Kai Herrmann, Hannes Voigt, Andreas Behrend, Jonas Rausch, Wolfgang Lehner

A Rollback Mechanism to Recover from Software Failures in Role-based Adaptive Software Systems

Erstveröffentlichung in / First published in:


DOI: [https://doi.org/10.1145/3079368.3079388](https://doi.org/10.1145/3079368.3079388)

Diese Version ist verfügbar / This version is available on:

[https://nbn-resolving.org/urn:nbn:de:bsz:14-qucosa2-752149](https://nbn-resolving.org/urn:nbn:de:bsz:14-qucosa2-752149)
A Rollback Mechanism to Recover from Software Failures in Role-based Adaptive Software Systems

Nguonly Taing¹ Thomas Springer¹ Nicolás Cardozo² Alexander Schill¹
¹Faculty of Computer Science, Technische Universität Dresden
²Systems and Computing Engineering Department, Universidad de los Andes
¹Dresden, Germany, ²Bogotá, Colombia
firstname.lastname@tu-dresden.de,n.cardozo@uniandes.edu.co

ABSTRACT
Context-dependent applications are relatively complex due to their multiple variations caused by context activation, especially in the presence of unanticipated adaptation. Testing these systems is challenging, as it is hard to reproduce the same execution environments. Therefore, a software failure caused by bugs is no exception. This paper presents a rollback mechanism to recover from software failures as part of a role-based runtime with support for unanticipated adaptation. The mechanism performs checkpoints before each adaptation and employs specialized sensors to detect bugs resulting from recent configuration changes. When the runtime detects a bug, it assumes that the bug belongs to the latest configuration. The runtime rolls back to the recent checkpoint to recover and subsequently notifies the developer to fix the bug and re-applying the adaptation through unanticipated adaptation. We prototype the concept as part of our role-based runtime engine LyRT and demonstrate the applicability of the rollback recovery mechanism for unanticipated adaptation in erroneous situations.

CCS CONCEPTS
-Software and its engineering → Frameworks;

KEYWORDS
Rollback, Recovery, Roles, Unanticipated Adaptation

ACM Reference format:
DOI: http://dx.doi.org/10.1145/3079368.3079388

1 INTRODUCTION
Context-dependent systems can be realized programmatically by using paradigms like Role-oriented Programming (ROP) [8] or Context-oriented Programming (COP) [11]. For this type of systems, testing is surely harder than the static systems, since the internal application configuration can be adapted at run time as a result of dynamic context changes. Moreover, current testing frameworks lack contextual constructs or are nonexistent for context-aware adaptive systems [11]. Hence, we argue that testing context-dependent applications is inherently challenging, and we cannot eliminate all the bugs during the testing phase. This problem is even more appealing in the presence of unanticipated adaptation as the new code is added or modified dynamically.

In this respect, we imply that bugs are inevitable in highly dynamic environments, and they cause runtime failure. Such implication is more realistic than the assertion of an error-free execution assumed by ROP and COP designers. As a consequence, adaptive systems are susceptible to software failures making them less applicable for creating robust and highly available applications. One approach to overcome this situation is making runtime systems for ROP and COP more robust against software failures.

This paper addresses a rollback mechanism to recover a runtime for adaptive software systems from software failures caused by bugs. Our idea is based on the notion of “If we cannot avoid bugs, embrace them, and let developers fix them for a subsequent adaptation.” The main concept of the mechanism is to automatically take system snapshots on a per-thread basis before performing an adaptation. Additionally, bug monitors utilizing exception handling are installed per-thread to detect bugs. Whenever a failure occurs, our runtime assumes this is a result of the latest adaptation and rolls gracefully back to the previous configuration snapshot. Furthermore, the runtime reports the defective system configuration with the intention that the bug can be fixed subsequently and an updated version of the code can be introduced to the system.

The main contribution of this paper is a rollback recovery mechanism that allows the adaptive runtime system to withstand the common software failures uncaught during testing and compiling phase and to avoid run-time crashes as a consequence of defective adaptations. Furthermore, we strengthen the applicability of the unanticipated adaptation as the runtime is ready to embrace the problem. We incorporated the concept as part of our role-based runtime engine LyRT [14]. Based on this proof-of-concept implementation, we demonstrate the applicability of our recovery mechanism in erroneous situations.

