ANALYZING NETWORK MANAGEMENT EFFECTS WITH SPIN AND CTLA

Gerrit Rothmaier, Andre Pohl, Heiko Krumm
Materna GmbH, Dortmund; LS 4, FB Informatik, Universität Dortmund

Abstract: Since many security incidents of networked computing infrastructures arise from inadequate technical management actions, we aim at a method supporting the formal analysis of those implications which administration activities may have towards system security. We apply the specification language *cTLA* which supports the modular description of process systems and facilitates the construction of a modeling framework. The framework defines a generic modeling structure and provides re-usable model elements. Due to *cTLA*'s connection to the temporal logic of actions *TLA*, formal analysis can resort to symbolic reasoning. Supplementarily, automated analysis can be applied. We focus here on automated analysis. It is supported by translation of *cTLA* specifications into suitable model descriptions for the powerful model checking tool *SPIN*. We outline the utilized methods and tools, and report on the modeling and *SPIN*-based analysis of *IP-Hijacking*.

Key words: security incidents, network management, formal analysis, *SPIN*, *cTLA*, *TLA*

1. INTRODUCTION

The current practice of those computer networks which form the basis of the IT infrastructure for companies and institutions shows, that a considerable part of the occurring security incidents has to be ascribed to vulnerabilities which result from inadequate technical management. One can perceive that network and system administrators are often overstrained by the complexity of the system structure, by the frequency of dynamic changes, and by the high number of existing service interdependencies. Moreover, since management operations can potentially control all components of a network, interferences of attacks and inadequate management may result in substantial vulnerability propagation and a

Y. Deswarte et al. (eds.), *Security and Protection in Information Processing Systems*

plethora of unforeseen threats. Therefore the security of IT-systems does not only depend on their suitable design, on the trustworthiness of their components and on the employment of suitable security services, but also on their proper technical management. Considering that background, our work focuses on the formal modeling and analysis of technical management effects in order to support clear descriptions of relevant attack procedures and management effects as well as a better understanding of the interrelations between management operations and attacks. Moreover, the work shall finally contribute to the discovery of new vulnerabilities and threats.

With respect to the type of modeling, we have to consider that management operations as well as attacks usually do not appear as atomic actions but rather as processes consisting of sequences of steps where each step performs a set of contiguous modifications of a system component. The interesting effects trace back to functional interactions between processes. Therefore we have to resort to a formal modeling technique for the functional aspects of concurrent process systems. The application of such modeling techniques for the analysis of security aspects is not new. In particular, they are applied in the context of protocol design for the verification of authentication protocols and in the context of trustworthy software system development for the verification of implementation code. In both fields the modeling regards rather narrow systems consisting of closely related elements which can mainly be represented in one major level of abstraction. Nevertheless, approaches of both fields report on the high complexity of models, on high costs and efforts for model design and analysis, and finally on the failure of automated analysis tools due to huge time and memory demands.

The application field of management operations and network-centered attacks imposes further requirements. Interesting scenarios comprise a broad spectrum of processes. Interactions can be relevant which concern operations on different abstraction levels. Therefore we attach special importance to the model specification technique. With respect to modeling it shall support the efficient description of broad models by means of composing re-usable specification modules. With respect to analysis, the defined models shall be concise and have a clear formal semantics. They shall be well-suited to the application of automated analysis tools, and – considering the limitations of these tools – they shall also support creative symbolic reasoning. Moreover, since most computer networks are very complex, there is a needs for partial modeling, i.e., one shall be able to focus the formal modeling on interesting parts and substructures of a network and the analysis of partial models shall supply clear contributions to the analysis of the network as a whole.

Our approach is based on the specification language *cTLA*, which supports the flexible and modular description of process systems [Her00]. In particular, in *cTLA* processes can describe the behavior of implementation components as well as general behavioral constraints (cf. [Vis88]). *cTLA*'s process composition has the character of superposition (cf. [Cha88, Kur93]), i.e. properties of subsystems are preserved in embedding systems. Therefore analysis results of partial models can be valid for systems as a whole. We already made good experiences with *cTLA* in the field of communication protocol verification where a formal modeling and verification framework was developed [Her00]. The framework consists of model architecture principles, re-usable specification modules, and theorems. It supports the efficient design of models as well as structured logical proofs of model properties. The semantics of *cTLA* specifications is directly defined by *TLA* formulas [Lam93] and the *TLA* methods for symbolic logical reasoning can be applied. Since *TLA* formulas refer to state transition systems, which is a well-understood and commonly used modeling technique, several suitable automated verification tools are available. Especially we translate *cTLA* specifications into the modeling language *Promela* and apply the corresponding model checker *SPIN* [Hol97], which is currently one of the most powerful and elaborated automated process system verification tools. We abstain from the direct use of *Promela* because it has a relatively complex C-language oriented semantics and does not directly support symbolic logical reasoning. Moreover, *Promela* specifications tend to be less abstract than *cTLA* specifications.

