On-The-Fly Model-Based Testing of Web Services with Jambition

Lars Frantzen[1,2], Maria de las Nieves Huerta[3], Zsolt Gere Kiss[4], and Thomas Wallet[3]

[1] Istituto di Scienza e Tecnologie della Informazione "Alessandro Faedo"
Consiglio Nazionale delle Ricerche, Pisa – Italy
[2] Institute for Computing and Information Sciences
Radboud University Nijmegen – The Netherlands
lf@cs.ru.nl
[3] Pragma Consultores – Argentina
{mhuerta,twallet}@pragmaconsultores.com
[4] 4D Soft – Hungary
zsolt.kiss@4dsoft.hu

Abstract. Increasing complexity and massive use of current web services raise multiple issues for achieving adequate service validation while sticking to time-to-market imperatives. For instance: How to automate test case generation and execution for stateful web services? How to realistically simulate web service related operation calls? How to ensure conformance to specifications? The Plastic validation framework tackles some of these issues by providing specific tools for automated model-based functional testing. Based on the Symbolic Transition System model, test cases can be generated and executed on-the-fly. This testing approach was applied for validating the Alarm Dispatcher eHealth service, aimed at providing health attention through mobile devices in B3G networks. In this paper we report how this modeling and testing approach helped to detect failures, support conformance, and reduce drastically the testing effort spent usually in designing test cases, validating test coverage, and executing test cases in traditional testing approaches.

1 Motivation

The usage of Web Services has been strongly growing during the last decade [3,26], uncovering new business possibilities and reaching a very broad public. Moreover Third Generation (3G) and Beyond Third Generation (B3G) mobile devices proliferation [1] reinforces such growth and leads to new massive business taking into account user mobility and connectivity [23]. As a consequence, the Web Service paradigm had to evolve to cope with emerging issues such as:

- More users directly connected and directly interacting with Web Services
- Users potentially connected from any place at any moment
- More complexity required to support new business possibilities

R. Bruni and K. Wolf (Eds.): WS-FM 2008, LNCS 5387, pp. 143–157, 2009.
© Springer-Verlag Berlin Heidelberg 2009

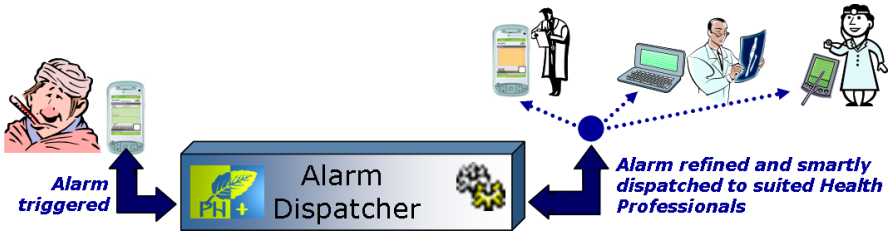

Fig. 1. Plastic eHealth Alarm Dispatcher Service

Such issues require complex Web Services, which may for instance be stateful and carry on related operations according to a complex logic, or interact directly with users through mobile devices.

The eHealth Alarm Dispatcher *Web Service Example*
We will now introduce the eHealth Alarm Dispatcher Web Service, which illustrates some of the issues mentioned above and will serve as a running example for the remainder of the paper. The Alarm Dispatcher Web Service is part of the Plastic eHealth services, aimed at providing medical attention through mobile devices over B3G networks. Plastic [25] is a European research project aimed at providing a service-oriented platform for adaptable and lightweight services in B3G networks. The Alarm Dispatcher is a passive robot which receives medical alarms triggered by patients from their mobile device (see Fig. 1). Its main goal is to interact with each patient in order to characterize and refine the alarm kind, so as to be able to forward it to suited health professionals according to the situation. For instance, the Alarm Dispatcher will ask to the patient the emergency type; get some patient information, etc. The Alarm Dispatcher finally broadcasts the refined alarm to the best suited health professionals. Based on this broadcast, some other Plastic eHealth Web Services enable the patient to select an available health professional who will start a remote diagnosis. The medical attention finishes with the professional sending his diagnosis or forwarding the alarm to other specific medical services (see [25] and [27] for technical details on Plastic eHealth Services Development).

Emerging Web Services Validation Issues
The validation of Web Services of the kind of the Alarm Dispatcher reveals some specific issues where new testing solutions must be provided in order to deal with the emerging complexity. For instance, the Alarm Dispatcher Web Service involves logical dependencies between its different operations, which cannot be invoked independently or in any order. Another aspect that increases validation complexity is that the Alarm Dispatcher is a stateful service, which means that some operation results depend on data from previously executed operations of the service. Moreover, most of the Alarm Dispatcher operations receive some inputs from the patient and an important part of the service logic is based on those inputs.

Validating Web Services like the Alarm Dispatcher requires dealing with these issues and should lead to the design and execution of numerous and complex test cases. Moreover, these test cases should take into account the operation dependencies, the service states, and the data to simulate user inputs. Test automation can drastically limit the testing effort due to such complexity, but should rely on detailed behavior specification models in order to ensure adequate validation coverage.

