Automatic Verification of UML-based System on Chip Design

Abstract

In this paper we apply a fully automated flow to formally verify System on Chip (SoC) UML models. In the proposed approach, we verify that the system model, specified by State Machines, satisfies the properties reported in independently drawn Sequence Diagrams. This allows the designers to formally verify the correctness of the system model before proceeding with further design steps, thus saving design time in case of errors in the model.

The paper describes the verification methodology along with the steps for implementing it by existing software tools. The methodology is also applied to a case study: an UML description of an IPSec accelerator.

1 Introduction

Electronic system complexity is being increased every day: each System on Chip (SoC) may be composed by a mix of processors, DSPs, specialized hardware units, and memories. The growth in complexity does not stop the demand of reduced time to market; thus, the design of complex SoCs is becoming a real challenge. In this context, an efficient product development cycle is of crucial importance. One of the key points for obtaining this efficiency is being able detecting possible errors as early as possible in the design process. Thus, a methodology for automatically verifying the correctness of the system design can help improving the design process efficiency. In fact, being able to find inconsistencies in the system model before proceeding to further design steps allows the designers to save time by avoiding error propagation through the design steps.

During the first design phases the system is described at a high abstraction level. As presented in [BLP05a] and [BLP05b], the Unified Modeling Language (UML) [Uml] provides a good level of abstraction for describing the system behavior and the system structure. UML descriptions are suitable both for hardware and software implementations. Though, some models will direct to SoC implementations due to their characteristics (e.g., required parallelism) and to performance constraints.

A wise solution for the consistency checking of UML models is to use formal verification. This approach can be applied through the Model Checking technique [CGP00]; this technique can be applied to a wide variety of systems as it supports partial verification of the requirements and it can be applied to complex systems (advanced techniques are required in this case for reducing the state explosion problem).

In this paper we describe how to apply Model Checking to the UML specifications of a SoC. We first show how to transform the UML model in a suitable form for model checkers; we then show how to apply the Model Checking technique on the model. To better describe the procedure we also present a case study. This case study is related to the design of a SoC for implementing the IPSec suite of protocols; the UML description of the SoC is translated into a meta language [KW06] and verified by a model checker.

The following sections are organized as follows. Section 2 discusses the existing approaches to the UML model verification problem. Section 3 gives an overview of the proposed methodology. Section 4 describes the case study, while section 5 concludes the paper.

2 Related Work

The formal verification of a system starting from its UML description is a topic that has been investigated in different research works. In [Bos99] the author proposes an approach to check desired properties of a class of distributed component based software architectures; these architectures are characterized by indirect connections via mediators and shared space. This approach is demonstrated in the context of an architectural design implementing the NetBill protocol for e-commerce. Wuwel et al. present in [WCH02] a toolset based on the semantic modeling. Abstract State Machines are used to validate both static and dynamic aspects of a model. An approach on how to model check UML State Machines is presented in [WJXZC99]; the authors present a method for verifying complex systems composed by multiple objects that are modeled by state machines and collaboration diagrams. In [PHP05] authors present a framework called Charmy for assisting software architects in designing and validating architectural specifications. The tool allows UML-like notations to graphically design the system and to automatically generate a formal prototype for simulation and analysis purposes.

Our work has been inspired by [KW06]; in this paper the authors describe a tool called Hugo/RT [Hug] which is able to automatically transform UML2.0 interactions into Büchi automata [CGP00]. These interaction automata, along with the State Machines describing the model of the system, are translated into a behavioral language for the SPIN model checker [Hol03]. The translation supports basic interactions, state invariants, strict and weak sequencing, alternatives, ignores, and loops as well as forbidden interaction frag-

ments. The translation of the model is integrated into the UML Model Checking tool Hugo/RT. All the above papers describe an approach to model check UML models of software complex systems. In our paper, we adapt and apply the verification flow, proposed in [KW06], to a hardware/software system, namely a System on a Chip (SoC).

3 Design Flow

System design is composed by different phases, also including modeling and verification. It is well known that an increasing percentage of the full design cost depends on the verification step, thus in this work we concentrate on that phase. In this section we describe the design flow that we propose.

