A Tool-Supported Method for the Design and Implementation of Secure Distributed Applications

Linda Ariani Gunawan, Frank Alexander Kraemer, and Peter Herrmann

Department of Telematics
Norwegian University of Science and Technology (NTNU)
Trondheim, Norway
{gunawan,kraemer,herrmann}@item.ntnu.no

Abstract. We describe a highly automated and tool-supported method for the correct integration of security mechanisms into distributed applications. Security functions to establish and release secure connections are provided as self-contained, collaborative building blocks specifying the behavior of several parties. For the security mechanisms to be effective, the application-specific model needs to fulfill certain behavioral properties, for instance, a consistent start and termination. We identify these properties and show how they lead to correct secured applications.

1 Introduction

Security is a significant aspect in the design and implementation of networked systems. Despite this fact, security is still an aspect that many developers consider as one of the last steps during system development [1]. One of the reasons is that developing secure applications needs substantial expertise [2, 3]. This level of expertise may exceed what one can expect from average developers who are rather experts in their specific application domains. For security experts, on the other side, it may likewise be difficult to cope with the domain-specific applications. In addition, even when security mechanisms themselves are sufficiently understood, their integration into an application needs careful consideration in order to be effective (see, for instance, [4] for integrating TLS [5] into application layer protocols). This means that both knowledge of a specific application domain and of the appropriate security mechanisms are needed.

Since security is an aspect that spreads and entangles with many components in an application [6, 7], it can be difficult to separate this aspect from the application's functional part. However, from our experience with model-driven design, analysis and refinement of distributed applications, we see that there are cases in which, given that certain preconditions hold, some security mechanisms can be effectively integrated into the system by a highly automated process which we can support by tools. The employed strategy here is three-fold:

1. We check that an initially unsecured system fulfills certain necessary structural and behavioral properties.

2. We automatically encapsulate parts of the specification with security mechanisms.
3. The protected specifications are integrated into the complete system model.

The preconditions address mainly functional properties and can be easily understood by domain experts and checked by tools. Due to the high degree of automation, the process of introducing the security mechanisms only requires basic knowledge. As a result, domain experts can spend most of their attention on their respective fields, while security experts may focus on the provision of general security mechanisms that can be applied to an entire class of systems.

In this paper, we present a highly automated and tool-supported approach that extends our previous method on model-driven engineering SPACE [8] with its tool-suite Arctis [9] and implements the strategy described above. As will be detailed later, the method benefits from the collaborative specification style of SPACE to model both functional and security aspects. We show that the secured applications produced by the method fulfill important security goals and hence the method correctly integrates security mechanisms into functional models.

1.1 Collaborative Specification Style

To specify a distributed application, we use a specification style that is based on collaborative building blocks [8]. These are the major design units which can cover both local behavior within components and interactions between them. As we will later see, this has the benefit that security mechanisms for communication, which are inherently collaborative, can be expressed by self-contained building blocks. Moreover, the specification style enables a rapid application development since, as shown in [10], on average more than 70 % of a system specification can be taken from reusable building blocks provided in various libraries [11]. The semantics of the specification is formally defined in [12] which makes it possible to guarantee important system properties, e.g., the correct usage of building blocks and the boundedness of communication, by the model checker included in Arctis [9].

As an example, we built an application for a telemedical consultation that enables patients to consult a physician by chat. Since the physician cannot determine the exact time when a patient can be served, the application contains an active waiting queue, in which the patient registers and gets informed about when the consultation begins. Figure 1 shows the specification of the application. It is a UML 2.0 activity consisting of two partitions, *patient* and *physician*, which denote the two participating entities for this system. The activity is composed from six building blocks that refer to subordinate activity diagrams (ignore for now the one labelled *Secure Chat*). The blocks have pins on their frames that are used to compose their behavior. A starting pin (see the legend in Fig. 1) is used to start a block which is only allowed when it is idle. Once a block is started, data can be passed in either direction via streaming pins. Termination of a block is modeled by terminating pins. The blocks *u1: Queue UI* and *u2: Chat UI* on the patient side encapsulate the user interfaces for the queueing function and the