Overview In this paper we will present how a model-based testing approach with the Jambition and Minerva tools of the Plastic validation framework was applied for testing the Alarm Dispatcher Web Service. We will enumerate the benefits found in the light of the experiment. Section 2 briefly introduces the Plastic validation framework which provides the model-based testing tools Jambition and Minerva, which are explained in more detail in Sect. 3. Section 4 presents the results and benefits of applying this approach for the validation of the Alarm Dispatcher Web Service, while presenting conclusions, related- and future work in Sect. 5.

2 Plastic and Its Validation Framework

The Plastic project [25] adopts and revisits service-oriented computing for B3G networks, in particular assisting the development of services targeted at mobile devices. The resulting Plastic platform enables robust distributed lightweight services in the B3G networking environment through:

- A development environment leveraging model-driven engineering for the thorough development of Service Level Agreement and resource-aware services, which may be deployed on various networked nodes, including handheld devices
- A service-oriented middleware leveraging multi-radio devices and multi-network environments for applications and services run on mobile devices, further enabling context-aware and secure discovery and access to such services
- A validation framework enabling off-line and on-line validation of networked services regarding functional and extra-functional properties

The Plastic development process is evolutionary and comprehensive, i.e., it encompasses the full service lifecycle, from development to validation, and exploits as much as possible model-to-model and model-to-code transformations, as well as model-based testing. To support such a comprehensive design approach we have defined the Plastic UML2 profile, which allows designers to create service models conforming to the Plastic domain. This Plastic UML2 profile includes *Symbolic Transition System* (STS) diagrams (explained in Sect. 3.1), which are used for specifying the functional behavior and for model-based testing of Web Services. The Plastic UML2 profile additionally provides five views of service models and their corresponding diagrams for other purposes outside the scope of this paper.

The PLASTIC *validation framework* provides different tools for *off-line* and *on-line* validation of Web Services. Off-line validation activities are performed while no user ("paying customer") is using the service. Hence, off-line validation of a system implies that it will be tested in one or more artificially evolving environments that simulate possible real interacting situations. On-line approaches concern a set of techniques, methodologies and tools to monitor the system after its deployment in one of its real working contexts.

This paper deals with the JAMBITION [31] and MINERVA [31] tools, which enable functional off-line validation of Web Services, based on a model-based testing approach. In this approach, the STS model is used for the automatic generation and execution of black-box test cases for a given Web Service, as explained in the next Section.

3 Modeling and Testing Services

As mentioned above, the functional behavior of a service is modeled using an automata model called *Symbolic Transition System*. STSs are a well studied formalism in modeling and testing of reactive systems [11]. STSs can be seen as a formal semantics for a variant of UML 2.0 state machines [24]. We have developed the MINERVA library, which transforms state machines modeled with MAGICDRAW [19] – a commercial UML modeling tool – into an STS representation understood by the STS-based testing tool JAMBITION. Firstly, we introduce the STS model in Sect. 3.1. Next, we summarize MINERVA in Sect. 3.2. Finally, we present JAMBITION in Sect. 3.3.

3.1 Symbolic Transition Systems

In our setting, STSs specify the functional aspects of a service interface. The ALARM DISPATCHER service from Fig. 1 has two interfaces - one to the patient and one to the health professionals. We focus here on specifying the interface to the patient.

Firstly, there are the static STS-constituents like types, messages, parameters, and operations. This information is commonly denoted in the *Web Services Description Language* (WSDL) [8]. Secondly, there are the dynamic constituents like states, and transitions between the states. STSs can be seen as a dynamic extension of a WSDL. They specify the legal ordering of the message flow at a service interface, together with constraints on the data exchanged via message parameters (called *parts* in the WSDL).

An STS can store information in STS-specific variables. Every STS transition corresponds to either a message sent to the service (input), or a message sent from the service (output). Furthermore, a transition can be guarded by a logical expression. After a transition has fired, the values of the variables can be updated. A special kind of transition is an *unobservable* transition, which does not specify a message, but represents an internal step the STS performs. Such a transition fires without any external trigger, and may update the variables. In the underlying theories such a transition is also referred to as a τ-transition.

Due to its extent and generality we do not give here the formal definition of STSs, which can be found in [11]. Instead, we exemplify the concepts in a setting relevant for this paper.

Let us consider a WSDL operation `receiveAlarm`. The input message has a part `patient` of type `Patient`; the output message has a part `return` of type `String`. The `Patient` type is a complex type sequence with the element `age` of type `Integer`. This operation could for instance correspond to a Java method `String receiveAlarm(Patient patient)`, together with the `Patient` class. A message in an STS corresponds to a message in the WSDL. Hence, we model the call of the `receiveAlarm` operation in the STS by two consecutive transitions. The first one with input message `receiveAlarm(patient:Patient)` represents the operation invocation, the second one represents the returned value via the `receiveAlarm(return:String)` output message.