As can be seen from Figure 1, the full verification process is divided into three parts; each one of them requires the use of specific tools. The starting point of the flow is the description of the SoC specifications by means of an UML model. The internal representation of the main diagrams (*Sequence Diagram*, *Class Diagram*, and *State Machine*) depends on the tool used; regardless the internal tool representation it is possible to write the UML model in *XML Metadata Interchange* (XMI) format. The commonly used UML modeling tools allow the user to export the UML model as an XMI file.

In our approach the implementation of the State Machines (the model) is checked against the temporal conditions posed in the Sequence Diagrams (the properties). This allows us to verify the implementation against its requirement, following the traditional Model Checking approach; this approach consists of proving that $M| = p$ (where M is the model that satisfies the property p). For this purpose, the system behavior description and the Büchi automata are extracted from the State Machines and the Sequence Diagrams, respectively. The automata is used to define the properties, that are translated in a behavioral code to be interpreted by the model-checker.

Both the properties and the system behavior description are used as input for the *model-checker* (the block at the bottom of Figure 1). The verification procedure depends on the particular tool used but, in most of the cases, after having determined whether the module is correct or not, the tool should give either a confirmation of the correctness of the model or a *counter-example* (i.e., an execution trace in which one of the properties is violated). If the model is marked as correct, it can be used in the further design steps. Otherwise, the model should be refined and re-checked. The last two steps should be repeated until the model can be marked as verified by the model checker.

The verification flow proposed is general and, as mentioned before, not depending on the particular tools used. Here we propose some open source tools that can be used to implement the aforementioned steps. UML diagrams can be drawn by *ArgoUML* [Arg]; this tool natively stores the model in *XMI* format, which is the format required for transforming Sequence Diagrams and State Machines into the model checker code. The properties and the behavioral description of the system are extracted by using *HugoRT* [Hug]. Those files contain the rules specified in the *Promela* (PROcess MEta LAnguage) [Pro] language and are input into *SPIN* [Hol03] model checker. *SPIN* verifies the model by checking that the State Machines satisfy the property using standard *interleaving semantics*. All possible execution traces are explored until it is ensured that the property holds.

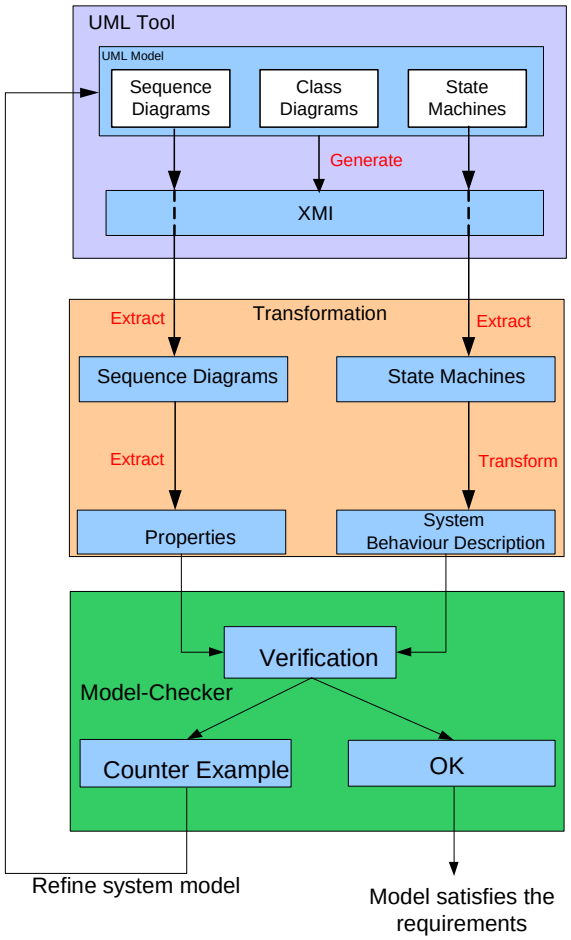

Figure 1: The overall approach

4 Case Study

In this section we describe a case study that we have used to test our verification flow; in this section the implementation of the flow steps is described in greater details. The following subsections describe the model of the system and show how we applied the verification flow to this model.

