Behavioral Singletons to Consistently Handle Global States of Security Patterns

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. Secure systems are usually complex since stateful security mechanisms like authentication and authorization have to be integrated into the functional behavior at various places. The security operations are, in general, interdependent such that events at one place may influence the behavior at other places. Thus, the composed specification of a system is neither easy to understand nor to analyze, and a faulty integration of the security mechanisms is often overseen. In this paper, we introduce the concept of singletons into our model-based engineering technique SPACE which facilitates a straightforward integration of security aspects. The behavior of a security protocol is encapsulated in a building block using a two-view interface contract. One view of the contract is quite simple and suffices for the correct integration of the block into a system specification. The other view is more complex but has to be considered only by the block designers to verify that the behavioral model in the block fulfills its interface contract. We exemplify the singletons by means of an authorization mechanism and discuss how to prove that the two views of its interface contract are consistent.

1 Introduction

Developing secure applications is a difficult and error prone task. One of the reasons for this is that many security mechanisms are hard to understand and realizing them requires in-depth knowledge [1]. Moreover, although developers with sufficient security expertise manage to provide implementations of those mechanisms, their integration in application designs poses yet another challenge due to the crosscutting nature of the security aspects [2–4]. Security functions are usually accessed by several parts of an application, such that a change in the security functionality has an impact on the rest of the application. This holds particularly for the integration of stateful security mechanisms into a system design which increases the complexity of the model dependencies further, since events at one place of a system may influence the behavior at other places.

Reusing proven, well-tested solutions is one way to address the problem of lack of expertise in some specific domains. It has also been recognized as an important key to build high-quality software systems in an efficient and cost-effective manner [5]. Solutions are more easily reusable when they are modular,

highly cohesive, and loosely coupled with external components [6]. Software libraries in the form of APIs are available for reuse also for security protocols. Nevertheless, the need to integrate these APIs into various places of an application aggravates understanding the composed system. Therefore, we require an approach that not only promotes reuse but also modularizes crosscutting concerns like security aspects and facilitates seamless integration with applications.

Previously, we developed the model-based approach SPACE [7] in which a system specification is modularized with self-contained building blocks describing the behavior of several functionalities in UML activities. In turn, these blocks can contain any number of inner activities referenced by call behavior actions. On average, more than 70% of an application specification can be composed from reusable blocks taken from various domain-specific libraries [8]. To integrate a block correctly into its environment, it is augmented with a contract in the form of a UML state machine which describes the behavior visible at the interface between the block and its environment. Moreover, the formal semantics of the activities [9] makes it possible to verify block compositions using the model checker included in Arctis [10], the tool-set for SPACE.

To deal with stateful, crosscutting concerns while also minimizing dependencies, we introduce the concept of behavior singletons which is inspired by the corresponding design pattern for programming described by Gamma et al. [11]. In particular, we discuss a new kind of block that may be represented by various call behavior actions in different parts of a system model which, however, all refer to a single behavior instance. Thus, a security aspect can be specified in this so-called *singleton block* independently from the application-specific functionality of a system. In contrast to Gamma et al.'s singleton, we also consider the behavior at the interface of singleton blocks by supplementing it with a behavior contract. The interface behavior is specified in such a way that inconsistencies due to the usage of the singleton at many parts of the application are ruled out if both the singleton and its environment obey the interface. Singleton blocks containing relevant security features can be easily reused since the automatic model checker included in Arctis is sufficient to prove that a singleton block instance is correctly integrated into its environment. The proof that a security mechanism guarantees the interface contracts of all singleton block instances, however, is a little more complex, but usually manageable as pointed out in Sect. 4.

2 A Social Application Example Modeled in SPACE

An Arctis building block may contain any number of inner activities referenced by call behavior actions that are connected through some "glue logic". Therefore, a system specification consists of building blocks forming a tree structure with a system activity (i.e., the outermost block) as the root and simple activities (i.e., blocks that do not contain any other call behavior action) as the leaves. Figure 1 shows the system specification of the example. It is an Android application which enables users to obtain their friends' geographical locations, manage their friend lists, and record their own current locations. This system consists of ten call

