An Approach for Automated Scenario-based Testing of Distributed and Heterogeneous Systems

Bruno Lima[1,2] and João Pascoal Faria[1,2]
[1] *Faculty of Engineering, University of Porto, Porto, Portugal*
[2] *INESC TEC, Porto, Portugal*

Keywords: Software testing, Distributed systems, Scenario-based Testing, Heterogeneous Systems, Systems of Systems.

Abstract: The growing dependence of our society on increasingly complex software systems, makes software testing ever more important and challenging. In many domains, such as healthcare and transportation, several independent systems, forming a heterogeneous and distributed system of systems, are involved in the provisioning of end-to-end services to users. However, existing testing techniques, namely in the model-based testing field, provide little tool support for properly testing such systems. Hence, in this paper, we propose an approach and a toolset architecture for automating the testing of end-to-end services in distributed and heterogeneous systems. The tester interacts with a visual modeling frontend to describe key behavioral scenarios, invoke test generation and execution, and visualize test results and coverage information back in the model. The visual modeling notation is converted to a formal notation amenable for runtime interpretation in the backend. A distributed test monitoring and control infrastructure is responsible for interacting with the components of the system under test, as test driver, monitor and stub. At the core of the toolset, a test execution engine coordinates test execution and checks the conformance of the observed execution trace with the expectations derived from the visual model. A real world example from the Ambient Assisted Living domain is presented to illustrate the approach.

1 INTRODUCTION

Due to the increasing ubiquity, complexity, criticality and need for assurance of software based systems (Boehm, 2011), testing is a fundamental lifecycle activity, with a huge economic impact if not performed adequately (Tassey, 2002). Such trends, combined with the needs for shorter delivery times and reduced costs, demand for the continuous improvement of software testing methods and tools, in order to make testing activities more effective and efficient.

Nowadays software is not more like simple applications but has evolved to large and complex system of systems (DoD, 2008). A system of systems consists of a set of small independent systems that together form a new system. The system of systems can be a combination of hardware components (sensors, actuators, etc.) and software systems used to create big systems or ecosystems that can offer multiple different services. Currently, systems of systems capture a great interest from the software engineering research community.

Testing these distributed and heterogeneous software systems or systems of systems, running over interconnected mobile and cloud based platforms, is particularly important and challenging. Some of the challenges are: the difficulty to test the system as a whole due to the number and diversity of individual components; the difficulty to coordinate and synchronize the test participants and interactions, due to the distributed nature of the system; the difficulty to test the components individually, because of the dependencies on other components.

An example of a distributed and heterogeneous system is the Ambient Assisted Living (AAL) ecosystem that was prototyped in the context of the nationwide AAL4ALL project (AAL4ALL, 2015). The AAL4ALL ecosystem comprises a set of interoperable AAL products and services (sensors, actuators, mobile and web based applications and services, middleware components, etc.), produced by different manufacturers using different technologies and communication protocols (web services, message queues, etc.). To assure interoperability and the integrity of the ecosystem, it was developed and piloted a testing and certification methodology (Faria et al., 2014), en-

Lima B. and Faria J..
An Approach for Automated Scenario-based Testing of Distributed and Heterogeneous Systems.
DOI: 10.5220/0005558602410250
In *Proceedings of the 10th International Conference on Software Engineering and Applications* (ICSOFT-EA-2015), pages 241-250
ISBN: 978-989-758-114-4

compassing the specification of 'standard' interfaces and component categories, the specification of unit (component) and integration test scenarios, and the test implementation and execution on candidate components by independent test labs. A major problem faced during test implementation and execution was related with test automation, due to the diversity of component types and communication interfaces, the distributed nature of the system, and the lack of support tools. Similar difficulties have been reported in other domains, such as the railway domain (Torens and Ebrecht, 2010). In fact, we found in the literature limited tool support for automating the whole process of specification-based testing of distributed and heterogeneous systems.

Hence, the main objective of this paper is to propose an approach and a toolset architecture to automate the whole process of model-based testing of distributed and heterogeneous systems in a seamless way, with a focus on integration testing, but supporting also unit (component) and system testing. As compared to existing approaches, the proposed approach and architecture provide significant benefits regarding efficiency and effectiveness: the only manual activity required from the tester is the creation (with tool support) of partial behavioral models of the system under test (SUT), using feature-rich industry standard notations (UML 2 sequence diagrams), together with model-to-implementation mapping information, being all the needed runtime test components provided by the toolset for different platforms and technologies; the ability to test not only the interactions of the SUT with the environment, but also the interactions among components of the SUT, following an adaptive test generation and execution strategy, to improve fault detection and localization and cope with non-determinism in the specification or the SUT.

The rest of the paper is organized as follows: Section 2 describes the state of the art. Section 3 presents an overview of the proposed approach and test process. Section 4 introduces the toolset architecture. Section 5 concludes the paper and points out future work. A running example from the AAL domain is used to illustrate the approach presented.

2 STATE OF THE ART

We analyze in this section work in areas most closely related with specification-based (model-based) testing of distributed and heterogeneous systems.