4.1 Description of the Model

In this section a description of the UML model that we have used as case study is given. The model describes

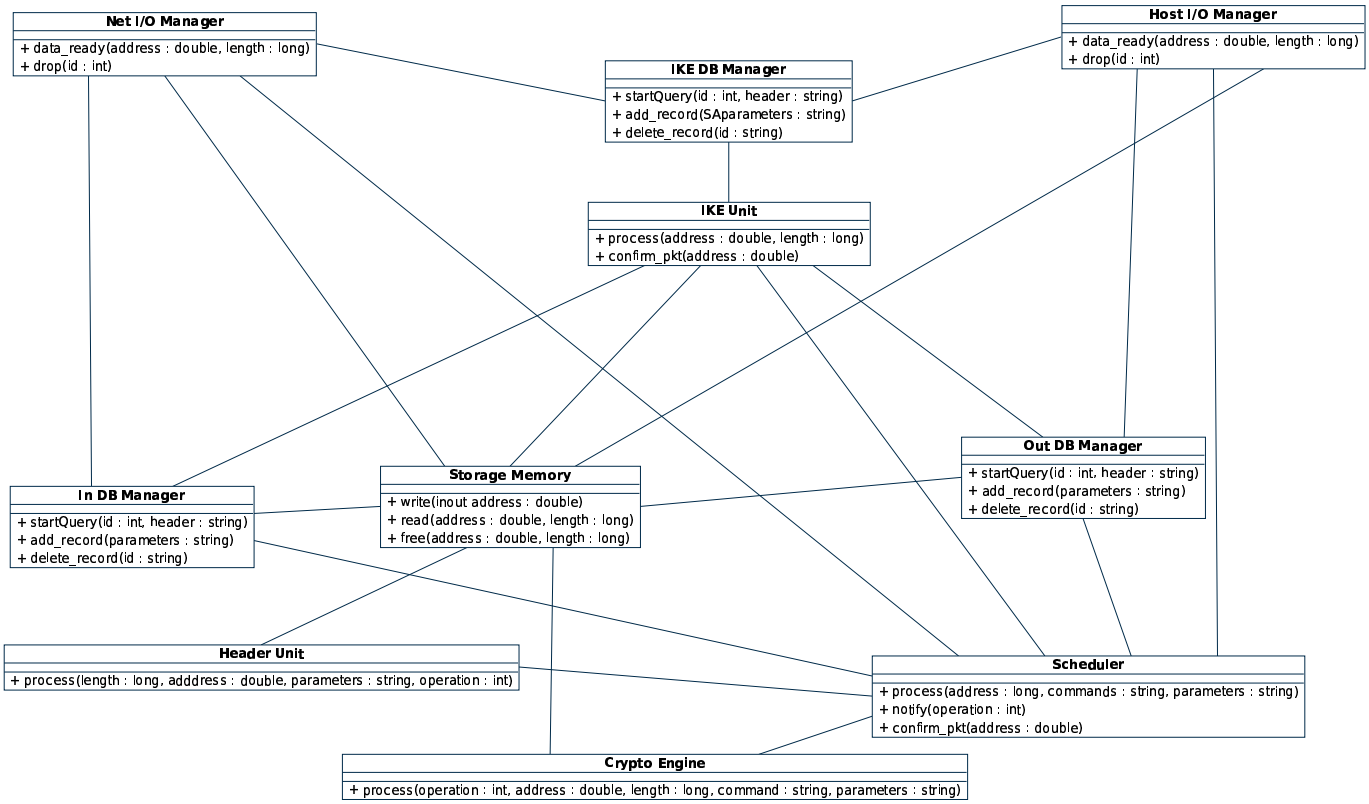

Figure 2: UML class diagram describing the relations among the different parts of the SoC.

an implementation of the IPSec (IP Secure) suite of protocols [YS01]. This model is especially suited to be implemented in a SoC. In fact, the different blocks of the model describe parts of the system that can operate in parallel. A purely software implementation is possible, but the final result will be a less optimized and slower system.