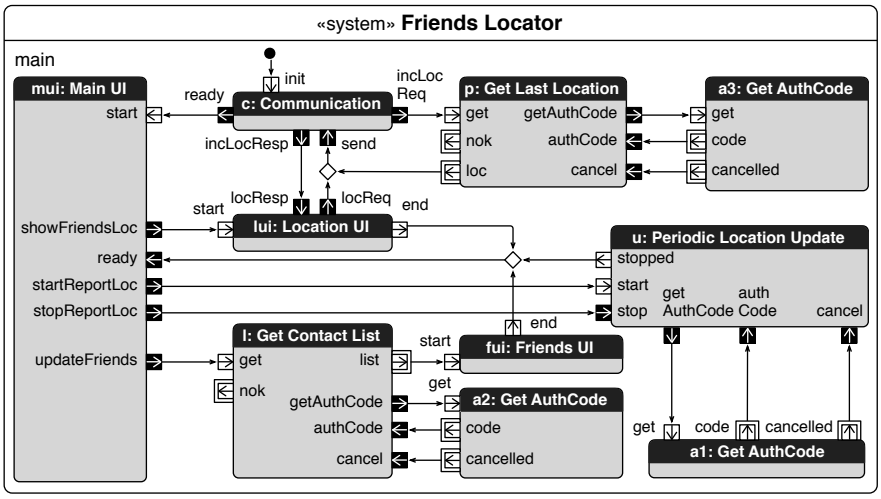

Fig. 1. Friends Locator Example

behavior actions which are further decomposed into other call behavior actions as depicted partially in Fig. 2.

The Petri net-like semantics of the activities models behavior as control respective data flows of tokens between the nodes of an activity via its edges. Block *c: Communication* is initialized when the application starts as delineated by the arrow from the initial node (•) in Fig. 1. This call behavior action refers to an activity that handles messages with the application's peers. Subsequently, the main user interface (UI), whose behavior is encapsulated in block *mui: Main UI*, is started. Thereafter, the user may choose to use any of the following features:[1] toggling the location update service (f_1), managing a list of friends (f_2), or retrieving friends' locations (f_3). Feature f_1 is managed by an event emitted from the UI block through the pins *startReportLoc* or *stopReportLoc*. It is supported by block *u: Periodic Location Update* which, when started, periodically reads the location of the device it is running on. To achieve this, the block utilizes corresponding Android APIs and stores the location on a remote server until the block is stopped. The other features which include retrieval of the user's contact list (f_2) and location data (f_3) from the server are modeled in a similar way with the remaining building blocks.

The example application employs Google services that can store information like locations, photos or contacts, while also providing RESTful APIs for third-party applications to access them. Here we utilize two APIs, namely Latitude [12] to handle the location data and Contacts [13] to retrieve the contact list. Since all requests to these services must be authorized, the application uses

[1] By *feature*, we refer to a part of an application that performs a certain function.

