Dynamic Secure Aspect Modeling with UML: From Models to Code

Jan Jürjens[1⋆] and Siv Hilde Houmb[2]

[1] Software & Systems Engineering, Dep. of Informatics, TU Munich, Germany
http://www4.in.tum.de/~juerjens
[2] Department of Computer and Information Science,
Norwegian University of Science and Technology, Norway
siv.hilde.houmb@idi.ntnu.no

Abstract. Security engineering deals with modeling, analysis, and implementation of complex security mechanisms. The dynamic nature of such mechanisms makes it difficult to anticipate undesirable emergent behavior. In this work, we propose an approach to develop and analyze security-critical specifications and implementations using aspect-oriented modeling. Since we focus on the dynamic views of a system, our work is complementary to existing approaches to security aspects mostly concerned with static views. Our approach includes a link to implementations in so far as the code which is constructed from the models can be analyzed automatically for satisfaction of the security requirements stated in the UML diagrams. We present tool support for our approach.

1 Introduction

Constructing security-critical systems in a sound and well-founded way poses high challenges. To support this task, we propose an Aspect-Oriented Modeling (AOM, see e.g. [EAK+01, EAB02, FRGG04, LB04]) approach which separates complex security mechanisms (which implement the security aspect model) from the core functionality of the system (the primary model) in order to allow a security verification of the particularly security-critical parts, and also of the composed model.

Since security requirements such as secrecy, integrity and authenticity of data are always relative to an unpredictable adversary, they are difficult to even define precisely, let alone to implement correctly within the development of security-critical systems. Being able to consider security aspects already in the design phase, before a system is actually implemented, is advantageous: Removing security flaws in the design phase saves cost and time. Thus, the goal is to develop security-critical systems that are secure by design. Towards this goal, the security extension UMLsec for the Unified Modeling Language (UML) has been defined

⋆ This work was partially funded by the German Federal Ministry of Education, Science, Research and Technology (BMBF) in the framework of the Verisoft project under grant 01 IS C38. The responsibility for this article lies with the author(s).

L. Briand and C. Williams (Eds.): MoDELS 2005, LNCS 3713, pp. 142–155, 2005.

in [Jür02, Jür04a]. It allows us to encapsulate knowledge on prudent security engineering as aspects and thereby make it available to developers which may not be specialized in security. In the current work, we present an approach which lets one weave in the security aspects specified as UMLsec stereotypes (such as secrecy) as concrete security mechanisms (such as a cryptographic protocol) on the modeling level. We demonstrate how to check whether the code which is meant to implement the models fulfills the security requirements by security verification with automated theorem provers (ATPs) for first-order logic. To support our approach, a tool is available over a web-interface and as open-source which, from control flow graphs generated from the source code and corresponding security requirements, automatically generates FOL logic formulas in the standard TPTP notation as input to a variety of ATP's [Jür04b].

In the next section, we give a short background on aspect-oriented modeling. In Sect. 3, we explain how one can specify security aspects in UMLsec models and how these are woven into the primary model using our approach. Section 4 explains our code analysis framework. Throughout the paper we demonstrate our approach using a variant of the Internet protocol Transport Layer Security (TLS). In Sect. 5, we report on experiences from using our approach in an industrial setting. After comparing our research with related work, we close with a discussion and an outlook on ongoing research.

2 Aspect-Oriented Modeling (AOM)

AOM techniques allow system developers to address crosscutting objectives, such as security requirements, separately from the core functional requirements during system design. An aspect-oriented design model consists of a set of aspects and a primary model. An aspect describes how a single objective is addressed in a design, and a primary model describes how core functional requirements are addressed. The aspects and the primary model are then composed before implementation or code generation. This is done by weaving the aspect and the primary model at the modeling level.

As illustrated in Fig. 1, aspect models consist of models describing the static and dynamic views. After weaving the security aspect with the primary model, our approach allows one to perform a security verification on the composed model. From the composed model, the code is constructed (either manually or by automatic generation). If one later performs changes in the code, as often necessary in industrial development, the primary and aspect models cannot be directly extracted from the code any more. Thus changes in the code cannot in general be un-weaved at the model level. Therefore, our approach furthermore allows us to directly verify the code constructed from the UML model and to make sure that, after the necessary manual adjustments, the code is still secure.