4.1.1 The IPSec Suite of Protocols

IPSec is a suite of protocols which adds security to communications at the IP level. IPSec is mainly composed of two protocols, Authentication Header (AH) and Encapsulating Security Payload (ESP). The former allows authentication of each IP datagrams headers or – depending on the operational mode that has been selected – of the entire IP datagram. The latter allows encryption – and optionally authentication – of the entire IP datagram or of the IP payload, depending on the operational mode that has been selected, namely the transport and the tunnel modes [KA98a, KA98b, HC98]. The concept of Security Association (SA) is fundamental to IPSec. A Security Association is a simplex "connection" that afford security services to the traffic carried by it [KA98c]. To secure typical bi-directional communication between two peers, two SAs (one in each direction) are required. Security services are afforded to a SA by the use of AH, or ESP, but not both. Security association establishment can be performed through a protocol named Internet Key Exchange (IKE) [HC98]. Two databases are involved in processing IP traffic relative to security associations. These two databases are the Security Policy Database (SPD) and the Security Association Database (SAD). The former specifies the policies that determine the disposition of all IP traffic. The latter contains parameters that are associated with each SA. For each packet traversing the IP communication layer, the SPD needs to be queried. If, in conformance with the SPD, an IP datagram needs to be processed by IPSec, the SAD needs also to be queried to discover the parameters of the considered SA. Information about whether a SA has already been created or not are contained in the SPD. If a suitable SA for the IP datagram to be processed does not exist, it needs to be established, for example through IKE. IPSec has proved to be computationally very intensive [MIK02, ANM+00, AVJ05]. Thus, some hardware acceleration is needed to support large network bandwidths, as may be required even in small secure gateways.

4.1.2 The UML Model

This subsection presents a model of a SoC designed providing a fast implementation of the IPSec suite of protocols. This SoC has been designed with the *flow-through* philosophy in mind: the accelerator is placed on the main data path and all the network packets flow through it [Fri04]. A full presentation of the aforementioned SoC was given in [Ref]. Figure 2 shows a class diagram representing the different functional units of the SoC and their relationships. The *Net I/O Man-*