Regarding the ALARM DISPATCHER service, the `receiveAlarm` operation is one of the five operations offered in the interface specification. They are summarized in the following table:

Operation	*Input Parameters*	*Output Parameters*
receiveAlarm	*patient* : Patient	*return* : String
cancelAlarm	—	—
confirmAlarm	*lifeRisk* : Boolean	*return* : String
emergencyType	*type* : TypeOfEmergency	*return* : String
consciousness	*con* : Boolean	*return* : String

The `TypeOfEmergency` is an enumeration having the values `fatal`, `heart`, `car`, `pregnancy`, `fire`, `home`, `pediatric`, `digestion`, and `other`. Figure 2 shows an STS specifying the ALARM DISPATCHER[1]. Initially, the STS is in state `1`. Now a user of the service (in our case study the patient's service) can invoke the `receiveAlarm` operation by sending a `Patient` object identifying the sender. This corresponds to the transition from state `1` to state `2`. The guard of the transition restricts the attribute `age` of parameter `patient` to be greater than 0, and less than 120. Next, the ALARM DISPATCHER has to return a `String` via the return parameter `return`. The string is interpreted as being displayed on the patient's mobile device. This string is determined by the guard to be `"Confirm Alarm!"` (transition from state `2` to state `3`). Next, two things can happen. Either, the patient cancels the alarm by sending the `cancelAlarm` message. This returns the STS to the initial state. Or the patient confirms the alarm via the `confirmAlarm` message, which additionally indicates if the life of the patient is at risk via the `lifeRisk` parameter, which is stored in the variable `risk` via the update statement `update = "risk = lifeRisk;"` (transition from state `3` to state `4`). If the life is at risk, the alarm is immediately forwarded to the emergency service (state `4` to state `9`), and the STS returns to its initial state via an unobservable transition. Otherwise the patient is queried for the type of

[1] In the picture you find the acronym `SSM`, which stands for *Service State Machine*, which is just another term for STS.