2.1 Model-based Testing

Model based testing (MBT) techniques and tools have attracted increasing interest from academia and industry (Utting and Legeard, 2007), because of their potential to increase the effectiveness and efficiency of the test process, by means of the automatic generation of test cases (test sequences, input test data, and expected outputs) from behavioral models of the system under test (SUT). However, MBT approaches found in the literature suffer from several limitations (Dias Neto et al., 2007). The most common limitation is the lack of integrated support for the whole test process. This is a big obstacle for the adoption of these approaches by industry, because of the effort required to create or adapt tools to implement some parts of the test process. Other common problems with existing MBT approaches are the difficulty to avoid the explosion of the number of test cases generated (in the first stages of the test automation process) and the difficulty to bridge the gap between the model and the implementation (in the last stages of test automation process, namely in the conversion of abstract test cases to concrete test cases). In recent MBT approaches (Moreira and Paiva, 2014; Faria and Paiva, 2014), researchers try to overcome the first problem (test case explosion) by the usage of behavioral models focusing on specific scenarios or patterns, and the second problem (test case concretization) by providing test concretization and execution mechanisms that require simple mapping information from the user.

2.2 Model-based Testing Approaches using UML Sequence Diagrams

Being a feature-rich industry standard, UML 2 sequence diagrams (SD) are particularly well suited for supporting scenario-based MBT approaches. With the features introduced in UML 2, parameterized sequence diagrams (SD) can be used to model both simple and complex behavioral scenarios, with control flow variants, temporal constraints, and conformance control operators. Although some works exist to derive test scenarios (partial behavioral specifications) from state machine based behavioral models (full behavioral specifications), the construction of partial behavioral specifications (SDs or natural language counterparts) seems more accessible for industrial adoption than the construction of full behavioral specifications. For that reason, we privilege the usage of UML SDs (irrespective of whether they are created from scratch by the user or generated automatically from other behavioral models).

In the literature it can be found some test automation approaches based on UML SDs, but those approaches fall short for the testing of distributed and heterogeneous systems.

Of particular relevance in the context of this paper is the UML Checker toolset developed in recent work of the authors (Faria, 2014; Faria and Paiva, 2014), with several advantages over other approaches, namely regarding the level of support of UML 2 features. The toolset supports the conformance testing of standalone object-oriented applications against test scenarios specified by means of so called test-ready SDs. Test-ready SDs are first translated to a form of extended Petri Nets (Event-Driven Colored Petri Nets) for efficient incremental conformance checking, with a limited support for parallelism and concurrency. Besides external interactions with users and client applications, internal interactions between objects in the system are also monitored using Aspect-Oriented Programming (AOP) techniques (Kiczales et al., 1997), and checked against the ones specified in the model. The testing of distributed systems is not supported, but some of the techniques developed have the potential to be reused for the testing of distributed and heterogeneous systems, where, instead of modeling and testing interactions between objects in a standalone application, one is interested in modeling and testing interactions between components in a distributed system.

Other examples of test automation approaches based on UML SDs are the SCENTOR tool, targeting e-business EJB applications (Wittevrongel and Maurer, 2001), the MDA-based approach of (Javed et al., 2007), and the IBM Rational Rhapsody TestConductor AddOn (IBM, 2013), targeting real time embedded applications. A comparison of the strengths and weaknesses of these approaches can be found in (Faria and Paiva, 2014).

2.3 Test Automation for Distributed Systems

Although we didn't find in the literature MBT approaches supporting in an integrated fashion the whole test automation process for distributed systems, we found several works supporting parts of the process, that can help in the construction of an integrated approach and toolset. The most relevant ones are mentioned next.

An additional difficulty in applying MBT techniques for distributed systems is that their distributed nature imposes theoretical limitations on the conformance faults that can be detected by the test components, depending on the test architecture used (Hierons et al., 2011; Hierons, 2014); finding a test architecture that simultaneously maximizes the fault detection capability and minimizes the overhead and delays caused by test coordination is still an open problem. Existent MBT approaches for distributed systems also lack the support for: internal interaction monitoring (between the SUT components), to improve fault detection and localization; adaptive (online) test generation strategies, to cope with non-determinism in the specification or the SUT (Hierons, 2014); feature-rich industry standard notations such as UML SDs.

Regarding test concretization and execution for distributed and heterogeneous systems, we found in the literature several reference architectures and frameworks that can be adapted for building a fully integrated test automation solution: test architectures for testing distributed systems proposed by (Ulrich and König, 1999); the STAF software testing automation framework, that can be used for coordinating distributed test components running on multiple platforms (STAF, 2014); the RemoteTest framework for testing distributed systems, which design was proposed in (Torens and Ebrecht, 2010); the FiLM runtime monitoring tool for distributed systems (Zhang et al., 2009); the DiCE approach for continuously and automatically exploring and checking the behavior of federated and heterogeneous distributed systems, which design was proposed in (Canini et al., 2011); the hybrid SUT test monitoring framework proposed in (Hierons, 2014).

3 APPROACH AND PROCESS

Our main objective is the development of an approach and a toolset to automate the whole process of model-based testing of distributed and heterogeneous systems in a seamless way, with a focus on integration testing, but supporting also unit (component) and system testing. The only manual activity (to be performed with tool support) should be the creation of the input model of the SUT.