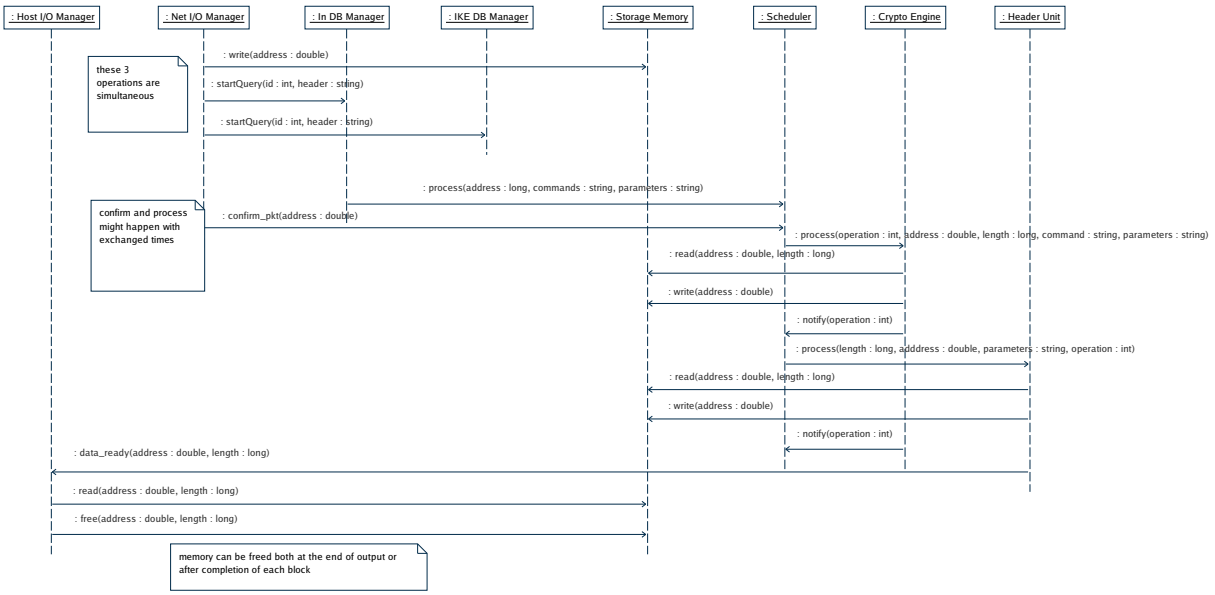

Figure 3: Sequence diagram describing the behavior of the system when an IPSec inbound packet is received.

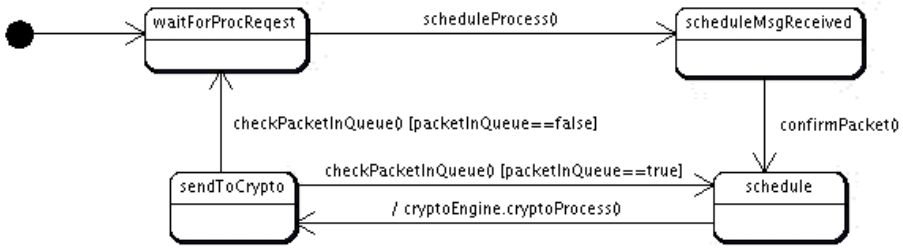

Figure 4: State Machines describing the *Scheduler* behavior.

ager and the *Host I/O Manager* manage the I/O of the chip on the network side and on the host computer side, respectively. These blocks can be further subdivided into two sub-blocks each: the first one manages the incoming packets; the other one manages the outgoing packets.

The DB Management units must manage the security policy and security association databases.

The *Scheduler* unit receives packets to be processed, and schedules them to the crypto-engine(s). Once a processing notify from the crypto unit is received, the unit should send the packet either again to the *Crypto-Engine* or, as it should happen more frequently, to the *Header* unit.

The *Crypto-Engine* might contain the implementation of different algorithms. A local control unit must be able to understand the commands sent by the *Scheduler* unit and to process packets accordingly. Either at the end or during the packet processing, this unit must write back the results to memory. This unit acts on each packet after the *Crypto-Engine* one, and it is responsible of verifying the signature of packets or to form the proper IPSec headers and to attack them to the packets.

The behavior of the chip and the interactions among different blocks are described by means of sequence diagrams. For example, Figure 3 describes the behavior of the chip when an IPSec inbound packet is received. The behavior of each functional block is described by a State Machine. For example, Figure 4 shows the behavior of the *Scheduler* block.

4.2 Flow Application

Among all the blocks described in Figure 2, we will consider, for our purposes, the behavior of the *Scheduler*. This block has the functionality of scheduling packets to be processed to the proper processing units (the *Crypto Engine* or the *Header Unit*). The *Scheduler* is notified about new packets and commands to be executed by the I/O blocks and by the DB Managers, respectively. If required, the *Scheduler* sends a proper command to the *Crypto Units*.

As explained in Section 3, our methodology aims at verifying the implementation against its requirement, following the traditional Model Checking approach that consists of proving that $M| = p$. In our case the model is described by means of a State Machines, and the properties are expressed by using a Sequence Diagram. The starting point is the description of the system behavior by means of a State Machines; the one depicted in Figure 4 is considered in this example application. The property for correctness verification are extracted from the sequence diagram shown in