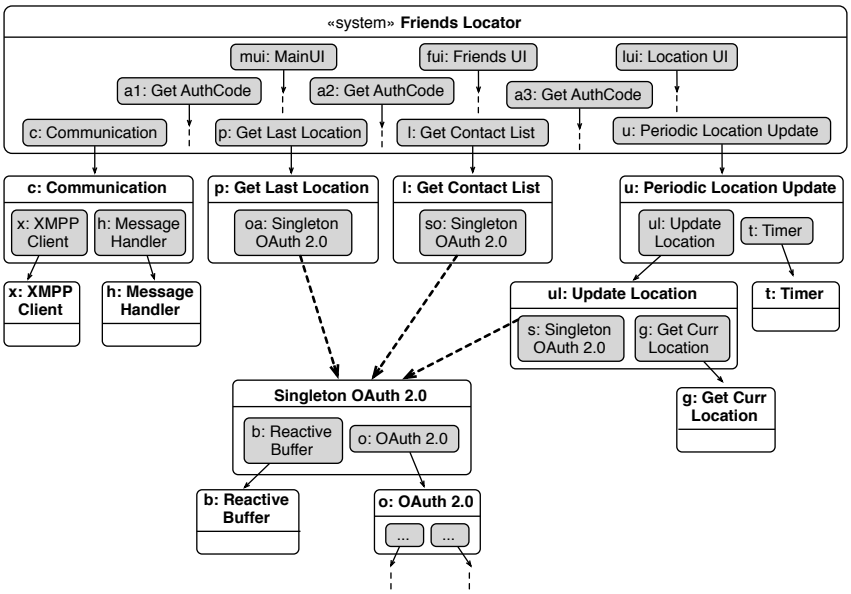

Fig. 2. Hierarchy of Blocks from The Friends Locator Example

the authorization protocol OAuth 2.0 [14]. This protocol is developed to enable a third-party application to obtain a limited access to protected resources hosted in an HTTP service with an access token instead of using the resource owner's (i.e., the user's) credentials directly. Before obtaining an access token, the application needs authorization from the user. This is achieved by redirecting the user to an authorization server via a web browser. After a positive authentication and authorization, the user is redirected back to the application with an authorization grant. Thereafter, the grant is used by the application to obtain an access token and a refresh token from the authorization server. By means of the access token, the application finally can access restricted resources for a limited duration. A refresh token is used to obtain a new access token when the current one expires.

The activity encapsulating the behavior of the OAuth protocol is not shown in this paper due to lack of space, but it is referenced by the block *o: OAuth 2.0*, that is depicted in Fig. 3. In the following, we sketch the external behavior of this block that handles one request to a protected resource at a time. The first request initiates a user's authentication and authorization process through pin *getAuthGrant*. This step is not managed by the block because it needs to invoke a web client which is treated differently on different platforms.[2] If the user grants

[2] In our example in Fig. 1, this is handled by three call behavior actions of type *Get AuthCode*. The referred activity utilizes *WebView*, a webpage viewer for Android.

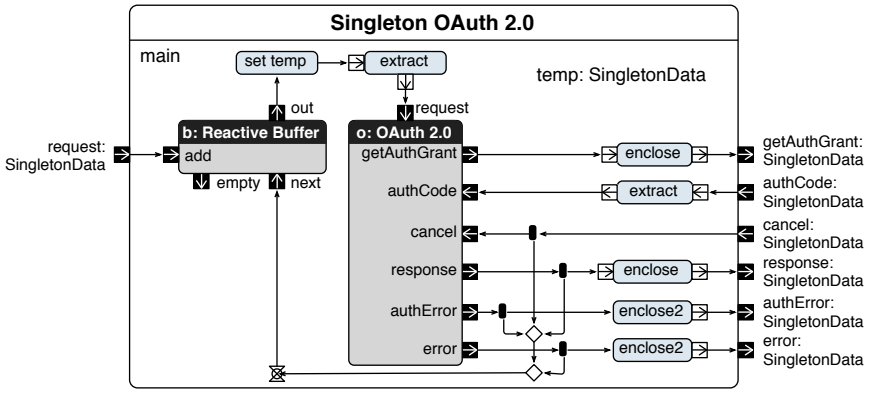

Fig. 3. Singleton OAuth 2.0 Activity

access rights, authorization code is received by block *OAuth 2.0* via pin *authCode* and later the result of executing the first request is forwarded via pin *response*. Thereafter, the following requests can be run without the authorization step. The incoming pin *cancel* of the OAuth block is used to indicate that the user denies the application's request to access his or her resources. The other two outgoing pins are used to report two types of errors: Pin *error* indicates an error related to the corresponding request, for example, if the requested resource is not found. Pin *authError* shows an authorization error, for instance, if the authorization code is incorrect or the user has revoked the access grant.

3 Singleton Building Blocks

In general, the behavior of authentication and authorization protocols is stateful. For instance, in OAuth the user verification and user's consent for access rights are executed *once*, i.e., upon receiving the first request, while subsequent requests directly use the stored token. Moreover, a new token may need to be obtained when the curent one expires. Such stateful behavior is also common in other protocols, e.g., TLS [15], the SASL framework [16], and single sign-on solutions such as SAML [17].

As mentioned, authorized access to a restricted resource is used at several features of a system. For instance, in our example we use authorization in all the features f_1 (current location update), f_2 (management of user's friend list), and f_3 (last location retrieval) which are not executed in *any strict sequence* since they can be independently triggered from each other. Therefore, a solution in which authentication is always done by *one* certain feature is not possible. Furthermore, in a large project, features are likely developed by different engineers, so that global constraints like demanding a "first requestor" would require undesirable additional coordination making the engineering process more complex.

