Compositional Verification of Application-Level Security Properties

Linda Ariani Gunawan and Peter Herrmann

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

Abstract. Automatic model checking can be employed to verify that security properties are fulfilled by a system model. However, since security requirements constrain most, if not all, functional modules of a system, such a proof needs to consider nearly all of the system's control and data flows. For complex real-life applications, that leads to a large state space to be explored effectively restricting the applicability of a model checker. To deal with this problem, we advocate a compositional approach utilizing the features of our model-based engineering technique SPACE. Both functional behavior and security-related aspects are specified using UML 2 activities. Further, we supplement each activity with an interface behavior description which will be extended by a security contract modeling certain security properties to be fulfilled by the activity. This enables us to verify application-level security properties by using contracts instead of their respective activities in model checker runs so that the number of states to be checked is significantly reduced. The approach is exemplified by an Android application example in which one's location must only be shared with certain recipients.

1 Introduction

An often underestimated reason for vulnerabilities and risks in application-level security is that development flaws in real-life software systems are overlooked. For instance, Iyer et al. [1] found out that 18% of all vulnerabilities listed in the *Bugtraq* database resulted from design errors. To avoid such development flaws, we extended our model-based approach SPACE for the development of reactive systems [2] and its tool-set Arctis [3] to support also the creation of secure software [4]. Engineering with SPACE and Arctis profits from the fact that models are a clearer and more concise way to express a system than traditional program code. That makes it easier to keep track of the system behavior. Moreover, due to its formal semantics [5], one can verify by syntactic inspection and model checking that application-level security goals are kept by the system model [6]. Finally, SPACE uses automatic code generation guaranteeing that the implementation is a correct realization of the model [2]. Thus, we can be sure that also the executed code complies with the proven security properties.

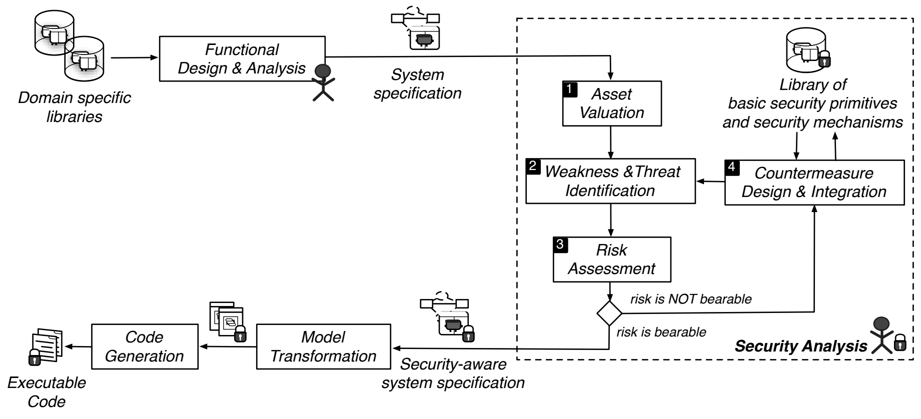
Domain specific libraries
Functional Design & Analysis
System specification
1 Asset Valuation
2 Weakness &Threat Identification
3 Risk Assessment
4 Countermeasure Design & Integration
Library of basic security primitives and security mechanisms
risk is NOT bearable
risk is bearable
Security Analysis
Security-aware system specification
Model Transformation
Code Generation
Executable Code

Fig. 1. Security-enhanced Development Method (taken from [4])

While model checking can be executed with a high degree of automatism, its weak point is the state explosion problem [7] which, in effect, constitutes the limiting factor for applying it to large systems. That is especially relevant if one wants to prove security requirements that often define and constrain all functional modules of a system such that its whole state space has to be considered (see [8]).

To tackle the state explosion problem, we advocate compositional verification that is already used to verify properties related to the functionality [3] and reliability [9] of a system. Here, we utilize the model composition mechanism of SPACE in which behavior is specified by an arbitrary number of UML 2 activities [10]. Like Petri-nets, those are graphs modeling behavior as a flow of tokens between the vertices via the edges. Activities are coupled with one another by call behavior actions that we call *building blocks*. From one viewpoint, a building block refers to a particular behavior expressed by an activity. From the other viewpoint, a designator of a block may be incorporated in another activity, and by so-called pins, tokens may flow between activities. Further, a building block is amended with a behavioral interface description specifying the order of token flows through its pins. One advantage of the approach is a high degree of reuse. A block modeling recurrent behavior can be created once and stored in a library. Thereafter, by adding its designator to other activities in a drag and drop fashion, the behavior modeled by the block can easily be added to various system models. According to our experience, on average 70% of a system model corresponds to building blocks taken from assorted libraries [5].

The other advantage of using building blocks is compositional verification [3]. Here, when proving that an activity fulfills a certain property, we can replace the activity of each of its building blocks by the block's behavioral interface description. As the interface description usually models a much simpler functionality than the activity, the number of states to be checked is vastly reduced (see, e.g., [9]). To use compositional verification also for the proof of security

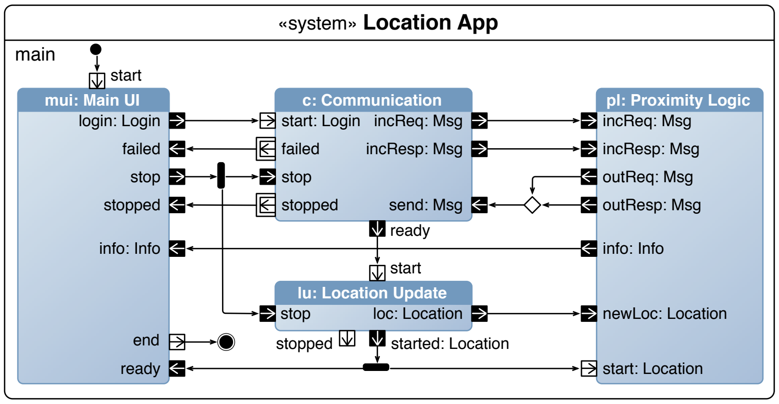

Fig. 2. The Location Application

properties, however, we have to extend the behavioral interface descriptions by so-called *security contracts* modeling security properties to be fulfilled by a block. In the model checker runs, we can then use these block-wise security properties to verify that the overall system fulfills the system-wide ones.