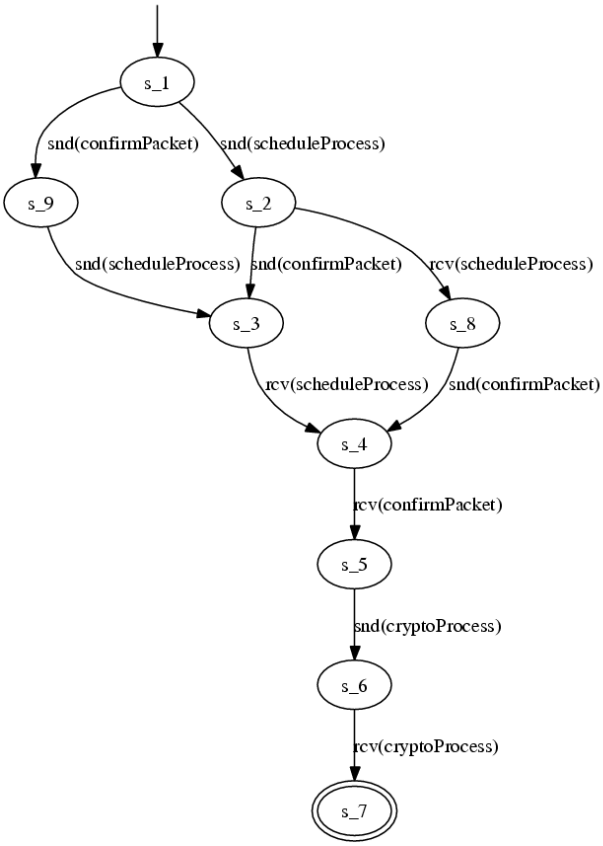

Figure 5: Büchi automaton of property 1)

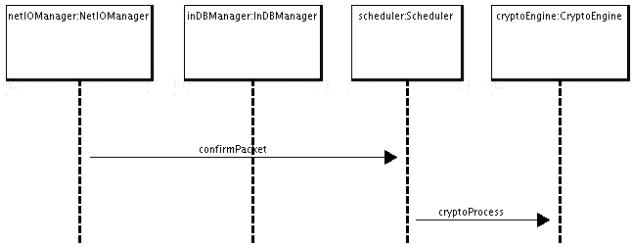

Figure 6: Sequence Diagram of property 2)

Figure 3. This diagram contains all the properties that the SoC blocks have to satisfy when an IPSec inbound packet is received.

The verification process begins with the generation of the XMI file; this file is extracted from the UML description by any of the UML design tools (e.g., ArgoUML). The XMI file is used by the Hugo/RT tool to generate the Promela file. Promela is a verification modeling language. It provides a way for making abstractions of distributed systems.

Unlike State Machines, Sequence Diagrams are expressed into Textual UML Format *UTE* [ute]. This representation is then used for extracting the properties. As an example, we have extracted two properties in order to verify the behavior of the *Scheduler* block. The first one is a property that must be satisfied by the model, while the second is a property that must NOT be satisfied by the model. In fact, the second one describes a behavior that the *Scheduler* must never assume. The two properties are described as follows:

1. after having received two signals, one from the *Net I/O Manager* and one from the *IN DB Manager*, the *Scheduler* must send the packet to the *Crypto-Engine*;

2. the *Scheduler* sends the packet to the *Crypto-Engine* without receiving any or only one signal from *Net I/O Manager* nor from the *IN DB Manager*, see also Figure 6.

The above properties will be transformed by Hugo/RT into Büchi automata and then translated into Promela behavioral code. Figure 5 shows the Büchi automaton of property 1), which describes all possible transitions that may occur. The condition for the model to satisfy property 1) is that, during the verification process, *SPIN* reaches the acceptance cycle *s_7* of Figure 5. Reaching the acceptance state of the property automaton proves that there is an execution of the state machine model that exaclty matches the sequence diagram used to express the property. Property 2) is *wrong* and should never hold. This can be verified in the same way as for property 1). Once we verify that a property is wrong, we should also verify that its negation holds.

The result provided by our verification flow is that our model satisfies property 1), but not property 2).

The Promela file corresponding to the above properties is then input to the SPIN model checker that verifies whether $M| = p$. SPIN provides two kinds of outputs: either it asserts that all the properties are correctly verified, or it generates a counter example. This counter example, as explained in Section 3 shows a case in which at least one of the properties is violated. If a counter example is provided, the UML model should be refined and verified again. If the properties are instead verified the design flow can continue as usual. In the considered case, the properties are verified, thus the model of this part of the SoC can be considered verified.

