Testing the Reconfiguration of Adaptive Systems

Kai Nehring, Peter Liggesmeyer
AG Software Engineering: Dependability
University of Kaiserslautern
Kaiserslautern, Germany
Email: nehring@cs.uni-kl.de, liggesmeyer@cs.uni-kl.de

***Abstract*—Adaptive systems can change their internal structure in order to respond to changes in their environment. These changes can cause malfunctions if not applied correctly. Current test approaches do not cover every aspect of the reconfiguration sufficiently. In this paper, we present a novel approach for testing the reconfiguration process of adaptive systems with respect to structural changes. The approach presents testers with guidelines for choosing the appropriate test strategy for a certain aspect, such as order of reconfiguration, state transfer, and transaction handling, of the reconfiguration procedure.**

***Keywords*-adaptive system; testing; reconfiguration; test process model; structural changes**

I. Introduction

Dynamically adaptive systems are a promising alternative to static systems when a system must modify its structure or behaviour due to changes in its executional context. Testing such systems, however, can be a challenging task since their structure and even their functionality may change at runtime — a test approach would have to take this into account. Current test- and verification approaches focus on the functionality and execution environment[4], the adaption policy[5][7], and automated evaluation of structural changes[3], but not on the reconfiguration itself on the executable system. Furthermore, some test approaches require complex (formal) models of the system in order to be applicable[6].

We have already illustrated how the visualisation and inspection of structural changes in adaptive systems can help to detect potential defects, and, at the same time, omit formal models[1]. However, other aspects, such as the state transfer from an object to its replacement, which are often interwoven with structural changes remained uncovered, too. We propose a novel approach to test the reconfiguration process of adaptive systems with respect to structural changes. The approach is designed as an additional test activity, hence it is a supplement and not a replacement for the aforementioned approaches.

In Section II, we propose a process model to test the reconfiguration procedure. Section III summarises our experience when we applied the test process on adaptive systems. Section IV completes with a conclusion and an overview of the future work.

II. Test Process Model

We propose a self-contained iterative test process model that addresses issues that may arise when structural changes are involved during the reconfiguration, such as replacing the instance of component X with an instance of component Y. The test process model is comprised of 6 iterations, each of which focuses on a specific aspect of the reconfiguration, such as the state transfer between a component and its replacement. Furthermore, the process model can be tailored to fit the system under test, i.e., iterations can be omitted if they focus on an aspect that is not supported in the system under test. Each iteration is further divided into three phases:

1) **Preparation**, in which workload, instrumentation probes, etc. will be prepared
2) **Execution**, in which the workload will be executed in order to collect data about the system's behaviour
3) **Evaluation**, in which the obtained runtime data is evaluated

Breaking down an iteration into phases not only allows the reuse of information from other iterations, but also to split and dispatch the procedure to multiple roles. The **Domain expert** provides the workload (i.e., the input data) that is executed at runtime. The expert also evaluates the result of the processing and determines whether the system processes the data correctly. **Developer** and **System Manager** execute the workload on the instrumented system. The **Architect** evaluates the structural changes.

The separation into phases has been omitted for simplicity reasons in the following outlines of the iterations.

A. Iteration 1 — System Overview

Understanding the internal structure of a system is essential to estimate effects of changes in its structure. Design documents are often a valuable source of knowledge, but they are, however, not always very reliable for several reasons:

- Development and maintenance can cause the system's structure to change over time. Changes in the source code are not always reflected in the design documents.
- The system has not been implemented as specified.
- etc.

Goal of Iteration 1 is to create a (virtually) complete runtime model of the system, which can later be used to select the actual components for further investigation. However, the iteration can be omitted if detailed and reliable knowledge of the system's runtime composition is available.

Assignments to attributes, which hold references to other objects and their values, must be tracked in order to create a runtime model. Since design documents can be outdated, inconsistent, or incomplete, the attributes are best collected from the source code. Instrumentation probes must be created for each attribute to track changes. Furthermore, a representative workload must be prepared in order to execute the system's functionality in different states/configurations, and to cause reconfigurations. An external trigger must be prepared if the reconfiguration is not induced by the workload itself.

The workload must be executed on the instrumented system. Depending on the approach that is used to implement the functionality, the workload might be required to run twice — before and after the reconfiguration — to track all utilised object–instances. Particularly components which utilise lazy loading would be incomplete otherwise.

Analysis of the collected information will not only unveil the object/component composition at runtime but also the order of changes during the reconfiguration. However, the vast amount of information may be overwhelming and may make further analysis more difficult since potentially lots of objects are displayed although they are not linked to the reconfiguration. Furthermore, the runtime behaviour of the system might be altered due to the instrumentation overhead, which may cause the system to change the reconfiguration strategy[2].