Consequently, we have to consider a certain stateful behavior that uses shared data (e.g., the access token) and a shared state (e.g., when authorization is ongoing) at various places of an application. In object-oriented programming, the access of shared data from various points can be handled using a *singleton pattern* [11]. This pattern ensures that a class has one shared instance only, and provides a single, global access point to this instance. We adapt this concept in the form of singleton blocks. Instead of referring to *different* instances of a behavior description, using a singleton pattern means that different call behavior actions of the same type refer to a *single* behavior instance. Hence, in Fig. 2 the three call behaviors *oa*, *so*, and *s* point to the same block, namely, *Singleton OAuth 2.0*.

The singleton pattern, however, does not solve the problem mentioned in the introduction that events at one feature may influence the behavior of the others which makes the understanding of the overall system behavior more difficult and is a common source of errors. Therefore, we provide the singleton blocks with interface behavior contracts (Sect. 3.2) that guarantee the absence of these inconsistencies.

3.1 Interactions with a Singleton

In SPACE and Arctis, a system consisting of various building blocks is implemented by an automatic two-step process: First, a component-based description modeled by state machines is derived from the building blocks using a special model transformation [18]. Second, code is generated from the state machines [19]. In the first step, a singleton is treated differently since it is implemented as a separate state machine. Thus, on the code level it communicates with the state machines of its environment asynchronously using message passing. Sending *towards* a singleton is easy: When a client feature[3] wants to send a message towards a singleton, it just uses the address of the state machine realizing the singleton and passes a message to it.

Sending *from* a singleton towards the client feature is more complicated since there may be more than one feature, and the singleton needs to determine the actual receiver. In some cases, the singleton only has to process a simple request that can be answered immediately such that no waiting is involved, and a request can be processed completely without being interrupted by other requests. In this case, the singleton simply has to send the response to the originating client feature of the request. However, this is not the general case. In most situations, the request needs to be buffered since other stateful behavior has to be executed to compute a response. Sometimes, the singleton may also require to store all known client features and dispatch certain events only to *some* of them. For this reason, the singleton must be able to handle addresses of the client features explicitly using variables. To realize this technically, we require that all incoming and outgoing parameters are of the special data type *SingletonData*. An object of this type contains the address of the originator that is used to determine to

[3] With the term *client feature* we refer to a feature that interacts with a singleton.

which client feature a signal has to be passed. Other data can be carried by this kind of object as well since it is a wrapper for object flows. Moreover, as several call behavior actions utilizing the same singleton may be synthesized into the same target state machine, we also need to keep track of which call behavior instance sent a certain request. This is also reflected by the address component of *SingletonData*.

The state machine implementing the singleton is created by the runtime support system when one of the other state machines realizing the system sends its first signal towards the singleton. For the client feature calling the singleton, this is transparent since sending the signal is buffered regardless. The singleton state machine is stopped once the main component terminates.

3.2 Behavioral Contract of a Singleton

As discussed in the introduction, an ordinary building block is protected by one behavioral contract specifying in which sequence its parameters may be used. Likewise, the interaction between a singleton and its environment is restricted which, however, is expressed by means of two different contracts:

Single-session: Here, the interaction between the singleton and a *single* client feature is described.

Multi-session: This contract models the overall interaction between the singleton and *all* client features that are linked to it.

The single-session contract is usually relatively simple and therefore suited to guide an integration of the singleton to a particular client feature. In contrast, the multi-session contract is more complex, since it handles several client features. Fortunately, however, this contract has to be considered only by the author of the singleton, who has to deal with the challenge to coordinate multiple invocations of a single stateful behavior anyhow.

A single-session contract is expressed in the form of a so-called External State Machine (ESM, [8]). This is a special UML state machine depicting the possible ordering of externally visible events on the activity parameter nodes. It specifies properties to be kept by both the activity itself and its environment; and thus shows how to compose the activity correctly with others. The semantics of activities is defined in [9], and the automatic verification that block compositions are correct is supported by the model checker included in Arctis [10]. The ESM depicting the single-session contract of our singleton OAuth example is presented in Fig. 4. In the begining, the initial transition to state *idle* is executed. Thereafter, only the transition labeled with *request/* to state *active* can be taken. It indicates that the singleton receives an incoming flow via parameter node *request* (see Fig 3). Afterwards, the singleton can output either one of the events *response*, *error*, *authError*, or *getAuthGrant*. The first three events trigger a transition to state *idle* modeling that the singleton is ready to receive another request. *getAuthGrant* triggers a transition to state *authorizing* which requires the environment to provide an incoming flow through node *cancel*