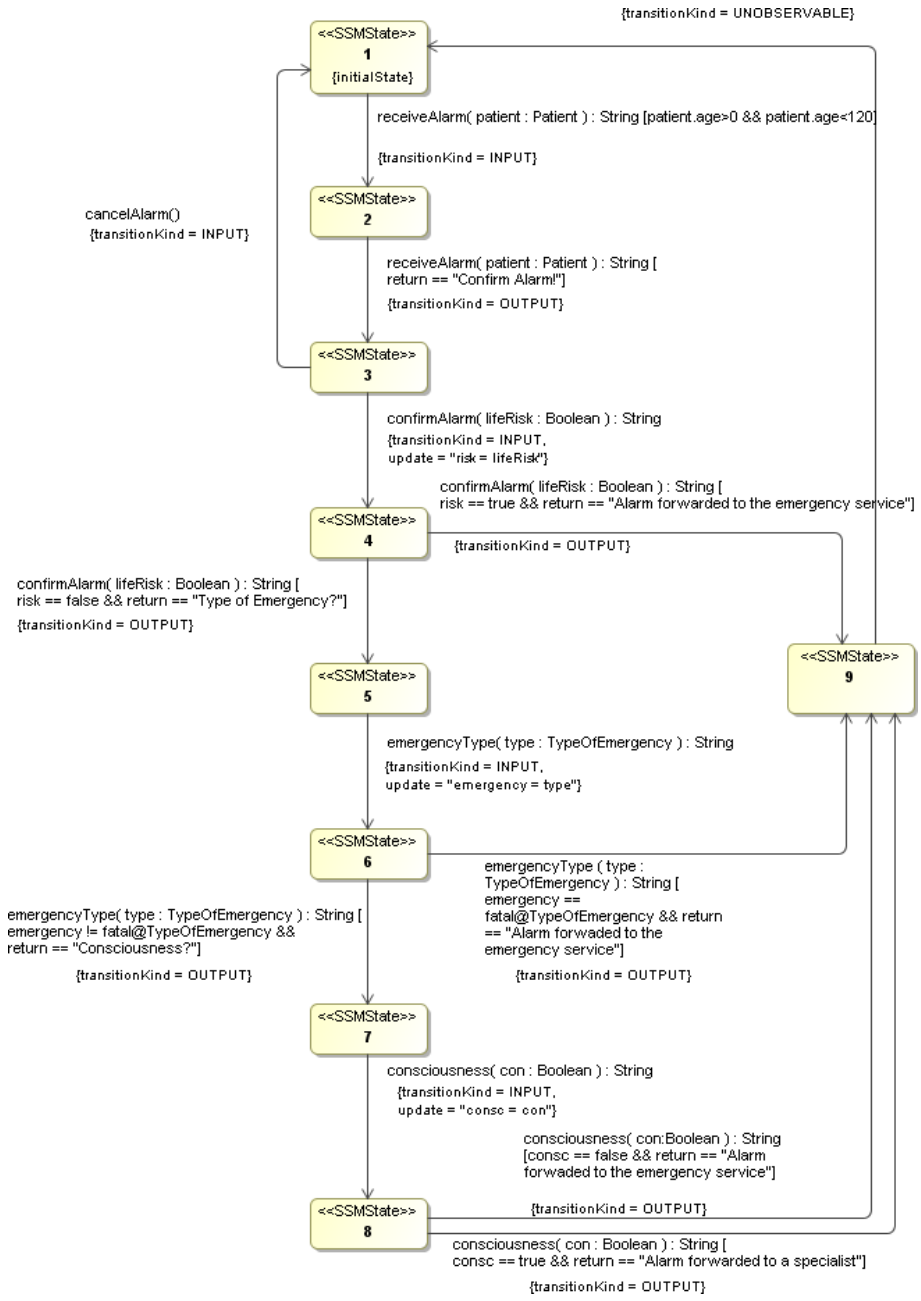

Fig. 2. An STS-Specification of the Alarm Dispatcher

emergency (state `4` to state `5`), which must be subsequently transmitted by the patient via the `emergencyType` operation (state `5` to state `6`). The type is saved in the `emergency` variable by the update statement. If the emergency type is `fatal`, the alarm is forwarded to the emergency service (state `6` to state `9`). For other types of emergencies, the state of consciousness is determined before either the emergency service or a specialist is contacted (state `6` to state `9` via states `7` and `8`).

Formal Testing
Semantically, STSs map to Labeled Transition Systems (LTSs). A rich set of formal testing theories has been defined on LTSs, see for instance [6]. A *testing relation* precisely defines when a System Under Test (SUT) conforms to its specification by relating the formal models representing SUTs with formal models representing the specifications. The gain of this effort is that one can unambiguously express what a testing algorithm is testing for, since the notions of *passing* or *failing* a test case are formally defined. Furthermore, the testing algorithm itself can be proven to be sound and complete for a given testing relation.

A well-accepted testing relation for LTSs is ioco [32]. The testing relation implemented in the JAMBITION tool is called sioco [11], a sound and complete adaption of ioco for STSs. These relations originate from the domain of reactive systems, which are inherently more complex than services. The main difference is, that a reactive system can actively send a message whenever it likes to, whereas a service sends a message only as a response to a previous request. Even though the WSDL allows in principle to specify *active* services via *solicit-response* and *notification* operations, such services are not in common use since they do not easily map to current programming paradigms and service deployment infrastructures. Due to the restriction to *passive* services the testing relations simplify, concepts like *quiescence* [32] are not relevant here. sioco simplifies to the requirement: *If the service produces a response message x after some specified trace σ, then the STS specification can also produce response message x after σ*. In other words, each observed response message must be allowed by the STS specification.

3.2 Minerva - Service Modeling

The MAGICDRAW modeling tool and the MINERVA PLASTIC tool facilitate the graphical development of services by designing and drawing the corresponding artifacts in UML diagrams, conforming to the PLASTIC UML2 Profile. By using this profile for all objects (classes, diagrams, etc.) one can ensure that the resulting product will comply with the PLASTIC conceptual model. A service in the PLASTIC concept is specified by two subviews: a *Structural View* and a *Behavioral View*. A Structural View is given by means of *Service Description diagrams* that describe the services which will constitute the final application. They provide the service interfaces and data structures. The Behavioral View specifies the dynamic service capabilities, modeled with STS diagrams.

Structural View: Defining the Data Types and Service Descriptions
The data structures supported are a commonly used subset of the XML Schema

types. There are the simple types `Integer`, `Boolean`, and `String`, and the so called complex types: literal enumerations and classes. Class types represent a sequence of types, either simple or complex. But defining recursive types like *lists* in this manner is not supported by the current framework.

Having the data types specified, the service interface can be modeled via a Service Description Diagram. Once the service description is ready, WSDL files, and the corresponding service stubs, can be generated by further tools from the PLASTIC toolchain.

Behavioral View: Creating the Symbolic Transition System
To define the dynamic behavior of a service, an `STS Diagram` is modeled, which specifies a conversation between a service interface and another actor. Figure 2 already showed an `STS Diagram` as it appears in MAGICDRAW. An STS transition has the following properties:

- **Operation** – This is the operation which triggered the transition
- **TransitionKind** – It can be:
  - `INPUT`, corresponding to an input transition (request)
  - `OUTPUT`, denoting an output transition (response)
  - `UNOBSERVABLE`, denoting an unobservable transition
- **Guard** – The guard is a boolean expression which has to hold for the transition to fire (like `patient.age>0 && patient.age<120`). To express a guard a simple language is used which offers most common operators known from programming languages. For details please refer to the JAMBITION [31] manual.
- **Update** – Variables of the STS are updated here, after the transition has fired. Such variables usually serve for recording state information, global to the conversation session.

After having created the model, the next step is creating the service implementation. Empty service stubs can be created automatically, as mentioned above. Next they have to be implemented, keeping in mind the conditions defined in the STS transition guards. Having the services in place, the STS model can be used by JAMBITION to automatically test the service. For this, it has to be exported, using the `Export to XMI feature`.

