Engineering Adaptation with *Zanshin*: An Experience Report

Genci Tallabaci
Department of Information Engineering and Computer Science, University of Trento, Italy
genci.tallabaci@studenti.unitn.it

Vítor E. Silva Souza*
Computer Science Department, Federal University of Espírito Santo (Ufes), Brazil
vitorsouza@inf.ufes.br

***Abstract*—The *Zanshin* framework adopts a Requirements Engineering perspective to the design of adaptive systems and is centered around the idea of feedback loops. Evaluation experiments conducted so far have used simulations, limiting the strength of our conclusions on the viability of our proposal. In this paper, we report on the experience of applying *Zanshin* to an existing base system, a software that simulates an Automated Teller Machine (ATM), available online, drawing conclusions on the applicability of the framework's potential in real-life situations.**

Index Terms*—Adaptive systems, adaptation, framework, experiment, case study, experience report, ATM, *Zanshin

I. Introduction

Motivated by the increasing complexity of software systems and the ever greater uncertainty of the environments within which they operate, Software Engineering (SE) researchers have recently taken interest in software *self-adaptation* as a research topic [1].

Taking a Requirements Engineering (RE) perspective and considering that feedback loops constitute an architectural solution for adaptation [2], we proposed the *Zanshin* method for the design of adaptive software systems, recently defended as a PhD thesis at the University of Trento [3].

In the research conducted thus far, we have performed simulation experiments with the purpose of evaluating the proposal. A framework, also called *Zanshin*, was developed and simulations of existing exemplars such as a Meeting Scheduler and an Ambulance Dispatch System (cf. [3]) were implemented and experimented with, showing that the framework analyzed augmented requirements models and produced sensible responses during simulations of system failures.

Simulations, however, are internal to the framework's architecture and provide limited evaluation scope. The main objective of this paper is to evaluate the effectiveness of the framework by applying it to an existing software system available in the public domain. In particular, we want to answer questions such as: Can *Zanshin* effectively adapt an existing base system at runtime? What is the effort required to integrate the managed system and the framework? In this experience report, we describe the application of our proposal to an existing software and its execution alongside our framework in a few scenarios.

* Work done while in Trento as PhD student/post-doc.

The system chosen for this experiment was an Automated Teller Machine (ATM) software publicly available on the Internet for non-commercial use. Although the software itself is a simulation of a real ATM (i.e., not the software of a physical ATM), under the perspective of our experiments with *Zanshin* it can be considered an external software. As shown later, results indicate that our prototype framework, if further developed, could well be applied to real-life situations and the work done herein constitutes a first step towards this direction.

The rest of the paper is structured as following: Section II describes the ATM exemplar and available requirements models that were used as baseline for the application of *Zanshin*; Section III explains the methodology that was followed, illustrating how *Zanshin* works; Section IV shows the augmented requirements models produced for an adaptive ATM, focussing on a couple of adaptation scenarios; Section V briefly presents our current framework implementation and the artifacts that had to be created in order to integrate the ATM with it; Section VI discusses the scenarios executed in this experiment, presenting their results; finally, Section VII concludes.

II. The ATM Exemplar

In an "attempt to give a complete example of object-oriented analysis, design, and programming applied to a moderate size problem", professor Russell C. Bjork of Gordon College, USA published complete documentation on a Software Engineering project for the development of a software that simulates an Automated Teller Machine (ATM), from requirements (use cases) to code (in Java™). The material is available freely for non-commercial educational purposes at http://www.math-cs.gordon.edu/courses/cs211/ATMExample/.

We should note that professor Bjork considers the ATM to be of moderate size for the purposes of education in his SE course, but one could argue that, for practitioners, the ATM is a small system, or even a toy example, compared to distributed, ultra-large systems that are developed nowadays. Still, we consider the ATM a good baseline for our experiments.

As stated earlier, the software designed and developed by professor Bjork is not a software for an actual physical ATM and, therefore, abstracts many issues present in a real ATM system. We therefore avoid the term "ATM system" and, instead, use "ATM exemplar" or simply "ATM". An exemplar is a shared, well-defined problem adopted by researchers of a

