Using SPIN and Eclipse for Optimized High-Level Modeling and Analysis of Computer Network Attack Models

Gerrit Rothmaier[1], Tobias Kneiphoff[2], and Heiko Krumm[3]

[1] Materna GmbH, Dortmund, Germany
gerrit.rothmaier@materna.de

[2] Bosch Rexroth AG, Witten, Germany
tobias@kneiphoff.com

[3] Universität Dortmund - FB Informatik, Dortmund, Germany
krumm@cs.uni-dortmund.de

Abstract. Advanced network attacks utilize complex, intertwined sequences of events on different hosts instead of just plain vulnerability exploitations. These sequences may contain protocol execution steps, attacker, and administrator actions. We propose a SPIN based approach for formal modeling and analysis of such attack sequences in scenarios where both protocol and network level aspects are relevant. Our approach allows to automatically find typical attack sequences and not yet considered variants in such an environment. The development of scenario models is supported by a modeling framework and the use of the high-level process specification language cTLA. For the purpose of automated analysis, the powerful model-checking tool SPIN is employed and a compiler provides optimized translation of the cTLA model descriptions to Promela. The development of models and analysis experiments are further facilitated by integrating the tools into the Eclipse universal tool platform. We outline the principles of our approach and focus on modeling structure, optimized translation and tool integration.

1 Introduction

Since security became an issue in computing, the objective of automatically analyzing system models and thus completely revealing immanent vulnerabilities and potential attack patterns exists. This objective is very ambitious, as we had to learn early, and may be reachable only under certain restrictions. Mainly there are two reasons why attempts for automated security analysis fail in practice. First, the development of suitable models is very expensive, since the model design is error-prone and tedious even if performed by well-educated and well-experienced designers. Second, analysis runs tend to exceed given time and memory limitations, since the analysis procedures have a high algorithmic complexity. These problems, by the way, are not restricted to automated security analysis but are already well-known in the general field of automated verification. Nevertheless, because security analysis of computer networks has to reason about unknown vulnerabilities, malfunctions and attack effects in comparably large systems, the search space is more complex and the problems therefore occur in an increased form.

Thus currently, a more realistic but still ambitious objective is to concentrate on a narrower field of interest and to lower the grade of automation by some forms of user guidance. One recognizes that formal modeling and analysis have a certain value, even if they are only employed for the representation and precise description of known attacks, since they can lead to a better understanding and insight into the phenomena and so probably indirectly contribute to future enhancements. In fact, the current status already provides some enhancements. Still, the costs of model development demand for the specialization to closer fields of interest and the analysis tool limitations demand for user guidance and restricted analysis scopes. Experience, however, showed that analysis runs which concentrate on a certain known and predefined class of attacks can find new unexpected variants.

A current trend in network attacks is more complex attack types making use of intertwined event sequences on multiple hosts instead of plain vulnerability exploitations [Ver04]. Typically such sequences

combine different aspects like carefully crafted attack steps with normal protocol execution steps and administrator actions. These advanced attacks are difficult to predict and analyze due to several reasons: Network and system administrators are often overstrained by the complexity of the system structure, by the frequency of dynamic changes, by the distribution of events on multiple hosts, and by the high number of existing service interdependencies. Furthermore, interoperability requirements force well-established basic management functions and protocols to remain in operation, even if they have been designed without special regard to security issues. Since these basic functions and protocols form the foundations of today's networks, successful attacks pose substantial threats. In view of that background we basically aim to support the clear modeling of complex attack sequences for a better understanding of the interrelations between different aspects like protocol weaknesses and attacker and administrator actions. Moreover the objective exists that models of interesting scenarios can automatically be analyzed in order to clarify supposed attack propagation processes and to find out possible variants.
With respect to the type of modeling, we have to reflect that management operations appear as processes consisting of sequences of steps where each step performs a set of contiguous modifications of a system component. The interesting effects therefore trace back to functional interactions between processes. Consequently, we resort to formal modeling and analysis techniques for the functional aspects of concurrent process systems.
Considering the problems of formal analysis which result from the expensive model design and the limitations of automated analysis tools, we follow up a combined approach which is mainly based on two elements. First, the system verification tool *SPIN* [Hol97, Hol03] is applied for automated analysis in order to profit from its powerful analysis procedures. Second, the development of models is supported by a modeling framework which provides model architecture guidelines and re-usable model definition components. The framework applies the high-level process specification language *cTLA* [HK00], which is a variant of Leslie Lamport's *Temporal Logic of Actions TLA* [Lam94] and provides for the modular definition of process types and the derivation of new process types by refinement and composition. Therefore, *cTLA* facilitates the efficient re-use and adaptation of framework elements, and, in comparison to *SPIN's* model description language *Promela*, supports more abstract and logically structured model definitions. The link to *SPIN* is provided by a compiler translating *cTLA* model definitions into *Promela*. Besides just translating models the compiler applies model optimizations. Moreover, the practical application of our approach is supported by means of a model development environment, which is implemented by extensions to the well-known software development tool *Eclipse* [Ecl05].
The approach has already been applied successfully to the modeling and analysis of different scenarios. [RPK04] presents the modeling and *SPIN*-based analysis of *ARP* spoofing like attacks respectively erroneous network management actions in a small LAN. Furthermore, [RK05] describes the modeling and analysis of *RIP* routing protocol attacks.
This paper focuses on the compositional structure of our models, the optimized translation of our models to *Promela* and the integration of *SPIN* and related tools into the *Eclipse* universal tool platform. As a next step, after addressing related work, we give an outline of the modeling framework and the model definition language *cTLA*. Two example models clarify the framework's application and highlight the compositional structure of our *cTLA* modeling. Then we discuss the principles of translating *cTLA* models to *Promela*. Model optimizations are outlined in the next section. Finally we describe how model editing, model translation, and *SPIN*-based analysis are integrated into *Eclipse*.

2 Related Work

Formal analysis and verification of security properties can be generally structured into *program and protocol verification*. Program verification shall enhance the trustworthiness of software systems (e.g. [BR00]). In protocol verification security weaknesses of protocols shall be found. Basic and cryptographic protocols (e.g. [MS02]) are particularly interesting. In both fields a variety of basic methods is applied, including classic logic and algebraic calculi (e.g. [KK03]), special calculi (e.g. [BAN89]), and process system modeling techniques (e.g. [LBL99]). Different kinds of analysis tools are used, including logic

programming environments like *Prolog*, expert system shells, theorem provers, algebraic term rewriting systems, and especially model checkers (e.g. [HJ04]). Some approaches even combine several analysis techniques [Mea96].
The formal modeling and analysis of complex, intertwined attack types in computer network scenarios is a relatively new field. Existing approaches either focus abstractly on protocols and disregard more low-level network aspects like topology, connectivity, and routing or the other way around. For example, in [RS98, RS02] the analysis of attack sequences resulting from the combined behavior of system components is described for a *single* host. A process model is used which is specified in a *Prolog* variant. Security properties are expressed by labeling states safe and unsafe. Execution sequences which lead to unsafe states and correspond to vulnerability executions are searched using a *Prolog* based programming environment.
In [AR00, NBR02] an approach called *topological vulnerability analysis* is presented. A network of hosts is checked for attack sequences consisting of combining predefined vulnerabilities. The host modeling consists of two sets representing existing vulnerabilities and attacker access level. Network topology is modeled using a multi-valued connectivity matrix. Protocols are represented very simply through fixed values in the connectivity matrix; no sending, receiving, or processing of protocol elements is modeled. Exploit definitions have to be given with the model. Using *SMV*, [Ber01] possible combinations of the given vulnerabilities leading to the violation of a property (e.g. attacker has root access level on a specified host) are analyzed.
Our approach supports modeling and analysis regarding both network and protocol level aspects in a single model. With respect to efficient modeling, the framework makes use of techniques invented for high-performance implementation of protocols. In particular we learned from the activity thread implementation model which schedules activities of different protocol layers in common sequential control threads [Svo89], and from integrated layer processing which combines operations of different layers [AP93]. Partial order reductions, proposed in [ABH97], have a strong relationship to the activity thread implementation model providing the basis for the elimination of nondeterministic execution sequences. Furthermore, approaches for *Promela* level model optimizations have to be mentioned. Many interesting low-level optimizations are described in [Ruy01].