3.3 Jambition

JAMBITION is a Java tool we have developed to automatically test Web Services based on STS specifications. As said above, the underlying testing relation is sioco. Furthermore the testing approach of JAMBITION is *random* and *on-the-fly*. This basically means that out of the set of specified input actions one input is chosen randomly, and then given to the service (i.e., an operation is invoked). Next, the returned message (if any) is received from the service. If that output message is not allowed by the STS, a failure is reported. Otherwise the next input is chosen – and so on.

For the Alarm Dispatcher service, as being specified by the STS from Fig. 2, this means that Jambition plays the role of a patient's service. Initially, being in state `1`, the only specified input message is the `receiveAlarm` invocation. Jambition has to construct here a `Patient` object, with the guard-requirement that the `age` attribute must be greater 0 and less than 120. To respect such requirements, the constraint solver of GNU Prolog [18] is queried via a Socket connection. Four heuristics can be applied:

- **min**: choose the smallest solution
- **max**: choose the greatest solution
- **middle**: choose the solution in the middle
- **random**: choose a random solution

If we decide to choose the smallest solution, we get `age = 1`. Since always choosing `1` might not be sufficient for achieving a desired coverage, a random solution is commonly good practice. Having the parameter object constructed, the `receiveAlarm` operation in invoked. Jambition moves the STS into state `2` and receives next the string the `receiveAlarm` operation returns. If this string does not equal `"Confirm Alarm!"`, a failure is reported. Otherwise state `3` is reached. Here, again randomly, Jambition chooses either to cancel the alarm (back to state `1`), or to invoke the `confirmAlarm` operation. In the latter case the `lifeRisk` parameter is not constrained, a random value (`true` or `false`) is chosen. Being in state `4` the returned string is received. Assuming a `lifeRisk` of value `true` had been generated, that string must equal `"Alarm forwarded to the emergency service"`. If the chosen risk was `false`, the string must instead be `"Type of Emergency?"`, and Jambition moves to state `5`, where an emergency type must be constructed to invoke the `emergencyType` operation. This process continues in this manner until either a failure is spotted, or the user halts the testing.

The *on-the-fly* approach differs from more classical testing techniques by not firstly generating a set of test cases, which are subsequently executed on the system. Instead, the test case generation, -execution, and -assessment happen in lockstep. So doing has, inter alia, the advantage of allaying the state space explosion problem faced by several conventional model-based testing techniques. The rationale here is that a test case developed beforehand has to consider all possible outputs the system might return, whereas the *on-the-fly* tester directly observes the specific output, and can guide the testing accordingly. Another cause of state space explosion is the transformation of symbolic models like STSs into semantical models like LTSs. Several tools do this step to apply test algorithms which are defined on the semantical model. Jambition also solves this issue by skipping this transformation step. Instead, its test algorithm directly deals with the STS, see [10] for details.

To visualize the ongoing testing process, and to understand a reported failure, Jambition can display the messages exchanged with the service while being tested in real-time via the Quick Sequence Diagram Editor [21], an external open-source visualizer for UML sequence diagrams. Furthermore, Jambition

displays the achieved state- and transition coverage of the STS. We will show these features of JAMBITION in more detail in the next section.

4 Results

The STS-based modeling and testing approach presented in the preceding chapter was used to specify, develop, and validate the functional behavior of the ALARM DISPATCHER service. In this section we present the experiences and benefits that the approach brought to the project.

eHealth ALARM DISPATCHER *Experiment Setup*
Starting from the requirements addressed by the ALARM DISPATCHER service, the experiment began with the modeling of the corresponding STS with the PLASTIC UML2 Profile in the MAGICDRAW tool. The resulting diagrams of this activity were a diagram of the ALARM DISPATCHER Web Service and operations, a diagram of the involved data types and a diagram of the STS. As depicted in Fig. 2 the ALARM DISPATCHER STS consists of 9 states and 13 transitions, specifying how the dispatcher must deal with different kinds of emergencies and other emergency attributes like life risk and consciousness. Based on these specifications, the Alarm Dispatcher Web Service was implemented via the Netbeans IDE [20] and deployed on the GlassFish Application Server [17]. At this point we finished the setup of the experiment. The experiment steps were the following:

1. *STS model exportation*: from MAGICDRAW we exported automatically the ALARM DISPATCHER STS.
2. JAMBITION *STS importation*: we loaded the STS file into JAMBITION. The importation includes checking the deployment of the corresponding service. During this step, the following types of errors where detected:
   (a) Consistency errors in the models (for instance, a data type was wrongly referenced in the STS).
   (b) Consistency errors between the models and the deployed service (for instance, a parameter of a service operation was declared with different types in the deployed service and in the model).
   (c) Service uncompleted deployment problems (for instance, the service deployment process could not be completed).
3. ALARM DISPATCHER *Validation*: we started the JAMBITION validation process that cycles continuously the STS till errors are found. During this step, the following types of errors where detected:
   (a) Transition guards violated (for instance, the emergency was not considered fatal in a case where the STS specified that it should be treated as fatal).
   (b) Never ending operations (for instance, an operation was not returning a result or had some bugs that stopped the operation).
   (c) States or transitions never reached (for instance, the validation coverage was never completed since some states or transitions were never reached due to missing features, or faults in the model itself, like non-reachable states).