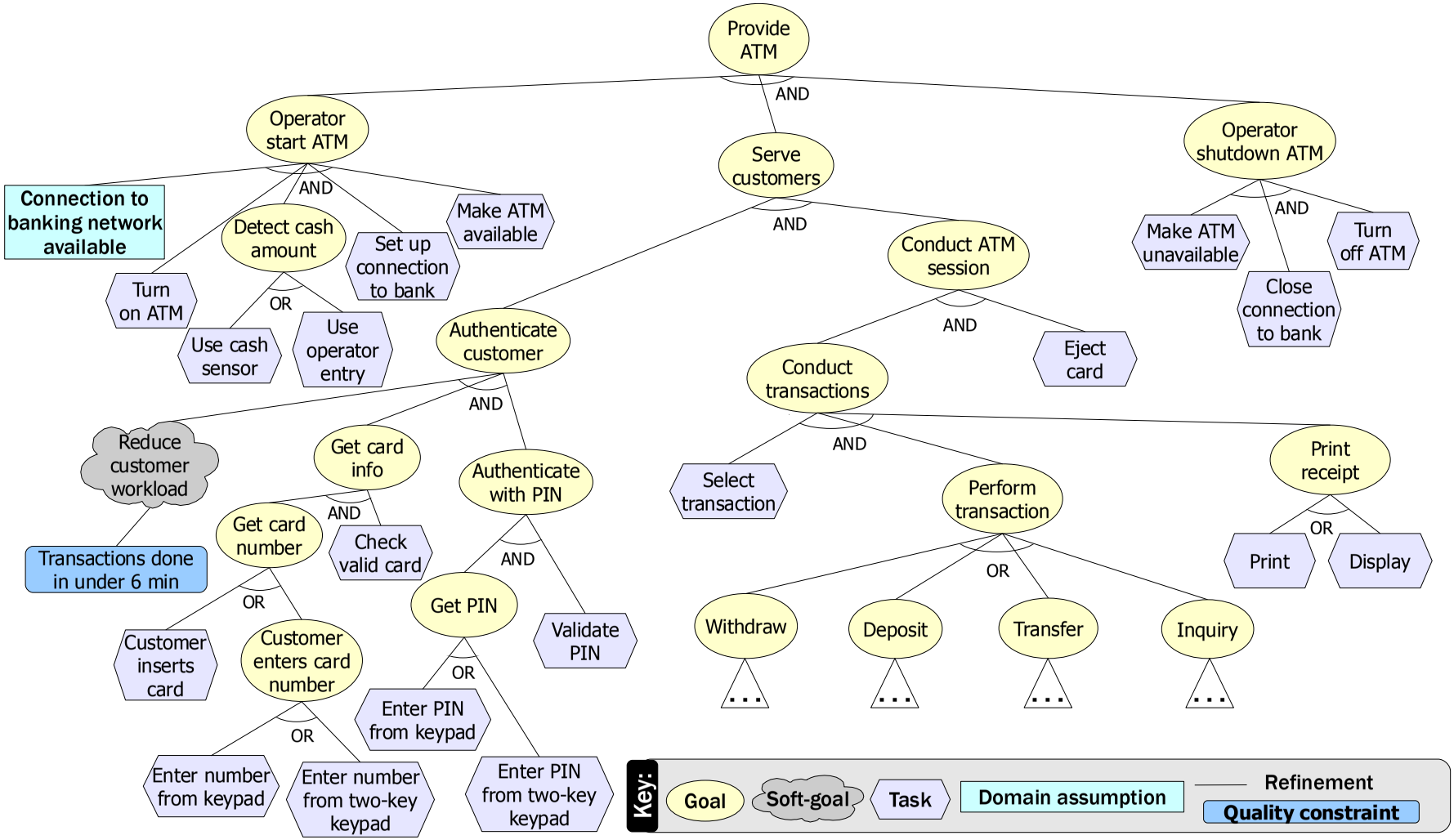

Fig. 1. Goal model representing the requirements for a software that simulates an ATM, based on [4].

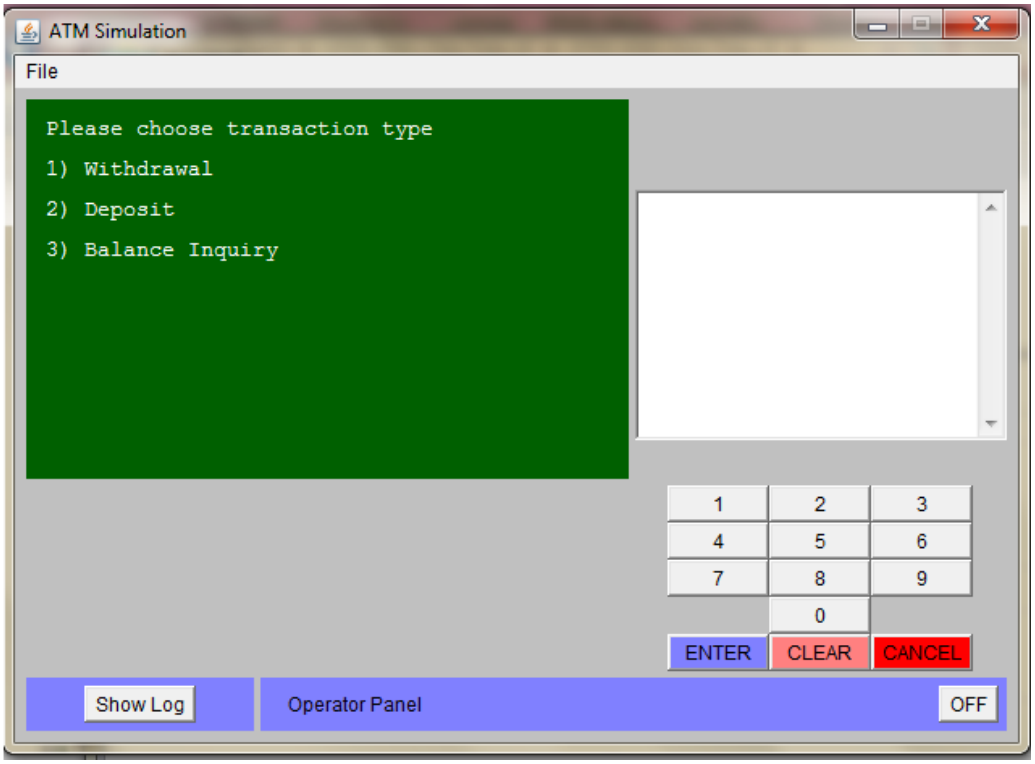

Fig. 2. Screenshot of the running ATM implementation.

specific field for presenting and comparing proposals. We hope that this work helps establishing the ATM as an exemplar in the area of adaptive systems.

Being a third-party software with extensive documentation and complete source code available made the ATM a good choice for this experiment. However, the material available online modeled requirements using UML Use Cases, whereas our framework is based on Goal-Oriented Requirements Engineering (GORE) [5] and represents requirements using goal models. Earlier research at the University of Toronto (see, e.g., [4]) had already used the ATM exemplar to experiment with GORE-based approaches, adding a goal model that captured system requirements. This model served as a starting point for this evaluation.

In his masters dissertation [6], Tallabaci used the ATM goal model as input to the *Zanshin* method for the design of adaptive systems (details in sections III and IV). Since the original requirements model was limited, it was initially extended to include also domain assumptions and quality constraints [7] in order to demonstrate these features of the framework. This extended model is shown in Figure 1. The original ATM implementation was extended as well to reflect these modifications.

The main goal of the ATM is to provide basic banking services, along with managerial services, such as having a bank operator turn the ATM on/off. Serving a customer means being able to authenticate their identity, then conduct a session of use, composed of one or more transactions. The latter can be a withdraw, a deposit, a transfer or an inquiry. Triangles with points of ellipsis under the goals that concern these four operations represent goal subtrees that are not relevant for the explanations contained herein and were removed to make the diagram simpler to read. Each transaction ends by printing a receipt.

The Java™ implementation of the ATM can be compiled and executed, simulating an automated teller machine. Figure 2 displays a screenshot taken after authentication of the client and before selecting the desired transaction. The green panel represents the ATM display, whereas the white one mocks the printing of receipts. Input is given through the keypad and an operations log is also available.