To that end, our approach is based on the following main ideas:

- the adoption of different 'frontend' and 'backend' modeling notations, with an automatic translation of the input behavioral models created by the user in an accessible 'frontend' notation (using industry standards such as UML (OMG, 2011)), to a formal 'backend' notation amenable for incremental execution at runtime (such as extended Petri Nets as in our previous work for object-oriented systems (Faria and Paiva, 2014));
- the adoption of an online and adaptive test strategy, where the next test input depends on the se-

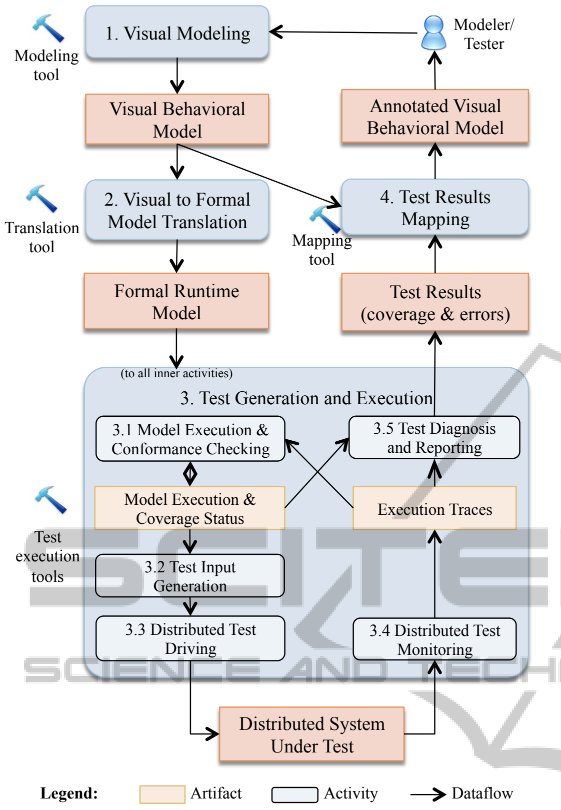

Figure 1: Dataflow view of the proposed test process.

quence of events that has been observed so far and the resulting execution state of the formal backend model, to allow for non-determinism in either the specification or the SUT (Hierons, 2014);

- the automatic mapping of test results (coverage and errors) to the 'frontend' modeling layer.

Figure 1 depicts the main activities and artifacts of the proposed test process based on the above ideas. The main activities are described in the next subsections and illustrated with a running example.

3.1 Visual modeling

The behavioral model is created using an appropriate UML profile (OMG, 2011)(Gross, 2005) and an existing modeling tool. We advocate the usage of UML 2 SDs, with a few restrictions and extensions, because they are well suited for describing and visualizing the interactions that occur between the components and actors of a distributed system. UML deployment diagrams can also be used to describe the distributed structure of the SUT. Mapping information between the model and the implementation, needed for test execution (such as the actual location of each component under test), may also be attached to the model with tagged values.

To illustrate the approach, we use a real world example from the AAL4ALL project, related with a fall detection and alert service. As illustrated in Figure 2, this service involves the interaction between different heterogeneous components running in different hardware nodes in different physical locations, as well as three users.

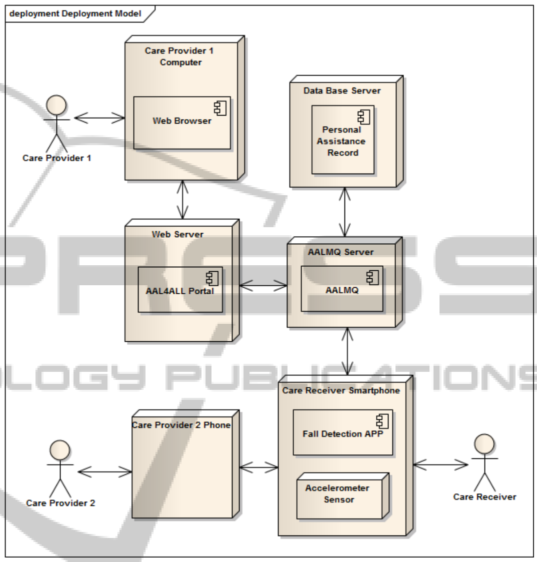

Figure 2: UML deployment diagram of a fall detection scenario.

A behavioral model for a typical fall detection scenario is shown in Figure 3. In this scenario, a care receiver has a smartphone that has installed a fall detection application. When this person falls, the application detects the fall using the smartphone's accelerometer and provides the user a message which indicates that it has detected a drop giving the possibility for the user to confirm whether he/she needs help. If the user responds that he/she does not need help (the fall was slight, or it was just the smartphone that fell to the ground), the application does not perform any action; however, if the user confirms that needs help or does not respond within 5 seconds (useful if the person became unconscious due to the fall), the application raises two actions in parallel. On the one hand, it makes a call to a previously clearcut number to contact a health care provider (in this case can be a formal or informal caregiver); on the other hand, it sends the fall occurrence for a Personal Assistance Record database and sends a message to a portal that is used by a caregiver (e.g. a doctor or nurse) that is responsible for monitoring this care receiver. The last two actions are performed through a central component of the ecosystem called AALMQ (AAL Mes-