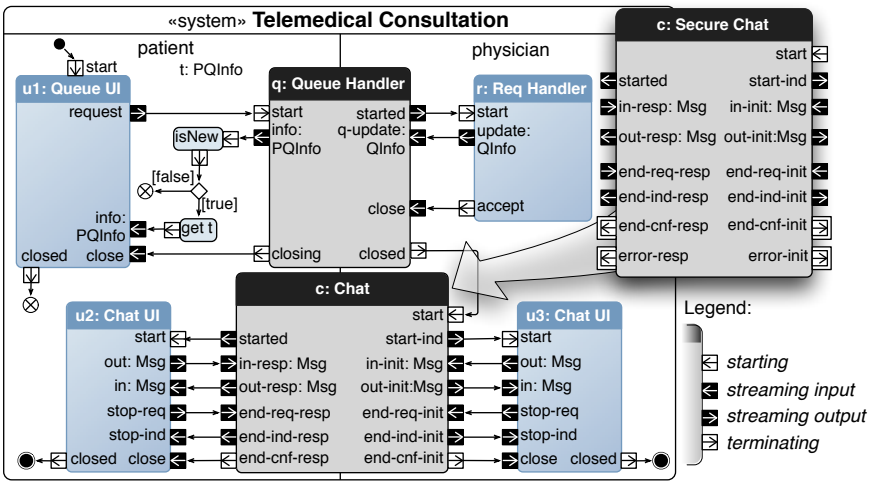

Fig. 1. Medical Consultation System

actual chat. The physician part has similar blocks, i.e., *r* and *u3*. The building blocks *q: Queue Handler* and *c: Chat* are assigned to both participants. They are collaborations, and their task is to manage interactions between the patient and the physician. The protocol to handle the queueing is encapsulated in *q: Queue Handler*, while *c: Chat* contains the necessary interactions for the chat.

The application is initiated on the patient side with the initial node (•) that starts *u1: Queue UI*. Then, via this UI, the patient can issue a request to get medical consultation, which is expressed by a flow from the streaming output *request* to the starting pin of *q: Queue Handler*. A signal carrying this request is sent to the physician part which further starts block *r: Req Handler*. Thereafter, the queue information in this block is appended, encapsulated in an *QInfo* object, and sent via pin *update*. Block *q* receives this, extracts necessary data, and forwards the data via streaming output *info*. This information is futher displayed to the patient. Updates on the queue information are also sent periodically in the same manner. However, an update is forwarded to the patient UI only if it is new, as denoted by operation *isNew* which contains a corresponding Java method and the two choices following a decision node (◇). When it is the patient's turn to be served, the block *r* emits an event *accept* and terminates. This event triggers the closing of blocks *u1* and *q*. Then, the *c: Chat* block is started.

As depicted in Fig. 1, the start of the chat collaboration also initiates the related UIs. Thereafter, both participants can send and receive messages at will via pins *in* and *out* on their respective UIs. This message exchange is governed by block *c: Chat* as shown by the flows through the corresponding pins. One side may decide at any time to terminate the chat, which is also managed by the chat block. To handle the termination consistently, it declares its intention via either pin *end-req-resp* or *end-req-init*, whereupon the other side receives an

indication. Any remaining chat messages that are obtained by the block before the indication are sent. Thereafter, the patient and subsequently the physician sides of the chat collaboration are closed. This also accounts for a situation in which both sides decide to terminate at the same time.

1.2 Security Goals

In the following, we focus on the protection of signals carrying sensitive or private information exchanged between two entities. Our method to integrate this protection into a distributed application such as the telemedical consultation in Fig. 1 ensures the fulfillment of three security goals as motivated below.

Due to the computation and resource penalty from executing cryptographic operations (like encryption or digital signature generation), the secure connection is generally only used as long as required. Therefore, in the life span of a distributed application, there may exist unsecure and secure phases. It is important to ensure all signals that require protection are only transferred in a secure mode. We formulate this goal as follows:

G1 While in secure mode, all signals are transferred with protection. In other words, during the secure mode either a correct, secured transmission of flows occurs or an attack is detected.

To avoid vulnerabilities due to the interference of application-specific communication with the security functions, no application signals can be transferred during secure mode establishment and termination. Therefore, our method realizes the following goal:

G2 The four phases, namely *unsecure mode*, *secure mode establishment*, *secure mode* and *secure mode termination*, are all distinct, i.e., signal transmissions belonging to different phases do not interleave.

As shown in [13–15], combining several secured security mechanisms can potentially lead to vulnerabilities. To avoid this problem, the applications obtained by using our method must not contain duplication of security mechanisms. We formulate this last goal as follows:

G3 The protected system specifications do not contain duplication of security mechanisms.