3 Modeling Framework

In order to foster re-use and reduce the effort needed for modeling, we looked into the possibility of creating a framework for formal modeling of computer networks. Because frameworks usually make heavy use of object-oriented mechanisms for composing elements and describing their relationships, we have to use a specification language that can express such concepts as well.

cTLA 2003 Specification Language

cTLA is based on *TLA* [Lam94], but supports explicit notions of process instances, process types, and process type composition [HK00]. Furthermore, *cTLA* 2003 adds object-oriented process composition types. Here we only give a conceptual overview of *cTLA* 2003. A language oriented description can be found in the technical report [RK03].
A *cTLA* specification describes a *state-transition system* which is composed of subsystems. These subsystems may be composed of further subsystems but are finally sets of process instances. Thus, after resolving subsystems, a *cTLA* system is always a composition of *process instances*.
Process instances belong to processes, which are typed. Each *process type* describes its own, self-contained state-transition system. In the simplest case, a process type does not use composition, but is completely self-contained. The state space of such a *simple process type* is completely defined through the set of local variables. Its transitions consist just of the process's actions. Actions are parameterized and

describe atomic transitions consisting of guards and effects. *Guards* define conditions that must be met to make the action executable, *effects* describe the state changes triggered by the action's execution.
Object oriented mechanisms are introduced with the *cTLA process composition type* `EXTENDS` and `CONTAINS`. These composition types allow new process types to be derived from other process types. The state space and the transitions of such process types depend not only on the locally defined variables and actions, but also on the state and transitions of the inherited process types.
Synchronization and communication between process instances is done via *joint actions*. Joint actions couple 1..n actions of process instances, i.e. their guards and effects are conjugated. All process instances not taking part in a joint action perform a *stuttering step*. On the system level, *system actions* have to be given to define the possible state transitions.
As an example, consider **Figure 1**. A simple *cTLA* System `Sys1` contains three process instances `NodeA`, `NodeB` and `PhysMedia` which are instances of types `Node` respectively `Media`. Each instance has it's own variables and actions. Four system actions, `na_rcv(pkt)`, `na_snd(pkt)`, `nb_snd(pkt)` and `nb_rcv(pkt)` are defined through *coupling* of instances' actions. For example, action `na_rcv(pkt)` couples `NodeA`'s `rcv` action with `PhysMedia`'s out action and `NodeB` performs a stuttering step.

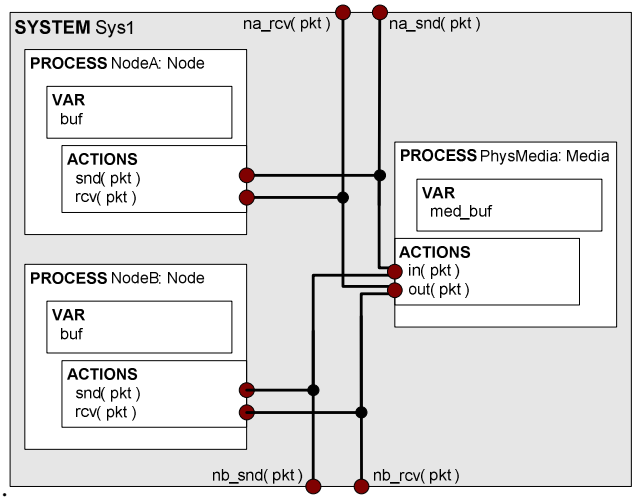

Figure 1: Action Coupling in a Simple cTLA System

Computer Network Modeling Framework

Frameworks as known from the world of object oriented programming consist of classes and their relationships. With process types and process composition types we have similar mechanisms available in *cTLA* 2003. We aim to transfer qualities known from object oriented frameworks like "natural modeling", broad level re-use of proven elements and architectures to formal modeling, especially of computer network related scenarios. Thus, we developed a modeling framework for TCP/IP based computer networks in *cTLA* 2003. We only give an overview of the framework here, an element-by-element description can be retrieved via our web site [Rot04].
The framework is structured into layers horizontally (c.f. **Figure 2**). The first layer, *Enumerations & Functions*, is used to define the network topology, initial address assignment and protocols desired for a model. For example, the enumeration `ZoneIdT` contains the model's zones (usually matching Ethernet segments), the function `fSrcToIa` is used to assign initial addresses and the enumeration `ProtocolT` symbolically lists protocols required in the model.
The second layer, *Data Types*, contains common data types for interfaces, packets and buffers used by other elements of the framework. For instance, the type `InterfaceT` combines attributes of an interface; `PacketT` is a record used to represent a packet and `PacketBufT` defines a buffer for such a packet.