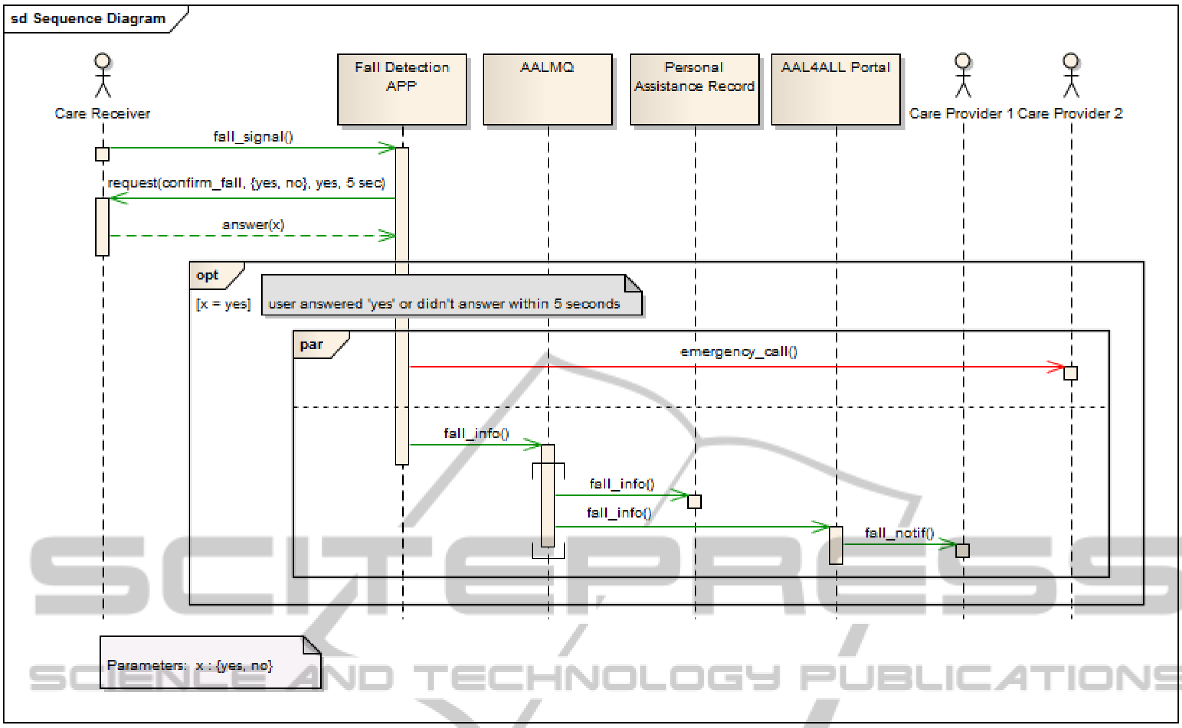

Figure 3: UML sequence diagram representing the interactions of the fall detection scenario. The diagram is already painted after a failed test execution in which the fall detection application didn't send an emergency call.

sage Queue), which allows incoming messages to be forwarded to multiple subscribers, according to the publish-subscribe pattern (Gamma et al., 1994). To facilitate the representation of a request for input from the user with a timeout and a default response, we use the special syntax *request(confirm_fall, {yes, no}, yes, 5 sec)*, where the first argument identifies the message, the second argument is the set of valid answers, the third is the default answer in case of timeout, and the last argument is the timeout time.

3.2 Visual to Formal Model Translation

For the formal runtime model, we advocate the usage of *Event-Driven Colored Petri Nets* – a sort of extended Petri Nets proposed in our previous work for testing object-oriented systems (Faria and Paiva, 2014), with the addition of time constraints as found in Timed Petri Nets. We call the resulting Petri Nets *Timed Event-Driven Colored Petri Nets*, or TEDCPN for short. Petri Nets are well suited for describing in a rigorous and machine processable way the behavior of distributed and concurrent systems, usually requiring fewer places than the number of states of equivalent finite state machines. Translation rules from UML 2 SDs to Event-Driven Colored Petri Nets have been defined in (Faria and Paiva, 2014). Rules for translating time and duration constraints in SDs to time constraints in the resulting Petri Net can also be defined.

Figure 4 shows the TEDCPN derived from the SD of Figure 3, according to the rules described in (Faria and Paiva, 2014) and additional rules for translating time constraints.

The generated TEDCPN is partitioned into a set of fragments corresponding to the participants in the source SD. Each fragment describes the behavior local to each participant and the communication with other participants via boundary places.

Transitions may be optionally labeled with an *event*, a *guard* (with braces) and a *time interval* (with square brackets). Events correspond to the sending or receiving of messages in the source SD. Guards correspond to the conditions of conditional interaction fragments in the source SD. Time intervals correspond to duration and time constraints in the source SD. A transition can only fire when there is at least one token in each input place, the event (if defined) has occurred, the guard (if defined) holds, and the time elapsed since the transition became enabled (i.e., since there is a token in each input place) lies within the time interval (if defined).