1.3 Overview of the Method

Figure 2 illustrates how our method transforms a functionally correct but unprotected system specification through three systematic steps into a protected one that fulfills the goals listed above. As a first step, the specification is assessed in order to determine which collaborations need to be protected. Risk-based assessment like in [16] can be applied here. This step is performed manually since the information to be protected is different in each application. Thereafter, using our analysis tools, both the system specification and the to-be-secured collaborations are checked for the fulfillment of properties which we will later discuss in detail.

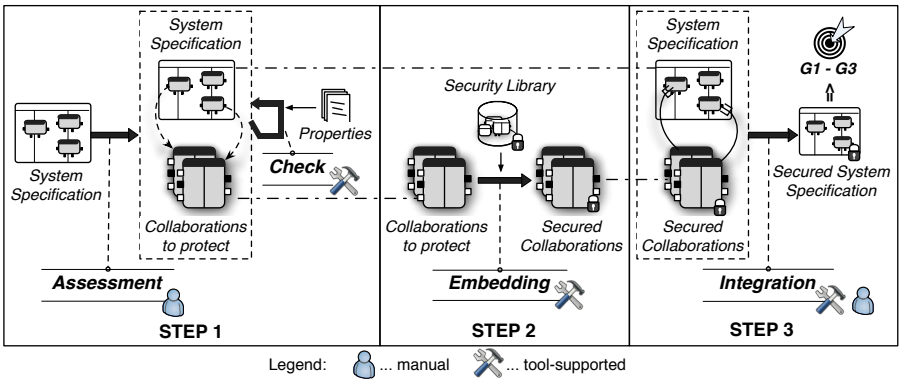

Fig. 2. Security Integration Method with Tool-Support

When all the properties are fulfilled, the next step is to embed security mechanisms into the collaborations to protect. These mechanisms that are also modeled as reusable collaborative building blocks are taken from a security library. This step is highly automated utilizing graph transformation techniques and produces, for each block to protect, a secured one that integrates the security blocks with the corresponding original block.

The last step to obtain a protected system specification is to integrate secured collaborations into the unprotected blocks. This step includes replacing the collaborations to be secured with their corresponding protected collaborations as illustrated by the *Secure Chat* block in Fig. 1. The substitution process is also highly automated. However, a manual inspection may still be needed since the new block contains additional pins as will be described further in Sect. 3.3.

In the following, we will describe the dedicated building blocks used to protect collaborations in Sect. 2. Thereafter, in Sect. 3 we detail the integration method illustrated in Fig. 2 and apply it to our telemedical consultation example. The discussion in Sect. 4 provides a proof-sketch that shows how specifications treated by our method fulfill the security goals stated above. We end with a discussion of related approaches in Sect. 5 and concluding remarks in Sect. 6.

2 Building Blocks for Secure Connections

As shown in Fig. 2, security functions are integrated by dedicated blocks. These blocks require certain mechanisms integrated into the underlying runtime-support system responsible for executing components, which we describe first.

2.1 Preparing the Runtime Support-System

To execute an application, our tool Arctis [9] automatically generates a software component for each partition of a system specification [17, 18]. As an example,

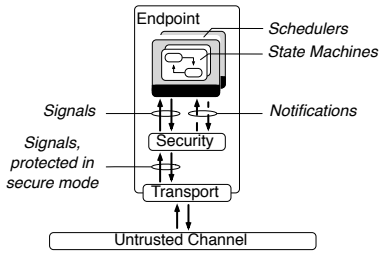

Fig. 3. An Endpoint

the system in Fig. 1 is realized by one component for the patient and one for the physician. Each component consists of four parts, namely state machines, schedulers, a security module, and a transport module, as depicted in Fig. 3. *State Machines* are automatically generated from the collaboration-oriented models and contain the application-specific execution logic in the form of states and transitions. Every transition is triggered by an event dispatched by the *Schedulers*. The *Transport Module* simply sends and receives signals using the underlying channels and performs the necessary serialization of data.

To protect signals conveyed by the untrusted channel, we use the *Security Module* between the schedulers and the transporter. This module handles the establishment and termination of secure modes with other endpoints. When operating in a secure mode, all signals between any state machine handled by a pair of endpoints are protected by encryption and integrity measures. We employ symmetric encryption and keyed message authentication code which use generated keys derived from a shared secret that is negotiated during a secure mode creation. SSLEngine [19], a transport-independent Java implementation of TLS [5], is used to provide this security feature.