An important observation here is that the model provided by the University of Toronto is a high variability model, including alternatives for many of the system's goals that do not exist in the original ATM implementation. Nevertheless, we have decided to use the original code in order to be able to make it publicly available for the readers to download and try the experiments for themselves. The scenarios described in this paper did not include any of these new alternatives, but future experiments might require some of them to be reimplemented.

III. Methodology

The objective of this experiment is the production of requirements models for an adaptive system based on an existing application and the execution of such application according to its models following a few scenarios of failure/adaptation. To

accomplish this, we follow a methodology composed of three phases.

In the **first phase**, the *Zanshin* method for the design of adaptive systems is carried out. It suggests a process that can be further subdivided in three steps:

1) **Elicitation of *Awareness Requirements* (*AwReqs*)**: *AwReqs* are requirements that talk about the states assumed by other requirements — such as their success or failure — at runtime. *AwReqs* represent situations to which stakeholders would like the system to adapt, in case they happen. Therefore, they constitute the requirements for the monitoring component of the feedback loop that implements the adaptive capabilities of the system. In this step, 18 *AwReqs* were identified for the model of Figure 1;
2) ***System Identification***: considering that *AwReqs* can be used as indicators of requirements convergence at runtime, a possible adaptation strategy when they fail is to search the solution space to identify a new configuration — i.e., values for system parameters — that would improve the necessary indicators. The System Identification process is concerned with discovering such parameters and modeling the effect that changes in them have in the satisfaction of the indicators (*AwReqs*). For the ATM, for example, 10 parameters were identified in Figure 1;
3) **Specification of *Evolution Requirements* (*EvoReqs*)**: the third and final step of the process is the precise specification of the adaptation strategy to be used for each *AwReq* failure. This is done via *EvoReqs*, which prescribe changes to the requirements model itself in order to adapt, representing the requirements for the adaptation component of the feedback loop. Strategies can consist of precise evolution instructions or use the information modeled during System Identification to reconfigure the system's parameters.

As the examples later illustrate, *AwReqs* and *EvoReqs* are combined in an Event-Condition-Action (ECA)-based framework in which *AwReq* failures represent the 'E', applicability conditions associated to *EvoReqs* compose the 'C' and the *EvoReq* itself (i.e., the adaptation strategy) is the 'A'. *AwReqs* are also associated with resolution conditions to indicate when the problem has been solved. Later, in Section IV, we show part of the models produced by the application of *Zanshin* to the ATM exemplar. Complete models can be found in [6] and more details about *Zanshin* can be seen in [3].

In the **second phase**, the existing ATM implementation was integrated with the *Zanshin* framework, in order for the latter to provide the former with a generic feedback loop of adaptation based on the models produced earlier. Since up to this point experiments had been done using simulations alone, which were implemented as part of the framework's architecture, this step included the development of a module that exposed a remote interface that allows external systems to communicate with *Zanshin*. Details in Section V.

Finally, the **third phase** consists of artificially adding failures that trigger a few adaptation scenarios in order to verify that *Zanshin*, both method and framework, is able to provide adaptation capabilities to a base system based on its requirements model. These scenarios are described in the next section, whereas their output at runtime will be shown in Section VI.

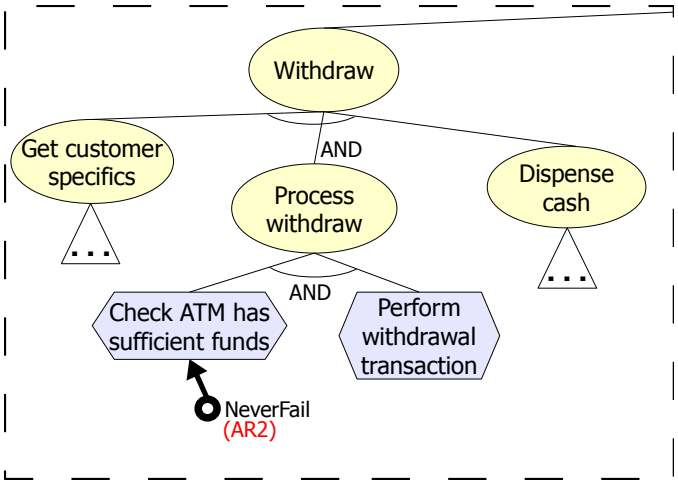

Fig. 3. Section of the ATM goal model showing *AwReq AR2*, which prescribes that task *Check ATM has sufficient funds* should never fail.

The interested reader can repeat the same methodology to execute the ATM experiment or a new experiment with a system of their own. Detailed instructions to obtain, install and use the *Zanshin* framework to execute one's own simulation or integrate an external system can be found in the project's *wiki* at https://github.com/sefms-disi-unitn/Zanshin/wiki.

IV. An Adaptive ATM

In this section, we present the models produced by the application of *Zanshin* to the ATM exemplar, based on the initial goal model provided in Figure 1. For reasons of space, we focus on a couple of scenarios that were modeled and implemented for this experiment: ATM printer malfunction and shortage of cash in the ATM dispenser.

A. Shortage of Cash

In Figure 1, the goals *Withdraw*, *Deposit*, *Transfer* and *Inquiry* were not detailed in order to make the model easier to read. Our first scenario of failure/adaptation, however, concerns the goal *Withdraw*, parts of which are now shown in Figure 3. To successfully perform a withdraw, the ATM checks that it has sufficient funds in its cash dispenser, represented in the model by the task *Check ATM has sufficient funds*. In case the bank notes available are not enough to serve the customer's request, this task fails and the whole operation is canceled.