In the sequel we firstly refer to related work in the security analysis field. Then, after outlining the specification language *cTLA*, we describe the translation of *cTLA* specifications into *SPIN*/*Promela* model definitions. The next section explains the generic model structure, all models of our approach shall comply with. Thereafter, the example problem "*IP Hijacking*" is introduced. Its modeling and *cTLA*-based model specification are discussed. Finally, we report on the *SPIN*-based analysis of this model. Concluding remarks refer to the current state and planned developments. More details of the work are described in [Rot03].

2. RELATED WORK

There is a wide spectrum of work which uses formal modeling and analysis of execution and interaction processes for the verification of security properties. In the main, program verification techniques are applied to enhance the trustworthiness of software systems (e.g. [Bal00]) and protocol verification techniques to detect vulnerabilities of cryptographic

protocols (e.g. [Mea95]). In both application fields, a variety of basic methods is employed covering classic logic and algebraic calculi (e.g. [Kaw03]), special calculi (e.g. [Bur89]), and process system modeling techniques (e.g. [Led99]). Many approaches consider tool support in order to facilitate the verification tasks and ensure the correctness of the reasoning. Most tools take advantage of existing more general analysis tools like Prolog, expert system shells, theorem provers, algebraic term rewriting systems, and reachability analysis based model checkers. Some powerful tools combine several analysis techniques [Mea96].

Formal model checking based analysis focusing on network management effects, network attacks, and vulnerability propagation is relatively new. [Ram98, Ram02] report on the analysis of computer systems for those vulnerabilities, which result from the combined behavior of system components. The underlying model is a composition of processes representing the components' and user's security-related behavior. The processes are specified in a special high level language and translated to Prolog. Mainly, simple security properties are modeled by labeling states safe or unsafe. A custom model checker built on top of a Prolog environment searches execution sequences which lead to unsafe states and correspond to vulnerability exploitations.

[Amm00] reports on checking networks of hosts. A network is statically modeled by a set of hosts, a set of vulnerabilities per host, an attacker access level per host, and a network connectivity matrix. The analysis investigates the possible combinations vulnerabilities and their stepwise propagation in the network. The model is implemented using the well-known model checker *SMV*. In [Noe02] this approach is extended and called topological vulnerability analysis. The model uses a multi-valued connectivity matrix which supports the representation of low level communication exploits.

3. CTLA

cTLA is based on TLA [Lam93]. In comparison with TLA it supplies explicit notions of processes, process types, and process composition [Her00]. Furthermore, there is a different look of *cTLA* specifications since canonical parts are not explicitly written down. A wide variety of data types can be defined, in particular those with finite (and preferably small) value sets in order to support the *SPIN*-based model checking.

3.1 TLA

TLA [Lam93] describes state transition systems by means of a so-called canonical formula. A state transition system STS ::= < S, S_0, T > is defined by a set of states S, a set of initial states $S_0 \subset S$, and a set of transitions $T \subset S \times S$. A canonical TLA formula f_{Sys} ::= *Init* $\wedge$ $\Box$ [*Next*]$_V$ $\wedge$ *FA* directly addresses the state transition system model. V is a set of state variables (e.g. $V = \{x_1, x_2, \ldots, x_n\}$) spanning the state space S. *Init* denotes the *Init*-predicate defining the set of initial states S_0. *Next* denotes the next state predicate which defines the set of transitions T and is structured into a set of actions: *Next* ::= $act_1 \vee act_2 \vee act_3 \ldots \vee act_m$. The always-subformula $\Box$ [*Next*]$_V$ means that each pair of two consecutive states $<s_i, s_{i+1}>$ of each execution state sequence of *Sys* has to fulfil the formula: *Next* $\vee$ ($x_1'=x_1 \wedge x_2'=x_2 \wedge \ldots \wedge x_n'=x_n$), i.e. the state pair has to comply with one of the actions of *Next* or represent a so-called stuttering step where s_i equals s_{i+1}. *FA* specifies liveness properties by a conjunction of fairness assumptions *FA* ::= WF(act_j) $\wedge$ WF(act_k) $\wedge \ldots \wedge$ SF(act_p) $\wedge \ldots \wedge$ SF(act_q). A weak fairness assumption WF(act_j) assumes, that the action act_i has to be executed in situations, where the action is enabled and continuously will be enabled until its execution. A strong fairness assumption SF(act_j) assumes, that the action act_i has to be executed, if the action will be enabled again and again until its execution.

TLA facilitates formal verification. It supplies inference rules supporting the deduction of implications of the form $f_{Sys} \Rightarrow f_{Prop}$, where the formula f_{Sys} describes a system *Sys* in more detail and the formula f_{Prop} represents a more abstract property of *Sys*. The deduction of the implication formally verifies that the system *Sys* has the property *Prop*.