Our approach facilitates the cooperation of application domain engineers with security experts (see Fig. 1). In a first step, the domain engineers develop a system specification utilizing blocks from the domain specific libraries. When the model passes all checks for functional correctness [3], it is handed over to the security experts who subject it to a security analysis. The outcome of this analysis is an amended system specification containing only application-level security risks that seem bearable. In a final step, the extended model is automatically transformed to executable code in a two-step process.

In the context of security analysis, the verification of system security properties is used to detect potential flaws in the design which make the system vulnerable against malicious attacks threatening its assets. Of course, such flaws form a formidable risk for the system and the obvious countermeasure is to change the system model such that its behavior fulfills the security properties.

2 Location Application – an Example

The system specification of our example is depicted in Fig. 2. It is an Android application that allows one to share one's current location, but only to a set of intended recipients, i.e., friends. The specification consists of four building blocks implementing various functionalities. The graphical user interface (UI) block *mui: Main UI* handles the user's input and displays relevant information for the user on the device interface. Block *c: Communication* handles the exchange of messages with peer applications running on other devices. This block encapsulates the *XMPP Client Android* block which allows one to transmit messages through an XMPP server. The current location of a device executing this application is reported by block *lu: Location Update*. Finally, block *pl: Proximity Logic* is responsible to manage location sharing, e.g., to respond to a location request from a friend. It contains three inner blocks as shown in Fig. 3.

The Petri-net like semantics of the UML activities models states as tokens resting in token places and state transitions as moving tokens along directed activity edges [10]. In SPACE, all behavior follows the run-to-completion characteristics [5]. This means, transitions are triggered by observable events, namely, the reception of signals and the expiration of local timers, and completed by reaching a stable state from which the next transition may be carried out.

The location application in Fig. 2 begins with a token flowing from the initial node (●) and activating the UI block. Thereafter, the system waits until the device user enters the necessary credentials to use an XMPP server. As soon as the credentials are received, a token carrying the data (in an object of type *Login*) moves from pin *login* to the starting pin of the communication block. Upon successful login, the application proceeds by initiating block *lu: Location Update* to obtain the present geographical location. Subsequently, a token carrying the location data emits from pin *started* of block *lu* and passes through a fork node which duplicates the token. One token is directed to pin *start* of block *pl: Proximity Logic*, while the other one is forwarded to pin *ready* of the UI block. Updates on the current position are reported by block *lu* via pin *loc* and consumed by block *pl* through pin *newLoc*.

The specification shown in Fig. 2 also includes behavior that handles unsuccessful credential verification and an input from the user to stop the application. However, for brevity we do not detail this here, but rather focus on the location sharing functionality which is handled mainly by block *Proximity Logic*.

As depicted on the left side of Fig. 3, block *Proximity Logic* becomes active when a token carrying location data flows from parameter node *start* and passes through a fork node. The downward pointing edge leaving the fork node shows that a token with the location data initializes block *h: Message Handler*. The other outgoing edge indicates that the Java operation *getFriendList* is executed. The output of this operation is a list of friends which is stored locally. This list is forwarded to a fork node with three outgoing edges, one of which initializes block *g: Req Generator*. The second one sends the list of friends to block *h* while the third directs a token through a merge node (◇) to a timer which is started. When the timer expires, block *g* generates location requests, one for each friend in the list, and emits them one-by-one via pin *aReq*. A token flowing through pin *done* indicates that all requests have been yielded and the next batch of requests can be generated when the timer expires again.

The inner block *b: Reactive Buffer* decouples message reception from message handling and, hence, is used to buffer messages while block *h* is busy processing one message. A message is received through pin *add* of the buffer. When the buffer is empty, it is emitted immediately via pin *out*; otherwise, it is buffered. Invoking pin *next* will get either the subsequent message in the buffer (via pin *out*) or an indication that the buffer is empty (pin *empty*). Three types of messages

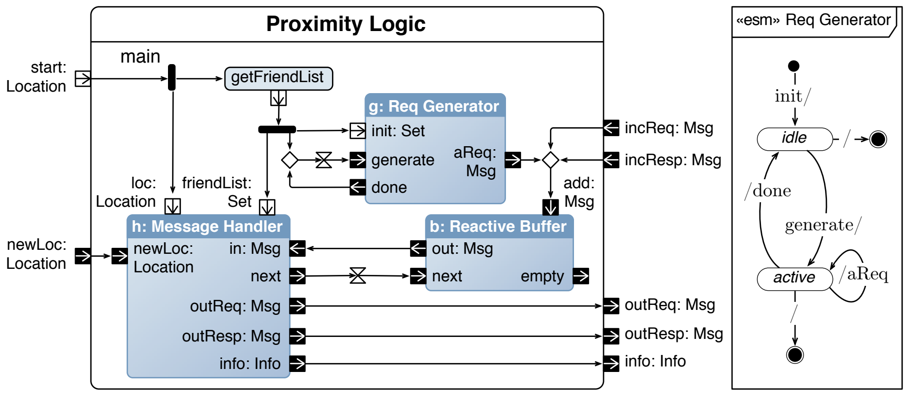

Fig. 3. Block Proximity Logic

are buffered and handled, namely, generated requests, requests from peer applications running in different devices, and responses to the generated requests. A message is received by the message handler block via pin *in*. Depending on the message type and additional constraints, one of the following four alternative behaviors is taken: (1) If the message is a generated request, it is emitted via pin *outReq*. (2) If the message is a request from a person in the friend list, a response containing the latest location is created and emitted via pin *outResp*. (3) If the message is a response to a generated request and the friend's location is near, a notification is emitted via pin *info* (4) For all other cases, the message is dropped. In addition, a token is emitted via the output pin *next* which after a certain latency guaranteed by a timer leads to obtaining the subsequent message from the buffer. The flows via the pins *outReq*, *outResp* and *info* of building block *h* are forwarded to the pins of the same name of the block *Proximity Logic* such that outgoing requests and responds are further sent to the communication block while notifications are forwarded to the UI block (see Fig. 2).