Table 1. Alarm Dispatcher Experiment Steps

Step	*Duration*	*Details*
STS model exportation	25 sec	This is a MagicDraw feature
Jambition STS importation	3 sec	For successful importations
Alarm Dispatcher Validation	8 sec	Reaching full STS coverage without errors

Errors found in steps two and three were corrected, and the experiment was repeated until all the failures disappeared, and a full coverage of states and transitions was reached. To achieve this, an average of 191 input and output messages were automatically processed by the random exploration of the STS. Table 1 presents average durations of the experiment steps[2].

Jambition *Added Value*
Benefits of well applied testing automation and model-based testing approaches such as testing effort saving, coverage assurance, test reuse, regression reliability and testing duration compression have been extensively discussed in literature (for instance, see [16],[2],[7]). In addition to these generic benefits of test automation, Jambition brought some specific added value when validating Web Services as in the Alarm Dispatcher experiment. Next we give a list of the Jambition benefits that were of particular interest for our Alarm Dispatcher experiment:

- Automatic and on-the-fly generation, execution and assessment of numerous test cases
- Real-time visualization of test cases execution coverage (percentage of STS states and transitions visited, as shown in Fig. 3)
- Test cases generation and execution time is very quick
- Debugging visualization facilities for tracking operations and data (with the Quick Sequence Diagram Editor, as shown in Fig. 4)
- Jambition and Minerva are released under the open source GPLv3 license

Testing Effort Reduction
A specific benefit of Jambition comes from the testing effort reduction provided by the tool automation. We will now give some considerations indicating the order of magnitude of this reduction by referring again to the Alarm Dispatcher example. For this purpose we are comparing only the test activities automated by Jambition, i.e., test case generation, -execution, and -assessment. We are not taking into account other activities usually involved in testing such as test planning, defect reporting, code debugging and correction, testing environment preparation, etc. Indeed we do consider that such activities, as well as the STS modeling, remain unchanged when testing with or without Jambition.

Comparing with traditional, manual testing techniques is not straightforward, since Jambition does not generate a set of test cases. To still have a metric at

[2] Experiment conduced on a standard Intel Core Duo CPU T2350 1,86GHz, 1GB RAM.

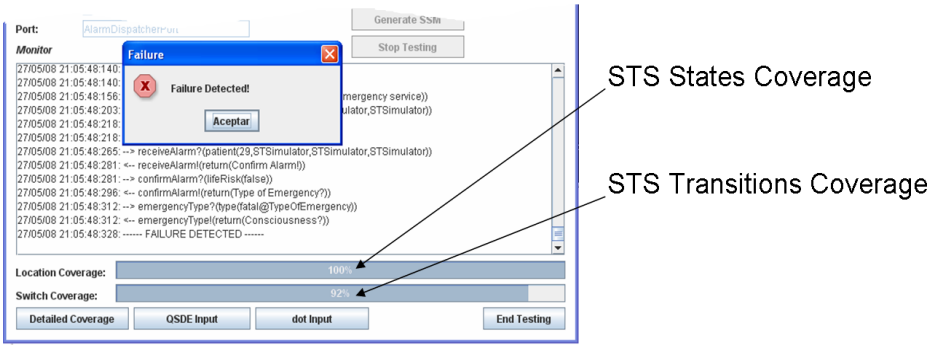

Fig. 3. JAMBITION Test Coverage Indicators

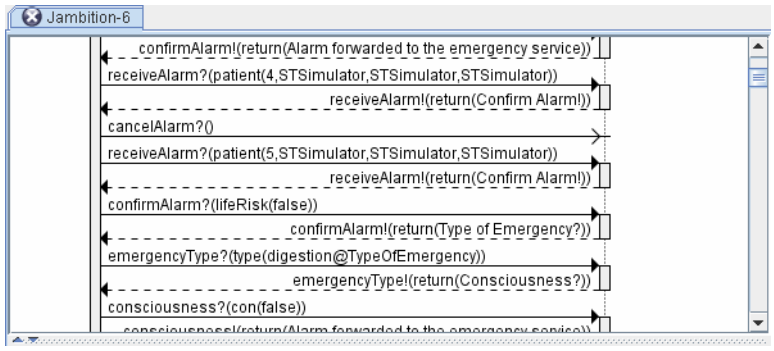

Fig. 4. Quick Sequence Diagram Editor

hand we first define what we mean by a test case, namely a path in the STS which starts and ends at the initial state 1. Such a path basically corresponds to a scenario, or transaction, like *receiving an alarm having fatal emergency*. To achieve full transition coverage, 5 such test cases are needed, see table 2.