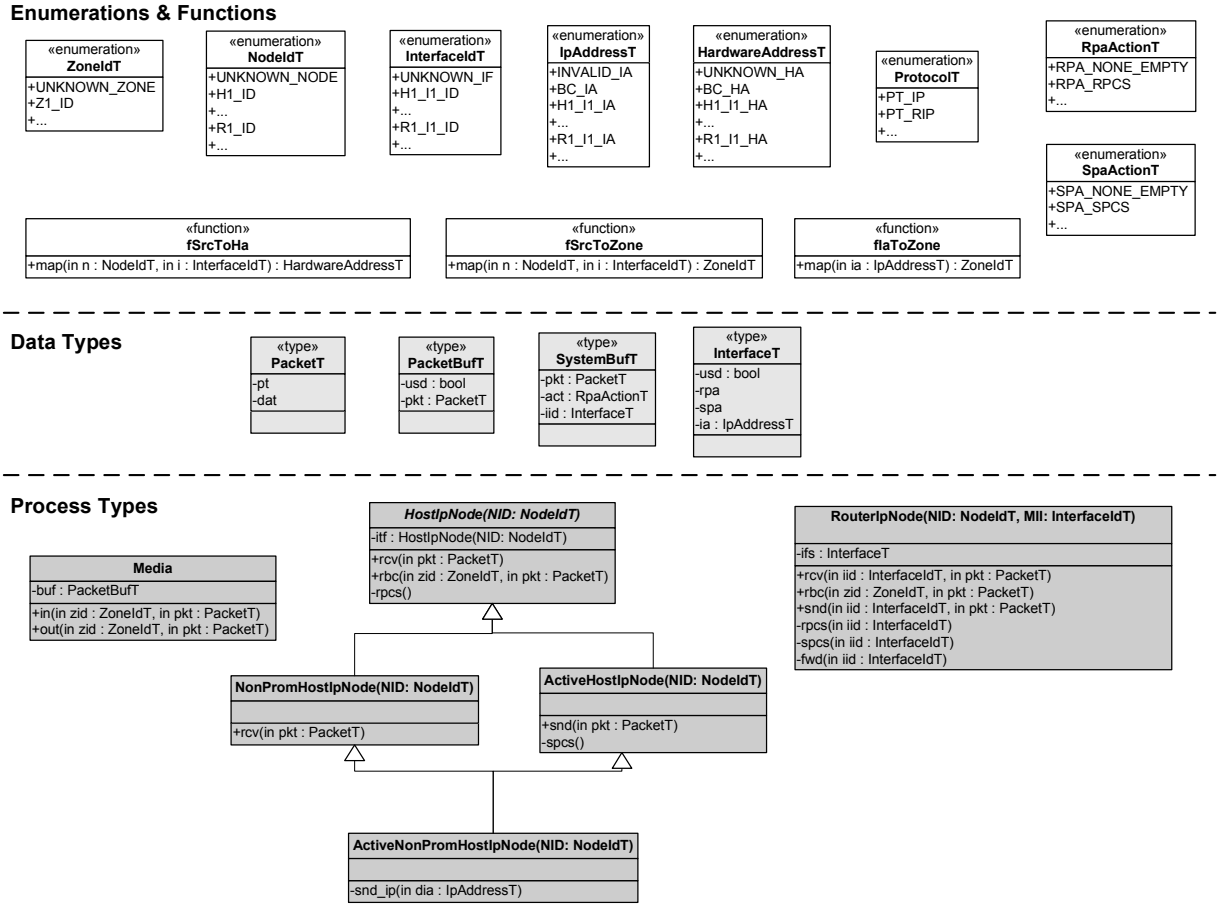

Figure 2: Framework Layers and Structure

Finally, the third layer, *Process Types*, provides core process types. For example, process type `RouterIpNode` models the basic behavior of a forwarding IP node and `HostIpNode` represents a passive IP host node. Through inheritance behavior is specialized. For example, `ActiveHostIpNode` adds behavior for the processing and sending of packets to `HostIpNode`.

Usually, all layers of the framework collaborate to model a conception. For example, a scenario's network topology is modeled by several functions (e.g. `fSrcToZone`) and enumerations (e.g. `ZoneIdT`), together with appropriate handling by processes (e.g. `Media`, `HostIpNode`, `RouterIpNode`) and their actions (e.g. `out`, `rcv`) which are parameterized with data types (e.g. `PacketT`).

During the design of the core process types we realized the usefulness of ideas known from *efficient protocol implementation techniques*. Especially the *activity thread* [Svo89] approach, which schedules activities of different protocol layers in common sequential control threads, and *integrated layer processing* [AP93], which combines operations of different layers, were helpful. Following these approaches allowed us to save on the required number of actions and buffers for the processing of packets. After translation to *Promela*, fewer actions and buffers lead to a reduced *SPIN* state-vector size and less possible transitions.

Example models use the basic structure and node types given by the framework. Of course, they usually have to add their own specific node types, e.g. `RipRouterIpNode`. These node types are derived from the framework's basic nodes and add data structures and behavior e.g. for processing additional protocols like RIP routing.

4 Example Models

Our approach has already been successfully used for the modeling and analysis of two example scenarios, the IP-ARP and RIP models. Here, we focus on the structure of the models and the relationship to the framework here.

IP-ARP Model

In the IP-ARP scenario, a small LAN with three hosts running a basic TCP/IP stack is modeled. This scenario is analyzed for confidentiality violations, i.e. packets received by non-intended recipients [RPK04]. The IP-ARP *cTLA* model structure – based on a preliminary version of the current framework – is depicted in **Figure 3**.

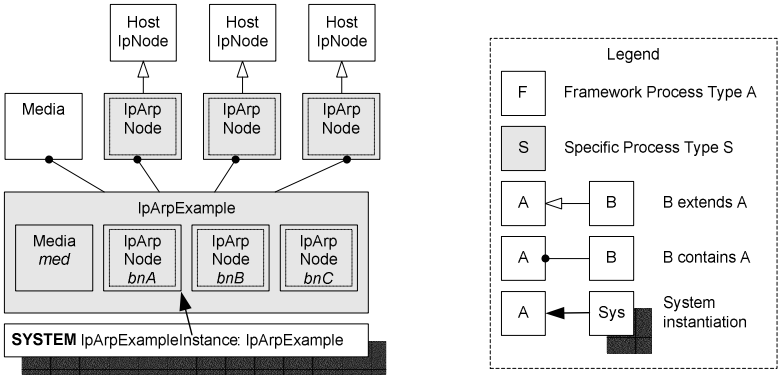

Figure 3: Compositional Structure of the IP-ARP Model

For the hosts, instances of the `IpArpNode` process type, which extends `HostIpNode` from the framework, are used. The `IpArpNode` process type adds support for a low-level ARP protocol layer. ARP queries are broadcasted for resolving yet unknown IP to hardware address mappings. ARP replies are processed and a local ARP cache is managed. On the IP level, changing of assigned IP addresses through management actions is added. Furthermore, packets to destination IP addresses with hardware addresses not yet in the ARP cache are buffered and the ARP layer is signaled. The LAN itself is modeled by a one zone `Media` instance with appropriate supporting enumerations (`ZoneIdT`, `NodeIdT`) and topology functions (`fSrcToZone`). Finally, the system is defined as an instance of the `IpArpExample` process type which is in turn a composition of one `Media` and three `IpArpNode` instances.
Depending on the inserted assertions modeling confidentiality properties, various violating sequences can be found. Interestingly, these sequences can be triggered by both ARP attacks and certain IP change management actions.

RIP Model

The RIP scenario consists of three LANs, connected by three routers R1, R2, R3 in a triangle-like fashion. Representative hosts H1, H2, HA are chosen from the LANs. The hosts are TCP/IP nodes, the routers additionally run the RIP protocol. In this scenario, man-in-the-middle attacks through forged RIP updates by host HA on communication between hosts H1 and H2 are analyzed [RK05].
In the RIP model, all routers are instances of `RipRouterIpNode` (cf. **Figure 4**), which is based on `RouterIpNode` from the framework. The `RipRouterIpNode` type adds functionality for processing and sending RIP update messages and updating its routing table accordingly. The hosts are modeled by different process types ultimately based on `HostIpNode` according to their role in the scenario (attacker, active or passive communication partner). The LANs are modeled by a six zones `Media` instance with

appropriate helper enumerations and functions. The system is an instance of the composed process type `IpRipExample` containing `Media`, three `RipRouterIpNode`, and three `HostIpNode` derived instances.