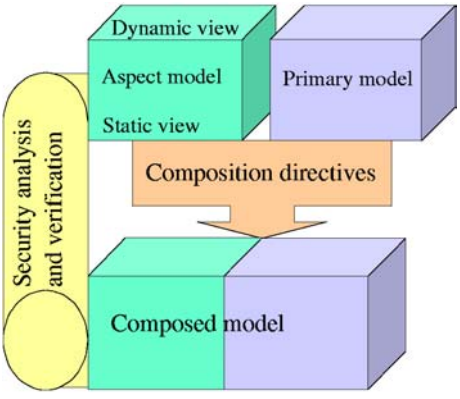

Fig. 1. Overview of the AOM approach for dynamic security aspects

3 Introducing Dynamic Security Aspects

3.1 Specifying Security Aspects

We can only shortly recall part of the UMLsec notation here for space reasons. A complete account can be found in [Jür02, Jür04a]. In Table 1 we give some of the stereotypes from UMLsec and in Table 2 the associated tags and corresponding adversary threats. The constraints connected to the stereotypes are formalized in first-order logic and can be verified by an automated first-order logic theorem prover, which is part of our UML analysis tool suite.

Stereotype	Base Class	Tags	Constraints	Description
Internet	link			Internet connection
encrypted	link			encrypted connection
LAN	link			LAN connection
secure links	subsystem		dependency security matched by links	enforces secure communication links
secrecy	dependency			assumes secrecy
secure dependency	subsystem		«call», «send» respect data security	structural interaction data security
critical	object	secret		critical object
no down-flow	subsystem		prevents down-flow	information flow
data security	subsystem		provides secrecy	basic datasec requirements
fair exchange	package	start,stop	after start eventually reach stop	enforce fair exchange

Table 1. UMLsec stereotypes (excerpt)

The general system model used here is the one that builds the foundation for a semantics for part of UML currently in development in a project with IBM Rational Software [BJCR05].

The primary model is a set of UML models and the dynamic aspect are weaved in by including the stereotypes defined above.

3.2 Weaving in Dynamic Security Aspects

Aspects encapsulate properties (often non-functional ones) which crosscut a system, and we use transformations of UML models to "weave in" dynamic security aspects on the model level. The resulting UML models can be analyzed as to whether they actually satisfy the desired security requirements using automated tools [Jür05]. Secondly, one should make sure that the code constructed from the models (either manually or by code generation) still satisfies the security requirements shown on the model level. This is highly non-trivial, for example because different aspects may be woven into the same system which may interfere on the code level in an unforeseen way. To achieve it, one has in principle two options: One can either again verify the generated code against the desired security requirements, or one can prove that the code generation preserves the security requirements fulfilled on the model level. Although the second option would be conceptually more satisfying, a formal verification of a code generator of industrially relevant strength seems to be infeasible for the foreseeable future. Also, in many cases, completely automated code generation may not be practical anyway. We therefore followed the first option and extended our UML security analysis techniques from [Jür04b] to the code level (presently C code, while the analysis of Java code is in development). The analysis approach now takes the generated code and automatically verifies it against the intended security requirement, which has been woven in as dynamic aspects. This is explained in Sect. 4. This verification thus amounts to a *translation validation* of the weaving and code construction process. Note that performing the analysis both at the model and the code level is not overly redundant: the security analysis on the model level has the advantage that problem found can be corrected earlier when this requires less effort, and the security analysis on the code level is still necessary as argued above. Also, in practice generated code is very rarely be used without any changes, which again requires verification on the code level.

The model transformation resulting from the "weaving in" of a dynamic security aspect p corresponds to a function f_p which takes a UML specification

Tag	Stereotype	Type	Multipl.	Description
secret	critical	String	*	secret data
start	fair exchange	$\mathcal{P}$(String)	1	start states
stop	fair exchange	$\mathcal{P}$(String)	1	stop states

Stereotype	$\mathsf{Threats}_{default}()$
Internet	{delete,read,insert}
encrypted	{delete}
LAN	$\emptyset$

Table 2. UMLsec tags (excerpt); Threats from the *default* attacker

$\mathcal{S}$ and returns a UML specification, namely the one obtained when applying p to $\mathcal{S}$. Technically, such a function can be presented by defining how it should act on certain subsystem instances[3], and by extending it to all possible UML specifications in a compositional way. Suppose that we have a set S of subsystem instances such that none of the subsystem instances in S is contained in any other subsystem instance in S. Suppose that for every subsystem instance $\mathcal{S} \in S$ we are given a subsystem instance $f_p(\mathcal{S})$. Then for any UML specification $\mathcal{U}$, we can define $f_p(\mathcal{U})$ by substituting each occurrence of a subsystem instance $\mathcal{S} \in S$ in $\mathcal{U}$ by $f_p(\mathcal{S})$. We demonstrate this by an example.

We consider the data secrecy aspect in the situation of communication over untrusted networks, as specified in Fig. 2. In the subsystem, the Sender object is supposed to accept a value in the variable d as an argument of the operation send and send it over the « encrypted » Internet link to the Receiver object, which delivers the value as a return value of the operation receive. According to the stereotype « critical » and the associated tag {secrecy}, the subsystem is supposed to preserve the secrecy of the variable d.