During the first step of the *Zanshin* method, this "failure" scenario resulted in *AR2*, an *AwReq* that indicates we would like to be aware of task *Check ATM has sufficient funds* failures, so that the system can respond by adapting. During System Identification, ten different parameters were identified for the ATM, one of which concerns this particular scenario: *NOA*, or *Number of Operators Available*. This control variable

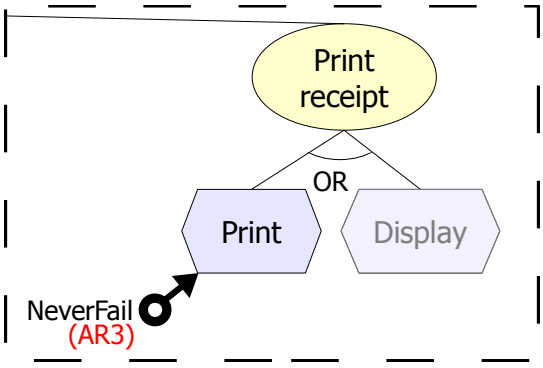

Fig. 4. Section of the ATM goal model showing *AwReq AR3*: task *Print* should also never fail.

tells the system how many employees of the bank should currently be available around the ATMs to assist customers in need. The following differential relation between this parameter and *AwReq AR2* was identified:

$$\Delta\left(AR2/NOA\right)\left[0, MaxOp\right] > 0 \tag{1}$$

Equation (1) tells us that if we increase *NOA*, the success rate of *AwReq AR2* will also increase. The rationale for this effect is that operators are allowed to refill the ATM with cash if their dispenser becomes empty, satisfying the customer's withdrawal request. The landmarks $[0, MaxOp]$ indicate this relation is valid only within this interval, $MaxOp$ being the maximum number of operators the bank can make available simultaneously (different branches could have different values for this variable).

Finally, in the last step of *Zanshin* the precise *EvoReq* to be associated with *AR2* is specified:

- **Adaptation strategy**: Reconfigure(∅);
- **Applicability condition**: always;
- **Resolution condition**: simple.

The above specification determines that whenever *AR2* fails, *Zanshin* should try to reconfigure system parameters in order to adapt. The argument ∅ indicates the default reconfiguration algorithm should be used, which chooses randomly from the available parameters the one that should be changed. This adaptation strategy is set to be always applicable and with simple resolution condition. The latter means that the problem will be considered fixed if a subsequent evaluation of *AwReq AR2* is satisfiable.

B. Printer Malfunction

We have already seen in Figure 1 that a session of use (goal *Conduct ATM session*) is composed of one or more transactions (goal *Conduct transactions*) and that each transaction ends with its receipt being printed (goal *Print receipt*). Considering that receipts can be very important to customers, *AwReqs AR3* is included in the model, constraining the task *Print* (which operationalizes the goal) to never fail. This section of the model is shown in Figure 4.

An obvious adaptation strategy in this case is to switch the operationalization of this goal and use the *Display* task instead. In this case, the goal's OR-refinement is considered a variation point and its value (i.e., which task is chosen) can be changed in one of two ways: through reconfiguration (as in the previous scenario) or via the `change-param()` instruction in an *EvoReq* (details in [3]).

However, as mentioned in Section II, in this experiment we use the original ATM implementation, which does not implement many of the variations included in the goal model provided by the Toronto research group. One of these variations is precisely the OR-refinement of Figure 4: the *Print* task is present in the ATM's original code, whereas the *Display* task is not. Figure 4 shows it dimmed down to represent the fact that it has not been implemented. We use a different adaptation strategy for this case, as below:

- **Adaptation strategy**: Retry(5 seconds);
- **Applicability condition**: at most twice per session;
- **Resolution condition**: simple.

Hence, whenever *AwReq AR3* fails, *Zanshin* makes the ATM wait for 5 seconds and then tries printing the receipt again. For each session, the *Retry* strategy is applicable at most twice, after which the framework aborts (the *Abort* strategy is by default the last resort for every *AwReq* of the system). Similar to *AR2*, the problem is considered solved when the *AwReq* succeeds.

V. Implementation

After applying *Zanshin* to the ATM goal model, we proceed to the second phase of the methodology described in Section III. At this point, we have the original ATM implementation, the *Zanshin* framework and the requirements model for the system, augmented with *AwReqs*, parameters, differential relations and *EvoReqs*, which specify the requirements for adaptation.

Our objective at this phase is to integrate the ATM implementation to the *Zanshin* framework so the latter would use the requirements model of the former in order to send adaptation instructions whenever the ATM presented a monitored failure. To accomplish this, we need to: (a) have *Zanshin* provide a remote API for use by external systems; (b) instrument the original ATM code to report failures of the monitored requirements; and (c) add a controller module to the ATM exemplar that would carry the adaptation instructions received from the framework.

A. Zanshin's Remote API

As mentioned in Section I, previous experiments with the *Zanshin* framework consisted of simulations of base systems, developed as part of the framework's architecture (cf. [3]). To be able to integrate the framework with external applications, we developed a remote server using Java™ RMI[1] technology. The **`IZanshinServer`** interface is shown in Figure 5.

[1]Remote Method Invocation, see http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136424.html.

```
1  package it.unitn.disi.zanshin.remote;
2
3  import java.rmi.Remote;
4  import java.rmi.RemoteException;
5
6  public interface IZanshinServer extends Remote {
7    String RMI_NAME = "Zanshin";
8    int RMI_PORT = 1099;
9
10   String registerTargetSystem(ITargetSystem
         targetSystem, String metaModel, String model)
         throws RemoteException;
11
12   Boolean isTargetSystemRegistered(String
         targetSystemId) throws RemoteException;
13
14   Boolean unregisterTargetSystem(String
         targetSystemId) throws RemoteException;
15
16   Long createUserSession(String targetSystemId)
         throws RemoteException;
17
18   Boolean disposeUserSession(String targetSystemId,
         Long userSessionId) throws RemoteException;
19
20   void logRequirementStart(String targetSystemId,
         Long userSessionId, String requirementsName)
         throws RemoteException;
21
22   void logRequirementSuccess(String targetSystemId,
         Long userSessionId, String requirementsName)
         throws RemoteException;
23
24   void logRequirementFailure(String targetSystemId,
         Long userSessionId, String requirementsName)
         throws RemoteException;
25
26   void logRequirementCancellation(String
         targetSystemId, Long userSessionId, String
         requirementsName) throws RemoteException;
27 }
```

Fig. 5. Remote interface of the *Zanshin* server.

As the code shows, the server exposes a few services that allow clients to: register themselves as a target (managed) systems in *Zanshin*, given their requirements models (line 10); verify if a given ID corresponds to a registered target system (line 12); unregister a target system, given its ID (line 14); create/dispose user sessions for a given target system (lines 16 and 18); and log a change of state in the life-cycle of a requirement instance (start, success, failure or cancellation), in the context of a particular target system and user session (lines 20–26).