5 Conclusions

In this paper we have shown that by using standard formal verification tools, it is possible to validate SoC UML models. Both the behavior of the system and the properties against which it should be validated are automatically extracted from the XMI files generated by any UML tool. These properties can be input to any available model checker, that uses finite state automata to represent properties.

To support our design and verification flow we made use of Argo UML version 0.24 to design Sequence Diagrams and State Machines, Hugo/RT version 0.50a to transform the Sequence Diagrams and State Machines

in the Promela behavioral code, and SPIN model-checker version 4.2.9 to verify our model against the properties.

We used our approach to verify a complex SoC that implements the IPSec suite of protocols. By using the methodology described in this paper, we were able to confirm the correctness of the model and that the methodology we have proposed is viable.

References

[ANM+00] S. Ariga, K. Nagahashi, M. Minami, H. Esaki, and J Murai. Performance Evaluation of Data Transmission Using IPSec Over IPv6 Networks. In *INET*, Yokohama, Japan, July 2000.

[Arg] http://argouml.tigris.org.

[AVJ05] Alberto Ferrante, Vincenzo Piuri, and Jeff Owen. IPSec Hardware Resource Requirements Evaluation. In *NGI 2005*, Rome, Italy, 18 April 2005. EuroNGI.

[BLP05a] A.S. Basu, M. Lajolo, and M. Prevostini. *A Methodology for Bridging the Gap between UML and Codesign.* UML for SOC Design, G. Martin and W. Mller (eds.), Springer, Dordrecht, The Netherlands, 2005, pp 119-146, 2005.

[BLP05b] A.S. Basu, M. Lajolo, and M. Prevostini. *Design and Synthesis of Reusable Platforms with Programmable Interconnects.* UML-SoC 2005 (DAC Workshop) Proc. pp 43-48, 2005.

[Bos99] P. Bose. *Automated translation of UML models of architectures for verification and simulation using SPIN.* 14th IEEE International Conference on Automated Software Engineering, Proc. pp 102-109, 1999.

[CGP00] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking.* The MIT Press, 2000.

[Fri04] Robert Friend. Making the Gigabit IPSec VPN Architecture Secure. *IEEE Computer*, 37(6):54–60, 06 2004.

[HC98] D. Harkins and D. Carrell. The Internet Key Exchange (IKE) – RFC2409. IETF RFC, 1998.

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

[Hug] http://www.pst.ifi.lmu.de/projekte/hugo.

[KA98a] S. Kent and R. Atkinson. IP Authentication Header – RFC2402. IETF RFC, 1998.

[KA98b] S. Kent and R. Atkinson. IP Encapsulating Security Payload (ESP) – RFC2406. IETF RFC, 1998.

[KA98c] S. Kent and R. Atkinson. Security Architecture for the Internet Protocol – RFC2401. IETF RFC, 1998.

[KW06] A. Knapp and J. Wuttke. *Model Checking of UML 2.0 Interactions.* CSDUML '06, 2006.

[MIK02] S. Miltchev, S. Ioannidis, and A. D. Keromytis. A Study of the Relative Costs of Network Security Protocols. Monterey, CA, June 2002. USENIX Annual Technical Program.

[PHP05] Paola Inverardi, Henry Muccini, and Patrizio Pelliccione. CHARMY: An Extensible Tool for Architectural Analysis. In *ESEC-FSE05*, Lisbon, Portugal, 5-9September 2005. ACM.

[Pro] http://www.spinroot.com.

[Ref] Reference removed for blind review.

[Uml] http://www.uml.org.

[ute] http://www.pst.ifi.lmu.de/projekte/hugo/#UTE.

[WCH02] S. Wuwel, K. Compton, and J. Huggins. *A toolset for supporting UML static and dynamic model checking.* 26th Annual International Computer Software and Applications Conference, Proc. pp 147-152, 2002.

[WJXZC99] D. Wei, W. Ji, Q. Xuan, and Q. Zhi-Chang. *Model checking UML statecharts.* Software Engineering Conference, Proc. pp 363-370, 1999.

[YS01] R. Yuan and W. Timoty Strayer. *Virtual Private Networks.* Addison Wesley, 2001.