A well-known implementation of this aspect is to encrypt the traffic over the untrusted link using a key exchange protocol. As an example, we consider a simplified variant of the handshake protocol of the Internet protocol TLS in Fig. 4. The notation for the cryptographic algorithms is defined in Fig. 3.

The goal of the protocol is to let a sender send a secret over an untrusted communication link to a receiver in a way that provides secrecy, by using symmetric session keys.[4] The sender S initiates the protocol by sending the message $\mathsf{request}(\mathsf{N}, \mathsf{K_S}, \mathcal{S}ign_{\mathsf{K_S}^{-1}}(\mathsf{S} :: \mathsf{K_S}))$ to the receiver R. If the condition $[\mathbf{snd}(\mathcal{E}xt_{\mathsf{K}'}(\mathsf{c_S}))=\mathsf{K}']$ holds, where K' and $\mathsf{c_S}$ are the second and third arguments of the message received earlier (that is, if the key K_S contained in the signature matches the one transmitted in the clear), R sends the return message $\mathsf{return}\big(\{\mathcal{S}ign_{\mathsf{K_R}^{-1}}(\mathsf{K} :: \mathsf{N}')\}_{\mathsf{K}'}, \mathcal{S}ign_{\mathsf{K_{CA}}^{-1}}(\mathsf{R} :: \mathsf{K_R})\big)$ back to S (where N' is the first argument of the message received earlier). Then if the condition

$$[\mathbf{fst}(\mathcal{E}xt_{\mathsf{K_{CA}}}(\mathsf{c_R}))=\mathsf{R} \wedge \mathbf{snd}(\mathcal{E}xt_{\mathsf{K}''}(\mathcal{D}ec_{\mathsf{K_S}^{-1}}(\mathsf{c_k})))=\mathsf{N}]$$

holds, where $\mathsf{c_R}$ and $\mathsf{c_k}$ are the two arguments of the message received by the sender, and $\mathsf{K}'' ::= \mathbf{snd}(\mathcal{E}xt_{\mathsf{K_{CA}}}(\mathsf{c_R}))$ (that is, the certificate is actually for R and the correct nonce is returned), S sends $\mathsf{transmit}(\{\mathsf{d}\}_\mathsf{k})$ to R, where $\mathsf{k} ::= \mathbf{fst}(\mathcal{E}xt_{\mathsf{K}''}(\mathcal{D}ec_{\mathsf{K_S}^{-1}}(\mathsf{c_k})))$. If any of the checks fail, the respective protocol participant stops the execution of the protocol.

Note that the receiver sends two return messages - the first matches the return trigger at the sender, the other is the return message for the receive message with which the receiver object was called by the receiving application at the receiver node.

[3] Although one could also define this on the type level, we prefer to remain on the instance level, since having access to instances gives us more fine-grained control.

[4] Note that in this simplified example, which should mainly demonstrate the idea of dynamic security aspect weaving, authentication is out of scope of our considerations.