3.2 cTLA Simple Process Type

Each *cTLA* specification module describes a process type, an instance of which corresponds to a state transition system modeling a process of this type. A *simple process type* declaration defines a state transition system directly. As an example, the following type *Media* models the physical packet transfer of a local area network and is part of the "*IP Hijacking*" model:

```
PROCESS Media();
  VAR buf: ARRAY[MAXZONES] OF PacketBufT;
  INIT::= FORALL i:MAXZONES: [ buf[i].usd = FALSE ];
ACTIONS
  in( pkt: PacketT ) ::=
   fSrcToZone( pkt.scn, pkt.sci ) != UNKNOWN_ZONE
   AND buf[ fSrcToZone( pkt.scn, pkt.sci ) - 1 ].usd = FALSE
```

```
    AND buf[ fSrcToZone( pkt.scn, pkt.sci ) - 1 ].usd' = TRUE
    AND buf[ fSrcToZone( pkt.scn, pkt.sci ) - 1 ].pkt' = pkt;
  out( pkt: PacketT ) ::= ...
END
```

The state space of the state transition system is spanned by state variables. In *Media* the variable `buf` is an array of packet buffers. The set of initial states is defined by the predicate `INIT` which in the example claims that all buffer fields are flagged as unused. The actions directly constitute the next state relation. Action `in` applies to those state transitions which model that the Media accepts and transfers a sent packet `pkt`. Aktion `out` models that a packet `pkt` is delivered to a set of receiving nodes. Syntactically, actions are predicates over state variables (referring to the current state, e.g. `buf`), so-called primed state variables (referring to the successor state, e.g. `buf'`), and action parameters (e.g. `pkt`).

A subset of actions can be classified as so-called internal actions. An internal action defines a set of state transitions in exactly the same way as a normal action. The difference between both sorts of actions concerns the composition of systems from processes. When a process instance is employed as a component in a system, the internal actions of the process cannot be coupled with actions of other processes. Each internal action is accompanied by stuttering steps of all other system components.

With respect to liveness properties, it has to be mentioned, that currently fairness assumptions are not used since we concentrate on safety properties.

3.3 cTLA Process Composition Type

A process composition type describes a process system which is composed from process instances of imported types. It is exemplified by the following type *IpArpExample* which models a small local computer network and is part of the "*IP Hijacking*" model:

```
PROCESS IpArpExample();
CONTAINS
  med: Media();   bnA: IpArpNode( NA_ID, NA_MII );
                  bnB: IpArpNode( NB_ID, NB_MII );
                  bnC: IpArpNode( NC_ID, NC_MII );
ACTIONS   /* send system actions */
  snd_A( pkt: PacketT ) ::= bnA.snd( pkt ) AND med.in( pkt );
  snd_B( pkt: PacketT ) ::= bnB.snd( pkt ) AND med.in( pkt );
  snd_C( pkt: PacketT ) ::= bnC.snd( pkt ) AND med.in( pkt );
          /* receive system actions */
  rcv_A( pkt: PacketT ) ::= bnA.rcv( pkt ) AND med.out( pkt );
  rcv_B( pkt: PacketT ) ::= bnB.rcv( pkt ) AND med.out( pkt );
  rcv_C( pkt: PacketT ) ::= bnC.rcv( pkt ) AND med.out( pkt );
          /* broadcast receive system actions */
  rbc( pkt: PacketT ) ::=
```

```
            bnA.rbc( pkt ) AND bnB.rbc( pkt ) AND bnC.rbc( pkt ) AND med.out( pkt );
    /* implicitly added: internal actions of the nodes, e.g. rpcs, spcs */
END
```

An *IpArpExample* network contains four process instances, one process `med` of the type *Media* which is described above, and three node processes `bnA`, `bnB`, and `bnC` of type *IpArpNode*. These processes are coupled according to the system actions (e.g. `snd_A`). The right side of a system action is a conjunction of process actions (e.g. `bnA.snd(pkt)`, `med.in(pkt)`) and lists those process actions which have to be performed jointly in order to realize a system action. Each process can contribute to one system action by at most one process action. If a process does not contribute to a system action, it performs a stuttering step. In the special *cTLA*-variant used here, process stuttering steps are not explicitly listed in the right side of system actions.

The state transition system which models an instance of a process composition type, is defined indirectly. Its state space is spanned by the vector of all state variables of all contained processes. Its *Init*-predicate is the conjunction of the *Init*-predicates of the contained processes, and its *Next*-predicate is the disjunction of the system actions. In this way the concurrent execution of processes is modeled according to interleaving semantics, while process interactions are modeled by joint actions where two or more processes simultaneously perform non-stuttering steps. System actions and process actions can be supplied with data parameters supporting the communication between processes.

The TLA formula describing a *cTLA* process system is equivalent to a conjunction of the process formulas and consequently a system implies its constituents. The process composition of *cTLA* has therefore the character of superposition (cf. [Cha88, Kur93]) which guarantees that a property fulfilled by a process or a subsystem is also a property of each system which contains this process or subsystem. In particular, superposition supports the so-called structured verification. In order to prove that a system *Sys* has a property *Prop* it is sufficient to find a subsystem of *Sys*, where *Prop* can be proven.

4. TRANSLATION TO SPIN/PROMELA

The model checker *SPIN* accepts models written in the *Promela* (Process Meta Language) specification language [Hol97]. *Promela*'s basic syntax resembles the *C* programming language. But most of the computational functions have been removed and constructs for modeling process systems have been added. These constructs are similar to Hoare's *CSP* (Communicating Sequential Processes) but more flexible. Process types can

be declared and dynamically executed through instantiation. Communications between processes are modeled with special channels or through shared variables. Guarded statements can be used for synchronisation.