B. Iteration 2 — Structural Changes

The order of instructions required to perform a reconfiguration is usually flexible to some degree. Although all considered execution paths eventually lead to a valid composition or configuration, quality-of-service and system integrity might be affected by a particular strategy. A reconfiguration strategy which focuses on system integrity might passivate all components before changes are performed — this might result in a reduced quality-of-service since the system is either unavailable or operates in a gracefully degraded mode (for a longer period of time). A quality-of-service based approach might try to minimise the downtime by performing as many steps as possible parallel to normal operation, which may increase the risk of inconsistencies in the system's data. The structural changes in the course of the reconfiguration are evaluated in Iteration 2.

Typically, only few components are affected by a reconfiguration. Tracing changes in such components can help in breaking up the complex system into partitions. The result of Iteration 1 can be used to eliminate unnecessary components, which results in a smaller set of probes to instrument the system. A component can be considered *unnecessary*, if it is neither involved in, nor affected by the reconfiguration.

The workload must fulfil the same requirements as in Iteration 1, and it can even be reused if Iteration 1 has been executed. It must then be executed on the instrumented system.

Analysis of the trace is best done using a graphical representation, such as a series of object diagrams[1]. In such an approach, the trace will be transformed into a series of object diagrams, at which each state, caused by a change in the structure, is expressed by a new object diagram. Developers and architects evaluate each diagram and decide whether the system passed through an illegal state. Unlike other methods, such as AMOEBA-RT[3], this approach does not require a formal model, such as a temporal logic model, to describe the system states.

C. Iteration 3 — Data Integrity

The system is in a transitional state during the reconfiguration. It can operate with either reduced or mixed functionality, be non-functional at all, or, in the worst case, in an inconsistent operational state. The approach used to achieve adaptability not only has a great influence on the observable behaviour, but also on system integrity and data integrity. Iteration 3 tests the data integrity in presence of a (increasing) load, such as incoming user requests.

The type of workload depends on the strategy used to achieve adaptability, and is distinguished between step load and ramp load. While *ramp load* slowly increases the workload on the system, *step load* suddenly puts a lot of pressure on the system, displayed in Figure 1. The system's behaviour on step load can point to potential defects, if the system implements a load-based adaptation strategy[2]. Furthermore, if the system is designed to buffer user requests while it is reconfigured, then a step load-like increase of requests could cause the buffer to run out of space before the reconfiguration has been completed. To test the behaviour of the system under these conditions, the load must be sized accordingly.

The system should not show unexpected behaviour, such as crashes, or lost requests, that can be traced back to the reconfiguration. Furthermore, it is advisable to verify that all structural changes during the reconfiguration have been applied correctly in order to preclude the possibility that a faulty trigger prevented the reconfiguration. If a queue (or, in more general, a "buffer") is used to buffer user requests, additional tests are required to ensure that it satisfies the following requirements:

- The queue/buffer must either be properly sized or be resizable in order to prevent the loss of user requests according to the quality requirements.
- The order of requests must be preserved, if not otherwise specified.

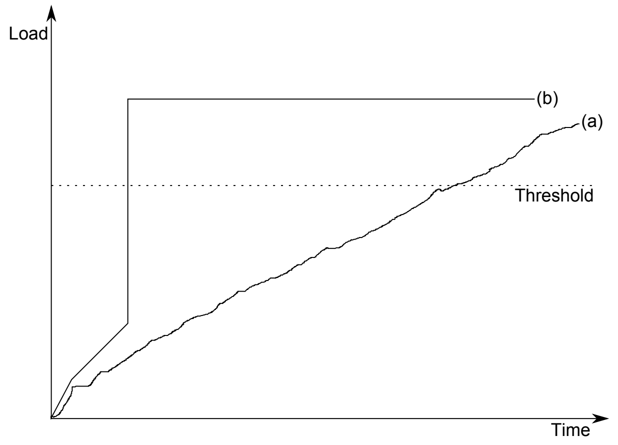

Figure 1. Ramp load vs. step load: ramp load (a) changes continuously whereas step load (b) changes suddenly. Reconfiguration of the system occurs if the load exceeds a predefined threshold.

- All buffered requests must be processed after the reconfiguration has been completed, or actions have to be taken if the reconfiguration failed. The order of the execution must be equal to the order in which the requests were buffered, if not otherwise specified.
- New requests must be buffered until all previously received requests have been executed, if not otherwise specified.