Incoming and outgoing arcs of a transition may be labeled with a pattern matching expression describing

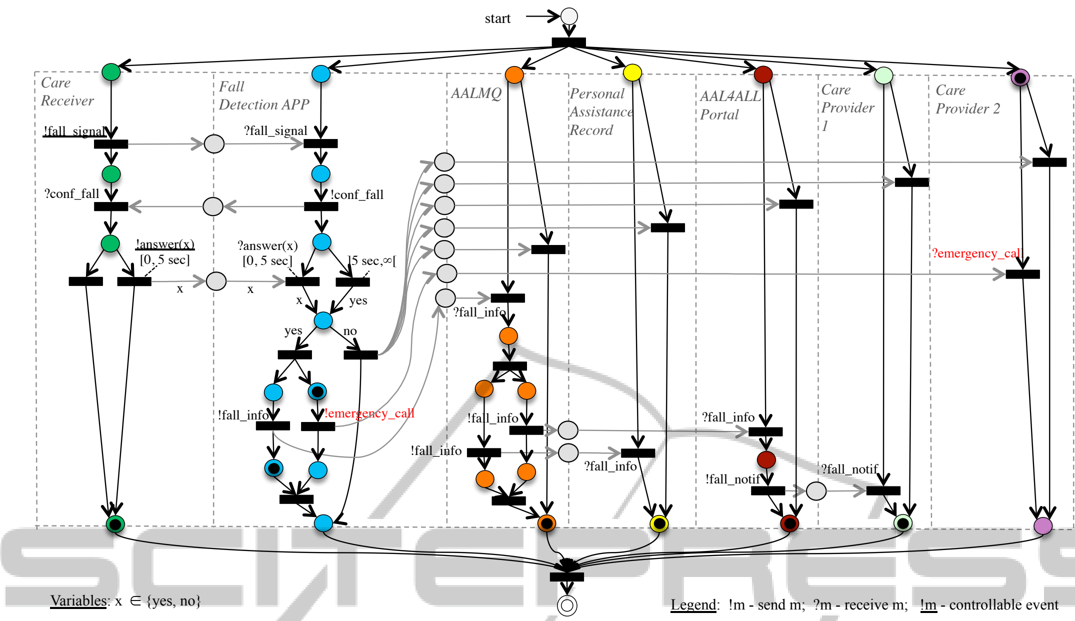

Figure 4: TEDCPN derived from the SD of Figure 3. The net is marked in a final state of a failed test execution in which the fall detection application didn't send an emergency call.

the value (token) to be taken from the source place or put in the target place, respectively, being 1 the default. For example, in Figure 4 the transition labeled "?answer(x)" has an input arc labeled "x", where "x" represents a local variable of the transition. The transition can only fire if the value of the token in the source place is the same as the value of the argument of the event. Then, the value of "x" is placed in the target place.

For testing purposes, the events in the runtime model are marked as *observable* (default) or *controllable*. Controllable events (underlined) are to be injected by the test harness (playing the role of a test driver, simulating an actor) when the corresponding transition becomes enabled. Controllable events correspond to the sending of messages from actors in the source SD. All other events are observable, i.e., they are to be monitored by the test harness. For example, when the TEDCPN of the example starts execution (i.e., a token is put in the start place), the initial unlabeled transition is executed and a token is placed in the initial place of each fragment. At that point, the only transition enabled is the one labeled with the "!fall_signal" controllable event, so the test harness will inject that event (simulating the user) and test execution proceeds.

This mechanism provides a unified framework with monitoring, testing and simulation capabilities. In one extreme case, all events in the model may be marked as observable, in which case the test system acts as a runtime monitoring and verification system. In the other extreme case, all events in the model may be marked as controllable, in which case the test system acts as a simulation system. This also allows, the usage of the same model with different markings of observable and controllable events for integration and unit testing.

3.3 Test Generation and Execution

3.3.1 Test Generation

Using the UML 2 interaction operators, a single SD, and hence the TEDCPN derived from it, may describe multiple control flow variants, that require multiple test cases for being properly exercised.

In the running example, from the reading of the set of interactions represented in Figure 3, one easily realizes that there are three test paths to be exercised (with at least one test case for each test path). The first test path (TP1) is the case where the care receiver responds negatively to the application and the application doesn't trigger any action. The second test path (TP2) is the situation where the user confirms to the application that he/she needs help and after that the application triggers the actions. The last test path (TP3) corresponds to the situation where the user doesn't answer within the defined time limit and the application triggers the remaining actions automatically. If one wants also to exercise the boundary values of allowed response time (close to 0 and close to 5 seconds), then two test cases can be considered for each of the test paths TP1 and TP2, resulting in a total of 5 test cases.

Equivalently, in order to exercise all nodes, edges and boundary values in the TEDCPN, several test cases are needed. In the example, one could exercise the two outgoing paths after the "?conf_fall" event, the two possible values of variable "x" in the "!answer(x)" event, and the two boundary values of the "[0, 5 sec]" interval, in a total of 5 test cases.