The remainder of this paper is organized as follows. Section 2 describes the motivating example and role concept, following by Section 3 explaining the idea of rollback recovery mechanism. Section 4 illustrates the prototype as a proof of concept and we discuss open issues in Section 5. Section 6 present related work. Finally, we conclude, and we outline future work in Section 7.
2 BACKGROUND

The idea developed in this paper targets to adaptive software systems in general. However, to keep the discussion of the paper focused, we discuss the case of ROP. Therefore, this section first brings a motivating example to illustrate the problem and then gives the foundation of roles.

2.1 Motivating Example

We use a simple chat application to motivate our work and to illustrate our solution. We assume the chat server serves multiple clients. The chat server adapts to individual client needs, i.e., the server can integrate specific behavior like message compression or encryption on a per-client basis.

![Figure 1: Chat application with per-thread adaptation](image)

The chat server uses a shared channel object for data sending and receiving, supporting a raw data transmission behavior as illustrated in Figure 1. The channel object can be adapted to provide different compression algorithms (e.g., LZ, or LZX) according to clients’ needs. The compression algorithms are implemented and deployed to the runtime beforehand.

During execution, chat server administrators may want to introduce a new encryption feature to the channel object (e.g., AES Encryption), to respond to privacy concerns of their users. Let us assume, although carefully tested, the encryption code contains a divide-by-zero bug. This bug is uncaught by the compiler and thus, not detected before deployment. The chat server runs on a role-based runtime that supports dynamic software updates as unanticipated adaptation. The encryption feature is dynamically applied by the running channel object to encrypt all future messages for a particular user session represented by “Thread2” in Figure 1. Due to the bug in the newly introduced code, an error exception may force the application or even the complete runtime to halt.

With the help of the rollback recovery mechanism that will be introduced in detail in Section 3, we enable the runtime to overcome such software failures by gracefully rolling back to a stable system configuration that was checkpointed in advance.

2.2 Nature of Roles

The role concept has been applied in many disciplines, ranging from data modeling, conceptual modeling to programming [8, 13]. It consists of three main abstractions player, role, and compartment. Players (also called core objects) implement the basic system behavior considered to be static over the complete lifetime. Roles encapsulate dynamic behavior that can dynamically extend or adapt the behavior of players. In our chat application example, the shared channel object is a player, representing the basic behavior for message transmission. Its behavior can be extended or adapted by roles that implement behavior for message compression or encryption.

Roles are considered to possess three natures: behavioral, relational, and context-dependent [8]. The behavioral nature characterizes the objects that can acquire and abandon roles dynamically, adapting the object’s behavior. For instance, the shared channel object (player) can play the LZX compression role. The relational nature represents constraining relations between roles. For example, the LZ compression role played by a channel object on the server side has a communication relation with another LZ role located on client side subject to many-to-one constraint. The context-dependent nature scopes roles and relations, encapsulating them within a context-dependent boundary, called compartment. For example, the shared channel object exclusively plays the AES encryption role under the secure environment compartment. Not only roles, but also relations are context-dependent because they are filled by roles within a compartment scope.

3 ROLLBACK RECOVERY MECHANISM

This section illustrates our proposed rollback recovery mechanism for role-based runtime systems to handle software failures introduced dynamically by unanticipated adaptation.

According to the role concept, an application configuration can be described as a set of compartments, a set of players, and a set of roles with bindings to players. We consider adaptation as a process of applying a set of adaptation operations to realize the transition of a role-based application from one valid configuration to another. Since dynamic behavior is encapsulated in roles, a system configuration can, for instance, be changed by performing a set of role binding and unbinding operations. In the case of unanticipated adaptation, adaptation operations for installing new roles or updating the implementation of a role with a new version are provided by the role-based runtime. We introduce the term adaptation transaction for a set of related adaptation operations that defines a scope to transfer an application from a source to a target configuration.

The main idea of our approach is to embrace failures caused by bugs if we cannot avoid them, and enable the runtime to recover by rolling back to a recent checkpoint. The system generates a checkpoint before initiating a new adaptation transaction. A checkpoint is a serialized representation of the current application configuration, i.e., the active compartments, a list of objects (representing roles) including their states and binding information reflecting the currently active plays relations between players and roles.

![Figure 2: Checkpointing before adaptation](image)
not been caught within that application configuration. Meanwhile, the runtime records the defective configuration to prevent it from being re-activated. The system also generates a notification to the developer responsible for the bugs. The adaptation transaction can be re-applied after the bug has been fixed and the new code version has been shipped.