In order to correctly apply the protection, the security module needs to communicate with the application logic through the following two mechanisms:

1. The application may obtain a handle to the security module and invoke particular methods.
2. The security module may send a notification about an occurrence of a certain event to the application. This notification is sent by using an internal signal to a specified state machine via its corresponding scheduler.

2.2 Building Block for the Secure Mode Establishment

The establishment of a secure mode (SM) between two components needs the cooperation of the security modules in both endpoints, which is why it is encapsulated as the collaboration shown in Fig. 4. After the block is activated, the initiator gets the required parameters which are encapsulated in an *SMParam* object and obtained via the block *m1: Key Manager*.[1] Next, the operation *pre-*

[1] This block manages the public/private keys of an entity and thus is part of a public key infrastructure. For brevity, this block is not discussed further in this paper.

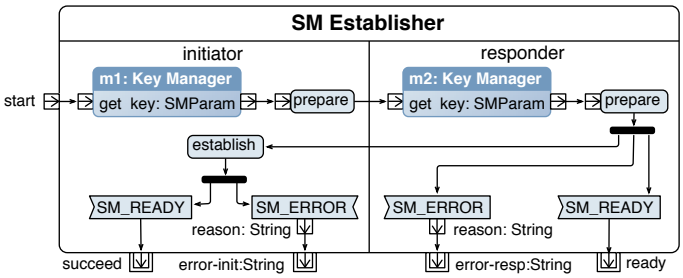

Fig. 4. Secure Mode Establisher Block

pare is invoked in which the parameters are passed to the corresponding security module as described by the first mechanism of communication explained in Sect. 2.1. Subsequently, the responder also obtains and sets its parameters, and is then ready to receive a notification from its own security module using the second communication mechanism. This notification is either *SM_READY* or *SM_ERROR*, indicating a successful respective failed attempt. Since these events are mutually exclusive, the terminating pins following these nodes are shown with additional boxes. After the preparation, the initiator starts the setup via operation *establish* that obtains a handler of its security module and invokes a provided method in the module. Thereafter, it is ready to receive a signal communicating the result.

The outcome of the establishment is communicated consistently by the security modules of both participants to their respective application partitions. If the secure connection is successfully established, the responder and subsequently the initiator get signal *SM_READY*. When the security module on the initiator side detects an error, the initiator and later the responder get *SM_ERROR*. Conversely, the responder and thereafter the initiator receive notifications if an error is detected by the responder's security module.

The establishment process implements the TLS Handshake protocol [5]. During the handshake, both communicating entities are mutually authenticated by means of a public key certification mechanism. Moreover, a shared secret is negotiated, from which the symmetric keys are generated. The TLS protocol guarantees that the negotiation is secure and reliable, i.e., the shared secret is only accessible to the participants and a modification in the messages is detected.

2.3 Building Block for the Secure Mode Termination

Termination of an SM must be handled carefully in order to thwart a truncation attack [5]. In our method, this process is executed by a pair of security modules implementing the TLS Close Alert protocol [5], and encapsulated in block *SM_Terminator*. Similarly to the establisher block, the terminator block is also a collaboration between an initiator and a responder. However, it does not necessarily mean that an entity that takes the initiator role for the setup must also be

the initiator of termination, since these roles can be assigned independently. The details of block *SM Terminator* is not shown here as it resembles the successful case of the establishment block. It is worth pointing out that this termination does not necessarily stop the transport module. Depending on its specification, the application may continue to communicate in the unprotected mode.

2.4 Building Block for the Secure Mode Error Listener

A security exception, such as a failed integrity check, may occur when transferring protected signals. If a security module of an endpoint detects this, it discards subsequent messages, sends an error alert to its peer, and informs the related application. Thus, the application must be prepared to receive this notification. We provide the *SM Err Listener* block (see Fig. 5) to implement this functionality. Upon activation, the block is ready to listen for a notification until it is stopped or a security exception does take place.

3 Integration of the Security Mechanisms

As outlined in Sect. 1.3, the integration of the security mechanisms is a process of three steps, which we will describe below.