In general, the required test cases can be generated using an offline strategy (with separate generation and execution phases) or an online test strategy (with intermixed generation and execution phases) (Utting et al., 2012). In an offline strategy, the test cases are determined by a static analysis of the model, assuming the SUT behaves deterministically. But that is not often the case, so we prefer an online, adaptive, strategy, in which the next test action is decided based on the current execution state. Whenever multiple alternatives can be taken by the test harness in an execution state, the test harness must choose one of the alternatives and keep track of unexplored alternatives (i.e., model coverage information) to be exercised in subsequent test repetitions.

3.3.2 Test Execution

Test execution involves the simultaneous execution of: (i) the set of components under test (CUTs); (ii) the formal runtime model (TEDCPN), dictating the possible test inputs and the expected outputs from the CUTs in each step of test execution; (iii) a local test component for each CUT, running in the same node of the CUT, able to perform the roles of test driver (i.e., send test inputs to the CUT, simulating an actor) and test monitor (i.e., monitor all the messages sent or received by the CUT).

The collection of monitored events (message sending and receiving events) forms an *execution trace*. Testing succeeds if the observed execution trace *conforms to* the formal behavioral model, in the sense that it belongs to the (possibly infinite) set of valid traces defined by the model.

Conformance checking is performed *incrementally* as follows: (i) initially, the execution of the TEDCPN is started by placing a token in the start place and firing transitions until a quiescent state is reached (a state where no transition can fire); (ii) each time a quiescent state is reached having an enabled transition labeled with a controllable event, the test harness itself generates the event (i.e., the message specified in the event is sent to the target CUT by the appropriate test driver) and the execution status of the TEDCPN is advanced to a new quiescent state; (iii) each time an observable event is monitored (by a test monitor), the execution state of the TEDCPN is advanced until a new quiescent state is reached; (iv) the two previous steps are repeated until the final state of the TEDCPN is reached (i.e., a token is placed in the final place), in which case test execution succeeds, or until a state is reached in which there is no controllable event enabled and no observable event has been monitored for a defined wait time, in which case test execution fails. The latter situation is illustrated in Figure 4. Depending on the conformance semantics chosen, the observation of an unexpected event may also be considered a conformance error.

To minimize communication overheads, the TEDCPN can itself be executed in a *distributed* fashion, by executing each fragment of the 'global' TEDCPN (describing the behavior local to one participant and the communication with other participants via boundary places) by a local test component. Communication between the distributed test components is only needed when tokens have to be exchanged via boundary places.

When a final (success or failure) state is reached, the *Test Diagnosis and Reporting* activity is responsible to analyze the execution state of the TEDCPN and the collected execution trace, and produce meaningful error information.

Model coverage information is also collected during test execution, to guide the selection of test inputs and the decision about when to stop test execution, as follows: when it is reached a quiescent state of the TEDCPN with multiple controllable events enabled leading to different execution paths, the test harness shall generate an event that leads to a previously unexplored path; when a final state of the TEDCPN is reached, test execution is restarted if there are still unexplored (but reachable) paths.

3.4 Test Results Mapping

At the end of test execution it is important to reflect the test results back in the visual behavioral model created by the user. As an example, the marking shown in the net of Figure 4 corresponds to the final state of a failed test execution in which the Fall Detection App didn't send an emergency call. By a simple analysis of this final state (and traceability information between the source SD and the TEDCPN), it is possible to point out to the tester which messages in the source SD were covered and what was the cause of test failure (missing "emergency call" message), as shown in Figure 3.

4 TOOLSET ARCHITECTURE

Figure 5 depicts a layered architecture of a toolset for supporting the test process and approach described in the previous section, promoting reuse and extensibility.

At the bottom layer in Figure 5, the SUT is composed by a set of components under test (CUT), executing potentially in different nodes (OMG, 2011). The CUT interact with each other (usually asynchronously) and with the environment (users or external systems) through well defined interfaces at defined interaction points or ports (Hierons, 2014; Gross, 2005).

The three layers of the toolset are described in the following sections.

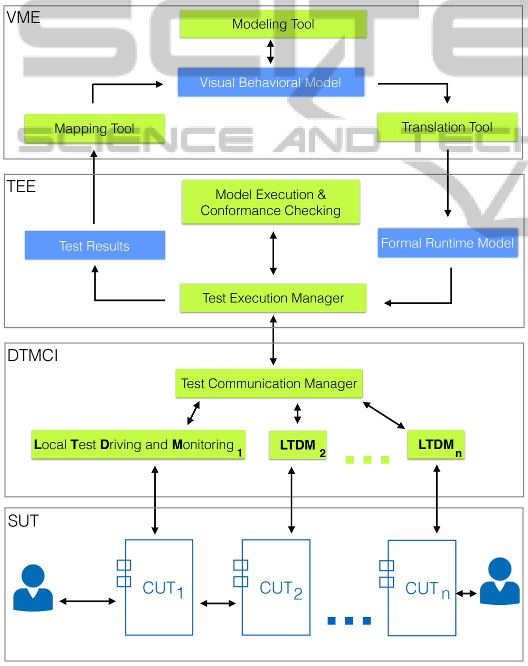

Figure 5: Toolset architecture.