3 Interface Contracts

Except for system-level blocks like the one in Fig. 2, building blocks are supplemented with behavioral interface descriptions. As modeling technique for the interface behavior, we use so-called External State Machines (ESMs) [11] that specify the possible ordering of events visible on the activity pins. The ESM of the block *Req Generator* is depicted on the right side of Fig. 3. It shows that this block starts by receiving a token through pin *init* and entering state *idle*. Thereafter, the block can receive a token via pin *generate* upon which it will emit requests, one at a time, via pin *aReq*. After having generated requests to a list of recipients, the block returns to state *idle* emitting a token via pin *done*. Later, the next batch of requests can be created upon receiving a new *generate* event. The transitions labeled with / show that block *Req Generator* allows its surrounding block, in our case the *Proximity Logic*, to terminate it anytime.

An ESM must be respected both by the activity and its environment in order to guarantee a correct interaction between them. Such property can be verified automatically by a model checker due to the formal semantics of the activities [5]. As mentioned in the introduction, the ESMs enable compositional verification of a system specification: After proving that an activity and its corresponding ESM are consistent, we can represent the blocks of an enclosing activity by their ESMs instead of their activities when model checking that the enclosing activity fulfills certain properties.

Compositional verification is also applied for the verification of reliability and dependability issues. Since the reliability of systems is often guaranteed by using several instances of a critical component and the ESMs are not suited to describe the interface behavior of such multi-instance components, we extended them with auxiliary variables which can be used in transition guards and effects. The resulting interface descriptions are named *Extended External State Machines* (EESMs) [12]. Further, an extension of the EESMs enables us to specify indeterministic interface behavior following from component failures, e.g., non-responsiveness or a reset to the initial state. In consequence, we could reduce the number of states to be model checked by several orders of magnitude (see [9]). This encouraging result has lead us to use compositional verification also for security properties which will be discussed in the following.

4 Modeling Security-Relevant Aspects

A highly relevant asset of applications running on modern smartphones is the phone's location which can be retrieved by the built-in GPS receiver or by triangulation of WiFi base stations. Of course, the location data must not leak to unauthorized principals since that would violate the privacy of the phone user and might also be a severe risk for her/his personal safety. Thus, with respect to application-level security we have to avoid that an erroneous system layout may lead to the unauthorized transmission of the location information. In the example presented in Sect. 2, for instance, we have to guarantee a security property $\mathcal{P}$ expressing that "*one's geographical position may only be sent to one's friends*".

As described in [3], the semantics of the SPACE approach and its tool-set Arctis is based on Leslie Lamport's *Temporal Logic of Actions* (TLA) [13]. This enables us to specify security properties like $\mathcal{P}$ by abstract system specifications or invariants in TLA and use the model checker TLC [14] to verify that they are fulfilled by the TLA representation of a SPACE model.

A suitable notation for the security contracts used to add security properties to the interface contracts are the EESMs [12]. They allow to insert additional variables and constants in transition guards and effects. As an example, we list the EESM of the block *Proximity Logic* in Fig. 4. It uses three control states, i.e., the initial state (●), *_idle* and *pl_active*. Besides of the pin identifiers similar to those used in the ESMs (see Sect. 3), a transition can be provided by a guard

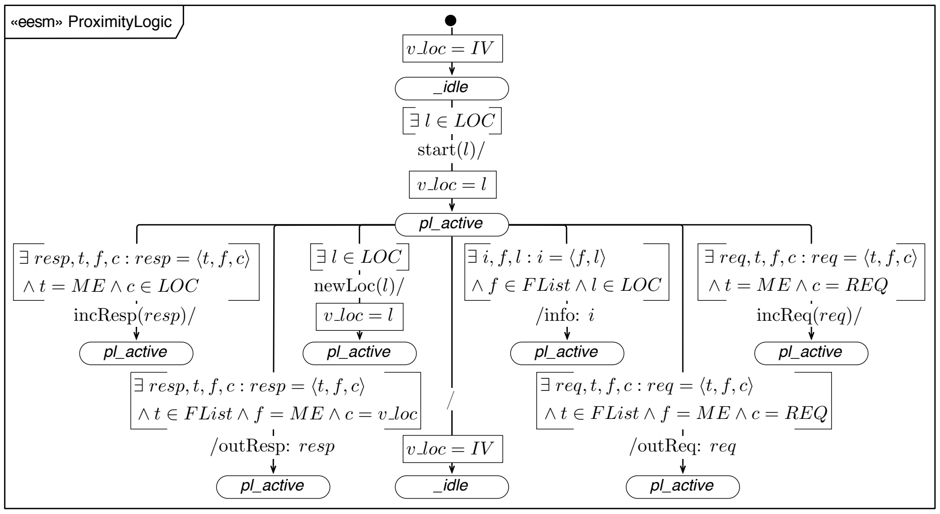
«eesm» ProximityLogic
$v\_loc = IV$
_idle
$\exists\, l \in LOC$
start(l)/
$v\_loc = l$
pl_active
$\exists\, resp, t, f, c : resp = \langle t, f, c\rangle \wedge t = ME \wedge c \in LOC$
incResp($resp$)/
pl_active
$\exists\, l \in LOC$
newLoc(l)/
$v\_loc = l$
pl_active
$\exists\, i, f, l : i = \langle f, l\rangle \wedge f \in FList \wedge l \in LOC$
/info: i
pl_active
$\exists\, req, t, f, c : req = \langle t, f, c\rangle \wedge t = ME \wedge c = REQ$
incReq(req)/
pl_active
$\exists\, resp, t, f, c : resp = \langle t, f, c\rangle \wedge t \in FList \wedge f = ME \wedge c = v\_loc$
/outResp: $resp$
pl_active
/
$v\_loc = IV$
_idle
$\exists\, req, t, f, c : req = \langle t, f, c\rangle \wedge t \in FList \wedge f = ME \wedge c = REQ$
/outReq: req
pl_active