3.1 Step 1: Risk Assessment and Check of Preconditions

First, a risk-based assessment is performed in order to determine which collaborations need protection. For the system in Fig. 1, block *t: Queue Handler* does not need to be secured due to the low value of information contained in the block. In contrast, collaboration *c: Chat* may transfer private information, e.g., medical records, and thus must be protected.

Next, the system level specification and the collaborations to be secured are checked for some properties. In order to apply our method of integrating security blocks in a highly automated way, we require that the system level specification fulfills the following properties:

- **S1** The system level does not directly contain any flows that cross partitions, i.e., all communication is encapsulated by collaborations.
- **S2** All collaborations have exactly two participants.
- **S3** While a collaboration to be secured is active, no other collaboration between the same pair of participants may be active as well.

Properties **S1** and **S2** are structural and can be checked by the syntactic inspection tool in Arctis, while the behavioral property **S3** is ensured by model checking. Our experience shows that many applications, including the one depicted in Fig. 1, can be designed to satisfy these properties. A specification that does not conform to the rules can be changed by, for instance, introducing collaborations that encapsulate other building blocks and direct communications.

In order to ensure that all application-specific signals of a secured collaboration are protected effectively, we identify the following properties that a collaboration to be protected must fulfill:

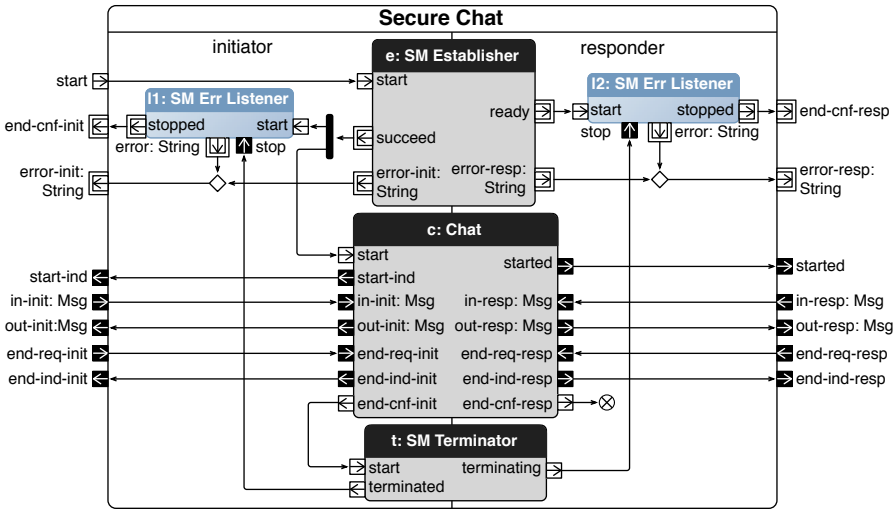

Fig. 5. Building Block for the Secure Chat

- **C1** The collaboration must have exactly one starting pin, which means that the collaboration is initiated by one party only.
- **C2** When a terminating pin of an activity partition is reached, there must not be any signals in the input queue of the partition or any other activity node that could trigger further behavior.
- **C3** The collaboration to be secured does not directly or indirectly contain any security blocks.

The structural properties **C1** and **C3** can be checked by a syntactic inspection, while property **C2** is behavioral and is ensured by model checking. An analysis of the telemedical example shows that the system specification fulfills the properties **S1** .. **S3**, and while **C1** .. **C3** are satisfied by the *Chat* block.

3.2 Step 2: Embedding Security Functions

Once all properties are satisfied, a collaboration to be protected and the blocks from Sect. 2 are automatically composed into a larger collaboration. Applying this step to block *Chat* results in the *Secure Chat* collaboration depicted in Fig. 5. *e: SM Establisher* and *t: SM Terminator* are built-in before and after *c: Chat*. The flow from the pin *succeed* of the establisher block to the starting pin of the application shows that the chat can only begin after a successful secure mode setup. Likewise, the termination is started when the *c: Chat* block is fully terminated, i.e., on the initiator side. In each partition, a local block *SM Err Listener* is added to make the collaboration ready to receive an error notification.

Note that the pins of block *Secure Chat* are the same as *c: Chat*'s plus pins *error-init* and *error-resp*. These additional terminating pins are used to report a security exception that may occur. Through them, a notification may indicate

either a failure during SM establishment or when the blocks *SM Err Listener* report exceptions between a successful setup and a normal termination. Thus, after being started, collaboration *Secure Chat* may proceed normally as the chat application or is stopped at any time due to an error.