In a traditional testing approach, these test cases must be designed and executed manually. We have said before that on average JAMBITION processed 191 messages to achieve the same coverage, which corresponds to ca. 37 test cases. Table 1 has shown that less than 8sec are needed to generate and execute these test cases. No matter how precisely you measure, this small example already shows how JAMBITION is an order of magnitude faster than the manual way. And, that JAMBITION executes more test cases to achieve the same coverage is another advantage, since more test cases simply can find more failures. For instance, to achieve full transition coverage, it is sufficient to make a test case with a fatal emergency type, and one with a non-fatal type. Since JAMBITIONexecutes on average 37 test cases, it is very likely, that it tests for more than just these two emergency types.

Table 2. Full Transition Coverage with Five Scenarios

Scenario	*State Sequence*
Cancelled Alarm	1 → 2 → 3 → 1
Risk of Life	1 → 2 → 3 → 4 → 9 → 1
Fatal Emergency	1 → 2 → 3 → 4 → 5 → 6 → 9 → 1
Consciousness	1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 1
No Consciousness	1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 1

But there is also a drawback if coverage depends on mere random decisions, since they may simply not be sufficient to reach each state and transition. We will point to that in the next and concluding Section.

5 Conclusions, Related- and Future Work

To the best of our knowledge, JAMBITION is the only current testing tool which allows for automatic and on-the-fly model-based testing based on symbolic models. Other testing approaches which are also based on (variants of) the STS model are [9,14]. Instead of on-the-fly testing, they use *test purposes* to deal with the state-space explosion problem. The specific value of random and on-the-fly testing has been demonstrated for instance in [4].

Several approaches exist to test Web Services based on other models than state machines. To name just a few, in [15] the authors propose to include graph transformation rules that enable the automatic derivation of meaningful test cases. To apply the approach they require that a service implements interfaces that increase its testability. A somewhat similar model than STSs is the *Business Process Execution Language* (BPEL) [29]. Since BPEL is an implementation language for service orchestration, its focus is different than ours. Several interesting testing and verification issues can be formulated on BPEL specifications, see for instance [5,12,13].

Despite its smallness the ALARM DISPATCHER validation experiment gave very promising results, satisfying the expectations raised by using test automation tools in a real development project. PLASTIC tools adoption for service modeling and testing was straightforward and resulted in an important reduction of the testing effort, which can scale up to very important savings for more complex services, representing an important productivity gain over conventional manual testing approaches. These conclusions settled the foundation for further experiments with JAMBITION in more complex Web Services development case studies.

The JAMBITION tool is based on a Java library which allows to model and simulate STSs, called STSIMULATOR [22]. One current limitation of the library is the lack of recursive data types like *lists*. Several web services use lists (via XML Schema unbounded sequences) to transmit data objects of variable length. We are currently investigating how to deal best with such recursive types.

We have already indicated that a mere random approach for test data selection is not necessarily sufficient to reach a full state- and transition coverage. It is even less appropriate for more sophisticated coverage criteria like condition coverage of the guards. Several approaches based on symbolic execution exist to guide the test case generation in a way that coverage becomes a search problem, see for instance [28,30]. Combining JAMBITION with such approaches is a major future goal. Also, techniques like equivalence partitioning and boundary value analysis can be a very fruitful combination with the random approach. For example, in its current version JAMBITION does not test for invalid equivalence class boundaries (like a patient at the age of 120). Also the combination of measuring model-coverage and code coverage can give further insight in the test efficiency.

STSs can in principle also be used to model the communication between several Web Services. To do so they have to embrace the message flow at several interfaces, like BPEL does. For instance, an STS could be modeled which also deals with the ALARM DISPATCHER interface to the health professionals. Testing based on such multi-interface STSs would allow to test more complex scenarios like coordinated and composed Web Services.

Acknowledgments. The PLASTIC Project is funded under FP6 STREP contract number 26955 by the Information Society Technologies (IST). Special thanks to Lorenzo Jorquera and Daniel Yankelevich for their detailed review of early versions of this paper. Lars Frantzen is further supported by the Marie Curie Network TAROT (MRTN-CT-2004-505121) and by the Netherlands Organization for Scientific Research (NWO) under project STRESS.