We developed a compiler, *CTLA2PC* (*cTLA* to *Promela* Compiler), for translating high level specifications written in *cTLA* into the more low level *Promela* language. *CTLA2PC* generates highly optimized *Promela* code and supports various switches. For example the `-optbitarrays` switch can decrease the state vector size considerably in comparison to a native *Promela* model (cf. Sec. 7.1).

The general steps of the translation process are as follows. First *CTLA2PC* builds a parse tree from the *cTLA* input. Using this tree the *compositional system* is then expanded to an equivalent *flat system.* The flat system representation is necessary because *Promela* specifications don't support systems of composed processes coupled through system actions. In the flat system only a single process with all actions exists. This process contains the system initialization (merged from all processes) as well.

The flat system is used for all further steps. Different back-ends can be used to generate code in a target language. A simple *cTLA* backend just outputs the flat system definition (switch `-ctla`). But for *Promela*, more transformations have to be done. Processes can't contain multiple actions in *Promela. CTLA2PC* therefore generates a non-deterministic selection with appropriate guards for each action. Furthermore, *Promela* doesn't support action parameters like *cTLA*. These parameters are implicitly existentially quantified. Hence *CTLA2PC* creates extra input generator processes in *Promela* and shared variables which replace the parameters. Each input generator uses a non-deterministic selection loop over all possible values of the parameter. The value selected is written to the parameter variable. An alternative approach would be to use *SPIN*'s channels instead of parameter variables for passing the value, but in our experiments *SPIN* needed more state space that way.

Many more points not mentioned here due to space reasons have to be taken into account during the compilation process. These include particularly the mapping of data type definitions, functions, and predicates.

5. GENERIC MODEL STRUCTURE

The goal of our modeling is to have a generic structure suitable for describing computer networks and applying automated verification techniques. It shall be abstract enough to hide low level details which would quickly enlarge the state space and make verification impossible. But it must

allow specific models to be derived easily and augmented with more detailed behavior required by the example, e.g. *ARP* protocol handling. Thus we choose a middle level of abstraction which can be extended both for low level packet oriented modeling and high level service oriented processes.

The modeling shall be as natural as possible. From a bird's eye view, a real computer network consists of several nodes transmitting packets over media. Thus these are the basic elements of our modeling (cf. *Fig.* 1). A *Node* process models any object (host, printer, ...) that sends (action `snd`) or receives (action `rcv`) packets with an interface over media. The processing of packets takes a layered approach. Which layers (e.g. *Network Interface*, *IP/ARP*) are needed depends on the specific model, but the generic background is as follows. For both sending and receiving, each layer has its own packet buffer and appropriate actions for processing (`spcs`, `rpcs`). After a packet arrives at the layer the processing action can be activated. The processing of the packet may either terminate within this layer or the packet may be marked as "ready" to be passed to the next layer. Corresponding actions (`smov`, `rmov`) for moving "ready" packets between the layers have to be included in the specific model. These actions as well as the processing actions belong to the internal actions of the node.

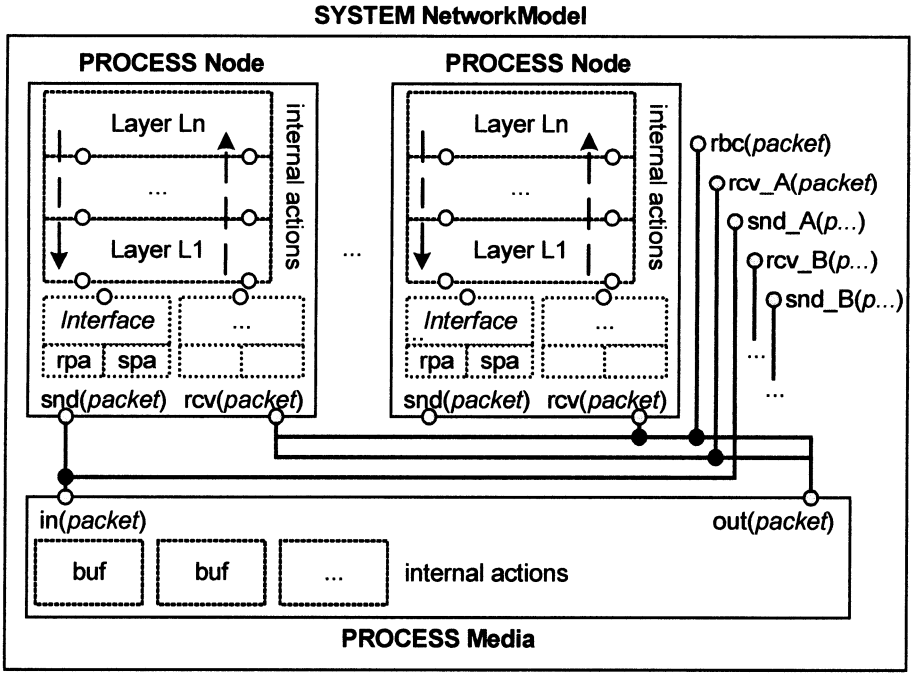

Figure 1: Generic model stucture

Because the generic model is independent of addressing schemes (e.g. Ethernet hardware addressing), no constraints on received packets (e.g. remove the packet if it isn't a broadcast packet and doesn't match the interface's address) are defined. These constraints are included in the low level packet processing of specific models. Similarly, stamping of packets with a specific source address (other than the model internal source node, source interface attributes) before sending has to be done there as well.