3.3 Step 3: Integrating the Secured Collaborations

The collaborations marked for protection in the system specification are replaced by their secured counterparts, as illustrated with collaboration *Chat* in Fig. 1, which is replaced by the *Secure Chat*. As noted above, the secured blocks are from the outside structurally and behaviorally similar, but with exception pins added. In general, it is up to the application to decide, what should happen upon such an exception. For the given example, one solution would be to inform users locally via the UIs and terminate both user clients. Since the collaborations to protect are on system level, the changes due to the additional pins are manageable; in the end, this is the necessary join point between a security function and an application logic that cannot be hidden.

4 Discussion and Proof

We ensure the correct integration of the secure mode solution by showing that the integration result, e.g., a secured, distributed application, fulfills the three security goals stated in Sect. 1.2. Before sketching the proof, we summarize the structural details of integrating the solution into an application-specific collaborative block C in so-called *implementation directives* as follows:

- **I1** **Establishment**: Block *SM Establisher* is built-in directly before the start of collaboration C, i.e., the termination of the block directly triggers the start of C. In particular,
  - **I1.1** Pins *error-init* and *error-resp* of block *SM Establisher* must never lead to the start of collaboration C.
  - **I1.2** Pin *succeed* of block *SM Establisher* must directly lead to the start of a local block *SM Err Listener*.
  - **I1.3** Pin *ready* of block *SM Establisher* must directly lead to the start of another local block *SM Err Listener*.
  - **I1.4** Pin *error* of both local blocks from **I1.2** and **I1.3** must not lead to the start of any other collaboration between the two participants of C including block *SM Terminator*.
- **I2** **Termination**: Block *SM Terminator* is built-in after the termination of collaboration C and before any other collaboration attached to the same pairs of participants is activated. In particular,
  - **I2.1** Pin *terminated* of block *SM Terminator* leads to the termination of local block *SM Err Listener* from **I1.2**.
  - **I2.2** Pin *terminating* of block *SM Terminator* leads to the termination of local block *SM Err Listener* from **I1.3**.

The property **S2** which states that all collaborations have exactly two participants guarantees that our integration method correctly applies the security functions designed for two entities. The rest of the properties for the system level specification and for the collaborations to be protected (Sect. 3.1) together with the implementation directives and the behavior of the security module in the execution environment satisfy the security goals **G1** to **G3** (Sect. 1.2). In the following, we will give the sketches of the corresponding refinement proofs:

The fulfillment of **G1** that ensures all signals are transferred protected during a secure mode phase is as follows: Due to **S3**, during the activity of a secured collaboration, no other collaboration between the same participants is active. Further, **S1** guarantees that there is no direct flow between those participants. Therefore, all communication between the two participants takes place in the secured collaboration. Due to **I1** and **I2**, the secured collaboration is only active after a successful SM establishment and before the start of SM termination. The security module of the execution environment assures that all signals exchanged between the participants of the secured collaboration are protected and a reception without resulting in the sending of error notification to the application means that no attack was detected.

A security attack can be detected by the application since **I1.2** and **I1.3** guarantee that during the lifetime of a secured collaboration the two local blocks of type *SM Err Listener* on both participants of the secured collaboration are active. Thus, an error will be detected due to the behavior of the security module in the execution environment. Furthermore, **I1.4** guarantees that the error leads to an error state and not into the normal proceeding of the function. ■

To prove **G2** claiming that all four phases are distinct, we show that in a normal condition they occur in sequence as follows:

- *unsecure mode* → *secure mode establishment*
  Due to **I1**, the *secure mode establishment* phase occurs before the secured collaboration is active. Further, **S3** guarantees that during this establishment process, no other collaboration between the participants of the secured collaboration is active. Due to **S1**, there is no other communication between the participants. Consequently, phases *unsecure mode* and *secure mode establishment* are mutually exclusive and the former leads directly to the later.
- *secure mode establishment* → *secure mode*
  Because of **I1** and **I1.1**, the secured collaboration is only started upon successful finishing of the SM establishment process.
- *secure mode* → *secure mode termination*
  This is guaranteed by **I2** and **C2**. Due to **I2**, the SM termination only begins when the secured collaboration is inactive. Moreover, **C2** guarantees that the secured collaboration contains no other signal in its queues.
- *secure mode termination* → *unsecure mode*
  Due to **I2** and **S3**, any collaboration between the participants of the secured collaboration can only be activated when SM termination process ends properly. Further, **I2.1** and **I2.2** guarantee that the remaining security function,

i.e., listening for exception, is terminated as well. This shows the sequence of phase *secure mode termination* to phase *unsecure mode*. ■