Fig. 4. Security Contract expressed as an EESM

consisting of a logical predicate framed by square brackets as well as operations on the variables which are described using Java-like statements in lined boxes. The EESM in Fig. 4 contains a variable *v_loc* storing the current location of the own device. The initial transition is carried out during system start and leads from the initial state to *_idle*. In its effect part, the variable *v_loc* is set to an initial value expressed by the constant *IV*. The activation of the block takes place when a token containing the current location as a parameter l reaches the pin *start*. The corresponding transition switches the control state from *idle* to *pl_active*. Further, it demands that l is indeed location information which is described in the transition guard and sets the variable *v_loc* to l. The block can only be terminated implicitly by closing down the overall system which, like in the ESMs, is expressed by the transition /. Here, the control state is set back to *idle* again and *v_loc* to the initial value *IV*.

Particularly interesting for the security proof of property $\mathcal{P}$ are the transitions *outResp* and *outReq* since the tokens leaving through them contain the messages to be sent by the communication block. The transition *outResp* uses a parameter *resp* specifying the message to be sent in the token. According to the guard of the transition, *resp* is a triple containing the address from the friends list expressed by the constant *FList* as the recipient address t (*to*). The sender address f (*from*) contains the user's address which is described by the constant *ME* while the content c includes the device's location data which is stored in the variable *v_loc*. The transition *outReq* is similar with the exception that the message content is a request (expressed by the constant *REQ*) asking the recipient for its geographical position. Thus, the EESM specifies that all messages passing pins *outResp* and *outReq* have a friend as a recipient address.

$$\text{MODULE } \mathit{EESMProximityLogic}$$

$\text{EXTENDS } \mathit{Naturals}$
$\text{VARIABLES } \mathit{state}, \mathit{v\_loc}$
$\text{CONSTANTS } \mathit{FList}, \mathit{ME}, \mathit{REQ}, \mathit{LOC}, \mathit{IV}$

$$\mathit{Init} \triangleq \mathit{state} = \text{"\_idle"} \land \mathit{v\_loc} = \mathit{IV}$$

$$\mathit{start}(l) \triangleq \mathit{state} = \text{"\_idle"} \land l \in \mathit{LOC} \land \mathit{state}' = \text{"pl\_active"} \land \mathit{v\_loc}' = l$$

$$\mathit{newLoc}(l) \triangleq \mathit{state} = \text{"pl\_active"} \land l \in \mathit{LOC} \land \mathit{v\_loc}' = l \land \text{UNCHANGED } \mathit{state}$$

$$\mathit{incReq}(\mathit{req}) \triangleq \mathit{state} = \text{"pl\_active"} \land \mathit{req.t} = \mathit{ME} \land \mathit{req.c} = \mathit{REQ}$$
$$\land \text{ UNCHANGED } \langle \mathit{state}, \mathit{v\_loc} \rangle$$

$$\mathit{incResp}(\mathit{resp}) \triangleq \mathit{state} = \text{"pl\_active"} \land \mathit{resp.t} = \mathit{ME} \land \mathit{resp.c} \in \mathit{LOC}$$
$$\land \text{ UNCHANGED } \langle \mathit{state}, \mathit{v\_loc} \rangle$$

$$\mathit{outReq}(\mathit{req}) \triangleq \mathit{state} = \text{"pl\_active"} \land \mathit{req.t} \in \mathit{FList} \land \mathit{req.f} = \mathit{ME} \land \mathit{req.c} = \mathit{REQ}$$
$$\land \text{ UNCHANGED } \langle \mathit{state}, \mathit{v\_loc} \rangle$$

$$\mathit{outResp}(\mathit{resp}) \triangleq \mathit{state} = \text{"pl\_active"} \land \mathit{resp.t} \in \mathit{FList} \land \mathit{resp.f} = \mathit{ME}$$
$$\land\ \mathit{resp.c} = \mathit{v\_loc} \land \text{UNCHANGED } \langle \mathit{state}, \mathit{v\_loc} \rangle$$

$$\mathit{implicit\_termination} \triangleq \mathit{state} = \text{"pl\_active"} \land \mathit{state}' = \text{"\_idle"} \land \mathit{v\_loc}' = \mathit{IV}$$

Fig. 5. Security Contract expressed as a TLA+ specification

EESMs can be automatically transformed into specifications in TLA$^+$ [13], the notation of TLA and the input language of the model checker TLC (see [12]). TLA is a linear-time temporal logic in which state transition systems are specified using variables for the states and actions (i.e., predicates on pairs of states) for the transitions. The TLA$^+$ specification of the EESM of the block *Proximity Logic* is listed in Fig. 5. It uses the variables *state* denoting the current control state of the EESM as well as the additional variable *v_loc*. *Init* is a predicate specifying the beginning state of the block, i.e., *_idle*. The other seven definitions model the transitions of the EESM in form of actions. Here, a simple variable identifier refers to the state before carrying out an action, whereas an identifier marked by a prime symbol ($'$) points to the state after its execution. For example, before triggering the action *start*, the variable *state* is equal to *_idle* while afterwards it carries the value *pl_active*. Further, this action is only enabled if its parameter l is of type LOC and the variable *v_loc* in the next state is l.