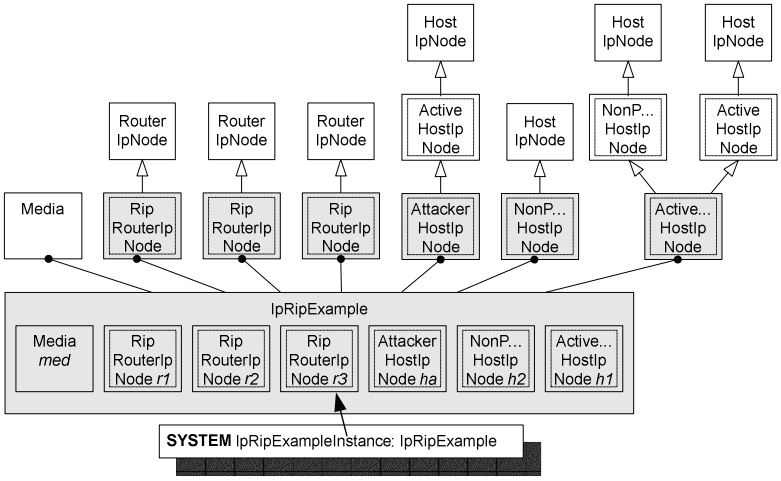

Figure 4: Compositional Structure of the RIP Model

Again, depending on the exact property modeling, various attack sequences can be found. **Figure 5** shows an example sequence. The sequences resemble typical routing attack ideas mentioned in [BHE01].

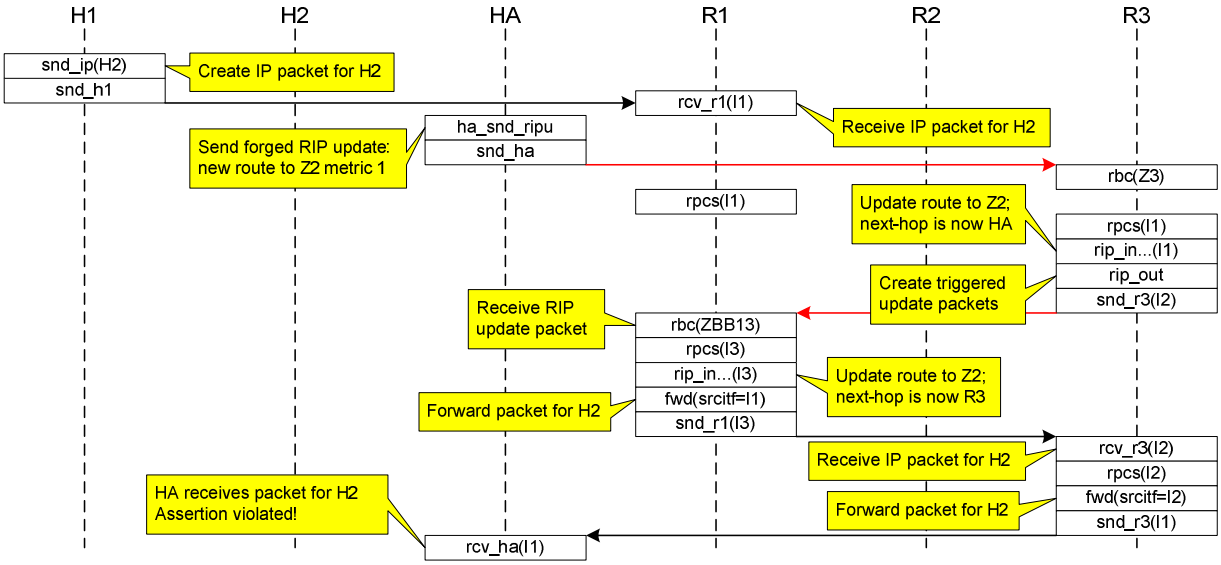

Figure 5: Simplified Attack Sequence in the RIP scenario

5 Translating cTLA Specifications to Promela

To be able to leverage both a high level *cTLA* based framework and *SPIN*'s powerful capabilities for checking *Promela* specifications, we engineered the *cTLA2PC* tool. It takes a *cTLA* specification as input and transforms it to an equivalent, optimized *Promela* specification. Alternatively, the output of a simplified, "flat" *cTLA* specification is possible as well. *cTLA2PC* is based on the ANTLR [PQ95] parser construction kit. In the following section, we give a short description of the most current version *cTLA2PC* version ("cTLA2PC 2").

The translation process starts with the scanner and parser components of *cTLA2PC*. If syntax errors are encountered, *cTLA2PC* prints an error message and the translation halts right after the parsing phase. After scanning and parsing, the semantic analysis is applied. Semantic analysis includes type checking of action parameters, function return values and assignments. Again, errors are flagged and stop the translation process.
The key phase for the translation of *cTLA* specifications to *Promela* is the *expansion* (cf. **Figure 6**). It transforms a compositional *cTLA* system to an expanded *cTLA* system. A compositional *cTLA* system (`CompSystemInstance`) is an instance of a process type (`CompSystemType`) containing process type instances (e.g. `pt1i1`, `pt2i1`, …). Each process type (`PT1`, `PT2`, …) may contain or extend further process types.

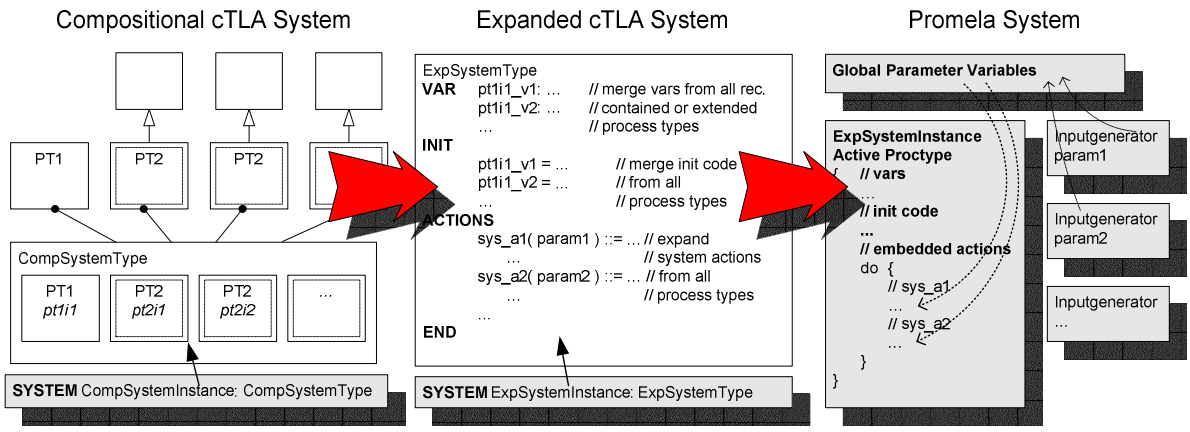

Figure 6: Transforming a Compositional cTLA System to Promela

Because such a model structure is not possible with *Promela*'s process types (`proctype`), extended and contained process types must be resolved prior to building the *Promela* system. This is done during the expansion phase. As an example, consider the expansion of an action from the IP-ARP model (cf. **Listing 1**). The compositional form of the action is given by the coupling of actions from the contained process type instances `bnA` (of process type `IpArpNode`) and `med` (`Media`). The expanded form of the action contains no process type instances. Instead, init code and variables from the instances have been merged directly into the generated expanded or *flat* system type (`ExpSystemType`). This allows the actions to be flat as well, i.e. to directly consist of the merged action code of the previously coupled instances.

```
// Original Action as defined in the Compositional System
snd_A( pkt: PacketT ) ::= bnA.snd( pkt ) AND med.in( pkt );

// Action after Expansion (Flat System)
snd_A( pkt: PacketT ) ::=
  pValidIf(pkt.sci, NA_MII)                    // guards
  AND bnA_ifs[pkt.sci - 1].usd = TRUE
  AND bnA_ifs[pkt.sci - 1].spa.usd = TRUE
  AND pkt = bnA_ifs[pkt.sci - 1].spa.pkt
  AND fSrcToZone(pkt.scn, pkt.sci) != UNKNOWN_ZONE
  AND med_buf[fSrcToZone(pkt.scn, pkt.sci) - 1].usd = FALSE
  AND bnA_ifs[pkt.sci - 1].spa.usd' = FALSE  // effects
  AND med_buf[fSrcToZone(pkt.scn, pkt.sci) - 1].usd' = TRUE
  AND med_buf[fSrcToZone(pkt.scn, pkt.sci) - 1].pkt' = pkt;
```