The no-duplication mechanism goal of **G3** is fulfilled by **C3** that guarantees neither SM setup nor termination is executed more than once in order. ■

Employing the highly automated method described above contributes to make the task of developing secure applications less daunting. The integration of the secure mode solution can be applied to various application domains since protection of sensitive data is generally required in distributed systems. Moreover, the specification style makes changes in both functionalities and security aspects manageable. However, further investigation is needed to determine the applicability of the method for integrating other protections effectively.

As a proof of the practicability of our method, we also implemented the presented example using Arctis. The *c: Chat* block in Fig. 1 is replaced with block *c: Secure Chat*. To inform the users about an event of security exception (Sect. 3.3), the additional pins need to be connected to the UI blocks. Therefore, a streaming input that is connected to the pin *closed* is added to block *Chat UI*. Then, two components are generated automatically using Arctis. The component for the physician is running on the Java SE platform and the one for the patient on an Android phone.

5 Related Work

Some approaches and tool supports have been proposed to integrate security aspects in distributed applications. Middleware technologies such as Java RMI [20], Web Services and CORBA [21] are extended with protection support in order to create secure applications. Li et al. develop the RMI toolkit [22] that enables developers of RMI-based application to adopt security feature. Security standards are defined for CORBA in [23] and Web Services in [24]. The implementation of these approaches is different from our model-based method, since manual changes in the code is required.

Model-based secure system development methods have also been suggested. UMLsec [25] is a profile on security requirements that can be attached to UML diagrams to evaluate a specification for security. SecureUML [26] is an extension of UML to specify role-based access control policies. These approaches may still require much expertise on security since security solutions are developed manually. However, tools are also developed to help analysing the secured system.

Other work that attempts to integrate security concerns is aspect-oriented modeling. Here, security mechanisms are specified as aspects, and weaved into base specifications at join points. Although there is no standard, some approaches and tool-support have been proposed, see for example [6, 7, 27]. Both our method and aspect orientation try to integrate protection mechanism in a highly automated way. However, many of aspect-oriented approaches do not consider the functional changes after the integration. Our method limits the changes only on the system level specifications so that they are manageable.

6 Concluding Remarks

We presented a comprehensive method to integrate a secure communication mechanism, in which building blocks realizing dedicated security functions could be automatically and consistently integrated into an application if certain preconditions are met. We have shown that, given that these preconditions hold, the security goals are in fact fulfilled.

In the future, we will extend our method in several ways. Similar to the building blocks facilitating the secure mode, we will add further blocks to our security library to support also other security mechanisms like access control. Due to the formal basis of our method, we can analyze the specifications expressed by UML activities thoroughly. This enables tool support that can ensure the correct integration of security patterns [28], so that they are effective.

We will also exploit the formal nature of our collaboration-oriented models for the security analysis. The strategy here is two-fold: We analyze an existing functional specification for behavioral and structural properties that are relevant for the security aspect and provide a recommendation of adequate protection mechanisms based on these properties. To reveal other weaknesses, the models are translated to input for other security analysis tools such as Scyther [29].