As an example, Figure 2 shows a system with an initial configuration denoted as AC0, and adapts twice to establish the application configurations AC1 and AC2. Checkpoints of AC0 and AC1 are saved before the corresponding adaptation transactions. A bug occurs after the adaptation to configuration AC2; so that we suppose the bug is introduced in configuration AC2 which needs to be rolled back to keep the application running in a valid configuration.

We propose a runtime architecture to realize our concept as an extension to LyRT [14], a role-based runtime engine. We then give an overview of LyRT followed by the extension components.

3.1 LyRT: A Role-based Run-time Engine

LyRT [14] is a role-based run-time engine reifying CROM [7], a unified model of compartments, roles and objects that captures the three natures of roles described in Section 2.2. LyRT is designed to support run-time variability through dynamic role-playing relations, enabling both anticipated and unanticipated adaptation.

LyRT relies on a dynamic instance binding mechanism which constructs the binding between players, roles and their associating compartment in a loosely coupled manner. Instances of the three types are kept separately and their binding information is stored in a lookup table. The lookup table is a mutable data structure holding the necessary binding relations which identifies the roles that are bound to different objects, the relations between roles, and the compartment where these bindings take place. The lookup table is used for dynamic method dispatching and thus adaptation.

To support unanticipated adaptation, the registry needs two additional functionalities. (1) A dynamic class loader is needed to load newly introduced roles into the runtime and to reload the existing roles in case their definition changes. (2) The registry must be able to process binding operations for those new roles. To handle these functionalities, LyRT uses an XML script to specify the existing objects and their relations to the newly adapting roles as well as the associating compartment. Once LyRT loads new roles and their relations are pushed to the lookup table, the specified objects adapt their behavior accordingly.

3.2 Rollback Recovery Architecture

The runtime architecture had to be extended as depicted in Figure 4 to incorporate the rollback recovery mechanism as part of LyRT. The extension comprises four components, namely a bug sensor, a control unit, a checkpoint manager, and a rollback unit. With our approach, the dynamic adaptation cycle is completed by (1) checkpointing the current system configuration, (2) adapting the application configuration, (3) detecting failures, (4) recovering from a software failure by rolling back to the previous application configuration and (5) notifying the developer about the failure.

Our proposal is based on the core of LyRT and we keep this core untouched. This is to ensure the portability of the solution to other systems. In addition to the four components to manage rollbacks, we introduce a language construct to represent Adaptation Transactions, described with the AT prefix. Snippet 1 shows a program execution containing an initial configuration AC0 established by the adaptation transaction AT0. Two further adaptation transactions AT1 and AT2 adapt AC0 to AC1 and AC2 respectively, as depicted in Figure 4. The adaptation transaction, in this case, is a block defining the adaptation operations attaching to objects declared within the try{} block as in Lines 4-7 and 9-12 of Snippet 1. The initial application configuration (AC0) constitutes the case where there are no binding operations (Line 2 in Snippet 1).

**Snippet 1: Adaptation transactions**

```
1 Registry reg = Registry.getRegistry();
2 Channel channel = reg.newCore(channel.class); //AT0
3 compartment.activate();
4 try{AdaptTrans at1 = new AdaptTrans(); //AT1
5 channel.bind(LZCompression.class);
6 .... //another set of adaptation operations
7 }
8 .......... // executing AC1
9 try{AdaptTrans at2 = new AdaptTrans(); //AT2
10 IRole r1=channel.getRole(LZCompression.class);
11} //end of adaptation transaction
```
Having this adaptation transaction block solves two problems. First, we can hook the checkpoint process into its implementation, transparently to developers. Second, operations executed in the block are atomic, either all or none of them execute. This atomicity determines whether the adaptation transaction is successful or not. For example, if the AESEncryption role in Line 11 of Snippet 1 has a bug, re-activating its binding will cause the same bug again. We avoid these situations.

**Checkpoint Manager.** In pursuit of supporting rollback, the runtime takes system snapshots as checkpoints, before performing an adaptation transaction. Given that the current active application configuration is persisted in the lookup table, creating checkpoints consists of serializing the records containing all the role instances and binding information, pushing them to the stack (Step 1 in Figure 4). Depending on the way we persist the data, serialization may be time-consuming, however, this is an effective method to preserve objects’ states and their dependencies, as opposed to shallow copying or cloning.