Each node has to have at least one interface, but may have multiple interfaces (e.g. a router node). Interfaces can be dynamically enabled or disabled (cf. *Unix ifconfig up/down*). Each interface has a buffer (variable `rpa`) to hold a packet after it has been received and before it is processed by the network stack on the node. Similarly each interface has a buffer (variable `spa`) to hold a packet that shall be transmitted to the media.

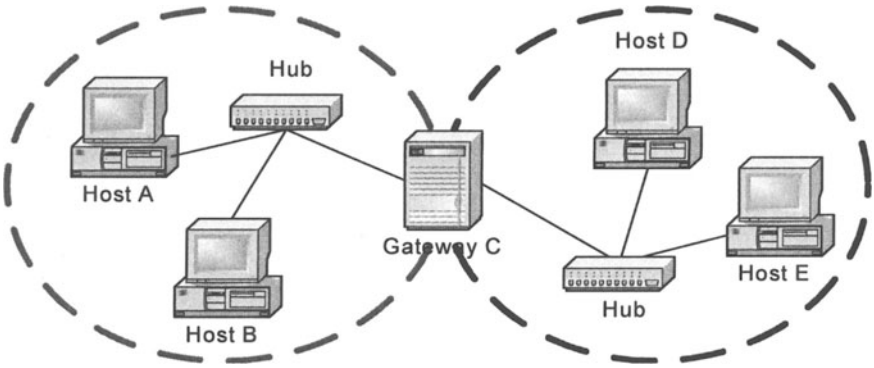

Figure 2: Example network

All interfaces which can directly reach other interfaces via the link layer (i.e. without using higher level, e.g. *IP*, routing functions on a node) belong to a *zone*. Because a node may have multiple interfaces, a node can be in multiple zones. A low level broadcast by a node with a selected interface will reach all others nodes with an interface in the zone. Zones are assumed to be static, i.e. a node may not be dynamically equipped with a new interface (but may enable or disable a previously existing interface).

The *Media* process is used for packet exchanges. For each zone *Media* buffers (variable `buf`) one packet currently in transit in the media. Only one packet can be in transit in any zone at one time. A node can only transmit a packet over an interface to the media (action `in`) if no packet is already in transit for the zone. Similarly a packet can only be received by a node over an interface from the media (action `out`) if a packet is currently in transit for the zone.

Within the generic model packets just contain two internal attributes for determining the zone of the packet. Without higher level routing a packet can't move between zones (by definition). For a specific model further attributes are typically added to packets.

The *System* process associates node and media instances. It defines system actions which couple actions of process instances. Each node *N* can send a packet to media (action `snd_N`, coupling node *N* and *Media*) or receive a unicast packet from *Media* (action `rcv_N`). Furthermore a broadcast packet may be received by all nodes in a zone (action `rbc`, coupling *Media* and all nodes of a zone).

6. EXAMPLE SYSTEM

The modeling structure described above provides a basis for concrete instances. Consider the example depicted in *Fig.* 2.

Two network zones are connected by a gateway. Zone 1 contains hosts *A*, *B* and the gateway *C*, zone 2 contains hosts *D*, *E* and the gateway *C*. Only the gateway has two interfaces. All hosts within a zone can directly reach each other without using the gateway. The hosts communicate over *IP*. We want to analyze the effects if an administrator (or ignorant user) hijacks an *IP* address (e.g. of the gateway) by changing the *IP* address of a host to an already used one.

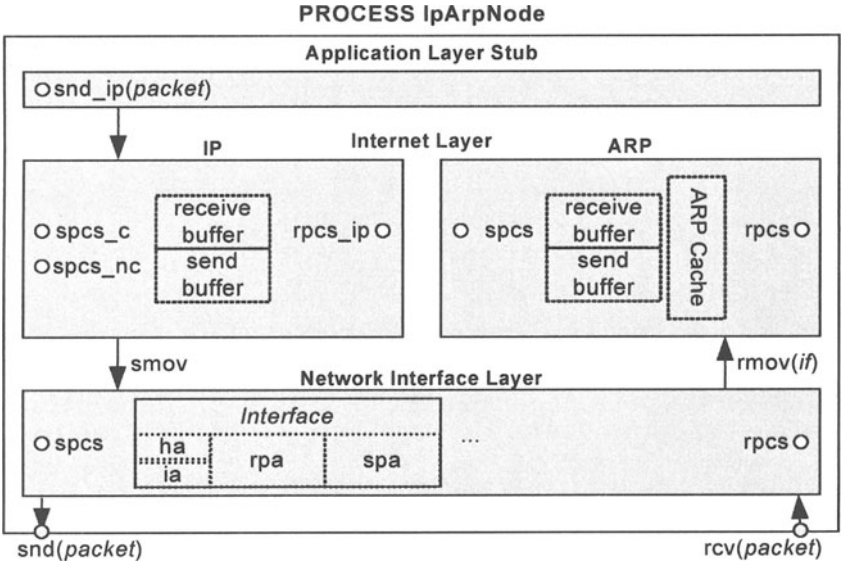

Figure 3: IpArpNode

For the modeling of this example we use the abstract node element as starting point. It has to be augmented with appropriate behavior and attributes for *IP* communication. As described in the generic model *IpArpNode*'s actions (cf. *Fig.* 3) are layered, in this case akin to the *TCP/IP* reference model [Com01]. The layers *Network Interface*, *Internet* (including *IP* and *ARP*) and a minimal *Application Layer Stub* are needed.