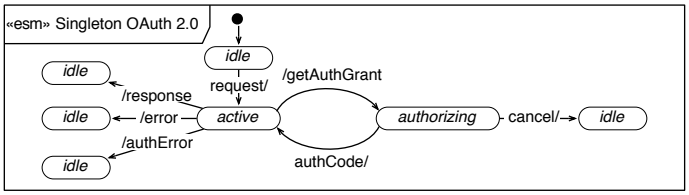

Fig. 4. Single-session Contract of The Singleton OAuth 2.0

or *authCode*. Transition *cancel/* makes the singleton waiting for a new request, while transition *authCode/* takes the singleton back to state *active*.

A multi-session contract is specified with an Extended ESM or EESM [20]. This extends the ESM with variables, transition guards, operations, arithmetic and predicate logic, such that event constraints related to the state of multiple invocations can be specified. The EESM of activity *Singleton OAuth 2.0* is shown in Fig. 5. We assume that the singleton is linked with n client features. The identifiers of those features that have sent a request but not yet received the corresponding result (i.e., the active features) are stored in the set variable *act* which, in consequence, uses the powerset $\mathcal{P}(1..n)$ as its data type. The variable *authno* of type $1..n$ indicates at which client feature an authorization step is taken place.

The initial transition sets *act* to the empty set.[4] When a request is received at the client feature i, the subsequent transition is taken. Here, the transition guard $i \notin act$ must be fulfilled to comply with the restriction that each feature can only send one request at a time. The operation $act = act \cup \{i\}$ specifies that feature i is an element of *act* after executing the transition. The operation $authno = i$ states that i is the feature via which the authorization procedure will be handled. The invocation of the authorization is indicated by the transition from state *unauthorized* to *authorizing*. When authorization code is received via pin *authCode*, the singleton enters state *authorized*. If the authorization is canceled, the singleton enters state *idle* provided that *authno* is the only feature that is currently active, or state *unauthorized* when there is at least one other active feature i. In the latter case, a new authorization starts at feature i. In the event of successful authorization, a request from one feature is answered by a response, an error message or an authorization error message as specified by the four corresponding transitions from state *authorized*. Since any inactive feature may send one request at any time, we use a self transition labeled with *request(i)* in states *unauthorized*, *authorizing*, and *authorized* of the EESM. Of course, the two contracts have to be consistent, and we discuss in the next section how that can be guaranteed.

[4] In the programming-like syntax of the EESMs an assignment is described by = while the equal operator is noted as ==.