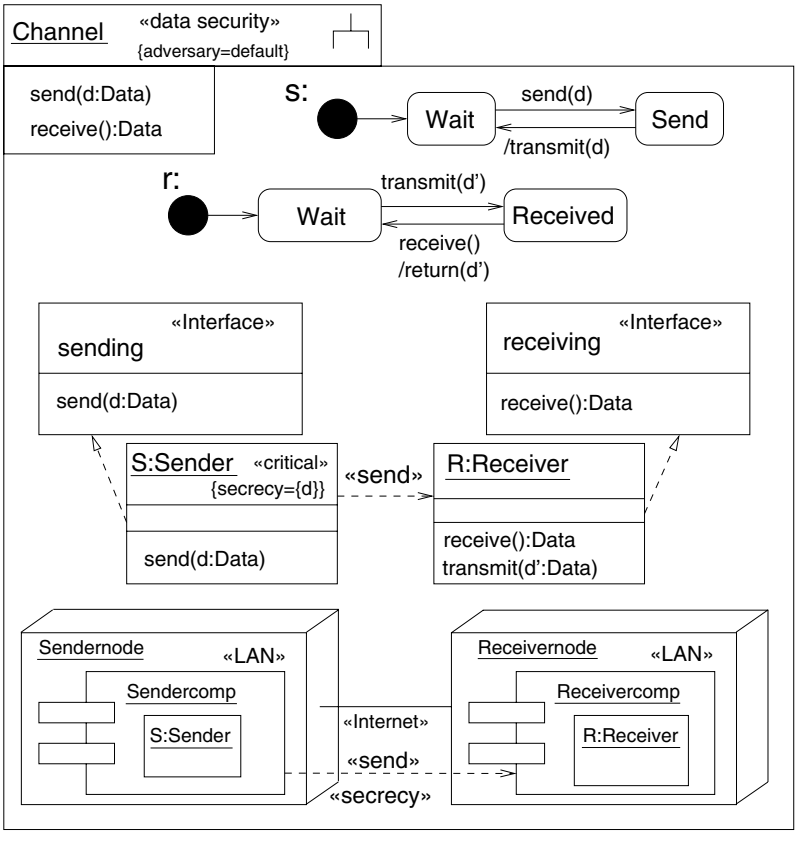

Fig. 2. Aspect weaving example: sender and receiver

- $\_ :: \_$ (concatenation)
- **head**(_) and **tail**(_) (head and tail of a concatenation)
- $\{\_\}_\_$ (encryption)
- $\mathcal{D}ec\_(\_)$ (decryption)
- $\mathcal{S}ign\_(\_)$ (signing)
- $\mathcal{E}xt\_(\_)$ (extracting from signature)

Fig. 3. Abstract Crypto Operations

To weave in this aspect p in a formal way, we consider the set S of subsystems derived from the subsystem in Fig. 2 by renaming: This means, we substitute any message, data, state, subsystem instance, node, or component name n by a name m at each occurrence, in a way such that name clashes are avoided. Then f_p maps any subsystem instance $\mathcal{S} \in S$ to the subsystem instance derived from that given in Fig. 4 by the same renaming. This gives us a presentation of

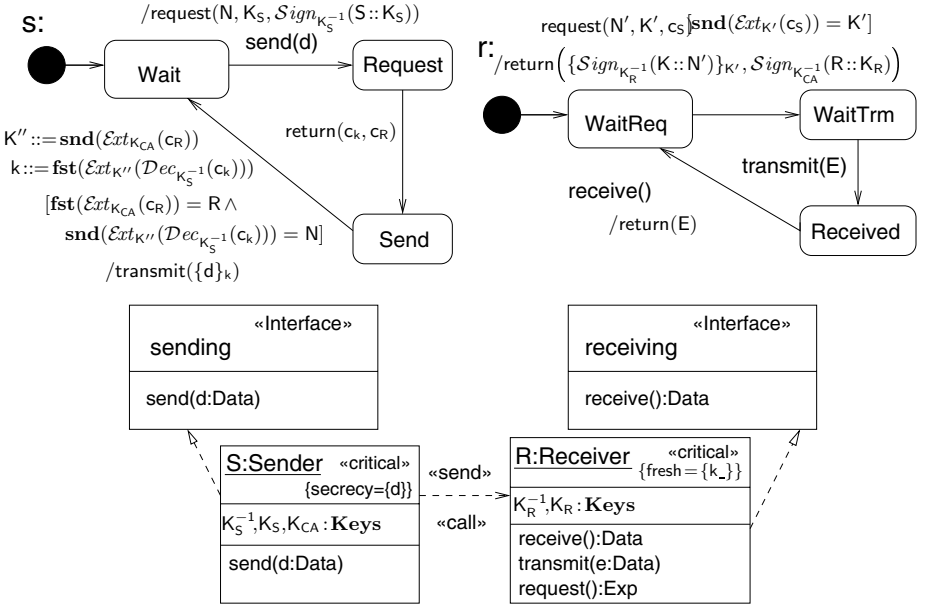

Fig. 4. Aspect weaving example: secure channel

f_p from which the definition of f_p on any UML specification can be derived as indicated above.

One can do the weaving by defining the transformation explained above using the model transformation framework BOTL developed at our group [BM03]. The overall tool-suite supporting our aspect-oriented modeling approach is given in Fig. 5. The tool-flow proceeds as follows. The developer creates a primary UML model and stores it in the XMI file format. The static checker checks that the security aspects formulated in the static views of the model are consistent. The dynamic checker weaves in the security aspects with the dynamic model. One can then verify the resulting UML model against the security requirements using the analysis engine (an automated theorem prover for first-order logic). One then constructs the code and also verify it against the security requirements using the theorem prover. The error analyzer uses the information received from the static and dynamic checkers to produce a text report for the developer describing the problems found, and a modified UML model, where the errors found are visualized.

4 Analyzing the Code

We define the translation of security protocol implementations to first-order logic formulas which allows automated analysis of the source code using automated first-order logic theorem provers. The source code is extracted as a control flow