Listing 1: Compositional and Expanded Form of an Action

Starting from the flat system, code optimizations can be applied. **Section 6** describes a few optimizations optionally done by *cTLA2PC* during this phase.

Depending on the chosen output, either the *cTLA* code generation or the *Promela* code generation phase follows. The key step in the *Promela* code generation phase is the handling of actions and their parameters. Because of the expansion phase only a single, simple process instance is left in the system. This instance, however, still contains multiple, parameterized actions. Thus, all actions are embedded into a *Promela* non-deterministic `do` selection loop. The translation of the actions themselves, which are structured into *guard and effect* statements, can be done quite easily. Quantified guards (*cTLA* keywords `FORALL`, `EXISTS`) and effects (`UPDATEALL`) are special cases which have to be handled through the introduction of local loop blocks and corresponding temporary variables (*Promela* keyword `hidden`) in the code.

```
typedef PacketT {                       // typedefs and constants
  NodeIdT scn;
  ...
}
...
PacketT param_PacketT;                  // global parameter variables
...
active proctype IpArpExampleInstance() // system instance
{...                                        // decl. of variables and INIT code
  do                                        // non-determin. action selection loop
  :: d_step {                               // ACTION: snd_A( pkt: PacketT )
      ...                                         // guards & effects
  }
  ...                                       // next action
  od;
}

active proctype pa...T_scnInputGen()   // input generator process
{ do
  :: param_PacketT.scn = 0;
  :: param_PacketT.scn = 1;
  ...
  od;
}
...                                     // further inputgens
```

Listing 2: Example Structure of a Basic Promela Specification Generated by cTLA2PC

Still, action parameters have to be handled. They are implicitly existentially quantified in *cTLA*, i.e. if parameter values exist that satisfy the action's guards, the action is executable with this parameter setting. Action parameters are handled through the introduction of shared global variables and input generator processes. At first, we tried using *Promela channels* instead of shared variables, but simple global variables proved to be more efficient. The actions' parameters are replaced by these variables. *Input generator* processes are used to allow the global variables to reach all possible values. The processes use the *randomness non-deterministic if* approach described in [Ruy01]. Different actions may (re-)use the same global variables and input generator processes, thus reducing the number of additional variables and processes. All together, the generated *Promela* specification is structured as the example depicted in **Listing 2**.

The described approach works fine and was successfully used in [RPK04], but is relatively costly in terms of possible transitions and – to a lesser extent – state space. In **Section 6** we discuss a more efficient approach to the handling of parameterized actions.
Finally, the *Promela* code generation follows. Thanks to the previously generated intermediate code, the *Promela* code is derived in a straightforward way. This concludes the translation process. Additional translation options, which are useful for special cases, are recognized by *cTLA2PC*. For example, the `--simulation` switch includes a control flow generator and symbolic action names into the *Promela* code. This allows scripted testing of partial execution sequences and symbolic choice of actions in *SPIN*'s

interactive simulation mode. Furthermore, the `--trace-points` switch inserts extra output statements to ease the understanding of trail file sequences during guided simulation.

6 Optimizations

Our current tool version, *cTLA2PC* 2, supports several switches for applying different optimizations to the *Promela* code. Some of the low-level optimizations are inspired from [Ruy01]. Ruys describes the *bitvector* optimization. *SPIN* internally stores each element of a bit array as a byte. This may lead to an eightfold increase in the size of the state vector. The bitvector optimization maps up to eight elements of a bit array into a single `byte` and replaces element accesses with appropriate macros. With cTLA2PCs `--optbitarrays` and `--opt-bool2byte` switches, we implement a *generalized bitvector* optimization. Arrays of records with multi bit fields – possibly of different size – are mapped into arrays of `byte`. Furthermore, `bool`s are mapped into `byte`s as well, because *SPIN* internally stores each `bool` as a `byte`. Using this generalized bitvector optimization we were able to significantly reduce the state vector for both the IP-ARP and the RIP model (cf. **Table 1**).
Higher level optimizations can lead to even better results. Our models have a special structure because of their *cTLA* origin. This of course leads to particular optimization possibilities. A rewarding area for optimizations is the transformation of actions. During this transformation several new processes and variables for handling action parameters are created (cf. **Section 5**).

```
// Action after Parameter Refinement: Split pkt into its Fields and Transform all
// Accesses
snd_A( x_pkt_scn, x_pkt_sci, x_pkt_sha, x_pkt_dha, x_pkt_dat ) ::=
  pValidIf(x_pkt_sci, NA_MII)
  AND bnA_ifs[x_pkt_sci - 1].usd = TRUE
  AND bnA_ifs[x_pkt_sci - 1].spa.usd = TRUE
  // pkt = bnA_ifs[pkt.sci - 1].spa.pkt // transformed into fields
  AND x_pkt_scn = bnA_ifs[x_pkt_sci - 1].spa.pkt.scn      // equality
  AND x_pkt_sci = bnA_ifs[x_pkt_sci - 1].spa.pkt.sci      // equal. with dependency
  ...
  AND x_pkt_dat = bnA_ifs[x_pkt_sci - 1].spa.pkt.dat      // equality
  AND fSrcToZone(x_pkt_scn, x_pkt_sci) != UNKNOWN_ZONE
  ...
  // AND med_buf[fSrcToZone(pkt.scn, x_pkt_sci) - 1].pkt' = pkt; // transform
  AND med_buf[fSrcToZone(x_pkt_scn, x_pkt_sci) - 1].pkt.scn' = x_pkt_scn;
  ...
  AND med_buf[fSrcToZone(x_pkt_scn, x_pkt_sci) - 1].pkt.dat' = x_pkt_dat;

// Action after Paramodulation: substitute all occurrences of x_pkt_scn, x_pkt_sha,
// x_pkt_dha, x_pkt_dat and remove obtained guards of the form a=a (always true)
snd_A( x_pkt_sci ) ::=
  pValidIf(x_pkt_sci, NA_MII)
  AND bnA_ifs[x_pkt_sci - 1].usd = TRUE
  AND bnA_ifs[x_pkt_sci - 1].spa.usd = TRUE
  AND x_pkt_sci = bnA_ifs[x_pkt_sci - 1].spa.pkt.sci // only this guard remains
  AND  fSrcToZone(bnA_ifs[x_pkt_sci-1].spa.pkt.scn, x_pkt_sci)!= UNKNOWN_ZONE
  ...
  AND  med_buf[fSrcToZone(bnA_ifs[x_pkt_sci-1].spa.pkt.scn, x_pkt_sci) -
     1].pkt.scn' = bnA_ifs[x_pkt_sci - 1].spa.pkt.scn;
  ...
  AND  med_buf[fSrcToZone(bnA_ifs[x_pkt_sci-1].spa.pkt.scn, x_pkt_sci) -
     1].pkt.dat' = bnA_ifs[x_pkt_sci - 1].spa.pkt.dat;
```