5 Compositional Verification

In TLA, the verification, that an application specification *Spec* fulfills a security property $\mathcal{P}$, corresponds to the implication proof *Spec* $\Rightarrow \mathcal{P}$. To carry out this proof, we transform the SPACE model of the application into TLA$^+$ specifications of the activities and EESMs that are coupled with each other in a constraint-oriented way (see [15]), forming the system specification *Spec*. The security property $\mathcal{P}$ is modeled as an abstract system specification or an invariant in TLA$^+$ as well. As discussed above, a system-level activity can contain any number of building blocks referring to other activities which in turn may encapsulate other activities (see Fig. 2 and Fig. 3 as an example). Reflecting that constraint-oriented composition corresponds to conjoining TLA formulas, *Spec* is defined as the conjunction of the TLA$^+$ specifications of all activities modeling the application:

$$Spec \triangleq \mathcal{A}_s \wedge \bigwedge\nolimits_{b \in Blocks} \mathcal{A}_b \tag{1}$$

Here, *Blocks* is the set of all building blocks, while $\mathcal{A}_b$ denotes the TLA$^+$ specification of the activity referenced by block b. With $\mathcal{A}_s$, we refer to the system activity. For our location application in Fig. 2, $\mathcal{A}_s$ corresponds to activity *Location App* while the set *Blocks* contains eight elements, namely, *mui*, *c*, *lu*, *pl*, *h*, *b*, *g*, and *x*. The first four elements refer to the activities *Main UI*, *Communication*, *Location Update*, and *Proximity Logic* respectively (see Fig. 2). The elements h, b and g point to the activities enclosed by the block *pl* (see Fig. 3), while x marks the activity *XMPP Client* which is enclosed by the communication block c.

To prove $Spec \Rightarrow \mathcal{P}$ by compositional verification, we have to conduct two major steps. First, we verify that all activities $\mathcal{A}_b$ (except the one on system level) are consistent with their corresponding EESMs. To clarify this proof, we perceive a system specification as a tree of activities. Here, an activity $\mathcal{A}_b$ is the parent of another activity $\mathcal{A}_c$ if the designator of the building block c referring to $\mathcal{A}_c$ is enclosed in $\mathcal{A}_b$. The system activity $\mathcal{A}_s$ forms the root of this tree, while those activities not containing any building blocks are the leaves.

We prove now for every activity $\mathcal{A}_b$ in the tree except for the root that it fulfills its EESM $\mathcal{E}_b$ whereupon we represent its children activities by their EESMs:

$$\mathcal{A}_b \wedge \bigwedge\nolimits_{c \in Children(b)} \mathcal{E}_c \Rightarrow \mathcal{E}_b \tag{2}$$

Proving equation (2) for all activities except for the system activity is sufficient since one can deduce by induction that all activities fulfill also the equation

$$\mathcal{A}_b \wedge \bigwedge\nolimits_{c \in Descendants(b)} \mathcal{A}_c \Rightarrow \mathcal{E}_b \tag{3}$$

in which *Descendants* refer to all the descendants of an activity in the tree. The starting step of the induction is the verification that equation (3) follows from (2) for all leaves of the tree. This proof is trivial since the leaf activities do not have any descendants at all. In the inductive step, we have to verify that an activity $\mathcal{A}_b$ fulfilling equation (2) also guarantees equation (3) as long as (3) holds also for all of its children. Likewise, this proof is easy since the descendants of the children of $\mathcal{A}_b$ are also its own descendants. Therefore,

$$\forall k \in Children(b) : \mathcal{A}_b \wedge \bigwedge\nolimits_{c \in Descendants(b)} \mathcal{A}_c \Rightarrow \mathcal{E}_k$$

holds and equation (3) can be directly deduced from (2).

In TLA, a verification of equation (2) is achieved by employing a refinement mapping [16], i.e., a mapping between the state spaces of $\mathcal{A}_b$ and $\mathcal{E}_b$ guaranteeing that an initial state of $\mathcal{A}_b$ is mapped to an initial state of $\mathcal{E}_b$, and that a TLA action of $\mathcal{A}_b$ is either mapped to an action in $\mathcal{E}_b$ or to a stuttering step in which the variables in $\mathcal{E}_b$ do not change. The refinement mapping proof can be automated by the model checker TLC, whereat the use of the EESMs of $\mathcal{A}_b$'s children keeps the number of states to check low. We cannot detail the verification process here, but the proofs are similar to the ones presented in [12]. One of the EESM proofs in our location example was $\mathcal{A}_{pl} \wedge \mathcal{E}_g \wedge \mathcal{E}_h \wedge \mathcal{E}_b \Rightarrow \mathcal{E}_{pl}$ stating that the activity *Proximity Logic* in Fig. 3 fulfills its EESM that is depicted in Fig. 4.

In the second major proof-step, we use the EESMs of the children of the root activity $\mathcal{A}_s$ to verify the security property $\mathcal{P}$:

$$\mathcal{A}_s \wedge \bigwedge\nolimits_{c \in Children(s)} \mathcal{E}_c \Rightarrow \mathcal{P} \tag{4}$$

From this equation and the fact that the children of $\mathcal{A}_s$ are blocks in the system specification *Spec*, we can infer $Spec \Rightarrow \mathcal{P}$ since for all the children of $\mathcal{A}_s$ equation (3) holds as well. For the proof of equation (4), we use the model checker TLC which again profits from using the EESMs of the inner blocks instead of their activities such that the number of states to be checked can be reduced in all of our TLC model checker runs.