When registering a system, the server receives a computer-readable version of the system's requirements model (e.g., the model of Figure 1 plus the adaptation elements described in Section IV) and prepares the system for management following a four-step process:

1) Stores the model in an internal repository;
2) Generates Java™ classes that represent each element of the model, founded on a meta-model of goal-based requirements;
3) Compiles these classes to bytecode;
4) Loads the classes in memory, associating them to a session manager.

For each user of the target system (e.g., each client performing transactions using an ATM), a new session should be registered in *Zanshin*. Through the session manager, the server creates new instances for the classes that were generated during system registration for every registered session, assigning it a unique ID.

At this point, the target system can notify *Zanshin* of changes in the life-cycle of requirements instances. Whenever the system starts pursuing a task or verifying a domain assumption or quality constraint, it should log a requirement start operation in the server. Likewise, when a particular task/assumption/constraint is satisfied, the system should log its success (failure and cancellation are analogous). *Zanshin* automatically propagates state changes up the goal hierarchy based on the type of refinement (*AND* or *OR*).

Provided with notification of changes of state in requirements instances, *Zanshin* monitors for *AwReq* satisfaction or failure[2] and, in the latter case, determines the *EvoReq* instructions to be sent back to the target system.[3] *Zanshin*'s internal monitoring/adaptation cycle is detailed in [3].

The *Zanshin* server communicates back with the target system through a remote callback reference that implements the interface **`ITargetSystem`**. This object is passed as argument during registration (see Figure 5, line 10) in order to be stored in the session manager associated with that particular system. Figure 6 shows the code for this Java™ interface.

The target system's controller is, therefore, supposed to implement all the methods declared in Figure 6, which correspond to all the possible primitive *EvoReq* commands that compose adaptation strategies (for a detailed table, see [3]). For instance, when reconfiguration is used, *Zanshin* will call the **`applyConfig()`** method (line 10 for local, in-session, changes; line 12 for global, from-now-on, changes), specifying the new system configuration as an argument.

The full source code of the *Zanshin* framework is available at a public version control repository: https://github.com/sefms-disi-unitn/Zanshin.

B. ATM Monitoring

With the remote API described in the previous subsection, we integrate the ATM original implementation to the *Zanshin* framework in order to build an adaptive ATM. Here, we describe the monitoring part of this integration, i.e., registering the ATM as a target system, creating user sessions and logging life-cycle changes of requirements instances.

A new module (a package called **`zanshin`**) adds to the original ATM code new components that are needed for the integration with *Zanshin*. The main class of this new module is **`TargetSystemController`**, a singleton class that implements the remote callback interface **`ITargetSystem`**. This class, heretofore referred to as *the controller*, mediates communication between *Zanshin* and the ATM. A remote reference to it is sent to the server upon registration of the ATM as a target system.

In order to keep the changes to the original ATM code to a minimum,[4] we use Aspect Oriented Programming [8] tech-

[2] At the moment of this paper's submission, however, the *AwReq* monitoring infrastructure described in [3] had not been integrated with the framework, which was limited to monitoring *AwReqs* of the type "requirement R should never fail". The list of current *Zanshin* limitations is also available at the project's *wiki*.

[3] Adaptation strategies were implemented in an as-needed basis, thus some of them may not be currently available. The interested reader may check the project's *wiki* and issue tracker. Contributions are welcomed.

[4] In the few cases one of the original classes had to be modified, we signaled the change with a comment: **`// <Zanshin Modification>`**.

```
1  package it.unitn.disi.zanshin.remote;
2
3  import java.rmi.Remote;
4  import java.rmi.RemoteException;
5  import java.util.Map;
6
7  public interface ITargetSystem extends Remote {
8    void abort(Long sessionId, String awreqName)
         throws RemoteException;
9
10   void applyConfig(Long sessionId, Map<String,
         String> newConfig) throws RemoteException;
11
12   void applyConfig(Map<String, String> newConfig)
         throws RemoteException;
13
14   void changeParameter(Long sessionId, String
         paramName, String value) throws
         RemoteException;
15
16   void changeParameter(String paramName, String
         value) throws RemoteException;
17
18   void copyData(Long sessionId, String srcReqName,
         String dstReqName) throws RemoteException;
19
20   void disable(String reqClassName) throws
         RemoteException;
21
22   void enable(String reqClassName) throws
         RemoteException;
23
24   void initiate(Long sessionId, String reqName)
         throws RemoteException;
25
26   void resume(Long sessionId, String reqName) throws
          RemoteException;
27
28   void rollback(Long sessionId, String reqName)
         throws RemoteException;
29
30   void sendWarning(Long sessionId, String actorName,
          String awreqName) throws RemoteException;
31
32   void suspend(Long sessionId, String reqName)
         throws RemoteException;
33
34   void terminate(Long sessionId, String reqName)
         throws RemoteException;
35
36   void waitFor(Long sessionId, Long timeInMillis)
         throws RemoteException;
37
38   void waitForFix(Long sessionId, String awreqName)
         throws RemoteException;
39 }
```

Fig. 6. Remote interface for a target system managed by *Zanshin*.

niques, in particular the *AspectJ* framework.[5] Figure 7 shows the code for the **Init** aspect, responsible for initializing the ATM–Zanshin integration.

Init defines four point-cuts (lines 6–11), each of which with a corresponding advice. During the execution of the ATM, single instances of classes **ATM** (which represents the ATM itself) and **Frame** (which holds the graphical user interface components) are expected to be created. Advices applied in point-cuts **atmCreation** (line 13) and **frameCreation** (line 18) intercept the moment in which these components are instantiated and inject them in the controller, which needs them for adaptation actions later. Moreover, methods **switchOn()** (line 23) and **switchOff()** (line 28) in class **ATM** are also intercepted and determine, respectively, the creation and disposal of a user session.