Listing 3: Refining Parameters and Paramodulating an Action

The *paramodulation* optimization makes use of coupling between parameters in *cTLA* system actions. Typically, some action parameters serve as output parameters of constituting process actions. Thus, value determining equalities exist. Using these equalities, parameters occurrences in the action can be

substituted and the parameters can be removed from the action's parameter list. As an example of a slightly more complicated case, reconsider action `snd_A` from **Listing 1**. The action's parameters are `snd_A( pkt: PacketT)`, where `PacketT` is a record and an equality `pkt = bnA_ifs[ pkt.sci-1 ].spa.pkt` exists. Substituting `pkt` using this equality does not work, because the right hand side depends on the field `sci` of `pkt`. However, after a *parameter refinement* of `snd_A`, i.e. splitting its parameter `pkt` into its fields *`scn, sci, sha, dha, dat`* and transforming all guards and effects containing `pkt` accordingly (cf. **Listing 3**), partial paramodulation becomes possible. Now, equalities without dependencies exist for all fields of `pkt` except `sci`. Accordingly, all parameters but `x_pkt_sci` can be substituted in `snd_A`, leading to the final version `snd_A(x_pkt_sci)` with just one simple parameter.

Paramodulation optimizes a model with respect to two aspects. First, the number of shared global variables is reduced, inducing a smaller state space. Second, corresponding input generator processes are saved as well. This leads to fewer possible transitions and accordingly smaller search depths for checking action sequences. In both the IP-ARP and the RIP model, the state vector is clearly smaller after applying the paramodulation optimization (cf. **Table 1**).

Model	Optimization	State Vector
IP-ARP	Standard	250 Bytes
	Paramodulation	210 Bytes
	Generalized Bitvector	168 Bytes
RIP	Standard	448 Bytes
	Paramodulation	424 Bytes
	Generalized Bitvector	344 Bytes

Table 1: Effects of Different Optimizations for both the IP-ARP and RIP Models

Even with paramodulation, however, a larger RIP based example scenario [RK05] could not be analyzed by *SPIN*. Input generator processes for setting parameter values substantially increase the complexity of the *Promela* model. The state vector is only enlarged by a small amount (about 4 bytes per process), but the number of possible transitions is expanded greatly. For example, each setting of a parameter value requires at least one step. If an action requires several parameters, usually a separate setting step is required for each parameter. Furthermore, because all input generators run freely as separate processes, two types of *useless sequences* are possible. First, sequences setting parameters not used by (and not defined for) an action may occur. Second, the same parameter may be set in several consecutive steps, each time overwriting the previous value. Only the last step of such a sequence before an action execution determines the parameter value. Because *SPIN* must consider all possible sequences for model checking, however, it has to follow the useless sequence types as well.

To prevent these useless sequence types, we evaluated making input generators and actions more intelligent. For that purpose, we enhanced *cTLA2PC* to add code to the beginning of an action that sets enable flags only for the input generators associated with the used parameters. Each input generator resets its enable flag after setting a parameter value. This approach prevents both useless sequence types mentioned above. Unfortunately, this approach proved to be counterproductive anyway. The additional flags and their management overhead usually add more complexity to the model than is saved through the prevention of the useless sequences.

Consequently, we developed a radically different approach for handling parameterized actions: *unrolling input generators*. Following this approach, no input generators are created by *cTLA2PC*. Instead, parameters are unrolled by copying the actions and replacing the parameters with fixed values. For example, an action with two parameters p1, p2 has to be copied $|p1|\cdot|p2|$ times, where $|p_i|$, i=1, 2 is the cardinality of the type associated with parameter p_i. In the copies, the parameters are successively replaced with fixed values for all possible values. Because all variable types in *cTLA* (and *Promela*) are finite, the number of fixed actions replacing a parameterized action is finite as well.

Of course, the unroll optimization may lead to very large *Promela* specifications, but this only increases translation time. *SPIN* itself copes well with such specifications[1]. We evaluated the effects of the unroll optimization with the afore-mentioned RIP scenario. For benchmark purposes a simple assertion was added to the `rcv` action of host H2. This assertion was analyzed using *SPIN* in breadth-first search mode. As **Table 2** shows, *SPIN* performs remarkably better with the *Promela* model generated using the unroll approach than with standard input generators.

Optimization	State Vector	Stored States	Transitions	Depth	Memory
Standard	332 Bytes	1.19E+06	2.3E+08	14	203 MBytes
Unroll	316 Bytes	1.99E+04	1.8E+06	11	11 MBytes

Table 2: Effects on state vector size, transitions and depth of the unroll optimization in the RIP model

As input generator steps are no longer needed, the search depth required for finding a violating path is reduced as is the number of possible transitions at each level. Furthermore, the state vector is decreased as well. The unroll optimization was the critical last step for the successful automated analysis of the RIP model with *SPIN*.
Finally, efficiency should be kept in mind right from the design phase of a model. There, the framework helps again. Derived models inherit ideas from efficient protocol implementation (cf. **Section 3**), thus saving buffers and actions right from the start.

7 Eclipse Integration

We aim to ease the application of our modeling and analysis approach through appropriate tool support and integration. The *Eclipse* workbench is a well-known, widely adopted "universal tool platform" [Ecl05]. It defines only a core set of services. A modern plug-in architecture [Bol03] allows extending and customizing *Eclipse*'s functionality. Current web directories [Ecl05b] contain over 700 *Eclipse* plug-ins, even if many are of an experimental nature. We engineered a prototypical integration of the *SPIN* and *cTLA2PC* tools into *Eclipse*.