D. Iteration 4 — State Transfer

A stateful component may be instructed to transfer its internal state to the replacement if it is about to be replaced. Iteration 4 checks whether the state of a component will be transferred correctly to its replacement, and, if necessary, type conversion is done in a way so that the replacement is fully operational. Furthermore, potential access violations due to the reconfiguration on concurrent systems can be checked.

A set of instrumentation probes should be prepared in order to monitor the runtime composition. If a component X is about to be replaced with Y, and both X and Y are instances of the same type, then the instrumentation facility must be able to distinguish them, e.g., by recording their memory addresses.

Furthermore, two workloads must be prepared. The first workload is applied on the system before the component is about to be replaced — the preload phase. It is responsible to set up the state of the instance of component X that must then be transferred to the replacement Y. The second workload must be executed either during or after the reconfiguration, depending on whether the system utilises concurrency.

On a purely single threaded system, the second workload is executed solely to verify that the new instance Y is fully operational and (optional required) conversion of the internal state has been successfully carried out.

On a multi threaded system, the second workload must be executed while the reconfiguration is running to test the following situations:

- The system must not alter the state of instance X once the state transfer has begun, i.e., user requests have to be stored in a buffer, if not otherwise specified.
- The system must be fully operational after the reconfiguration, i.e., the state transfer has been successfully carried out.

After the state transfer (of instance X) has been performed and workload 2 has been executed, the replacement (instance Y) should be comprised of the required data of instance X and of the new data. Depending on the type of system, this can manifest in the following situations:

- All data, e.g., items in a shopping trolley, have been added to the replacement.
- All data, e.g., GPS coordinates of a route, have been added to the replacement and the order has been preserved. In addition, all new data have been appended to the previous ones.
- A new state, e.g., a new random number, has been calculated correctly using the previous state of instance X.

E. Iteration 5 — Transaction Handling

The reconfiguration of a system can impact transaction capable components either directly, if the component is target of a reconfiguration, or indirectly, if the transaction capable component utilises a component that is part of the reconfiguration or vice versa. Iteration 5 checks the transaction handling during a reconfiguration.

The system specification should provide information about the observable behaviour when a reconfiguration is triggered while a transaction is running. It can most likely be narrowed down to the following two situations:

1) The reconfiguration must be delayed until the transaction has been finished. A finished transaction can be either successful or unsuccessful in which case rollback has to be performed.
2) The transaction must be aborted and a rollback must be performed to undo changes. Nevertheless, the reconfiguration must be delayed until the rollback has been completed to ensure data integrity.

The progress of a transaction is, however, of no interest, i.e., a component X must not be passivated (e.g., in order to replace it) even if its job is done. In case of a rollback, which can still occur if the last operation in the transaction fails, the component X might be needed again.

A set of instrumentation probes should be prepared in order to monitor the component composition. Also a workload must be prepared to utilise the system. Furthermore, the reconfiguration must be triggered while a transaction is running.

During the execution, a snapshot of the datasource that is about to be altered should be made to simplify the integrity check after the reconfiguration has been performed. The reconfiguration must not only be triggered while the workload is executing but also while a transaction is running.

The evaluation of the reconfiguration encompasses several steps. First, the system composition must be checked — the system must be in an operational state. This includes that no crashes or deadlocks occurred at runtime due to an incomplete system composition (e.g., crashes due to null pointer exceptions) or passivated components.

Depending on the specification of the system, the transaction must be either be rolled back or the reconfiguration must be delayed until the transaction has been finished. The latter includes a rollback, i.e., the system must not change its composition while a transaction is running.

In the last step, the data integrity must be evaluated. The data must either be unaltered (in case of a rollback) or fully updated.

F. Iteration 6 — Identity

It is in some cases important to know whether one or more components use the same instance of a component X or instances of the same type T, where X is an implementation or a subtype of T. Iteration 6 focuses on the identity of the instances.

A typical workload must be executed on an instrumented system. The instrumentation facility must be able to distinguish multiple instances of the same type. Common practice is to record the memory address of each instance. Developers then compare the utilised objects and determine whether the usage scenario is acceptable.