An excerpt of the TLA$^+$ specification of our location application is depicted in Fig. 6, in particular, $\mathcal{A}_{LocationApp} \wedge \mathcal{E}_{mui} \wedge \mathcal{E}_c \wedge \mathcal{E}_{lu} \wedge \mathcal{E}_{pl} \Rightarrow \mathcal{P}$. Variables and constants used in the specification are declared in the section *Variables and Constants Declaration*. Most of them represent the variables and constants defined in the EESMs, including the ones modeling security related aspects. The section *Using the EESMs of Inner Blocks* contains four instantiation statements used to couple TLA$^+$ specifications into the system description. For instance, in the last statement, module *EESMProximityLogic* (see Fig. 5) is instantiated and denoted as *pl*. Here, the variables *state* and *v_loc* of the instantiated module are respectively substituted by variables *pl_state* and *pl_v_loc* of the application specification. Likewise, all the constants in the EESM are instantiated, albeit implicitly since they are substituted with constants of the same name. By the other three statements, the EESMs of the block instances *mui*, *c*, and *lui* are composed. The instances enable us to refer to EESM transitions by $\langle instance \rangle ! \langle EESM\_transition \rangle$ (e.g., $pl!outResp(resp)$). These references are used to specify events in an enclosing activity as exemplified in Fig. 6 by the section labeled with *System Actions*. The TLA action $pl\_outResp(resp)$ defines that response *resp* emitted by block *Proximity Logic* ($pl!outResp(resp)$) is sent by block *Communication* ($c!send(resp)$) which, among others, stores a message sent to another station in the auxiliary variable[1] *c_v_out*. Moreover, the UNCHANGED statement points out that the blocks *Main UI* and *Location Update* are not involved in the action and do not change their variables.

[1] Auxiliary variables do not influence the behavior of a system but support verification.

MODULE *Location_App*

Variables and Constants Declaration

VARIABLES $pl\_state, c\_state, mui\_state, lu\_state, pl\_v\_loc, c\_v\_out, c\_v\_enOut, c\_v\_enLgn$

CONSTANTS $FList, ME, REQ, LOC, IV, Any, Ciphertext, Login$

Using the EESMs of Inner Blocks

$mui \triangleq$ INSTANCE $EESMMainUI$ WITH $state \leftarrow mui\_state$

$c \triangleq$ INSTANCE $EESMCommunication$ WITH $state \leftarrow c\_state, v\_out \leftarrow c\_v\_out,$
$v\_enOut \leftarrow c\_v\_enOut, v\_enLgn \leftarrow c\_v\_enLgn, Recepient \leftarrow FList \cup Any$

$lu \triangleq$ INSTANCE $EESMLocationUpdate$ WITH $state \leftarrow lu\_state$

$pl \triangleq$ INSTANCE $EESMProximityLogic$ WITH $state \leftarrow pl\_state, v\_loc \leftarrow pl\_v\_loc$

System Actions

$pl\_outResp(resp) \triangleq pl!outResp(resp) \land c!send(resp)$
$\land$ UNCHANGED $\langle mui\_state, lu\_state \rangle$
$\ldots$

TLA$^+$ System Specification

$Init \triangleq pl!Init \land c!Init \land mui!Init \land lu!Init$
$Next \triangleq$
$\lor \exists\, r \in [t : FList \cup Any,\ f : \{ME\} \cup Any,\ c : \{REQ\} \cup LOC \cup Any] : pl\_outResp(r)$
$\lor \ldots$
$vars \triangleq \langle pl\_state, c\_state, mui\_state, lu\_state, pl\_v\_loc, c\_v\_out, c\_v\_enOut, c\_v\_enLgn \rangle$
$Spec \triangleq Init \land \Box[Next]_{\langle vars \rangle}$

$\mathcal{P} \triangleq \Box((c\_v\_out = IV) \lor (c\_v\_out.c \in LOC \Rightarrow c\_v\_out.t \in FList))$

Fig. 6. Excerpt from the TLA+ specification of the Location App

In section *TLA$^+$ System Specification* of Fig. 6, the TLA$^+$ specification modeling block *LocationApp* is written as the so-called canonical formula $Spec \triangleq Init \land \Box[Next]_{\langle vars \rangle}$. It expresses that the initial state of the application fulfills the predicate *Init* and that every state change follows one of the system actions which are disjuncts of the next state relation *Next*. By $[\ldots]_{\langle vars \rangle}$, one models that stuttering steps in which the list of variables *vars* do not change are also allowed.

The security property $\mathcal{P}$, i.e., "*one's locations are only sent to one's friends*" is expressed by the TLA invariant that is listed in the bottom part of Fig. 6. It states that at all times the variable *c_v_out* storing the messages sent to other recipients carries either the initial value *IV* (i.e., no message has been sent yet) or that a sent message containing location information is sent to the address of a friend. We use the TLC model checker to verify both, $\mathcal{P}$ and the security property "*the credentials used to login to an XMPP server are sent via a secure communication channel*". The performance issues of the model checker runs proving these two security properties will be discussed below.

Table 1. Verification effort: compositional approach vs. direct approach

		Number of elements for each set			
		1	**2**	**3**	**4**
compositional approach	states (largest)	984	53 855	823 174	6 568 677
	states (total)	1 204	56 200	837 810	6 630 076
	time (total)	2 sec	12 sec	44 sec	331 sec
direct approach	states	6 248	1 047 503	> 25 M	-
	states (x largest)	6.35 x	19.45 x	> 30 x	-
	states (x total)	5.19 x	18.64 x	> 29 x	-
	time	6 sec	224 sec	> 2 hours	-
	time (x total)	3 x	18.67 x	> 163 x	-

▪ buffer size = 2

6 Model Checking Performance

To evaluate the advantage of employing EESMs for verification of security properties, we compare the result of model checking the example application with two approaches: The first one is the compositional technique described above, i.e., proving formula (4) for the system block *Location App* in Fig. 2 and formula (2) for the eight inner blocks *mui*, *c*, *lu*, *pl*, *h*, *b*, *g*, and *x*. The other one is the direct method in which the TLA$^+$ specifications of all activities of the system are used, i.e., equation (1).