Another aspect, this one responsible for monitoring the *Print* task for the scenario described in Section IV-B, is called **Print** and is shown in Figure 8. It defines a single point-cut (line 7) over the **printReceipt()** method in class **ReceiptPrinter**, responsible for printing receipts at the

[5]http://www.eclipse.org/aspectj/.

```
1  import atm.ATM;
2  import java.awt.Frame;
3  import zanshin.TargetSystemController;
4
5  public aspect Init {
6    pointcut atmCreation(): call(ATM.new(..));
7    pointcut frameCreation(): call(Frame.new(..));
8    pointcut sessionStart():
9      execution(void ATM.switchOn());
10   pointcut sessionEnd():
11     execution(void ATM.switchOff());
12
13   after() returning(ATM atm): atmCreation() {
14     TargetSystemController controller =
           TargetSystemController.getInstance();
15     controller.setAtm(atm);
16   }
17
18   after() returning(Frame frame): frameCreation() {
19     TargetSystemController controller =
           TargetSystemController.getInstance();
20     controller.setFrame(frame);
21   }
22
23   after() returning(): sessionStart() {
24     TargetSystemController controller =
           TargetSystemController.getInstance();
25     controller.startSession();
26   }
27
28   after() returning(): sessionEnd() {
29     TargetSystemController controller =
           TargetSystemController.getInstance();
30     controller.endSession();
31   }
32 }
```

Fig. 7. **Init** aspect, responsible for initialization.

```
1  import atm.physical.ReceiptPrinter;
2  import banking.Receipt;
3  import zanshin.AtmRequirement;
4  import zanshin.TargetSystemController;
5
6  public aspect Print {
7    pointcut printReceipt(): execution
8      (void ReceiptPrinter.printReceipt(Receipt));
9
10   before(): printReceipt() {
11     TargetSystemController controller =
           TargetSystemController.getInstance();
12     controller.logRequirementStart(AtmRequirement.
           T_PRINT_RECEIPT);
13   }
14
15   after() throwing(): printReceipt() {
16     TargetSystemController controller =
           TargetSystemController.getInstance();
17     controller.logRequirementFailure(AtmRequirement.
           T_PRINT_RECEIPT);
18   }
19
20   after() returning(): printReceipt() {
21     TargetSystemController controller =
           TargetSystemController.getInstance();
22     controller.logRequirementSuccess(AtmRequirement.
           T_PRINT_RECEIPT);
23   }
24 }
```

Fig. 8. **Print** aspect, which monitors the *Print* task.

end of transactions. Three advices are applied to this point-cut: before the method executes a *start* is logged (line 10), if the method throws an exception a *failure* is logged (line 15) and if the method returns properly a *success* is logged (line 20).

Another aspect, **CashDisp**, responsible for monitoring the task *Check ATM has sufficient funds* for the scenario described in Section IV-A, is very similar to the **Print** aspect and is thus not shown. Monitoring could be added to other tasks of the system in the same fashion: identify the method that implements the task, add a point-cut to its execution and apply advices around this point-cut to identify the start, success and failure of the task. Cancellation would probably involve a separate method and require another point-cut and advice.

Locating the proper point-cuts for monitoring could be challenging in more complex systems, but nonetheless feasible if traceability to requirements exists. Moreover, previous studies [9] have shown that AspectJ scales to large systems.

C. ATM Adaptation

As mentioned in the previous subsection, the controller class implements the **`ITargetSystem`** interface and a reference to it is sent to the *Zanshin* server during registration. It therefore receives from the framework calls to the methods listed in Figure 6 whenever an adaptation strategy is being executed in response to an *AwReq* failure.

The controller handles adaptation using a separate thread, implemented in class **`AdaptationThread`**. When the singleton instance of the controller is initialized, it creates and runs an instance of this thread and both instances share a blocking queue, on which the thread waits for adaptation actions to process. Whenever one of the **`ITargetSystem`** methods is called, an object representing the called method is created and placed in this queue. Hence, *EvoReq* instructions are handled in a first-come, first-served basis. More sophisticated controllers could support prioritization, preferably calculated by the framework itself (future work).

Parts of the source code for the **`AdaptationThread`** class are shown in Figure 9. The method in line 13 is called for every item taken from the queue and identifies what is the adaptation action to be carried out. The figure shows the parts of the code corresponding to the printer malfunction scenario (cf. § IV-B).

The *Retry* strategy that is associated with this scenario contains, among others, the instructions *wait* (a few seconds) and *initiate* (the new copy of the requirement instance, so we can try again). These adaptation instructions are captured, respectively, in lines 16 and 21 of the method that processes queue items. The former calls the **`displayWaitDialog()`** method (line 30), whereas the latter delegates to **`retryPrinterReceipt()`** (line 43) if the *initiate* instruction refers, indeed, to the *Print* task.

To display a wait dialog, the adaptation thread creates an instance of a new UI **`Panel`** with a wait message and, using the reference to the main UI frame injected in the controller (as shown earlier in Figure 7), replaces the normal panel with the new one, sleeps for the time specified in the adaptation instruction and switches back to the normal UI afterwards.

Retrying the printer is a bit more elaborate and also involves a modification in the original ATM class **`Transaction`**, in package **`atm.transaction`**. This class has been modified to capture exceptions coming from a defective printer component and wait (i.e., sleep under the controller's monitor) for *Zanshin* to respond with an adaptation strategy (more precisely, the *Retry* strategy). The adaptation thread, therefore, notifies (wakes up) the main ATM thread (line 50), asking the customer using the ATM if she is ready to proceed and try printing the receipt again.