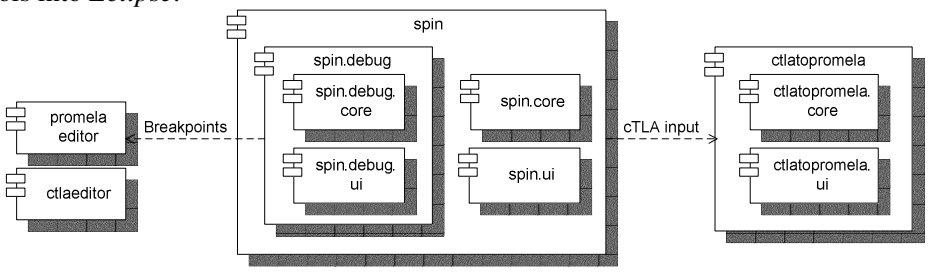

Figure 7: Eclipse Integration: Plug-in Architecture (UML Component Diagramm)

Architecture

Our integration is comprised of 8 *Eclipse* plug-ins (cf. **Figure 7**) implemented by 70 Java classes, totaling about 12,000 lines of code.
Except for the `promelaeditor` and `ctlaeditor` plug-ins, all plug-ins are separated into a `.ui` and `.core` component. User interface elements are implemented by the `.ui` component, the corresponding non-graphical functionality is implemented by the `.core` component. The underlying architectural pattern of the *Eclipse* framework is that different UI implementations can be used to present the same core functionality. Communication between UI and core components is handled via events. Unfortunately, the

[1] At least the Linux version, the Windows mingw32 port produced a `yacc` stack overflow during parsing.

conceptional separation between different plug-ins cannot be implemented to the last consequence. Some dependencies exist. As an example, the `spin.debug.ui` plug-in has to have knowledge of the name of the `promelaeditor` plug-in in order to provide the breakpoint-functionality correctly and not to interfere with other plug-ins.
The plug-ins are activated over the *Eclipse*-Plug-in-Architecture. During start-up, the plug-ins folder is scanned for MANIFEST-files, normally called `plugin.xml`, which is provided for every plug-in. The MANIFEST-file contains a description of the plug-in by indicating which *extension points* are implemented by the plug-in. This information is saved in a temporary database. Thus the plugin-code itself will only be loaded if necessary.

Features

Taken together, our plug-ins provide the following key features of the *SPIN Eclipse* integration:

Editing of Specifications. *Promela* and *cTLA* specifications can be edited with all standard capabilities (Search/Replace, Cut/Paste, Open/Save). Furthermore, keywords and comments are highlighted in different colors for both *Promela* and *cTLA* specifications. To support the debugging features described below, breakpoint markers can be set in the editors.
The described functionality is implemented by two plug-ins (`promelaeditor`, `ctlaeditor`) which both extend the `TextEditor` class. Thus all of *Eclipse*'s standard editing capabilities are available within *Promela* and *cTLA* specifications as well.

Specification Translation. Translation of *cTLA* (to *Promela* with *cTLA2PC*) and *Promela* specifications (with *SPIN* to C) is supported from within *Eclipse*. Different translation options can be given and saved in configurations. Each syntax error creates a new entry in *Eclipse*'s tasks pane. Double clicking an entry in the tasks pane leads to the corresponding source line in the editor.
This functionality is implemented by the two plug-ins `ctlatopromela.ui` and `ctlatopromela.core` respectively `spin.ui` and `spin.core`. The `.ui` plug-ins contain a dialog for configuring translation options. The `.core` plug-ins run the *cTLA2PC* respectively the *SPIN* tool in the background and capture the output. This is done with the help of *Eclipse*'s *Launching* architecture for external tools. For each tool, a matching environment is derived from `LaunchConfigurationType`. This type specifies a method `launch` which executes the tool with a given configuration. A `Configuration` is a set of parameters as name-value pairs. Actual tool executions with actual parameter values are instances of `LaunchConfiguration`. Default values for source and destination file are derived from the currently selected workspace resource. A `LaunchConfigurationDialog` shows the parameter values and allows their modification prior to launching the tool. Furthermore, additional parameters may be given. The captured output is parsed for translation errors which are transferred to the tasks pane using *Eclipse*'s *Markers* mechanism.

Simulation and Debugging. Simulation of *Promela* specifications is supported from within *Eclipse*. In random simulation mode, *SPIN*'s output is simply captured and transferred to *Eclipse*'s console window. For interactive simulations, the output is parsed and an interactive selection dialog is displayed for each non-deterministic choice (cf. **Figure 8**). Choices marked by *SPIN* as "unexecutable" are not displayed in the selection dialog. Furthermore, "debugging" of *Promela* specifications is supported as well. Breakpoints can be set in the *Promela* editor. If the corresponding line of the specification is hit, the simulation will be stopped. The user can then resume the specification simulation or step through it. Additionaly, variables can be added to the watch window. That means that the current value of such a variable is always displayed by *Eclipse*.
The plug-in `spin.core` implements the functionality to run the *SPIN* tool in the background based on the *Launching* architecture as described above. A new `LaunchConfigurationsType` is defined for *SPIN* simulation. The `spin.ui` plug-in contains a dialog for setting additional *SPIN* options based on *Eclipse*'s

`LaunchConfigurationDialog` and the selection dialog for interactive simulation. The `spin.debug.core` plug-in parses *SPIN* output and detects changes of watched variables, hit breakpoints etc. If breakpoints are defined, a `CodeModifier` is applied to the *Promela* file prior to starting the simulation. Its purpose is to add `printf(\"MSC: break?" + nextBreakpoint.getFileName() + ":" + nextBreakpoint.getSourceLineNumber()+ "\\n\" );` code at the appropriate lines. The plug-in captures *SPIN*'s output using a limiting buffer and scans it for the `MSC:` marker. If the marker is found, a breakpoint has been hit. The breakpoint's file and line number can be extracted from the extra information after the question mark. This implementation of breakpoints resembles *XSpin* [Spi05].

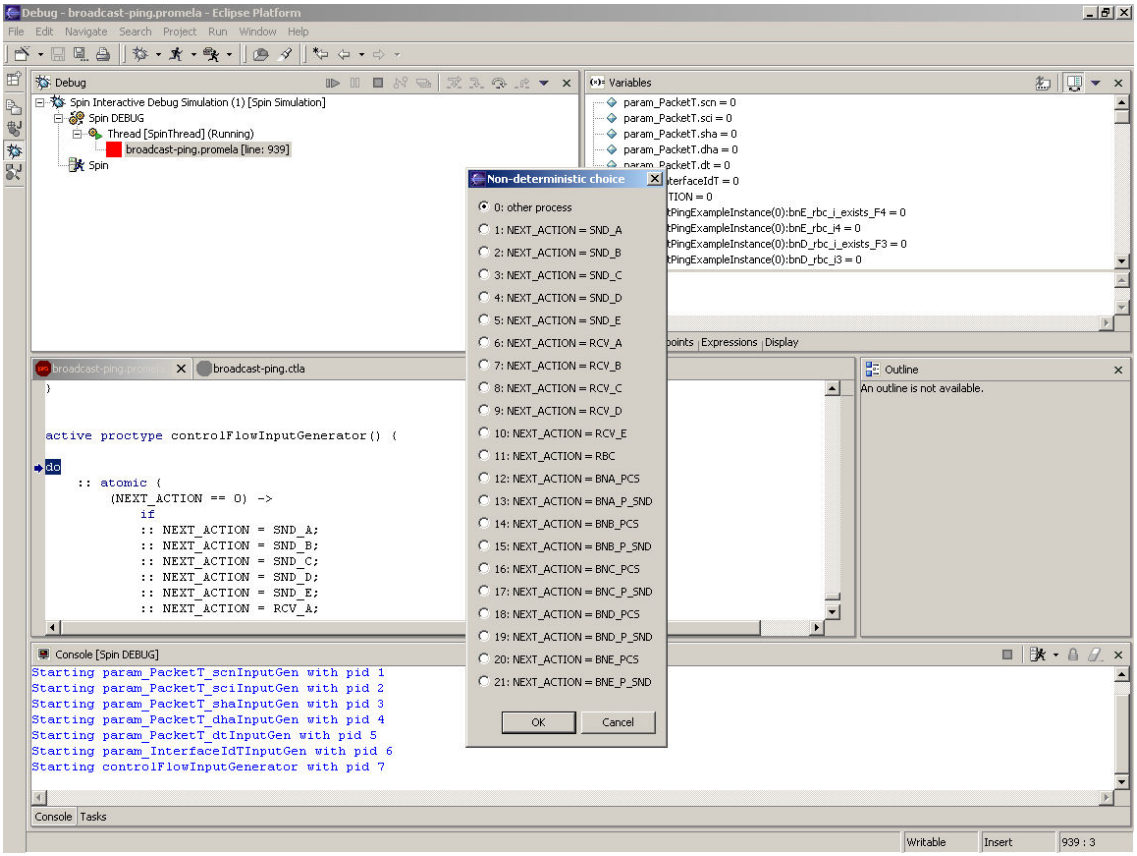

Figure 8: Interactive Simulation of a Promela Specification in Eclipse's Debug Perspective