Submitted packets from the application (action `snd_ip`) are stored in a send buffer. Then the packet can be processed by the *IP* layer. Basically *IP* addresses have to be resolved to hardware addresses. If the *IP* address to hardware address assignment is not available within the cache, a broadcast *ARP* query is created (action `spcs_nc`), processed through the layers (especially *ARP*) and sent (action `snd`). After a matching *ARP* reply (or a query by the sought-after node) has been received and processed, the *ARP* cache is updated. Then the *IP* packet can be processed and finally sent. If the address was resolved before and is still in the *ARP* cache no *ARP* request is generated by the *ARP* layer (action `spcs_c`).

Received packets (action `rcv`) are stored in the interface's buffer and processed layer by layer as defined in the generic model. For this model, the low-level *Network Interface* layer checks the packet's hardware destination address against the interface's hardware address (if the interface is neither in promiscuous mode nor the packet is a broadcast packet). Invalid packets are thrown away immediately and don't reach the *Internet* layer. Similarly, the send packet processing (action `spcs`) has been extended to set the hardware source address of the packet to the hardware address of the interface.

Besides the layers and their actions each of *IpArpNode*'s interfaces is extended with a hardware (variable `ha`) and an *IP* address (variable `ia`). The initialization of *IpArpNode* has to be enhanced to include these as well. Furthermore the packet structure is augmented for the *Network Interface* layer with hardware source and destination address and frame type. Depending on the frame type, a packet has either *ARP* or *IP* attributes on *Internet* layer. *ARP* packet attributes include a query type and pairs of hardware and protocol addresses. *IP* packet attributes are the *IP* source and destination addresses.

To model the administrator, we introduce a new action for changing the *IP* address of a host to another address (action `chg_ip`). The system initialization is generated by *CTLA2PC* from the processes initialization automatically. So no further explicit system initialization action has to be added.

7. ANALYSIS

Most of the time model checking is a challenging task. Even small models quickly exceed given time and memory constraints. Marginal model additions can have a strong adverse impact on the performance of the verification tool. This effect is well-known as state space explosion.

7.1 Example System Optimizations

In order to enable automated analysis, we started with two major model optimizations. First, we realized that a smaller subsystem suffices to study the effects of the administrator's actions. The two zones are similar, so modeling just one zone, e.g. zone 1 with the hosts *A*, *B*, and *C*, satisfies our requirements. Due to the superposition property of *cTLA*'s process composition, the results can be carried over to embedding systems. Also, because all hosts are of the same type, we can assume that a certain host, namely *B*, changes its *IP* address to the address of host *C*.

Second, we apply an approach of the field of efficient protocol implementation in order to save on buffers and action steps. We refer to the *activity thread* approach which combines all those actions of several protocol layers into one integrated and non-interruptible execution thread, which processes the same stimulating packet request [Svo89]. Accordingly, we modify the host model by merging actions and eliminating interface buffers. As a result, just one processing action and working buffer for each direction (sending and receiving) remains. Furthermore we include the basic interface setup of the nodes (i.e. activation and assignment of original *IP* addresses) into the system initialization to save some steps.

To evaluate the possibility of checking the model we translated it to *Promela* using *CTLA2PC* and performed several *SPIN* runs. The state vector size was about 250 bytes. A *DFS* search depth of over one million was reached within a minute and *SPIN* quickly used up over 512 MB of memory. Approximate (supertrace and bit-state-hashing) *SPIN* verification modes prevented the memory overrun but did not give any results in a reasonable amount of time. One of these runs was cancelled after over 150 hours.

Hence we looked at the *cTLA* model again in order to cut down the state vector size further. In particular the action parameters were promising. *cTLA* action parameters correspond to existentially quantified value variables, and their translation to *Promela* introduces extra processes and state variables (cf. Sec. 4). We observed that for most of the action parameters value determining equalities exist, since many parameters of flat system actions serve as output parameters of constituting process actions. Therefore it is possible to replace these parameters with the corresponding symbolic output

value. In turn this "paramodulated" version of the model was translated to *Promela*. Now the *SPIN* state vector size was shortened to about 210 bytes. Additionally we improved the code generation of *CTLA2PC* to consider specific oddities of *SPIN*. In particular, *SPIN*'s built-in handling of arrays of those variables which use only a few bits, requires much more state space than necessary. With the switch `-optbitarrays` *CTLA2PC* wraps such an array into an integer type following a generalized version of the bitvector approach suggested by [Ruy01] and appropriately maps all read and write accesses to array elements. This led to a further considerable reduction of the state vector to a size of 168 bytes.