III. Evaluation

We have evaluated the test approach on three systems so far:

1) adaptive Tic-Tac-Toe: a version of the well known game Noughts and Crosses, which automatically adjusts the game level of the computer player in relation to the human player's skills
2) an adaptive ERP system, which utilises a local cache if the connection to the off-site master ERP system is faulty. It automatically synchronises and utilises the off-site ERP as soon as it becomes available again
3) an Emergency Detection System, which can be used to monitor humans in an ambient assisted living scenario. The system utilises a variety of sensors, such as pulse sensors, fall sensors, etc. A new sensor will be utilised after it becomes available to the system and if the quality-of-service level can be improved by the new sensor

A. adaptive Tic-Tac-Toe

The only reconfigurable component in the program is the computer player, whose playing strategy (Easy, Medium, and Advanced) can be adjusted after each match according to the following predefined rules:

1) The game level shall be increased to the next higher level if the computer loses two consecutive matches. The game level remains unaltered if the highest game level has already been reached.
2) The game level shall be decreased to the next lower level if either
    a) two consecutive matches end with a draw, or
    b) the computer player wins two consecutive matches.

    The game level remains unaltered if the lowest game level has already been reached.

Applicable iterations are:

- Iteration 2 to create an overview of the reconfiguration process and to track structural changes
- Iteration 6 for an overall view of the used instances

The structure of the game is rather simple since only the play strategy can be varied. In the course of tailoring, the remaining iterations have been removed for the following reasons:

- The structure of the computer player is simple; an complete overview is unnecessary
- The play strategy cannot be replaced while a match is running
- No state transfer among the game strategies is supported
- The game does not utilise transactions

Furthermore, the tracing of the program would deliver identical results for Iteration 2 and Iteration 6, which is why we reused the tracing results of Iteration 2 for further analysis in Iteration 6.

The evaluation of the trace unveiled abnormal behaviour whenever the human player repeatedly won matches in the game level *Advanced*. The computer player reconfigured itself even though it already utilised the *Advanced* strategy level, which contradicts the requirements. Also, we could observe a strange behaviour where the *Advanced* level was replaced with another instance of the same class, which does not change the behaviour of the computer player. Further examinations located the defect in the method which increases the game level. The method did not identify the level *Advanced* correctly. Traditional testing did not uncover the defect since *Advanced* is the highest level in the current version of the game. Furthermore, the game did not expose unusual behaviour to justify further investigations. However, the defect would have caused stagnancy in the level *Advanced* once further levels would have been added to the game.

B. ERP System

The class diagram in Figure 2 shows a simplified overview of the system. The `CashRegisterController` comprises a reference to an implementation of the `ERP` interface. This reference is subject to change if the master ERP is replaced with the cache, and vice versa.

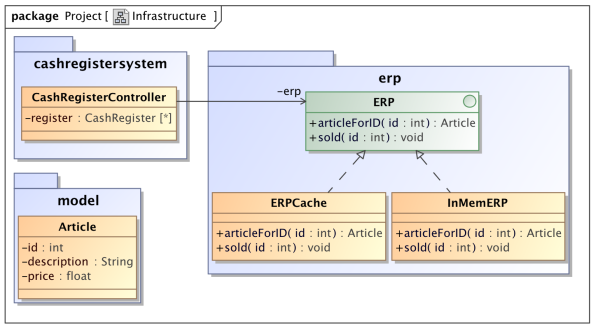

Figure 2. ERP class diagram (Key: UML class diagram)

A full system trace in not required since the reconfigurable portion of the system is rather small. The current version does not allow a cash register system to operate while the system is being reconfigured, which is why Iteration 3 is not applicable. Neither is Iteration 5 since the system does not utilise transactional components beyond the database management system. Besides the reconfiguration process itself (Iteration 2) and the state transfer test (Iteration 4), the component identity (Iteration 6) is subject to test to ensure that the same master ERP system is used, i.e, the same server address and identical login credentials are used.

Two sets of data were prepared in order to test the state transfer. The first set was used to preload the system with information, which were stored in the master ERP's database. The second data set was used to populate the ERPCache after the reconfiguration "master ERP to ERPCache". That data was transferred to the master ERP once it became available again, which resulted in a combined set of data.

Two snapshots were created to gather the internal state of the master ERP — one before the first reconfiguration and one after the final reconfiguration. The analysis of the ERP-data did not unveil deviations from the expected data, i.e., the state transfer has been implemented correctly.

The tracing results of Iteration 2 were reused to check whether the master ERP was the same before the first reconfiguration (transition to the ERPCache) and after the second reconfiguration (transition back to the master ERP). The hash codes of the utilised master ERP instances were equal in both cases, i.e., the same ERP system was used.