References

1. Mouratidis, H., Giorgini, P.: Integrating Security and Software Engineering: Advances and Future Vision. IGI Global (2006)
2. Anderson, R.J.: Security Engineering: A Guide to Building Dependable Distributed Systems. John Wiley & Sons, Inc. (2008)
3. Lampson, B.W.: Computer Security in the Real World. Computer **37** (2004) 37–46
4. Rescorla, E.: SSL and TLS: Designing and Building Secure Systems. Addison-Wesley (2001)
5. Dierks, T., Rescorla, E.: The Transport Layer Security Protocol (TLS) version 1.2. The Internet Engineering Task Force (IETF). (August 2008) RFC 5246.
6. Georg, G., Ray, I., Anastasakis, K., Bordbar, B., Toahchoodee, M., Houmb, S.H.: An Aspect-Oriented Methodology for Designing Secure Applications. Information and Software Technology **51**(5) (2009) 846 – 864 Special Issue: Model-Driven Development for Secure Information Systems.
7. Mouheb, D., Talhi, C., Lima, V., Debbabi, M., Wang, L., Pourzandi, M.: Weaving security aspects into uml 2.0 design models. In: AOM '09: Proceedings of the 13th workshop on Aspect-Oriented Modeling, ACM (2009) 7–12
8. Kraemer, F.A.: Engineering Reactive Systems: A Compositional and Model-Driven Method Based on Collaborative Building Blocks. PhD thesis, Norwegian University of Science and Technology (August 2008)
9. Kraemer, F.A., Slåtten, V., Herrmann, P.: Tool Support for the Rapid Composition, Analysis and Implementation of Reactive Services. Journal of Systems and Software **82**(12) (December 2009) 2068–2080
10. Kraemer, F.A., Herrmann, P.: Automated Encapsulation of UML Activities for Incremental Development and Verification. In: Proceedings of the 12th Int. Conference on Model Driven Engineering, Languages and Systems (Models). Volume 5795 of LNCS., Springer (2009) 571–585

11. Arctis Website. http://www.arctis.item.ntnu.no/
12. Kraemer, F.A., Herrmann, P.: Reactive Semantics for Distributed UML Activities. In: Formal Techniques for Distributed Systems. Volume 6117 of LNCS. Springer (2010) 17–31
13. Datta, A., Derek, A., Mitchell, J.C., Pavlovic, D.: Secure Protocol Composition. In: FMSE '03: Proceedings of the 2003 ACM workshop on Formal Methods in Security Engineering, ACM (2003) 11–23
14. Krawczyk, H.: The Order of Encryption and Authentication for Protecting Communications (or: How Secure Is SSL?). In: CRYPTO '01: Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology, Springer (2001) 310–331
15. Cremers, C.: Compositionality of Security Protocols: A Research Agenda. Electronic Notes Theoretical Computer Science **142** (January 2006) 99–110
16. Baskerville, R.: Information Systems Security Design Methods: Implications for Information Systems Development. ACM Computing Surveys **25**(4) (1993) 375–414
17. Kraemer, F.A., Herrmann, P.: Transforming Collaborative Service Specifications into Efficiently Executable State Machines. In: Proceedings of the 6th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2007). Volume 7 of Electronic Communications of the EASST., EASST (2007)
18. Kraemer, F.A., Herrmann, P., Bræk, R.: Aligning UML 2.0 State Machines and Temporal Logic for the Efficient Execution of Services. In: Proceedings of the 8th International Symposium on Distributed Objects and Applications (DOA). Volume 4276 of LNCS., Springer (2006) 1613–1632
19. SSLEngine from JSSE. http://java.sun.com/javase/6/docs/api/javax/net/ssl/SSLEngine.html
20. Java Remote Method Invocation. http://java.sun.com/javase/technologies/core/basic/rmi/
21. Object Management Group: Common Object Request Broker Architecture (CORBA/IIOP), version 3.1. (January 2008) formal/2008-01-08.
22. Li, N., Mitchell, J.C., Tong, D.: Securing Java RMI-Based Distributed Applications. In: Proceedings of the 20th Annual Computer Security Applications Conference, ACSAC'04, IEEE Computer Society (2004) 262–271
23. Object Management Group: CORBA Security Service, version 1.8. (March 2002) formal/2002-03-11.
24. OASIS: Web Services Security, version 1.1. (February 2006)
25. Jürjens, J.: Secure System Development with UML. Springer-Verlag (2004)
26. Basin, D., Doser, J., Lodderstedt, T.: Model Driven Security: From UML Models to Access Control Infrastructures. ACM Transactions on Software Engineering and Methodology **15**(1) (2006) 39–91
27. Pavlich-Mariscal, J., Michel, L., Demurjian, S.: Enhancing UML to Model Custom Security Aspects. In: AOM '07: Proceedings of the 11th Workshop on Aspect-Oriented Modeling. (2007)
28. Schumacher, M., Fernandez-Buglioni, E., Hybertson, D., Buschmann, F., Sommerlad: Security Patterns : Integrating Security and Systems Engineering (Wiley Software Patterns Series). John Wiley & Sons (March 2006)
29. Cremers, C.J.: The Scyther Tool: Verification, Falsification, and Analysis of Security Protocols. In: CAV '08: Proceedings of the 20th International Conference on Computer Aided Verification, Springer (2008) 414–418