4.1 Visual Modeling Environment

At the top layer, we have a visual modeling environment, where the tester can create a *visual behavioral model* of the SUT, invoke test generation and execution, and visualize test results and coverage information back in the model.

This layer also includes a *translation tool* to automatically translate the visual behavioral models created by the user into the formal notation accepted by the test execution manager in the next layer, and a *mapping tool* to translate back the test results (coverage and error information) to annotations in the visual model.

The model transformations can be implemented using existing MDA technologies and tools (Völter et al., 2013).

4.2 Test Execution Engine

At the next layer, the test execution engine is the core engine of the toolset. It comprises a *model execution & conformance checking engine*, responsible for incrementally checking the conformance of observed execution traces in the SUT against the *formal runtime model* derived from the previous layer, and a *test execution manager*, responsible for initiating test execution (using the services of the next layer), forward execution events (received from the next layer) to the model execution & conformance checking engine, decide next actions to be performed by the local test driving and monitoring components in the next layer of the system, and produce *test results* and diagnosis information for the layer above.

The model execution & conformance checking engine can be implemented by adapting existing Petri net engines, such as CPN Tools (Jensen et al., 2007).

4.3 Distributed Test Monitoring and Control Infrastructure

We adopt a hybrid test monitoring approach as proposed in (Hierons, 2014), combining a centralized 'tester' and a local 'tester' at each port (component interaction point) of the SUT, that was shown to lead to more effective testing than a purely centralized approach (where a centralized tester interacts asynchronously with the ports of the SUT) or a purely distributed approach (where multiple independent distributed testers interact synchronously with the ports of the SUT).

Hence, the Distributed Test Monitoring and Control Infrastructure comprises a set of *local test driving and monitoring* (LTDM) components, each communicating (possibly synchronously) with a component under test (CUT), performing the roles of test monitor, driver and stub; and a *test communication manager* (TCM) component, that (asynchronously) dispatches control orders (coming from the previous layer) to the LTDMs and aggregates monitoring information from the LTDMs (to be passed to the previous layer).

During test execution, the TEDCPN may be executed in a centralized or a distributed mode, depending on the processing capabilities that can be put in the LTDM components. In centralized mode, the LTDM components just monitor all observable events of interest and send them to the central TEM; they also inject controllable events when requested from the central TEM. In distributed mode, a copy of each fragment (up to boundary places) is sent to the respective LTDM component for local execution. When there is the need to send a token to a boundary place, the LTDM sends the token to the central TEM, which subsequently dispatches it to the consumer LTDM. Because of possible delays in the communication of tokens through boundary places, the LTDM components must be prepared to tentatively accept observable events before receiving enabling tokens in boundary places.

This infrastructure may be implemented by adapting and extending existing test frameworks for distributed systems, such as the ones described in Section 2.3.

Different LTDM components will be implemented for different platforms and technologies under test, such as WCF (Windows Communication Foundation), Java EE (Java Platform, Enterprise Edition), Android, etc. However, a LTDM component implemented for a given technology will be reused without change to monitor and control any component under test that uses that technology. For example, in our previous work for automating the scenario-based testing of standalone applications written in Java, we developed a runtime test library able to trace and manipulate the execution of any Java application, using AOP (aspect-oriented programming) instrumentation techniques with load-time weaving. In the case of a distributed Java application, we would need to deploy a copy of that library (or, more precisely, a modified library, to handle communication) together with each Java component under test. In the case of a distributed system implemented using other technologies (with different technologies for different components in case of heterogeneous systems), similar test monitoring components suitable for the technologies involved will have to be deployed.

5 CONCLUSIONS

In this paper, it was presented a novel approach and process for automated scenario-based testing of distributed and heterogeneous systems. It was also presented the architecture of a toolset able to support and automate the proposed test process. Based in a multilayer architecture and using a hybrid test monitoring approach combining a centralized 'tester' and a local 'tester' this toolset promotes reuse and extensibility. In the approach proposed, the tester interacts with a visual modeling front-end to describe key behavioral scenarios of the SUT using UML sequence diagrams, invoke test generation and execution, and visualize test results and coverage information back in the model using a color scheme (see Figure 3). Internally, the visual modeling notation is converted to a formal notation amenable for runtime interpretation (see Figure 4) in the back-end. A distributed test monitoring and control infrastructure is responsible for interacting with the components of the SUT, under the roles of test driver, monitor and stub. At the core of the toolset, a test execution engine coordinates test execution and checks the conformance of the observed execution trace with the expectations derived from the visual model. For better understanding the approach and toolset architecture proposed, a real world example from the AAL domain was presented along the paper.

As future work we will implement a toolset following the architecture (represented in Figure 5) and working principles presented in this paper, taking advantage of previous work for automating the integration testing of standalone object-oriented systems. To experimentally assess the benefits of the approach and toolset, industrial level case studies will be conducted, with at least one in the AAL domain.

With such a toolset, we expect to significantly reduce the cost of testing distributed and heterogeneous systems, from the standpoint of time, resources and expertise required, as compared to existing approaches.