Besides the state stored in attributes of objects that represent roles, role’s behaviors might manipulate externally stored data, e.g., by utilizing file or database operations. For a complete rollback, such external state needs to be considered for checkpoint and rollback. However, for the sake of simplicity, we currently do not support this feature.

**Bug Sensors.** Bug sensors are run-time components designed to detect software failures that are uncaught during testing or compilation. Bug sensors are used to signal the Control Unit for recovery, as depicted in step 3 of Figure 4. Inspired by the software fault tolerance domain [10], two types of bug sensors can be deployed during run time. The first type of sensors exploits the exception handling system used by the application. The second type of bug sensors handles memory-related bugs, by detecting buffer overflow, memory leaks, etc. However, garbage-collected languages, such as Java, resolve most of the memory related issues. Therefore, we only implement the first type of bug sensors to monitor the run-time failures. This technique tackles only the failures raised by run-time exception. Bug sensors are installed either globally in the main program to intercept all the exceptions raised from all threads, or locally to each thread, capable of catching only a particular thread’s exceptions. We provide the default implementation of a bug sensor while developers can customize it to suit their needs.

**Rollback Unit.** The runtime rolls back by destroying the current defective configuration and restoring the latest checkpoint from the stack (step 4 in Figure 4). This process involves deleting the current records in the lookup table and inserting the serialized records from the checkpoint. The lookup table manages instance relations and their executing thread information. Therefore, the rollback is on a per-thread basis relevant to the only thread encountering failures. This design minimizes the overall system disruption and data loss.

The rollback procedure may present data loss in a period between the adaptation and a failure detection. This may raise a lot of problems for which the solution is application specific. Considering our motivating example, each user has their own representation of the data transmission (e.g., without compression, with compression, with encryption, or both). The new encryption feature is applied first to a testing group before system-wide adoption. An error in the encryption impacts only on the testing group while it has no effect on the existing clients.

**Control Unit.** It is a central component handling checkpoints, rollback, and bug notifications. The Control Unit communicates with the adaptation activation to generate a checkpoint between each adaptation and listens to the signal from the bug sensors to execute a rollback. The Control Unit records the defective adaptation configurations as to avoid the same configuration from being later activated, unless fixes have been made. Meanwhile the Control Unit notifies developers about the failure, and re-apply the configuration once the bug is solved, by means of unanticipated adaptation as illustrated in step 5 of Figure 4.

## 4 IMPLEMENTATION

As a proof of concept, we prototype the rollback recovery mechanism on top of LyRT [14], briefly described in Section 3.1. LyRT is built using the Java 1.8 specification that supports all the language features necessary to implement the proposed concept.

Checkpoints and rollbacks are developed inside a ControlUnit class containing two additional functions. First, the reportBugs() method logs the bugs found in a configuration for developers to fix. The hasDefectiveRoles() method checks and prevents the defective roles from being re-activated. We attach the checkpoint execution to the implementation of the adaptation transaction, the AdaptTrans class, while the rollback process is handled in the bug sensor implementation.

#### Snippet 2: Implementation of AdaptTrans class

```java
11  r1.bind(AESEncryption.class); //role plays role
12  }                   // bug occurs while executing AC2

public class AdaptTrans implements AutoCloseable {
  public AdaptTrans()
    ControlUnit.checkPoint(); //snapshot
  }

@Override
  public void close() {
    if(ControlUnit.hasDefectiveRoles())
      //Rollback the binding; otherwise commit
    Registry.cancelAdaptation(this);
  }
}

The try-with-resources of Java 1.7 or later can be specialized for AdaptTrans by implementing the AutoCloseable interface. The blueprint of the AdaptTrans class is illustrated in Snippet 2. The statement ControlUnit.checkPoint() inside the constructor generates the checkpoint while the overridden close() method executes after applying the adaptation to decide whether an adaptation is successful. If the configuration contains the defective roles that have been reported earlier, the whole adaptation should be canceled (Line 9). Therefore, within the try-block, role binding operations are executed atomically, either all or none of them execute.

### 4.1 Checkpoint and Rollback

Serialization can be used to implement a checkpoint. In Java, serialization requires implementing the Serializable interface. However, not all objects can be serialized, for instance, file pointer,
socket, database connection, etc. Performance is a well-known issue in the Java serialization library due to the code compatibility with previous versions. Therefore, we use a deep copy method to perform checkpoints by copying the relations and role instances in the lookup table and pushing them to the stack. Just as serialization, the deep copy ensures the role instances and their dependencies are copied by using direct assignment. This technique consumes more memory for large objects, but it is faster than serialization.

To rollback, we first remove the current configuration (i.e., binding information and role instances) from the lookup table. Then, we pop the recent snapshot from the stack and reinstate it to the lookup table.

4.2 Exception Handling as the Bug Sensor

By taking advantage of application level exception handling, we can hook an exception handler for all threads, a group of threads or a particular thread to intercept the raising error for handling a recovery process. We can set the default handler for any uncaught exception with UncatchedExceptionHandler interface to implement the bug sensor by overriding the default method where the logic of the recovery process takes place. Snippet 3 depicts the bug sensor implementation, installed at the beginning of the main code as shown in Snippet 4, or attached to each thread.

Snippet 3: Bug sensors implementation