7.2 Checking Assertions & Analyzing Trails

Based on this version we wanted to check the property that a node cannot receive non-broadcast *IP* packets destined for other nodes. We inserted the following assertion at the end of the non-broadcast receive action of each node:

```
assert( !(bnN_ifs[1-1].rpa.pkt.l2t==L2_IP &&
          BVGET(bnN_ifs[1 - 1].rpa.pkt.dat, 3, DI_IDA) != bnN_ifs[1 - 1].ia ));
```

It checks that a received *IP* packet's destination address equals the interface's *IP* address that received the packet. We had in mind, that through a change of the *IP* address of node *B* to the *IP* address of node *C* (*IP* hijacking) and a subsequent *ARP* query by *B* the entry for *C* in the *ARP* cache of the other nodes might be updated with *B*'s hardware address (*ARP* cache poisoning). This could lead to further packets intended for *C* to be received by *B* and violate the assertion. So we started a *SPIN* verification run. As expected the assertion was violated:

```
pan: assertion violated  !(((bnB_ifs.rpa.pkt.l2t==1) && (((bnB_ifs.rpa.pkt.dat>>(1*3)) &&
                         ((1<<3)-1))!=bnB_ifs.ia)))          (at depth 12)
pan: wrote ip-arp-example-verif-comp-flat-para-bitopt.promela.trail
(Spin Version 4.1.0 -- 6 December 2003)
   Warning: Search not completed + Using Breadth-First Search + Partial Order Reduction
   ...
State-vector 168 byte, depth reached 12, errors: 1
```

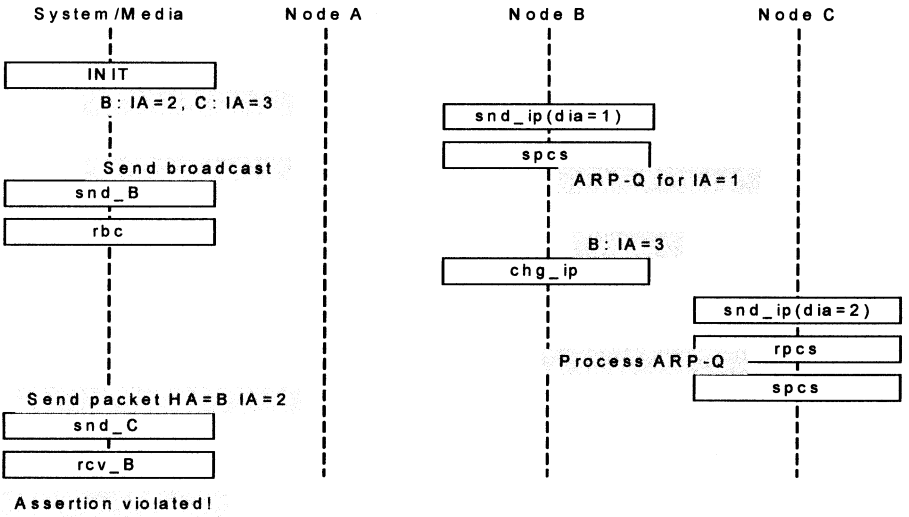

Figure 4: Violating path

But subsequent guided simulation with the created trail file revealed that the violation occurs in an unforeseen way (cf. *Fig.* 4). Node *B* changed its *IP* address (`chg_ip`) to 3 but has not yet updated the *ARP* cache of another node. Instead, a previously received (`rbc`) *ARP* query from node *B* still contains the old *IP* address and node *C* updates its *ARP* cache from this query (`rpcs`) just now. Thus an already buffered *IP* packet for the (now unused) *IP* address 2 can be processed (`spcs`) with the cached *IP* address to hardware address assignment. The packet is submitted to *Media* and received by node *B* (due to the matching hardware address). But the interface's *IP* address is different and this violates the assertion. A typical (non promiscuous) *IP* stack would throw away such a packet. Thus this demonstrates the simple fact that an *IP* change may lead to lost packets until all nodes in the subnet have updated their *ARP* caches correctly. This is especially relevant to hosts with static *ARP* configurations since their settings have to be maintained manually.

To analyze a scenario analogous to the originally intended situation another assertion can be used, which has to be included in the send action of node *A*:

```
assert( !(bnA_ifs[1 - 1].spa.pkt.l2t==L2_IP &&
        BVGET( bnA_ifs[1 - 1].spa.pkt.dat, 3, DI_IDA) ==
                bnC_ifs[1 - 1].ia && bnA_ifs[1 - 1].spa.pkt.dha != bnC_ifs[1 - 1].ha ));
```

The resulting trail file shows node *B* immediately changing its *IP* address and poisoning the *ARP* cache of the other nodes as described above. Then the assertion is violated by node *A* sending an *IP* packet intended for node *B* to node *C*.