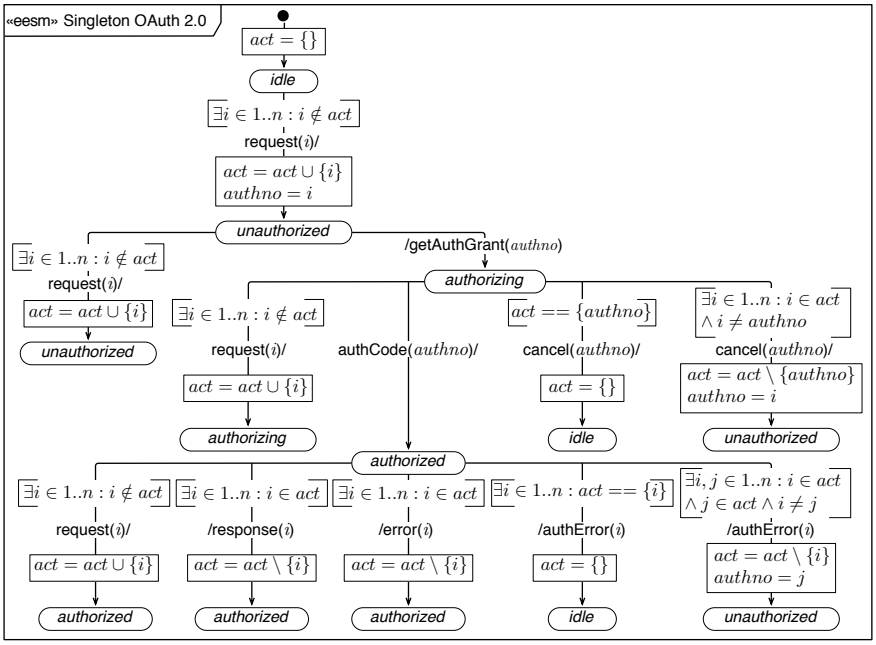

Fig. 5. Multi-session Contract of The Singleton OAuth 2.0

4 Consistency of Singleton Contracts

To guarantee consistency of the two singleton contracts, we have to prove formally that the multi-session contract is a correct refinement of the single-session one. As an example, we sketch in the following parts of the proof that the *Singleton OAuth 2.0* EESM in Fig. 5 fulfills the ESM in Fig. 4 for each client feature in a specification. Intuitively, it shows that the first transition of the EESM is mapped to the initial transitions of the ESMs for all client features. Each of the other EESM transitions corresponds to one ESM transition of a client feature while all other features do not execute any transition.

We mark the EESM as Φ while $\Psi[k]$ refers to the ESM for the client feature k. In [21], Abadi and Lamport verify that so-called refinement mappings (RM) can be used for refinement proofs, i.e., that $\forall k \in 1..n : \Phi \Rightarrow \Psi[k]$ holds. An RM is a function from Φ to $\Psi[k]$ that fulfills the following two properties:[5]

1. An initial state of Φ is mapped to an initial state of $\Psi[k]$.
2. A transition of Φ is mapped to a transition of $\Psi[k]$ or to a stuttering step in which the variables of $\Psi[k]$ do not change. In the former case, the labels of the transitions have to be identical.

[5] In [21], two other properties are mentioned that, yet, are not relevant in our case.

For the verification that our EESM Φ in Fig. 5 implements the ESM Ψ of every client feature k depicted in Fig. 4, we use the following mapping:

$$
\begin{array}{lll}
\Psi[k].state \triangleq & \text{IF } \Phi.state = initial & \\
& \text{THEN } initial & \\
& \text{ELSE IF } k \notin \Phi.act & \\
& \quad \text{THEN } idle & \\
& \quad \text{ELSE IF } \Phi.state = authorizing \wedge \Phi.authno = k & \\
& \quad\quad \text{THEN } authorizing & \\
& \quad\quad \text{ELSE } active &
\end{array}
$$

It reflects that the initial state of the EESM is mapped to the one of the ESM. The ESM of client feature k is in the state *idle* if in the EESM k is not an element of the variable *act*. If Φ is in state *authorizing* and k is the value of variable *authno*, then the authorization process at feature k was invoked and, in consequence, its ESM is in state *authorizing*. In all other cases, the ESM is in state *active*.

We have to verify that the mapping implements the two properties listed above. The first property is trivially fulfilled by the first IF-statement of the mapping. To prove the second property, we have to verify for every transition of the EESM that it is mapped to a transition or a stuttering step of the ESM. As a full proof sketch would exceed the space limit, we exemplify that by verifying the EESM transition *request*(i) from state *idle* to state *unauthorized*. Here, we distinguish the cases $i = k$ and $i \neq k$. In the former case, k is not an element of *act* due to the transition guard and, according to the mapping, the ESM is in the state *idle* before the execution. Thereafter, k is an element of *act*. Since the EESM reaches the state *unauthorized*, the ESM will be in state *active* afterwards. In consequence, the EESM transition is mapped to the ESM transition from *idle* to *active* which, moreover, uses the same label *request*.

The case $i \neq k$ is also straightforward. If k is in the variable *act* prior to the execution of the transition, the ESM is in state *active* while it is otherwise in *idle*. Since we assume $i \neq k$, k is neither added to nor removed from *act* by executing the transition such that the ESM does not change its state. Thus, in this case the EESM transition is mapped to a stuttering step.

While the proof steps are similarly simple for most of the other EESM transitions, in three of them we need additional invariants about the EESM which have to hold in the initial state and are not falsified by any transition. Due to the simple structure of the EESM with only two variables, however, it was very easy to find suitable invariants.

To assure that the singleton activity in Fig. 3 fulfills the EESM and, as proven above, also the ESMs, we have to carry out another refinement proof. Due to the space limit, this verification cannot be pointed out here but similar proofs are discussed in [20].