```java
class BugSensor implements UncatchedExceptionHandler{
    void uncaughtException(Thread t, Throwable e) {
        ControlUnit.reportBugs(t + " thrown: " + e);
        ControlUnit.rollback(t); // perform rollback
        // Restart the user thread ....
    }
}
```

Snippet 4: Bug sensors installation

```java
public static void main(String... args){
    BugSensor sensor = new BugSensor();
    Thread.setDefaultUncaughtExceptionHandler(sensor);
    ... // the rest program
}
```

The current implementation of the bug sensor catches only the uncaught exceptions, such as DivideByZeroException, ArrayOutOfBoundException, etc., raised by the JVM. For the caught exceptions such as IOException, there are two options to handle and detect them with the bug sensor. First, application developers need to manually re-throw all caught exceptions; so that the JVM can perform a stack trace of the caught exceptions until it reaches the exception block in the bug sensor. The second option requires no application developers involvement, but still, it needs to re-throw caught exceptions. Exception re-throwing can be injected automatically during class loading, by a bytecode re-writing framework such as ASM\(^2\). For our implementation, we chose the first option.

5 DISCUSSION

This section discusses how the proposed run-time mechanism and its implementation solve the problem in the motivating example in Section 2.1, and it raises some open issues to consider.

Run-time adaptation and managing unanticipated adaptation are the central pieces of LyRT \[14\]. The channel object can carry out its own behavior to transmit raw data or to use the compression adaptation. LyRT, in this case, allows channel objects to bind to a new compression role in a compartment activation (Section 3.1). In our implementation, we surround the binding operation with the AdaptTrans try-block as shown in Snippet 1.

Unanticipated adaptation allows an unforeseen and previously unknown encryption feature to be installed at run time. According to the role concept and our motivating example, the AES encryption functionality is implemented as role that can be dynamically loaded, deployed and bound to the shared channel object in LyRT. As shown in Lines 9-12 of Snippet 1 the AESEncryption role is bound to the LZCompression role, to perform both compression and encryption of transmitted data in application configuration AC2.

When a method of AESEncryption is executed, a DivideByZeroException is thrown. Our runtime reacts with the help of the bug sensors and repels the adaptation by rolling back to the recent checkpoint (application configuration AC1) which contains only LZCompression. Additionally, the developers are notified to fix the bug. Due to the per-instance binding of the runtime, this bug does not affect other existing user connections, but only the affected thread needs to restart to use the previous configuration for data transmission. Note that, although the thread is restarted and data gets lost within a time window, the user session is still maintained. Therefore, client and server can still communicate even though without the encryption feature. After developers fix the divide-by-zero bug, the feature is recompiled and the system can re-adapt with the new version of encryption.

Although the proposed mechanism responses to our setting scenario, there are some limitations and open issues, which we describe in the following.

How does rollback affect the program execution flow? Once bugs are detected, the runtime rolls back the affected thread to the previous configuration and re-spawns the thread. This seems to be odd (i.e., data might get lost) because program flow should resume where the error is caught after the rollback. This is a limitation of which we have not found any better solution because we exploit exception handling for our bug sensor and Java does not offer an on-error-resume feature. As a result, data loss has to be handled at application level. For the chat application, the effect of the bug would cause the loss of chat messages that could be detected by the application-level chat protocol and handled by retransmitting messages.

Bugs are detected after several adaptations. Suppose errors are introduced with configuration AC1 but the faulting method has never been executed. Then, the runtime adapts to configurations AC2 and AC3. In AC3, the runtime catches the bugs of AC1. In this case, the system rolls back to AC2. This is the wrong choice, which we consider for future enhancement.

What is the effectiveness of the bug sensor? The degree of fault tolerance of the proposed mechanism lies under the power of the bug sensors. Our current implementation relies on the exception handling mechanism to detect the caught and uncaught exceptions at the application level. Nonetheless, JVM related issues such as OutOfMemoryError that shuts downs the JVM cannot be handled. In this regard, implementing a bug sensor at the JVM level could be more efficient, but it may face compatibility issues.