C. Emergency Detection System

The Emergency Detection System (EDS) is a monitoring system, applicable for example in an ambient assisted living scenario. It constantly evaluates informations which it acquires from several sources, such as blood pressure sensors, pulse sensors, and location sensors. If it detects a critical situation, it can execute a variety of protocols, e.g., notify emergency medical staff. A critical situation can be caused, for example, by a sudden change in vital signs, or by a fall of the monitored person. The system automatically selects a sensor configuration to offer the highest possible service quality. Furthermore, it reconfigures itself to utilise newly added sensors without service interruption.

If a new sensor is registered at the `EDSManager`, which is a centralised administration component to keep track of all available sensors, its contribution to the system is analysed. The system will be reconfigured to utilise the new sensor if the new sensor is considered *valuable*. A sensor is rated valuable if the EDS-evaluation-algorithm can create a sensor configuration that results in a higher quality-of-service, either by adding new kind of sensor, which was not available before (e.g., a fall sensor), or by replacing an existing sensor with a higher grade sensor (e.g., higher Safety Integrity Level (SIL)). The sensors are connected to a an implementation of the `IEDSHandler`-interface. Each supported configuration is represented by its own implementation, i.e., an implementation that supports only a pulse sensor and a pressure sensor can be distinguished from an implementation that supports pulse sensor, pressure sensor, and a location sensor. Each `IEDSHandler`-implementation processes the sensor-signals and sets off the alarm if a critical situation is detected. This design offers flexibility since a new handler can be set up in the background. If it is fully constructed, the new handler replaces the old one without service interruption.

According to the system description, sensor signals are not processed while the handler is replaced, and historic data is not transferred either. Hence, Iteration 3 and Iteration 4 are not applicable. The system does not support transaction, which is why Iteration 5 is not applicable, too. The component identity (Iteration 6) is of interest to identify which sensors are utilised at a particular point in time.

The reconfiguration test had been divided into three stages:

1) The initial system configuration comprises a blood pressure sensor (SIL 2) and a pulse sensor (SIL 2)
2) A location sensor (SIL 1) is added — the system should integrate the location sensor.
3) A new pulse sensor (SIL 3) is added — the system should replace the previously used SIL 2 pulse sensor with the SIL 3 pulse sensor.

Starting from the initial configuration, the location sensor was added. The system incorporated the location sensor without service interruption. Then the new pulse sensor was added. The evaluation of the object diagrams showed that a new handler was created, which utilised the pressure sensor, the location sensor, and the new (SIL 3) pulse sensor.

The system went through a total of 33 states from startup to the final configuration. Analysis of the object diagrams showed that the integrity was not compromised at any time.

IV. Conclusion and Future Work

Structural changes may not be the only concern in an adaptive system when a reconfiguration is performed. In order to test state transfer, transaction handling, etc., a more rigour testing strategy is necessary. In this paper, we presented an approach to test the reconfiguration procedure of adaptive systems with respect to structural changes having regard to the special properties, such as state transfer. The iterative process model can be tailored to fit the system under test. Furthermore, the process model is designed to be an additional test activity, not a replacement for other processes and therefore focuses on the reconfiguration only, i.e., component test, etc. remain unaffected.

There are extensions to this work, which have not been discussed yet. Quality requirements, such as maximum tolerable reconfiguration time, have not been included to the test process model, which is to be addressed in future work.

References

[1] K. Nehring and P. Liggesmeyer, "Tracing structural changes of adaptive systems," in *ADAPTIVE 2010: The Second International Conference on Adaptive and Self-Adaptive Systems and Applications*, 2010, pp. 142–145.

[2] S.-W. Cheng, D. Garlan, and B. Schmerl, "Architecture-based self-adaptation in the presence of multiple objectives," in *Proceedings of the 2006 international workshop on Self-adaptation and self-managing systems*, SEAMS '06, 2006, pp. 2–8.

[3] H. J. Goldsby, B. H. Cheng, and J. Zhang, "AMOEBA-RT: Run-Time Verification of Adaptive Software," 2008, pp. 212–224.

[4] Component+ Partners, "Built-in testing for component-based development," in *EC IST 5th Framework Project IST-1999-20162 Component+*, Technical Report D3, 2001.

[5] F. Munoz and B. Baudry, "Artificial table testing dynamically adaptive systems," *CoRR*, abs/0903.0914, 2009.

[6] J. Zhang and B. H. C. Cheng, "Using temporal logic to specify adaptive program semantics," in *Journal of Systems and Software*, Volume 79(10), 2006, pp. 1361–1369.

[7] J. Zhang, H. J. Goldsby, and B. H. Cheng, "Modular verification of dynamically adaptive systems," in *AOSD '09: Proceedings of the 8th ACM international conference on Aspect-oriented software development*, 2009, pp. 161–172.