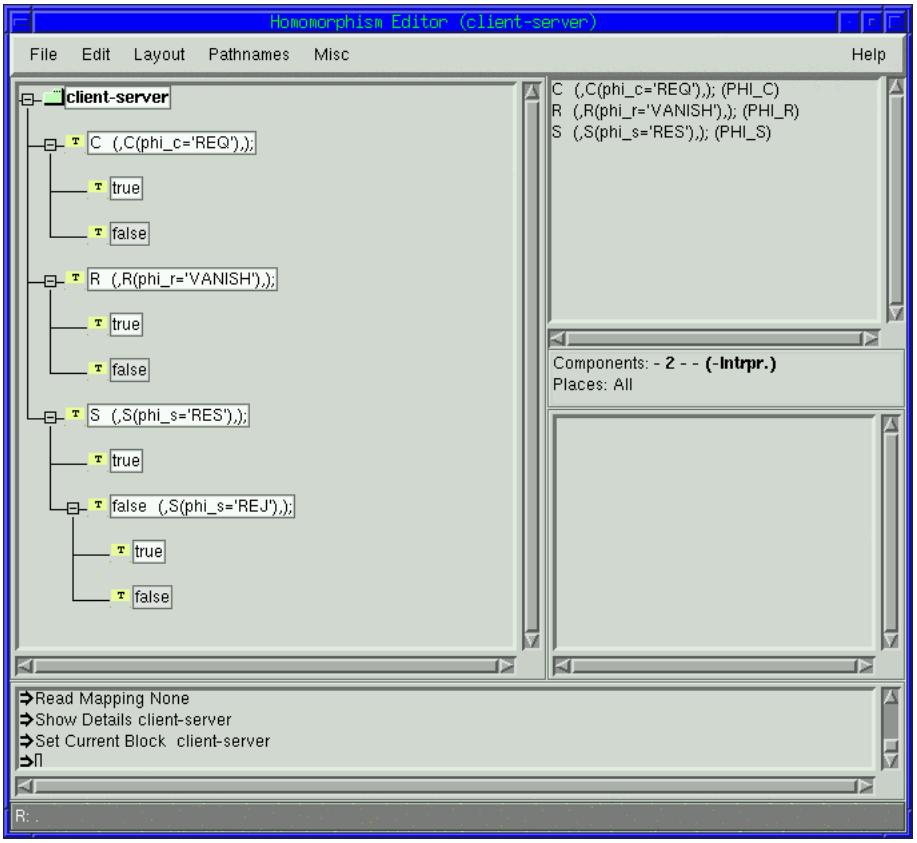

Figure 16: Client / Server Homomorphism

To make the result more readable the predicates are given short names that make the minimal automaton look like the one in figure 12.

References

[Bas99] G. Basak. Sicherheitsanalyse von Authentifizierungsprotokollen – model checking mit dem SH-Verification tool. Diploma thesis, University of Frankfurt, 1999.

[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.

[FhG] FhG – Institute for secure Telecooperation, Darmstadt. *Simple Homomorphism Verification Tool – Manual*.

[Fox98] S. Fox. Sezifikation und Verifikation eines Separation of Duty-Szenarios als verbindliche Telekoopertation im Sinne des Gleichgewichtsmodells. GMD Research Series 21, GMD – Forschungszentrum Informationstechnik, Darmstadt, 1998.

[Kap02] Kappes, S. SET (Secure Electronic Transaction) Formale Modellierung und Analyse des Bezahlvorgangs zwischen Kunde und Händler mit Produktnetzen basierend auf Annahmen des Gleichgewichtsmodells. Diploma thesis, University of Frankfurt (in preparation), 2002.

[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.

[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.

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

[Och94a] 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.

[Och94b] P. Ochsenschläger. Verifikation von Smartcard-Anwendungen mit Produktnetzen. In B. Struif, editor, *Tagungsband des 4. GMD-SmartCard Workshops*. GMD Darmstadt, 1994.

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

[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.

[ORR00a] P. Ochsenschläger, J. Repp, and R. Rieke. Abstraction and composition – a verification method for co-operating systems. *Journal of Experimental and Theoretical Artificial Intelligence*, 12:447–459, June 2000.

[ORR00b] P. Ochsenschläger, J. Repp, and R. Rieke. The SH-Verification Tool. In *Proc. 13th International FLorida Artificial Intelligence Research Society Conference (FLAIRS-2000)*, pages 18–22, Orlando, FL, USA, May 2000. AAAI Press.

[ORR00c] P. Ochsenschläger, J. Repp, and R. Rieke. Verification of Cooperating Systems – An Approach Based on Formal Languages. In *Proc. 13th International FLorida Artificial Intelligence Research Society Conference (FLAIRS-2000)*, pages 346–350, Orlando, FL, USA, May 2000. AAAI Press.

[ORRN99] Peter Ochsenschläger, Jürgen Repp, Roland Rieke, and Ulrich Nitsche. The SH-Verification Tool – Abstraction-Based Verification of Co-operating Systems. *Formal Aspects of Computing, The International Journal of Formal Method*, 11:1–24, 1999.

[Roß98] J. Roßmann. Formale Analyse der Business-Phase des First Virtual Internet Payment Systems basierend auf Annahmen des Gleichgewichtsmodells. Diploma thesis, University of Frankfurt, April 1998.

[Rud98] Carsten Rudolph. Analyse krypotgraphischer Protokolle mittels Produktnetzen basierend auf Modellannahmen der BAN-Logik. GMD Research Series 13/1998, GMD – Forschungszentrum Informationstechnik GmbH, 1998.

[Rud99] Carsten Rudolph. Automated Analysis of Cryptographic Protocols, A Tableau Method for Logics of Authentication. 1999. Submitted to *FLAIRS-2000* Special Track on Validation, Verification & System Certification.

[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.