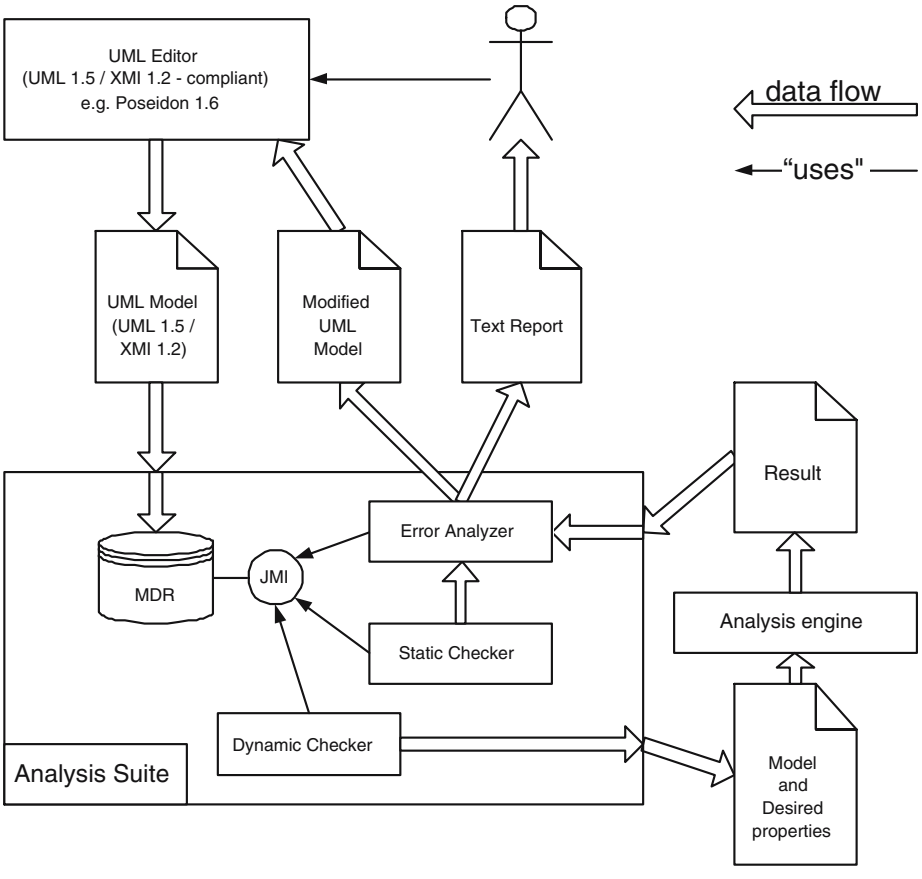

Fig. 5. UML verification framework: usage

graph using the aiCall tool [Abs04]. It is compiled to first-order logic axioms giving an abstract interpretation of the system behavior suitable for security analysis following the well-known Dolev-Yao adversary model [DY83]. The idea is that an adversary can read messages sent over the network and collect them in his knowledge set. He can merge and extract messages in the knowledge set and can delete or insert messages on the communication links. The security requirements are formalized with respect to this adversary model. For example, a data value remains secret from the adversary if it never appears in the knowledge set of the adversary. As with similar approaches such as [SFWW03], our approach works especially well with nicely structured code. For example, we apply an automated transformation which abstracts from pointers before applying our security analysis.

We explain the transformation from the control flow graph generated from the C program to first-order logic, which is given as input to the automated

theorem prover. For space restrictions, we restrict our explanation to the analysis for secrecy of data. The idea here is to use a predicate knows which defines a bound on the knowledge an adversary may obtain by reading, deleting and inserting messages on vulnerable communication lines (such as the Internet) in interaction with the protocol participants. Precisely, $\mathsf{knows}(E)$ means that the adversary may get to know E during the execution of the protocol. For any data value s supposed to remain confidential, one thus has to check whether one can derive $\mathsf{knows}(s)$.

From a logical point of view, this means that one considers a term algebra generated from ground data such as variables, keys, nonces and other data using symbolic operations including the ones in Fig. 3. In that term algebra, one defines the equations $\mathcal{D}ec_{K^{-1}}(\{E\}_K) = E$ and $\mathcal{E}xt_K(\mathcal{S}ign_{K^{-1}}(E)) = E$ (for all $E \in \mathbf{Exp}$ and $K \in \mathbf{Keys}$) and the usual laws regarding concatenation, head(), and tail(). This abstract information is automatically generated from the concrete source code.

The set of predicates defined to hold for a given program is defined as follows. For each publicly known expression E, the statement knows(E) is derived. To model the fact that the adversary may enlarge his set of knowledge by constructing new expressions from the ones he knows, including the use of cryptographic operations, formulas are generated which axiomatize these operations.

We now define how a control flow graph generated from a C program gives rise to a logical formula characterizing the interaction between the adversary and the protocol participants. We observe that the graph can be transformed to consist of transitions of the form trans(state, inpattern, condition, action, truestate), where inpattern is empty and condition equals true where they are not needed, and where action is a logical expression of the form localvar = value resp. outpattern in case of a local assignment resp. output command (and leaving it empty if not needed). If needed, there may be additionally another transition corresponding to the negation of the given condition, where we safely abstract from the negated condition (for logical reasons beyond this exposition).