\(^2\)http://asm.ow2.org/
6 RELATED WORK
This work is influenced by fault tolerance mechanisms adopted for adaptive software systems, as we envision to have adaptations regardless of bugs. The design of Rx [10] also inspires our proposed architecture. Rx is a safe technique to repel bugs by shifting the application execution environment under the control of checkpoint and rollback mechanism. Rx treats bugs as allergies relating to the executing environment. Changing the environment for the same code execution is likely to remove bugs. For example, a buffer overflow is caught; so that the program rollbacks to the recent checkpoint and re-execute in a different environment. Rx allocates new memory buffer that has been misplaced by the developers. As a result, the program continues its operation without intrusion on the user codes. Rx is the OS level approach implemented to manage the user-space execution environment. In contrast, our rollback mechanism is an application-level technique built as part of the runtime that requires programmers’ awareness to manage fault. It is designed specifically to handle failures for context-dependent applications that is hardly achieved by the OS-level approaches. Therefore, application level fault tolerance is necessary [3].

The linguistic mechanisms explicitly supporting contextual runtime adaptation have been developed underpinning the ROP [8] and COP [11]. Adaptation is assumed successfully installed and executed in an error-free environment. This assertion is not always realistic in the production systems though developers can rely on static analysis or bug finding tools such as FindBugs [5]. The tools, nonetheless, are incapable of capturing all the bugs but rather find them as many as possible.

ROP languages [4, 9, 14, 15] address composition and dynamic adaptation by playing roles under the activation of a placeholder, similar to our compartment, but it is named differently. None of them takes into account on adaptation pitfalls resulting from software bugs. In the recent development, SCROLL [9] provides a static model checking to bring type safety which reduces unnecessary bugs arising from MethodNotFoundException when a core object attempts invoking a method of a missing role. However, execution in an error-free environment is still an assumption.

Similarly, some COP languages [1, 2, 6] offer verification between adaptable entities that are composed to core objects to assure desired system properties. Nonetheless, detecting and avoiding bugs are still open for exploration.

Unanticipated adaptation support in LyRT and the Dynamic Software Update (DSU) [12] share a similarity regarding code updating. However, DSU allows programmers to update any part of the system regardless of software architectures. In LyRT, developers need to follow the principle of adaptive software systems, i.e., explicitly define static and dynamic parts, and only the dynamic part is updated. To the best of our knowledge, there is currently no DSU approach supporting a comparable rollback recovery mechanism.

7 CONCLUSION
In this paper, we argue that testing context-dependent applications written in ROP or COP is inherently challenging. Since we cannot eliminate all the bugs during the testing phase, we need to embrace them during execution. We proposed a rollback recovery mechanism as part of a runtime system, realized in ROP, to recover the runtime from software bugs. The ROP runtime is enhanced with four functionalities for this purpose, namely bug capturing, checkpoint, rollback, and control unit. Our proposed runtime performs checkpoints before applying any adaptation transaction. A bug sensor component, based on exception handling, permanently waits for bugs likely to appear in method execution and signals the runtime to roll back, if detected, to the previous configuration by instating from a saved snapshot. The defective adaptation configuration is secured from being used by other threads, giving time to system developers to fix the bugs before re-applying the adaptation by means of unanticipated adaptation. This technique ensures runtime progress and fits the motivating example allowing clients and servers to communicate despite the missing encryption feature.

As future work, we will investigate the run-time overhead and the effectiveness of bug sensors. We also intend to address the limitations discussed in Section 5 to improve our approach.

ACKNOWLEDGMENTS
This work is partially funded by the Erasmus Mundus Program and the German Research Foundation (DFG) through the Research Training Group on Role-based Software Infrastructures for Continuous-Context-Sensitive Systems (RoSi) (GRK 1907).

REFERENCES