8. CONCLUDING REMARKS

Technical management processes and their security implications constitute a new and practically relevant application field for formal modeling and analysis. It, however, has distinct demands. The modeling results in comprehensive models and shall therefore be supported by a modular and flexible specification technique, while on the other hand analysis tools shall be able to uncover unknown effects automatically. As shown by means of the *IP-Hijacking* example, the proposed combination of modeling guidelines, the *cTLA* specification language, and *SPIN*-based analysis is a feasible approach which can substantially contribute to improve the understanding of management effects.

Current work studies intertwined network and security service reconfiguration processes. Moreover we focus on an enhancement of the technique. The modeling framework will be completed by re-usable process type definition modules. The analysis of complex models will be supported by model architecture principles which are borrowed from the field of efficient communication protocol implementation. The provision of improved tool support will be based on the integration of *cTLA* and *SPIN* into an *Eclipse*-based development environment (cf. [Ecl02]).

We thank the anonymous reviewers for their valuable suggestions directing our future work towards methodic modeling abstractions for complex systems.

REFERENCES

[Amm00] P. Ammann, R. Ritchey: *Using Model Checking to Analyze Network Vulnerabilities*. IEEE Symposium on Security and Privacy, May 2000.

[Bal00] M. Balser, W. Reif et al.: *Formal System Development with KIV*. In: T. Maibaum (ed.), Fundamental Approaches to Software Engineering. Springer LNCS 1783, 2000.

[Bur89] M. Burrows, M. Abadi, R. Needham: *A Logic of Authentication*. In: Proceedings of the Royal Society, Volume 426, Number 1871, 1989, und in: William Stallings, Practical Cryptography for Data Internetworks, IEEE Computer Society Press, 1996.

[Com01] D. E. Comer, R. E. Droms: *Computer Networks And Internets*. Prentice Hall, 2001.

[Cha88] K.M. Chandy, J.Misra: *Parallel Program Design – A Foundation*, Addison-Wesley, Reading, 1988.

[Ecl02] Eclipse.org: *Eclipse Project FAQ*. URL: http://www.eclipse.org/eclipse/faq/eclipse-faq.html, 2002.

[Her00] P. Herrmann, H. Krumm: *A framework for modeling transfer protocols*. Computer Networks, 34(2000)317-337.

[Hol97] G.J. Holzmann: *The model checker spin*, IEEE Trans. Soft. Eng. 23 (1997) 279-295.

[Kaw03] K. Kawauchi, S. Kitazawa et al.: *A Vulnerability Assessment Tool Using First-Order Predicate Logic*. IPSJ SIGNotes Computer SECurity No.019 (2003)

[Kur93] R. Kurki-Suonio: *Hybrid models with fairness and distributed clocks*, in: Lecture Notes in Computer Science, Vol. 736 (Springer, New York, 1993) pp. 103–120.

[Lam93] L. Lamport: *The temporal logic of actions*, ACM Trans. Prog. Lang. and Sys. 16 (1994) 872-923.

[Led99] G. Leduc, O. Bonaventure, L. Leonard, E. Koerner, C. Pecheur: *Model-based verification of a security protocol for conditional access to services*. Formal Methods in System Design, Kluwer Academic Publishers, vol.14, no.2, March 1999 p.171-91.

[Mea95] C. Meadows: *Formal Verification of Cryptographic Protocols: A Survey*. In: Proc. Asiacrypt, Intern. Conf. on Theory and Application of Cryptology, LNCS, Springer-Verlag, 1995.

[Mea96] C. Meadows: *The {NRL} Protocol Analyzer: An Overview*. Journal of Logic Programming, 26, 2 (1996) pp. 113-131.

[Noe02] S. Noel, B. O' Berry, R. Ritchey: *Representing TCP/IP connectivity for topological analysis of network security*. Computer Society, IEEE (ed.), Proceedings of the 18th Annual Computer Security Applications Conference, Dec. 2002, p. 25-31.

[Ram02] C. Ramakrishnan, R. Sekar: *Model-Based Analysis of Configuration Vulnerabilities*. Journal of Computer Security, Vol. 10, Nr. 1, Jan. 2002, p. 189-209.

[Ram98] C. Ramakrishnan, R. Sekar: *Model-Based Vulnerability Analysis of Computer Systems*. Second International Workshop on Verification, Model Checking, and Abstract Interpretation, Pisa, Italy, Sep. 1998.

[Rot03] G. Rothmaier, H. Krumm: *Formal Modeling of Security Properties of Computer Networks*, Internal Technical Report, RvS Group, FB Informatik, University of Dortmund, URL: http://ls4-www.cs.uni-dortmund.de/RVS/P/ForSecMod.pdf, 2003.

[Ruy01] T. C. Ruys: *Towards Effective Model Checking*. PhD Thesis, Univ. Twente, 2001.

[Svo89] L. Svobodova: *Implementing OSI Systems*. IEEE Journal on Selected Areas in Communications 7 (7),(1989), pp.1115–1130.

[Vis88] C.A. Vissers, G. Scollo, M. van Sinderen: *Architecture and specification style in formal descriptions of distributed systems*, in: S. Agarwal, K. Sabnani (Eds.), Proceedings of the PSTV VIII, Elsevier, Amsterdam, 1988, pp. 189-204.