We model checked the compositional and direct approaches on a 2.4 GHz, 8 GB personal computer. The result is presented in Tab. 1. Both versions use the same sets representing various types of data (e.g., *FList* denotes a list of friends). Since TLC works by generating behaviors that satisfy a specification, we needed to declare the elements of those sets. We used the same elements for each set in the specifications of both approaches and decided to use the size of a set as the parameter to compare the verification effort. Further, one restriction, i.e., a maximum buffer size, was required since, otherwise, the specifications related to the reactive buffer would have infinitely many reachable states due to arbitrarily many sequences of messages stored by the buffer. For the compositional approach, we present three types of data in Tab. 1: The values in the first row, obtained from proving formula (2) for block *pl: Proximity Logic*, show the largest number of states created in a single model checker run reflecting the maximum amount of memory needed. In the second row, we simply add the number of states checked in all nine runs. Similarly, the total amount of time to model check all nine blocks is shown in the third row. These values are compared with the respective number of states and verification time of the direct approach.

Observing Tab. 1, we see that the number of states found by the model checker for the direct version is much higher compared to the compositional version. In consequence, also the execution time of the model checker runs grew. For example, using sets consisting of two elements, the state space of the direct version is more than 18 times larger than the composed version's. Further, it also takes about 18 times longer to verify the direct version than the compositional one. Moreover, the state and time differences between the compositional and the direct verification increase with a growing set size. Indeed, the direct approach effectively fails when the set size reaches the value 4 while the compositional verification is still manageable in a few minutes.

Altogether, these results confirm our experience with functional and reliability checks mentioned above that utilizing the SPACE building blocks and their interface descriptions for model checking effectively reduces the state explosion problem. Thus, it helps to make automatic analysis more feasible for real-life systems. In addition, the effort to verify systems that are developed with already proven blocks is further reduced since ensuring the conformity of a block to its interface contract only needs to be done once.

7 Related Work

Various methods have been proposed to support the development of secure systems. UMLsec [17] is a UML profile that is used to incorporate security-related information such as fair exchange and secure communication links in various UML diagrams. SecureUML [18] is a modeling language tailored to integrate Role-Based Access Control policies into application models defined with the UML. Similarly, integration of Mandatory Access Control with UML is proposed in [19]. Approaches based on aspect-orientation, modeling security mechanisms as aspects which are automatically weaved in at joint points of a specification, have also been proposed (see, e.g., [20–23]). The CORAS approach [24] defines a modeling language to support security risk analysis for systems designed with UML. Its UML diagrams are mainly devoted to model the various steps of a security analysis while the purpose of ours is to express system behavior.

Since systems are usually composed from numerous parts, specifying security aspects in the components and verifying system-wide security properties has been the focus of a number of approaches. To support the development of security-critical applications, Moebius et al. proposed SecureMDD, a model-driven technique that includes verification of application-specific properties [25]. For large systems, they take an incremental approach for which some functionality is added in every step such that security proof needs to be repeated in every iteration [26]. In contrast, in our approach functional behaviors are composed in a constraint-oriented way. Deng et al. proposed a method to model security system architectures and verify whether required security constraints are assured by the composition of the components [27]. However, unlike our work, it employs a top-down approach. Security policies are specified as application-wide constraint patterns which are further decomposed onto the individual components of the system. In [28], Khan et al. present a framework to construct compositional security contracts based on the required and ensured security properties exposed by the atomic components. Although the contracts help engineers to characterize the security aspects of a composed system, the framework does not include validation whether the contracts fulfill the security requirements of the system. Other work that aims to address security issues in software systems consisting of simpler components can be found in [29–31].

8 Conclusion

In this paper, we introduced *security contracts* that encapsulate both functional and security aspects of a building block. Due to the formal semantics of the contracts, model checking can be employed to ensure that the contracts and their corresponding blocks are consistent. Furthermore, we showed that such contracts enable compositional verification of application-level security properties which significantly reduces the number of states to be checked and, consequently, also the verification time. On the whole, these behavior and security interface descriptions facilitate model-based development of secure systems: Security mechanisms enclosed in building blocks [4, 6] are easily integrated with blocks modeling other functionalities, and both kinds of blocks can be (re-)used in various application designs. The security contracts specify security properties fulfilled by the blocks.

Currently, we are investigating in separating the development of the security contracts from using them in proofs of system-wide security properties. This supports the nature of SPACE-based system engineering that building blocks are often developed independently from the applications and stored in libraries. To achieve that, we need to find a way that relevant security aspects of a block can be anticipated, modeled in a security contract, and proven without knowing the applications using the block. As a solution, we consider to employ application domain-oriented information security ontologies stating relevant assets, vulnerabilities and threats (see also [32]). For example, an ontology for Android may contain passwords and location information as typical assets of Android devices and leaking them as a typical confidentiality threat. Based on that, a security expert might annotate the building blocks of the Android library by security contracts addressing the elements of this ontology. Moreover, an ontology may contain a list of system-wide security properties to be fulfilled by systems of that domain (e.g., an Android device may never send a password to anybody).

Like the security contracts which are represented by EESMs, one can also define the system-wide security properties in a more comprehensible syntax than plain TLA$^+$ effectively reducing the required expertise in formal methods. This complements our experience with functional system development in which engineers analyze their models by just pushing a button which leads to a message containing that everything is correct or a list of errors in an easily understandable format. Further, the trace towards a state violating a property is animated directly on the SPACE models [3] such that the engineer does not need to understand the formalism of the model checker running in the background at all. We want to achieve a similar procedure for the verification of security properties. By describing the system security properties in an easily understandable way, at least basic security protection can be done directly by the domain engineers without involving the security experts in excess of the creation of the building block security contracts. This will ease the development of more secure software.