Verification. Finally, *Promela* specifications can be verified from within *Eclipse*. Parameters for verifier generation (e.g. `-a`), verifier compilation (e.g. `-DBFS`) and verifier execution (e.g. `-m1000`) can be modified with a dialog by the user. *SPIN*'s verification output is then displayed in *Eclipse*'s console window.

The described functionality is implemented through a further `LaunchConfigurationType` and `LaunchConfigurationDialog` in the `spin.core` respectively `spin.ui` plug-ins.

Thanks to core services inherited from *Eclipse*, our integration covers further aspects, e.g. aggregation of files related to a specification into a project as well.

8 Concluding Remarks

The presented modeling framework and tool support facilitates the experimentation with small to medium size formal computer network models substantially and – as our experience showed – can be used not only

for the precise description of known scenarios and attack processes but also for the automated detection of unknown attack variants. The development of models, however, is still a demanding task, since each model design decision about whether at all and how a certain detail of the real scenario is to be represented in the model, may yield either too strong an increase of the set of reachable states or the loss of relevant analysis results. Therefore, our current work continues to investigate approaches of efficient protocol implementation in order to achieve further enhancements of the modeling framework. Moreover, we study the integration of symbolic reasoning into the approach. Particularly symbolically proven state invariants shall help to justify model simplifications.

References

[ABH97] R. Alur, R.K. Brayton, T.A. Henzinger, S. Qadeer, S.K. Rajamani: *Partial order reduction in symbolic state space exploration*. In Proc. of CAV 97: Computer-Aided Verification, LNCS 1254, pp. 340-351. Springer-Verlag, 1997.

[AP93] M. Abbott, L. Peterson: *Increasing network throughput by integrating protocol layers*. IEEE/ACM Transactions on Networking, pp. 600–610, 1, 1993.

[AR00] P. Ammann, R. Ritchey: *Using Model Checking to Analyze Network Vulnerabilities*. IEEE Symposium on Security and Privacy, May 2000.

[Ber01] S. Berezin: *The SMV web site*. URL: http://www.cs.cmu.edu/~modelcheck/smv.html/, 1999.

[BHE01] Blackhat Europe Conference: *Routing and Tunneling Protocol Attacks*, URL: http://www.blackhat.com/html/bh-europe-01/bh-europe-01-speakers.html#FX, 2001.

[Bol03] A. Bolour: *Notes on the Eclipse Plug-in architecture*. URL: http://www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html

[Ecl05] Eclipse.org: *Main Page*. URL: http://www.eclipse.org/

[Ecl05b] Eclipse-Plugins.info: *Plugin Overview*. URL: http://eclipse-plugins.info/eclipse/plugins.jsp

[HJ04] Yu-Tong He, Ryszard Janicki: *Verifying Protocols by Model Checking - A Case Study of the Wireless Application Protocol and the Model Checker SPIN*. CASCON '04: Proc. of the 2004 Conference of the Centre for Advanced Studies on Collaborative research, IBM Press, pp. 174-188, 2004.

[HK00] P. Herrmann, H. Krumm: *A Framework for Modeling Transfer Protocols*. Computer Networks, vol. 34, pp. 317-337, 2000.

[Hol97] G. J. Holzmann: *The Model Checker SPIN*, IEEE Transactions on Software Engineering, vol. 23(5), pp. 279-295, 1997.

[Hol03] G. J. Holzmann: *The SPIN Model Checker: Primer and Reference Manual*. Addison Wesley, 2003.

[Lam94] L. Lamport: *The Temporal Logic of Actions*, ACM Transactions on Programming Languages and Systems, vol. 16(3), pp. 872-923, 1994.

[LBL99] G. Leduc, O. Bonaventure, L. Leonard et al: *Model-based Verification of a Security Protocol for Conditional Access to Services*. Formal Methods in System Design, Kluwer Academic Publishers, vol. 14(2), pp. 171-191, 1999.

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

[MS02] Paolo Maggi, Riccardo Sisto: *Using SPIN to Verify Security Properties of Cryptographic Protocols*. Proc. 9th Int. SPIN Workshop on Model Checking of Software, LNCS 2318, Springer-Verlag, pp. 187-204, 2002.

[NB02] S. Noel, B. O' Berry, R. Ritchey: *Representing TCP/IP connectivity for topological analysis of network security*. Computer Society, IEEE (ed.), Proc. of the 18th Annual Computer Security Applications Conference, pp. 25-31, 2002.

[PQ95] T. Parr and R. W. Quong: *ANTLR: A Predicated-LL(k) Parser Generator*. Software – Practice and Experience, Vol. 25(7), pp. 789-810, 1995.

[RS02] C. Ramakrishnan, R. Sekar: *Model-Based Analysis of Configuration Vulnerabilities*. Journal of Computer Security, Vol. 10(1), pp. 189-209, 2002.

[RS98] C. Ramakrishnan, R. Sekar: *Model-Based Vulnerability Analysis of Computer Systems*. Second International Workshop on Verification, Model Checking, and Abstract Interpretation, Pisa, Italy, 1998.

[RK03] G. Rothmaier, H. Krumm: *cTLA 2003 Description*. Internal Technical Report, URL: http://ls4-www.cs.uni-dortmund.de/RVS/MA/hk/cTLA2003description.pdf, 2003.

[RPK04] G. Rothmaier, A. Pohl, H. Krumm: *Analyzing Network Management Effects with SPIN and cTLA*. Proc. of IFIP 18th World Computer Congress, TC11 19th Int. Information Security Conference, pp. 65-81, 2004.

[Rot04] G. Rothmaier: *cTLA Computer Network Specification Framework*. Internal Report. URL: http://www4.cs.uni-dortmund.de/RVS/MA/hk/framework.html

[RK05] G. Rothmaier, H. Krumm: *Formale Modellierung und Analyse protokollbasierter Angriffe in TCP/IP Netzwerken am Beispiel von ARP und RIP*. To be published Proc. of Sicherheit 2005, 2005.

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

[Spi05] Spinroot.com: *Promela Reference – printf(4)*. URL: http://spinroot.com/spin/Man/printf.html

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

[Ver04] Verisign: *Internet Security Intelligence Briefing / November 2004 / Vol. 2 / Issue II*. URL: http://www.verisign.com/Resources/Intelligence_and_Control_Services_White_Papers/internet-security-briefing.html