Now assume that the source code gives rise to a transition $\mathsf{TR1} = \mathsf{trans}(\mathsf{s1}, \mathsf{i1}, \mathsf{c1}, \mathsf{a1}, \mathsf{t1})$ such that there is a second transition $\mathsf{TR2} = \mathsf{trans}(\mathsf{s2}, \mathsf{i2}, \mathsf{c2}, \mathsf{a2}, \mathsf{t2})$ where $\mathsf{s2} = \mathsf{t1}$. If there is no such transition TR2, we define $\mathsf{TR2} = \mathsf{trans}(\mathsf{t1}, [], \mathsf{true}, [], \mathsf{t1})$ to simplify our presentation, where [] is the empty input or output pattern. Suppose that c1 is of the form $\mathsf{cond}(\mathsf{arg}_1, \ldots, \mathsf{arg}_n)$. For i1, we define $\bar{\mathsf{i1}} = \mathsf{knows}(\mathsf{i1})$ in case i1 is non-empty and otherwise $\bar{\mathsf{i1}} = \mathsf{true}$. For a1, we define $\bar{\mathsf{a1}} = \mathsf{a1}$ in case a1 is of the form localvar = value and $\bar{\mathsf{a1}} = \mathsf{knows}(\mathsf{outpattern})$ in case $\mathsf{a1} = \mathsf{outpattern}$ (and $\bar{\mathsf{a1}} = \mathsf{true}$ in case a1 is empty). Then for TR1 we define the following predicate:

$$\mathsf{PRED}(\mathsf{TR1}) \equiv \bar{\mathsf{i1}} \& \mathsf{c1} \Rightarrow \bar{\mathsf{a1}} \& \mathsf{PRED}(\mathsf{TR2}) \tag{1}$$

The formula formalizes the fact that, if the adversary knows an expression he can assign to the variable i1 such that the condition c1 holds, then this implies

```
void TLS_Client (char* secret)
{  char Resp_1 [MSG_MAXLEN];
   char Resp_2 [MSG_MAXLEN];
   // allocate and prepare buffers
   memset (Resp1, 0x00, MSG_MAXLEN);
   memset (Resp2, 0x00, MSG_MAXLEN);
   // C->S: Init
   send (n, k_c, sign(conc(c, k_c), inv(k_c)));
   // S->C: Receive Server's respond
   recv (Resp_1, Resp_2);
   // Check Guards
   if ( (memcmp(fst(ext(Resp_2, k_ca)), s, MSG_MAXLEN) == 0) &&
         (memcmp(snd(ext(dec(Resp_1, inv(k_c)),
            snd(ext(Resp_2, k_ca)))), n, MSG_MAXLEN) == 0) )
   {  // C->S: Send Secret
       send (symenc(secret, fst(ext(dec(Resp_1,
        inv(k_c)), snd(ext(Resp_2, k_ca)))))); }}
```

Fig. 6. Fragment of abstracted client code

that $\bar{\mathsf{a1}}$ will hold according to the protocol, which means that either the equation $\mathsf{localvar} = \mathsf{value}$ holds in case of an assignment, or the adversary gets to know outpattern, in case it is send out in a1. Also then the predicate for the succeeding transition TR2 will hold.

To construct the recursive definition above, we assume that the control flow graph is finite and cycle-free. As usual in static code analysis, loops are unfolded over a number of iterations provided by the user. The predicates PRED(TR) for all such transitions TR are then joined together using logical conjunctions and closed by forall-quantification over all free variables contained.

Figure 6 gives a simplified C implementation of the client side of the TLS variant considered earlier. From this, the control flow graph is generated automatically. Although the complete graph cannot be shown here, we show as an example a fragment of the client side in Fig. 7. The main part of the transformation of the client to the e-SETHEO input format TPTP is given in Fig. 8. We use the TPTP notation for the first-order logic formulas [SS01], which is the input notation for many automated theorem provers including the one we use (e-SETHEO [SW00]). Here & means logical conjunction and $![\mathsf{E1}, \mathsf{E2}]$ forall-quantification over E1, E2. The protocol itself is expressed by a for-all quantification over the variables which store the message arguments received.

Given this translation of the C code to first-order logic, one can now check using the automated theorem prover that the code constructed from the UMLsec aspect model still satisfies the desired security requirements. For example, if the prover can derive knows(secret) from the formulas generated by the protocol, the adversary may potentially get to know secret. Details on how to perform this analysis given the first-order logic formula are explained in [Jür05].