As Figure 9 shows, adaptations were written directly in Java code in the adaptation thread. More complex systems would require some kind of modularization. If architectural models are provided, one could even use special purpose adaptation languages, such as, for instance, Stitch [10].

```
1  package zanshin;
2
3  import atm.ATM;
4  import atm.physical.CustomerConsole.Cancelled;
5  import java.awt.Component;
6  import java.awt.Frame;
7  import java.util.Map;
8  import java.util.concurrent.BlockingQueue;
9
10 class AdaptationThread extends Thread {
11   /* ... */
12
13   private void processAdaptationAction(
         AdaptationAction action) {
14     switch (action.getInstruction()) {
15
16     case WAIT:
17       long waitTime = Long.parseLong(action.
           getParams()[1].toString());
18       displayWaitDialog(waitTime);
19       break;
20
21     case INITIATE:
22       String reqName = "" + action.getParams()[1];
23       if (AtmRequirement.T_PRINT_RECEIPT.matches(
           reqName)) retryPrintReceipt();
24       break;
25
26     /* Other cases... */
27     }
28   }
29
30   private void displayWaitDialog(long waitTime) {
31     Component guiPanel = frame.getComponent(0);
32     WaitPanel waitPanel = new WaitPanel(waitTime);
33     frame.remove(guiPanel);
34     frame.add(waitPanel);
35     frame.revalidate();
36
37     Thread.sleep(waitTime);
38     frame.remove(waitPanel);
39     frame.add(guiPanel);
40     frame.revalidate();
41   }
42
43   private void retryPrintReceipt() {
44     String question = "Ready to proceed?";
45     String menu[] = { "Yes", "No" };
46     int answer;
47     answer = atm.getCustomerConsole().
         readMenuChoice(question, menu);
48     if (answer == 0)
49       synchronized (controller) {
50         controller.notifyAll();
51       }
52   }
53
54   /* ... */
55 }
```

Fig. 9. Parts of the **`AdaptationThread`** class.

In the next section, we demonstrate the steps to execute this experiment, including the shortage of cash scenario of Section IV-A, showing the result of the adaptation actions that have just been described.

VI. Experimentation

With the models described in Section IV and the implemented ATM–*Zanshin* integration presented in Section V, we now execute the adaptation scenarios in order to see the *Zanshin* framework in action. In what follows, we present the results for the shortage of cash (cf. § IV-A) and printer malfunction (cf. § IV-B) scenarios, concluding with a brief discussion of evaluation results.

A. Shortage of Cash

Back in Figure 2, we can see that the main ATM user interface offers an ON/OFF button in the bottom-right corner

```
1  Processing method call: fail / GDetectCashAm
2  Processing state change: AR2 -> failed
3  (Session: S1) Created new session for AR2
4  (Session: S1) The problem has not yet been solved...
5  (Session: S1) Selected strategy:
       ReconfigurationStrategy
6  (Session: S1) Applying strategy
       ReconfigurationStrategy(qualia; class-level)...
7  Parameters chosen: [NOA]
8  Values to increment in the chosen parameters: [1.0]
9  Produced new config. with 1 changed parameter(s)
10 (Session: S1) AR2 has been evaluated to false
11 (Session: S1) Evaluating resolution: false
12 (Session: S1) The problem has not yet been solved...
13
14 Processing method call: start / GDetectCashAm
15 Processing method call: success / GDetectCashAm
16 Processing state change: AR2 -> succeeded
17 (Session: S1) Retrieved existing session for AR2,
       one event already in the timeline
18 (Session: S1) AR2 has been evaluated to true
19 (Session: S1) Evaluating resolution: true
20 (Session: S1) Problem has been solved. S1 ended.
```

Fig. 10. Excerpt from the *Zanshin* log for the shortage of cash scenario.

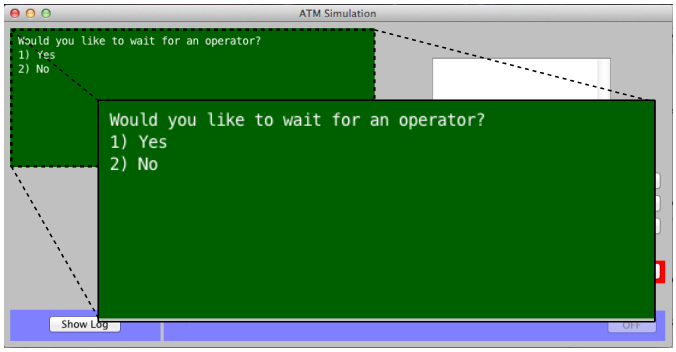

Fig. 11. Asking the customer if she would like to wait for an operator (zoomed in for better visualization).

of the window. When started, the ATM is initially off. When turned on, the system simulates the sensor that would be responsible for indicating how many bank notes there are in the cash dispenser by asking the user to inform the amount.

Later, if a withdraw operation is selected and the amount to withdraw is superior to the amount of bank notes informed when the ATM was tuned on, a failure of *AR2* is triggered in *Zanshin*, as can be seen in the excerpt from the log shown in Figure 10. As prescribed by the model, the framework uses reconfiguration as adaptation strategy, incrementing the *NOA* parameter in 1 unit.

Considering the ATM as a socio-technical system [11] composed of ATM hardware, software and bank employees, in a real deployment of this system the increase of *NOA* would trigger a notification to a human operator asking them to load a particular ATM dispenser with more cash. Instead, in the ATM simulation a question is presented to the customer, asking if she would like to wait for an operator, as shown in Figure 11. If the answer is yes and the customer tries the withdraw again, the dispenser will have the necessary amount of bills the second time and the operation will succeed (also shown in the log of Figure 10).

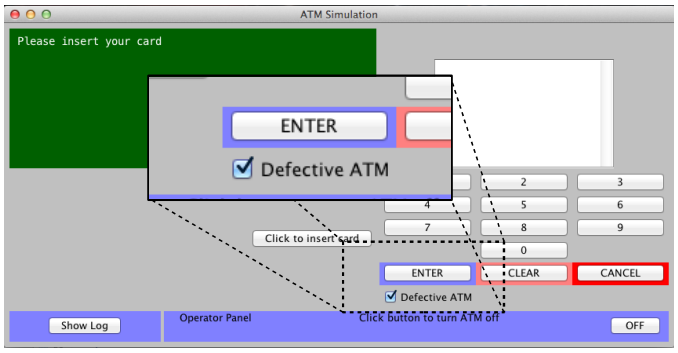

Fig. 12. A checkbox allows the simulation of a defective ATM (zoomed in for better visualization).

B. Printer Malfunction

To simulate malfunctions in the ATM's printer, the original implementation is modified to present a checkbox under the keypad, as displayed in Figure 12. When checked, the ATM behaves in a defective manner and, for this particular scenario, throws exceptions whenever its printer is requested to print a receipt.

With *defective mode* on, any print request (which happens automatically after any transaction) will trigger the monitoring–adaptation loop of the *Print* task via *AwReq AR3*, presented as illustration of the the ATM controller's implementation in Section V.