References

1. Iyer, R.K., Chen, S., Xu, J., Kalbarczyk, Z.: Security Vulnerabilities - from Data Analysis to Protection Mechanisms. In: Proceedings of the Ninth IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS) 2003. (2003) 331–338
2. 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)
3. 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
4. Gunawan, L.A., Herrmann, P., Kraemer, F.A.: Towards the Integration of Security Aspects into System Development using Collaboration-Oriented Models. In: Proceedings of the International Conference on Security Technology (SecTech) 2009, Jeju Island, Korea, December 10-12, 2009. Volume 58 of CCIS, Springer (2009) 72 – 85
5. 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
6. Gunawan, L.A., Kraemer, F.A., Herrmann, P.: A Tool-Supported Method for the Design and Implementation of Secure Distributed Applications. In: Engineering Secure Software and Systems. Volume 6542 of LNCS, Springer (2011) 142–155
7. McMillan, K.L.: Symbolic Model Checking: an Approach to the State Explosion Problem. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA (1992)
8. Davis, A.M.: Software Requirements: Objects, Functions and States. 2nd edn. Prentice-Hall, Inc., Upper Saddle River, NJ (1993)
9. Slåtten, V., Kraemer, F.A., Herrmann, P.: Towards Automatic Generation of Formal Specifications to Validate and Verify Reliable Distributed Systems: A Method Exemplified by an Industrial Case Study. In: Proceedings of the 10th ACM International Conference on Generative Programming and Component Engineering, New York, NY, USA, ACM (2011) 147–156
10. Object Management Group: Unified Modeling Language: Superstructure, version 2.3. (May 2010) formal/2010-05-05.
11. 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), Denver, Colorado, USA, October 4-9, 2009. Volume 5795 of LNCS, Springer (2009)
12. Slåtten, V., Herrmann, P.: Contracts for Multi-instance UML Activities. In: Proceedings of the joint 13th IFIP WG 6.1 and 30th IFIP WG 6.1 international conference on Formal techniques for distributed systems. FMOODS'11/FORTE'11, Springer (2011) 304–318
13. Lamport, L.: Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley Professional (2002)
14. Yu, Y., Manolios, P., Lamport, L.: Model checking TLA+ specifications. In: Correct Hardware Design and Verification Methods, Springer (1999) 54–66

15. Herrmann, P., Krumm, H.: A Framework for Modeling Transfer Protocols. Computer Networks **34**(2) (2000) 317–337
16. Abadi, M., Lamport, L.: The Existence of Refinement Mappings. Theoretical Computer Science **82**(2) (May 1991) 253–284
17. Jürjens, J.: Secure System Development with UML. Springer-Verlag (2005)
18. 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
19. Doan, T., Demurjian, S., Ting, T.C., Ketterl, A.: MAC and UML for Secure Software Design. In: Proceedings of the 2004 ACM workshop on Formal methods in security engineering. FMSE '04, New York, NY, USA, ACM (2004) 75–85
20. 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.
21. Mouheb, D., Talhi, C., Nouh, M., Lima, V., Debbabi, M., Wang, L., Pourzandi, M.: Aspect-Oriented Modeling for Representing and Integrating Security Concerns in UML. In: Software Engineering Research, Management and Applications 2010. Volume 296 of Studies in Computational Intelligence. Springer (2010) 197–213
22. Jürjens, J., Houmb, S.H.: Dynamic Secure Aspect Modeling with UML: From Models to Code. In: International Conference on Model Driven Engineering Languages and Systems, MoDELS 2005, Montego Bay, Jamaica, October 2-7, 2005. Volume 3713 of LNCS, Springer (2005) 142–155
23. Jézéquel, J.M.: Model Driven Design and Aspect Weaving. Software and System Modeling **7**(2) (February 2008) 209–218
24. Lund, M.S., Solhaug, B., Stølen, K.: Model-Driven Risk Analysis - The CORAS Approach. Springer (2011)
25. Moebius, N., Stenzel, K., Reif, W.: Formal Verification of Application-Specific Security Properties in a Model-Driven Approach. In: Engineering Secure Software and Systems. Volume 5965 of LNCS, Springer (2010) 166–181
26. Moebius, N., Stenzel, K., Borek, M., Reif, W.: Incremental Development of Large, Secure Smart Card Applications. In: Proceedings of the 1st Model-Driven Security Workshop (MDSec) 2012. (2012) To appear.
27. Yi, D., Wang, J., Tsai, J.J., Beznosov, K.: An Approach for Modeling and Analysis of Security System Architectures. IEEE Transactions on Knowledge and Data Engineering **15**(5) (sept.-oct. 2003) 1099 – 1119
28. Khan, K., Han, J., Zheng, Y.: A Framework for an Active Interface to Characterise Compositional Security Contracts of Software Components. In: Proceedings of the 2001 Australian Software Engineering Conference. (2001) 117–126
29. Herrmann, P.: Information Flow Analysis of Component-Structured Applications. In: Proceedings of the 17th Annual Computer Security Applications Conference (ACSAC), New Orleans, ACM SIGSAC, IEEE Computer Society Press (2001) 45–54
30. Mantel, H.: On the Composition of Secure Systems. In: Proceedings of the IEEE Symposium on Security and Privacy, IEEE Computer Society (May 2002) 88–101
31. Bartoletti, M., Degano, P., Ferrari, G.: Security Issues in Service Composition. In: Formal Methods for Open Object-Based Distributed Systems. Volume 4037 of LNCS, Springer (2006)
32. Vasilevskaya, M., Gunawan, L.A., Nadjm-Tehrani, S., Herrmann, P.: Security Asset Elicitation for Collaborative Models. In: Proceedings of the 1st Model-Driven Security Workshop (MDSec 2012). (2012) To appear.