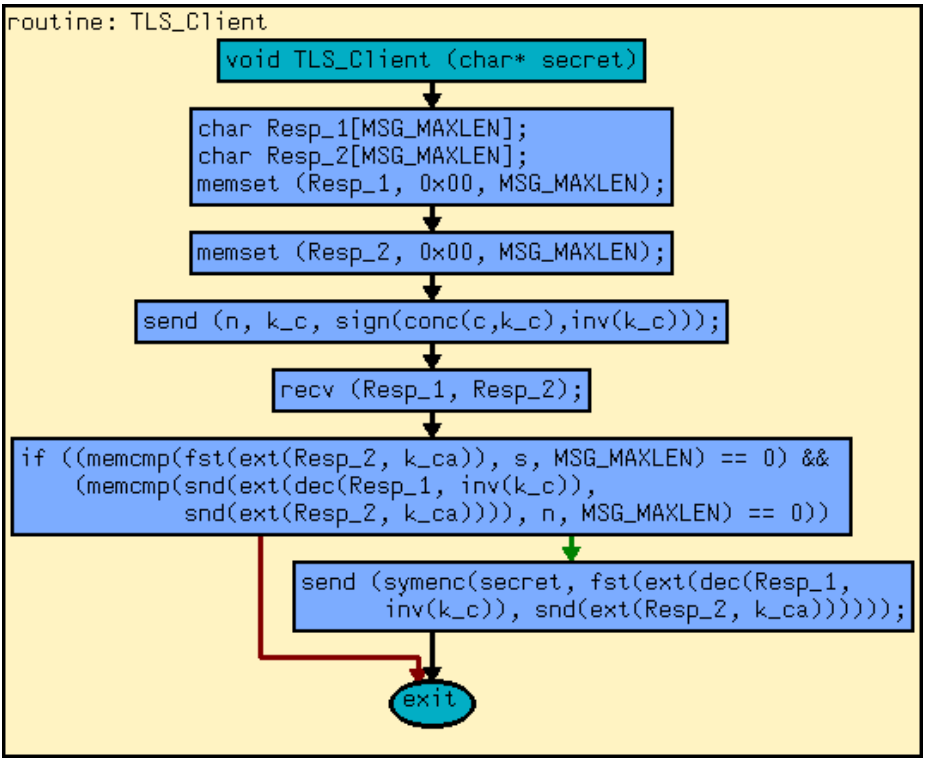

Fig. 7. Control graph for client

```
input_formula(protocol,axiom,(
 ![Resp_1, Resp_2] : (((knows(conc(n, conc(k_c,sign(conc(c,conc(k_c,eol)),inv(k_c)))))
         & ((knows(Resp_1) & knows(Resp_2)
         & equal(fst(ext(Resp_2,k_ca)),s)
         & equal(snd(ext(dec(Resp_1,inv(k_c)),snd(ext(Resp_2,k_ca)))),n))
     => knows(enc(secret,fst(ext(dec(Resp_1,inv(k_c)),snd(ext(Resp_2,k_ca))))))))).
```

Fig. 8. Core protocol axiom for client

5 Industrial Application

We are currently applying our method in an industrial project with a major German company. The goal is the correct development of a security-critical biometric authentication system which is supposed to control access to a protected resource. Because the correct design of such cryptographic protocols and the correct use within the surrounding system is very difficult, our method was chosen to support the development of the biometric authentication system. Our approach has already been applied at the specification level [Jür05] where several

severe security flaws had been found. We are currently applying the approach presented here to the source-code level for a prototypical implementation we constructed from the specification. The security analaysis results achieved so far are obtained with the automated theorem prover within less than a minute computing time on an AMD Athlon processor with 1533 MHz. tact frequency and 1024 MB RAM.

6 Related Work

In [FKGS04, FRGG04], aspect models are used to describe crosscutting solutions that address quality or non-functional concerns on the model level. A rigorous technique for specifying pattern solutions in UML is described. It is explained how to identify and compose multiple concerns, such as security and fault tolerance, and how to identify and solve conflicts between competing concerns. [GS04] proposes an approach which models application requirements and designs separately from security requirements and designs in the UML notation. Security requirements are captured in security use cases and encapsulated in security objects separately from the application requirements and objects. One of the benefits of aspect-oriented approaches is reuse of models or patterns and code. [EAK+01] discusses an approach to enhance reuse of code for requirements such as synchronization and scheduling. The authors present a formal design methodology to model the system's concerns based on aspect-orientation. Aspects of AOP are discussed more generally in [EAB02]. [LB04] focuses on the importance of subsystem (pattern) reusability. They propose an Aspect-Oriented Development Framework (AODF) where functional behaviors are encapsulated in each component and connector, while non-functional requirements are tuned separately. To support the modularity of non-functional requirements, they devise Aspectual Composition Rules (ACR) and Aspectual Collaborative Composition Rules (ACCR). Related to the source-code analysis side of our work, [MSRM04] addresses the problem of concept location using an advanced information retrieval method, Latent Semantic, that supports software maintenance and reverse engineering of source code.

Note that although dynamic aspects have been one major focus of aspect-oriented approaches in general, in the case of security, most approaches so far have not concentrated on an integrated approach for weaving in dynamic security aspects at the design level and for constructing and analyzing the code.