5 Using and Designing Singleton Building Blocks

Integrators who create applications that *use* a singleton do not have to be experts in a specific domain like, for instance, security. Moreover, when developing a client feature, they do not need to take into account whether other features in a system use the same singleton. Only the single-session contract attached to the singleton block must be respected. This includes alternative behaviors as exemplified by the OAuth example for which the result of a request can be "delayed" by an authorization step or given out immediately since the authorization has been executed in another client feature. The ESM in Fig. 4 shows these alternatives as separate transitions starting from state *active*. A client feature must be ready to handle all alternatives since failing to fulfill one may result in a deadlock. In summary, the client feature must obey the ESM contract which can be guaranteed without having to coordinate certain global constraints. In addition to the contract, the integrators have to ensure that every flow *into* a singleton block carries an object of type *SingletonData* that is provided with data if required. Since the flows *from* the block are also of type *SingletonData*, data that is sent from the singleton has to be extracted from the corresponding object in the received token.

A client feature i may notice the existence of other client features (e.g., by not needing to authorize a user). However, since the interferences by these client features do not change the state of the single-session contract of i, the already proven consistency between the two contract views effectively rules out that the other features may harm i. Thus, the singletons fulfill also the demand stated in Sect. 3 that different developers should be able to build in blocks without having to coordinate global constraints with other engineers.

The "price" to offer singletons enabling independent integration, however, is that their *development* is more complex. This mainly results from the fact that due to the single instance and global access principles, multiple incoming flows can occur anytime, in any order, and from many different features. These flows may access a shared resource that needs to be kept consistent. Therefore, the authors are often required to design a singleton which maintains mutual exclusion among those events. One way to make multiple incoming flows mutually exclusive is to buffer them and handle them one by one according to, for example, their arrival sequence. Applying this strategy to the OAuth 2.0 protocol results in the *Singleton OAuth 2.0* activity depicted in Fig. 3. Block *b: Reactive Buffer*, which is taken from one of our libraries, is used to buffer requests. If a request is received when the buffer is empty, the request is given out immediately, otherwise it is buffered. The pin *next* is used to get a subsequent request, if available, after the inner block *o: OAuth 2.0* completes the execution of the current request.

In order to send messages to correct client features, the singleton must handle their addresses explicitly in the singleton block design. The singleton author, therefore, needs to ensure that every output of a singleton block contains such an address. This address can be obtained from a *SingletonData* object which is carried by every incoming flow and set automatically by the Arctis runtime support system. If the singleton needs to execute other stateful behavior, the

address of an incoming flow can be stored temporarily in a variable, e.g., *temp* in the singleton OAuth example.

Furthermore, the author of a singleton block needs to create the single-session and multi-session contracts and to verify that they are consistent as sketched in Sect. 4. Since a block is created once and, in general, reused many times (see [8]), however, the higher cost of constructing such a block will be rewarded. For instance, the singleton block OAuth 2.0 is sufficiently versatile to be applied in merely all systems using this authorization protocol [14].

6 Related Work

To the best of our knowledge, this is the first article on singleton patterns that also include behavior descriptions with interface contracts and are above programming language level. In [11], Gamma et al. introduce *singletons* as a creational pattern, but do not handle elaborate behavior.

Related to the crosscutting nature of security concerns, many approaches that use aspect-oriented concepts, such as AOP, AOM, AOD, and AOSD, have been proposed (see, e.g., [2–4, 22–24]). Security mechanisms are modeled as aspects which are automatically weaved in at joint points of a primary specification based on rules defined as pointcuts. This approach, like ours, modularize security aspects so that they can be analyzed in isolation. However, our singleton-based integration of security aspects into a system specification is facilitated by explicit behavior contracts. In the aspect-oriented approaches, the binding between security and primary models is tightly coupled [25], hidden and dispersed in the joint points, pointcuts, and weaving algorithm.