ACKNOWLEDGEMENTS

This work is supported by project NORTE-07-0124-FEDER-000059, financed by the North Portugal Regional Operational Programme (ON.2 - O Novo Norte), under the National Strategic Reference Framework (NSRF), through the European Regional Development Fund (ERDF), and by national funds, through the Portuguese funding agency, Fundação para a Ciência e a Tecnologia (FCT).

REFERENCES

AAL4ALL (2015). Ambient Assisted Living For All. http://www.aal4all.org.

Boehm, B. (2011). Some Future Software Engineering Opportunities and Challenges. In Nanz, S., editor, *The Future of Software Engineering*, pages 1–32. Springer Berlin Heidelberg.

Canini, M., Jovanović, V., Venzano, D., Novaković, D., and Kostić, D. (2011). Online Testing of Federated and Heterogeneous Distributed Systems. *SIGCOMM Comput. Commun. Rev.*, 41(4):434–435.

Dias Neto, A. C., Subramanyan, R., Vieira, M., and Travassos, G. H. (2007). A Survey on Model-based Testing Approaches: A Systematic Review. In *Proceedings of the 1st ACM International Workshop on Empirical Assessment of Software Engineering Languages and Technologies: Held in Conjunction with the 22Nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007*, WEASELTech '07, pages 31–36, New York, NY, USA. ACM.

DoD (2008). Systems Engineering Guide for Systems of Systems. Technical report, Office of the Deputy Under Secretary of Defense for Acquisition and Technology, Systems and Software Engineering Version 1.0.

Faria, J. (2014). A Toolset for Conformance Testing against UML Sequence Diagrams. https://blogs.fe.up.pt/sdbt/.

Faria, J. and Paiva, A. (2014). A toolset for conformance testing against UML sequence diagrams based on event-driven colored Petri nets. *International Journal on Software Tools for Technology Transfer*, pages 1–20.

Faria, J. P., Lima, B., Sousa, T. B., and Martins, A. (2014). A Testing and Certification Methodology for an Open Ambient-Assisted Living Ecosystem. *International Journal of E-Health and Medical Communications (IJEHMC)*, 5(4):90–107.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design patterns: elements of reusable object-oriented software*. Pearson Education.

Gross, H.-G. (2005). *Component-Based Software Testing with UML*. Springer Berlin Heidelberg.

Hierons, R. M. (2014). Combining Centralised and Distributed Testing. *ACM Trans. Softw. Eng. Methodol.*, 24(1):5:1–5:29.

Hierons, R. M., Merayo, M. G., and Núñez, M. (2011). Scenarios-based testing of systems with distributed ports. *Software: Practice and Experience*, 41(10):999–1026.

IBM (2013). IBM® Rational® Rhapsody® Automatic Test Conductor Add On User Guide, v2.5.2.

Javed, A., Strooper, P., and Watson, G. (2007). Automated Generation of Test Cases Using Model-Driven Architecture. In *Automation of Software Test , 2007. AST '07. Second International Workshop on*, pages 3–3.

Jensen, K., Kristensen, L., and Wells, L. (2007). Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems. *International Journal on Software Tools for Technology Transfer*, 9(3-4):213–254.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. (1997). Aspect-oriented programming. In Akşit, M. and Matsuoka, S., editors, *ECOOP'97 — Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer Berlin Heidelberg.

Moreira, R. M. and Paiva, A. C. (2014). PBGT Tool: An Integrated Modeling and Testing Environment for Pattern-based GUI Testing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 863–866, New York, NY, USA. ACM.

OMG (2011). OMG Unified Modeling LanguageTM (OMG UML), Superstructure. Technical report, Object Management Group.

STAF (2014). Software Testing Automation Framework (STAF).

Tassey, G. (2002). The Economic Impacts of Inadequate Infrastructure for Software Testing. Technical report, National Institute of Standards and Technology.

Torens, C. and Ebrecht, L. (2010). RemoteTest: A Framework for Testing Distributed Systems. In *Software Engineering Advances (ICSEA), 2010 Fifth International Conference on*, pages 441–446.

Ulrich, A. and König, H. (1999). Architectures for Testing Distributed Systems. In Csopaki, G., Dibuz, S., and Tarnay, K., editors, *Testing of Communicating Systems*, volume 21 of *IFIP — The International Federation for Information Processing*, pages 93–108. Springer US.

Utting, M. and Legeard, B. (2007). *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Utting, M., Pretschner, A., and Legeard, B. (2012). A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability*, 22(5):297–312.

Völter, M., Stahl, T., Bettin, J., Haase, A., and Helsen, S. (2013). *Model-driven software development: technology, engineering, management*. John Wiley & Sons.

Wittevrongel, J. and Maurer, F. (2001). SCENTOR: scenario-based testing of e-business applications. In *Enabling Technologies: Infrastructure for Collaborative Enterprises, 2001. WET ICE 2001. Proceedings. Tenth IEEE International Workshops on*, pages 41–46.

Zhang, F., Qi, Z., Guan, H., Liu, X., Yang, M., and Zhang, Z. (2009). FiLM: A Runtime Monitoring Tool for Distributed Systems. In *Secure Software Integration and Reliability Improvement, 2009. SSIRI 2009. Third IEEE International Conference on*, pages 40–46.