7 Conclusion

We explained how to develop and analyze specifications and implementations wrt. dynamic security aspects using aspect-oriented modeling. The approach separates complex security mechanisms from the core functionality to allow a security analysis and verification of the particularly security-critical parts and also of the composed model. Being able to consider security aspects already in the design phase (before a system is actually implemented) is advantageous,

since removing security flaws in the design phase saves cost and time. In practice usually at least part of the code construction is still done manually and is thus again prone to security flaws. We therefore extended our approach to be able to check whether code obtained in the end actually fulfills the security requirements, using an automated security analysis with first-order logic theorem provers.

Experiences from the industrial application project mentioned in Sect. 5 indicate that our approach is quite suited to increase the security of systems developed in practice (exemplified also by the number of security flaws found and removed so far).

Since we focus on the dynamic views of a system, our work is complementary to existing approaches mostly concerned with static views. For future work, it would therefore be very interesting to try to integrate these approaches with the one proposed here. Note that although we concentrate on security aspects in this paper, which pose specific challenges (such as the correct use of cryptographic operations), our approach can be generalized to other non-functional aspects such as dependability by using a suitable extension of UML (see e.g. [Jür03]).

Acknowledgements Assistance from Mark Yampolskiy on the material for the example in this paper is very gratefully acknowledged.

References

[Abs04] AbsInt. aicall. http://www.aicall.de/, 2004.

[BJCR05] M. Broy, J. Jürjens, V. Cengarle, and B. Rumpe. Towards a system model for UML. Technical report, TU Munich, 2005.

[BM03] P. Braun and F. Marschall. The BOTL tool. http://www4.in.tum.de/~marschal/botl/index.htm, 2003.

[DY83] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, IT-29(2):198–208, 1983.

[EAB02] T. Elrad, O. Aldawud, and A. Bader. Aspect-oriented modeling: Bridging the gap between implementation and design. In Don S. Batory, Charles Consel, and Walid Taha, editors, *GPCE*, volume 2487 of *Lecture Notes in Computer Science*, pages 189–201. Springer, 2002.

[EAK+01] T. Elrad, M. Aksit, G. Kiczales, K.J. Lieberherr, and H. Ossher. Discussing aspects of AOP. *Commun. ACM*, 44(10):33–38, 2001.

[FKGS04] R.B. France, D. Kim, S. Ghosh, and E. Song. A UML-based pattern specification technique. *IEEE Trans. Software Eng.*, 30(3):193–206, 2004.

[FRGG04] R.B. France, I. Ray, G. Georg, and S. Ghosh. Aspect-oriented approach to early design modelling. *IEE Proceedings - Software*, 151(4):173–186, 2004.

[GS04] H. Gomaa and M.E. Shin. Modeling complex systems by separating application and security concerns. In *ICECCS*, pages 19–28. IEEE Computer Society, 2004.

[Jür02] J. Jürjens. UMLsec: Extending UML for secure systems development. In J.-M. Jézéquel, H. Hußmann, and S. Cook, editors, *UML 2002 – The Unified Modeling Language*, volume 2460 of *LNCS*, pages 412–425. Springer, 2002.

[Jür03] J. Jürjens. Developing safety-critical systems with UML. In P. Stevens, editor, *The Unified Modeling Language (UML 2003)*, volume 2863 of *LNCS*, pages 360–372. Springer, 2003.

[Jür04a] J. Jürjens. *Secure Systems Development with UML*. Springer, 2004.

[Jür04b] J. Jürjens. Security analysis tool (webinterface and download), 2004. http://www4.in.tum.de/csduml/interface.

[Jür05] J. Jürjens. Sound methods and effective tools for model-based security engineering with UML. In *27th International Conference on Software Engineering (ICSE 2005)*. IEEE Computer Society, 2005.

[LB04] J.-S. Lee and D.-H. Bae. An aspect-oriented framework for developing component-based software with the collaboration-based architectural style. *Information & Software Technology*, 46(2):81–97, 2004.

[MSRM04] A. Marcus, A. Sergeyev, V. Rajlich, and J.I. Maletic. An information retrieval approach to concept location in source code. In *WCRE*, pages 214–223. IEEE Computer Society, 2004.

[SFWW03] J. Schumann, B. Fischer, M.W. Whalen, and J. Whittle. Certification support for automatically generated programs. In *HICSS*, page 337, 2003.

[SS01] G. Sutcliffe and C. Suttner. The TPTP problem library for automated theorem proving, 2001. Available at http://www.tptp.org.

[SW00] G. Stenz and A. Wolf. E-SETHEO: An automated^3 theorem prover. In R. Dyckhoff, editor, *TABLEAUX 2000*, volume 1847 of *LNCS*, pages 436–440. Springer, 2000.