With regards to model-based secure system development, various approaches have been published [26–28]. UMLsec [26] is a UML profile that is used to incorporate security-related information such as fair exchange and secure communication links in various UML diagrams and hence facilitating the development of security-critical systems. SecureUML [27] is a modeling language designed to integrate information relevant to Role-Based Access Control policies into application models defined with the UML. Similarly, integration of Mandatory Access Control with UML is proposed in [28]. However, in these approaches, security aspects are tightly coupled with the functional models and therefore designing a secure application requires in-depth security knowledge or a close cooperation with security experts. With our approach, domain-specific specialists can work separately with their own building blocks and due to the behavioral contracts of blocks their models can be integrated seamlessly.

7 Conclusion

We have applied the singleton pattern known from object-oriented programming to a component-driven (resp. collaborative) approach using model-based engineering. The novelty lies in the consideration of the behavioral aspect by means of contracts, i.e., in which sequence data may be sent between the singleton

and the feature(s) using it. With these contracts, we can determine that the singleton is used in a correct way. Further, the need to support single-session contracts compels the author to guarantee that the client features are treated independently from each other by the singleton. The approach is theoretically sound (see Sect. 4) and also appealing from a practical viewpoint. The presented example application is a prototype of a realistic application and was fully implemented with our tool-support.

With respect to the expertise required to use the approach, there is an inherent advantage: *Using* the singleton blocks is rather simple, since here only the simplified version of the contract has to be considered. *Creating* a singleton is more subtle, since one has to take the handling of multiple invocations of a stateful behavior from client features into account. Nevertheless, this is part of the intricacies of, e.g., a security protocol that have to be understood by the author of a singleton anyhow. We are also investigating the applicability of behavioral singletons for other problem domains such as communication and dependability.

In the future, we plan to extend the behavioral contracts of building blocks to include security properties, e.g., confidentiality, integrity, and authenticity, in such a way that the security evaluation of an application specification using those blocks is largely based on the security properties of each block. Our goal is to support modular reasoning of a system not only for functional aspects but also for security issues.

References

1. Mouratidis, H., Giorgini, P.: Integrating Security and Software Engineering: Advances and Future Vision. IGI Global (2006)
2. Viega, J., Bloch, J.T., Chandra, P.: Applying Aspect-Oriented Programming to Security. Cutter IT Journal **14**(2) (2001) 31–39
3. 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.
4. 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 Berlin / Heidelberg (2010) 197–213
5. Heineman, G.T., Council, W.T.: Component-Based Software Engineering: Putting the Pieces Together. Addison-Wesley, USA (2001)
6. Stevens, W.P., Myers, G.J., Constantine, L.L.: Structured Design. IBM Systems Journal **13**(2) (1974) 115 –139
7. 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)
8. 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)

9. 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
10. 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
11. Gamma, E., Helm, R., Johnson, R.E., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, MA (1995)
12. Google Latitude API. `http://code.google.com/apis/latitude/`
13. Google Contacts API. `http://code.google.com/apis/contacts/`
14. Hammer-Lahav, E., et al.: The OAuth 2.0 Authorization Protocol. Internet-Draft (September 2011) draft-ietf-oauth-v2-22
15. Dierks, T., Rescorla, E.: The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard) (August 2008)
16. Melnikov, A., Zeilenga, K.: Simple Authentication and Security Layer (SASL). RFC 4422 (Proposed Standard) (June 2006)
17. Cantor, S., et al.: Assertions and Protocols for the OASIS Security Assertion Markup Language (SAML) v2.0 (March 2005)
18. 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)
19. 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), 2006, Montpellier, France. Volume 4276 of LNCS, Springer–Verlag Heidelberg (2006) 1613–1632
20. 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, Berlin, Heidelberg, Springer-Verlag (2011) 304–318
21. Abadi, M., Lamport, L.: The Existence of Refinement Mappings. Theoretical Computer Science **82**(2) (1991) 253–284
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. Pavlich-Mariscal, J., Michel, L., Demurjian, S.: Enchancing UML to Model Custom Security Aspects. In: AOM '07: Proceedings of the 11th Workshop on Aspect-Oriented Modeling. (2007)
24. Jézéquel, J.M.: Model Driven Design and Aspect Weaving. Software and System Modeling **7**(2) (February 2008) 209–218
25. Alexander, R.T., Bieman, J.M.: Challenges of Aspect-oriented Technology. In: Workshop on Software Quality, 24th Int'l Conf. Software Engineering. (2002)
26. Jürjens, J.: Secure System Development with UML. Springer-Verlag (2004)
27. 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
28. 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