In particular, Section V-C showed that the *Retry* strategy associated with *AR3* (cf. § IV-B) includes, among others, the instructions *wait* and *initiate*. When received by the controller, the *wait* instruction results in an informational message appearing for 5 seconds.

Right afterwards, the *initiate* instruction makes the controller ask the customer if she is ready to proceed, giving her the opportunity to turn *defective mode* off before the ATM requests the receipt to the printer one more time. If the system is still defective, the *Retry* strategy will present the same behavior the second time, but since its applicability condition limits its use to at most twice per session, the third consecutive error will show a different informational message and abort.

C. Discussion

The results presented earlier suggest that *Zanshin* can be applied to external systems, adding adaptation features to them through a feedback loop based on requirements models. The presented scenarios were very simple, but already give an indication of the framework's potential in real-life situations. More complex adaptation requirements can be seen in the complete model for the ATM [6] or the models for the Meeting Scheduler and the Ambulance Dispatch System presented in [3].

Our experience so far with the application of *Zanshin* to design different adaptive systems is that the framework presents appropriate concepts for the elicitation of the requirements for adaptation: *AwReqs* define explicitly the states of requirements of which stakeholders would like to be aware (monitoring),

whereas *EvoReqs* prescribe what should be done in response to some of these situations (adaptation).

Given that feedback loops are in the centerpiece of any adaptation approach, making them first-class citizens in Requirements Engineering can facilitate the job of system analysts when adaptation is an important aspect of the system-to-be. The feedback loop concepts present in the modeling language help analysts and stakeholders to think and communicate in terms of 'which failures should we be aware of?' and 'what should we do in case they happen?'

Indeed, being based on feedback loops and having its generic functionality already implemented eased the process of designing and developing the ATM's adaptation scenarios. Instead of implementing each scenario from scratch, it was enough to provide a declarative model of the adaptation requirements and integrate the ATM with the *Zanshin* framework through aspects (monitoring) and a blocking queue (adaptation).

The most challenging step of the process was finding the proper point-cuts for monitoring and tweaking the original implementation to support the adaptation actions. Our experiment, however, didn't include elicitation of requirements from stakeholders, which is traditionally a very challenging task, nor reverse-engineering of the requirements model from an existing software, which is not a trivial job either. Having the goal model provided allowed us to jump to later stages of the process and focus on the application of our framework to the problem at hand.

There are, however, some limitations of the proposal and threats to validity of this evaluation. First, evaluation involving practitioners outside our research group have not been conducted. Moreover, the framework offers little tool support and is currently a prototype under development. Also, although scalability tests of *Zanshin* have been presented before (cf. [3]), one that involves a base system as the ATM would be necessary to evaluate the performance of the Remote API.

Nonetheless, the fact that *Zanshin* was successfully integrated with an existing application, using an amount of effort proportional to its size, and that it effectively augmented such application with adaptation features can be considered an interesting contribution to the maturity of our proposal.

VII. Conclusions

In this paper, we reported on the experience of using *Zanshin* for the design of adaptive systems to an existing application: a software that simulates an ATM. Models of the system's adaptation requirements were produced, the managed system was integrated with the adaptation framework and a few scenarios of adaptation were executed to test their occurrence in practice. Results indicate that the approach and framework have the potential to be applied in industrial settings, given proper development of the prototype framework and tool support.

Further evaluation efforts should concentrate on measuring the ease of use of the approach by practitioners, on the applicability of the framework in more complex scenarios and on comparing *Zanshin* to other proposals for the design of adaptive systems. Efforts regarding this last goal have already begun in our research group.

Acknowledgment

This work has been supported by the ERC advanced grant 267856 "Lucretius: Foundations for Software Evolution" (April 2011 – March 2016, http://www.lucretius.eu) as well as Brazilian foundation FAPES (http://www.fapes.es.gov.br) through the PRONEX grant #52272362.

References

[1] B. H. C. Cheng *et al.*, "Software Engineering for Self-Adaptive Systems: A Research Roadmap," in *Software Engineering for Self-Adaptive Systems*, ser. Lecture Notes in Computer Science, B. H. C. Cheng, R. de Lemos, H. Giese, P. Inverardi, and J. Magee, Eds. Springer, 2009, vol. 5525, pp. 1–26.

[2] Y. Brun *et al.*, "Engineering Self-Adaptive Systems through Feedback Loops," in *Software Engineering for Self-Adaptive Systems*, ser. Lecture Notes in Computer Science, B. H. C. Cheng, R. de Lemos, H. Giese, P. Inverardi, and J. Magee, Eds. Springer, 2009, vol. 5525, pp. 48–70.

[3] V. E. S. Souza, "Requirements-based Software System Adaptation," PhD Thesis, University of Trento, Italy, 2012.

[4] Y. Wang and J. Mylopoulos, "Self-Repair through Reconfiguration: A Requirements Engineering Approach," in *Proc. of the 2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2009, pp. 257–268.

[5] A. van Lamsweerde, "Goal-Oriented Requirements Engineering: A Guided Tour," in *Proc. of the 5$^{\text{th}}$ IEEE International Symposium on Requirements Engineering*. IEEE, 2001, pp. 249–262.

[6] G. Tallabaci, "System Identification for the ATM System," Master Thesis, University of Trento, Italy, 2012.

[7] I. Jureta, J. Mylopoulos, and S. Faulkner, "Revisiting the Core Ontology and Problem in Requirements Engineering," in *Proc. of the 16$^{\text{th}}$ IEEE International Requirements Engineering Conference*. IEEE, 2008, pp. 71–80.

[8] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *ECOOP'97 — Object-Oriented Programming*, ser. Lecture Notes in Computer Science, M. Akşit and S. Matsuoka, Eds. Springer, 1997, vol. 1241, pp. 220–242.

[9] A. Singh, "The Scalability of AspectJ," Masters Thesis, University of British Columbia, Canada, 2007.

[10] S.-W. Cheng and D. Garlan, "Stitch: A language for architecture-based self-adaptation," *Journal of Systems and Software*, vol. 85, no. 12, pp. 2860–2875, 2012.

[11] V. Bryl, "Supporting the Design of Socio-Technical Systems by Exploring and Evaluating Design Alternatives," PhD Thesis, University of Trento, 2009.