References

1. 3G Americas. 2.5 Billion GSM Subscribers Worldwide - More than a Million New Users Daily (press release), `http://www.3gamericas.org/English/News_Room/DisplayPressRelease.cfm?id=2982&s=ENG`, June 5 (2007)
2. Apfelbaum, L.: Automated Functional Test Generation. In: Proceedings of the Autotestcon 1995 Conference. IEEE, Los Alamitos (1995)
3. Business Wire Article. SOA Software Products Drive More Than 10 Billion Web Service Transactions, `http://findarticles.com/p/articles/mi_m0EIN/is_2006_Sept_18/ai_n16728778`, September 18 (2006)
4. Belinfante, A., Feenstra, J., de Vries, R.G., Tretmans, J., Goga, N., Feijs, L., Mauw, S., Heerink, L.: Formal test automation: A simple experiment. In: Csopaki, G., Dibuz, S., Tarnay, K. (eds.) TestCom 1999, pp. 179–196. Kluwer Academic Publishers, Dordrecht (1999)
5. Bianculli, D., Ghezzi, C., Spoletini, P.: A model checking approach to verify BPEL4WS workflows. In: IEEE SOCA 2007, pp. 13–20. IEEE Computer Society Press, Los Alamitos (2007)
6. Brinksma, E., Tretmans, J.: Testing transition systems: an annotated bibliography. In: Cassez, F., Jard, C., Rozoy, B., Dermot, M. (eds.) MOVEP 2000. LNCS, vol. 2067, pp. 187–195. Springer, Heidelberg (2001)
7. Bruno, G., Varani, M., Vico, V., Offerman, C.: Benefits of using model-based testing tools. In: Nesi, P. (ed.) Objective Quality 1995. LNCS, vol. 926, pp. 224–235. Springer, Heidelberg (1995)

8. Christensen, E., et al.: Web Service Definition Language (WSDL) ver. 1.1 (2001), http://www.w3.org/TR/wsdl
9. Clarke, D., Jéron, T., Rusu, V., Zinovieva, E.: STG: a Symbolic Test Generation tool. In: Katoen, J.-P., Stevens, P. (eds.) TACAS 2002. LNCS, vol. 2280, p. 470. Springer, Heidelberg (2002)
10. Frantzen, L., Tretmans, J., Willemse, T.A.C.: Test generation based on symbolic specifications. In: Grabowski, J., Nielsen, B. (eds.) FATES 2004. LNCS, vol. 3395, pp. 1–15. Springer, Heidelberg (2005)
11. Frantzen, L., Tretmans, J., Willemse, T.A.C.: A symbolic framework for model-based testing. In: Havelund, K., Núñez, M., Rosu, G., Wolff, B. (eds.) FATES 2006 and RV 2006. LNCS, vol. 4262, pp. 40–54. Springer, Heidelberg (2006)
12. Fu, X., Bultan, T., Su, J.: Analysis of interacting BPEL web services. In: Proc. of WWW 2004, New York, USA, May 17-22, pp. 17–22 (2004)
13. García-Fanjul, J., Tuya, J., de la Riva, C.: Generating test cases specifications for compositions of web services. In: Bertolino, A., Polini, A. (eds.) WS-MaTe2006, pp. 83–94 (2006)
14. Gaston, C., Le Gall, P., Rapin, N., Touil, A.: Symbolic execution techniques for test purpose definition. In: Uyar, M.Ü., Duale, A.Y., Fecko, M.A. (eds.) TestCom 2006. LNCS, vol. 3964, pp. 1–18. Springer, Heidelberg (2006)
15. Heckel, R., Mariani, L.: Automatic conformance testing of web services. In: Cerioli, M. (ed.) FASE 2005. LNCS, vol. 3442, pp. 34–48. Springer, Heidelberg (2005)
16. Hoffman, D.: Cost Benefits for Test Automation. In: STAR West 1999 (1999)
17. GlassFish Application Server homepage, https://glassfish.dev.java.net
18. GNU Prolog homepage, http://www.gprolog.org/
19. MagicDraw homepage, http://www.magicdraw.com
20. Netbeans IDE homepage, www.netbeans.org
21. Quick Sequence Diagram Editor homepage, http://sdedit.sourceforge.net/
22. STSimulator homepage, http://www.cs.ru.nl/~lf/tools/stsimulator/
23. Lau, J.: The State of European Enterprise Mobility in 2006, October 13, 2006. Forrester Research (2006)
24. Object Management Group. UML 2.0 Superstructure Specification, ptc/03-08-02 edition. Adopted Specification
25. PLASTIC project homepage, http://www-c.inria.fr/plastic
26. Web Host Industry Review. Web Services to Reach $21 Billion by 2007: IDC, http://www.thewhir.com/marketwatch/idc020503.cfm
27. Rong, L., Wallet, T., Fredj, M., Georgantas, N.: Mobile Medical Diagnosis: An m-Health Initiative through Service Continuity in B3G. In: Middleware 2007 Demos - ACM/IFIP/USENIX Middleware Conference (November 2007)
28. Sen, K., Agha, G.: CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-Checking Tools. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 419–423. Springer, Heidelberg (2006)
29. Business Process Execution Language for Web Services version 1.1 specification, http://www-128.ibm.com/developerworks/library/specification/ws-bpel/
30. Tillmann, N., de Halleux, J.: Pex - White Box Test Generation for.NET. In: Beckert, B., Hähnle, R. (eds.) TAP 2008. LNCS, vol. 4966, pp. 134–153. Springer, Heidelberg (2008)
31. PLASTIC tools homepage, http://plastic.isti.cnr.it/wiki/tools
32. Tretmans, J.: Test generation with inputs, outputs and repetitive quiescence. Software—Concepts and Tools 17(3), 103–120 (1996)