Let The Puppets Move!
Automated Testbed Generation for Service-oriented Mobile Applications

Antonia Bertolino, Guglielmo De Angelis, Francesca Lonetti, Antonino Sabetta
Istituto di Scienza e Tecnologie dell'Informazione "A. Faedo"
Consiglio Nazionale delle Ricerche, Pisa, Italy
{antonia.bertolino, guglielmo.deangelis, francesca.lonetti, antonino.sabetta}@isti.cnr.it

Abstract

There is a growing interest for techniques and tools facilitating the testing of mobile systems. The movement of nodes is one of the relevant factors of context change in ubiquitous systems and a key challenge in the validation of context-aware applications. An approach is proposed to generate a testbed for service-oriented systems that takes into account a mobility model of the nodes of the network in which the accessed services are deployed. This testbed allows a tester to assess off-line the QoS properties of a service under test, by considering possible variations in the response of the interacting services due to node mobility.

1 Introduction

Advances in networking technologies make available an open wireless environment where services can be provided *anytime* and *anywhere* to mobile users.

The proper functioning of a service depends on the resources and services provided in the current configuration of a network. Changes in network connectivity and locations may lead to unpredictable variations in contextual information. As a device moves across the network, the environment that surrounds it changes as well: new services, offered by mobile nodes may become available, whereas other services that where used previously, may no longer be reachable. This dynamic transformation of the network topology and, consequently, of the configuration of the environment must be taken into account when developing a networked service, especially in the testing phase.

Context-awareness and adaptation are key features for services that are to be deployed in different environments and on hardware platforms with different characteristics. Applications must be able to react to context changes and to adapt themselves so as to continue to provide services within the levels of Quality of Service (QoS) that were previously agreed. Therefore, in order to construct an efficient and effective application, developers should test it in advance in all possible network scenarios. However, at development time it is difficult to anticipate all possible configurations in which a service will be executed. A solution for off-line testing is to provide a testbed in which the behavior of the underlying platform and network can be simulated in a realistic way. In a mobile network, the most typical context change is due to the movement of nodes.

We have investigated for some time the design of an off-line validation framework enabling extra functional testing of service-oriented systems. At the core of our approach is a model-based stub generator, called PUPPET (Pick UP Performance Evaluation Testbed) [2].

PUPPET supports the validation of service-oriented applications, aiming at evaluating the desired QoS characteristics for a specific service under development before it is deployed. PUPPET can automatically derive stubs for the invoked services from their published functional and extra functional specifications, thus generating an environment within which the composite service can be run and tested.

In this work we "let the PUPPETs move": we equip PUPPET with the capability to simulate the runtime movement of the nodes that host the generated stubs. To this end, we exploit the mobility models obtained with the NS-2 network simulator [1]. NS-2 is an open source tool widely used in the networking research community to simulate wired and wireless network scenarios and to produce timestamped movement traces of each network node for the whole simulation time.

The contribution of this paper is thus the automated generation of a testing environment whose QoS properties depend on the mobility models of the devices where the services of the testbed are deployed. The benefit of introducing a mobility model in PUPPET is twofold. On the one hand, the proposed approach is able to validate the QoS features of the Service Under Test (`SUT`) by taking into account the availability of the other interacting services (in earlier works we did not consider mobility and we assumed services were

always available). We assume that a service is available in a network scenario if, according to the used mobility model, the node where this service is deployed is reachable. On the other hand, our testbed generator permits to use in the testing phase the same mobility model that can be used during the design and analysis phases.

In the next section, we provide a general scenario motivating our proposal, then after describing the PUPPET tool and the NS-2 simulator in Sec. 3 and Sec. 4 respectively, we illustrate our approach in Sec. 5. An exploratory example is presented in Sec. 6, while related work is summed up in Sec. 7. Finally, we draw some conclusions in Sec. 8.

2 Motivating Example

This section illustrates a motivating example that we refer in the rest of the paper to illustrate the proposed approach. Specifically, the scenario refers to an ideal service-oriented system used in an hospital in order to forward (non critical) assistance requests from patients to doctors. Three kinds of entities are involved in the proposed example: the doctors, the patients, and the Hospital Operator Service (HOS) (see Fig. 1).

Each doctor is equipped with a PDA. Each PDA hosts a set of services[1] according to the role of the doctors in the hospital and the kind of assistance they can give. Also each patient has a mobile device that can be used to request assistance from a doctor. In particular, the patient queries the HOS for a doctor. Depending by the kind of the request, the HOS starts invoking the services offered by each doctor looking for a match between the patient request and the doctor availability. When a match is found, the HOS informs the patient that the request has been successfully scheduled.

Let us assume that the interactions between the HOS and the services deployed on the doctor's wearable device are regulated by QoS agreements that impose constraints on the latency of replies and on the reliability of the services. QoS agreement may also be established between patients and the HOS.

In this scenario we realistically consider that a doctor moves within the hospital. Furthermore, he/she is not always present (i.e., reachable) or could be busy in some activities. Thus the set of available services varies according to some mobility models. As introduced above, we consider that doctors play different roles in this scenario; thus we also assume that the maximum tolerated response latency of a service offered by a doctor's PDA may vary.

Now, the behavior (both functional and extra functional) of the HOS depends on the run-time behavior of doctors. The more the doctors are available, the higher is the probability to find a doctor matching a patient request. Hence, the agreement on the response time that the HOS can offer to patients varies depending on the roles played by the doctors that are available.

[1] In the remaining of the paper we refer to Web Services indifferently using both "Web Service" and "service" terms.

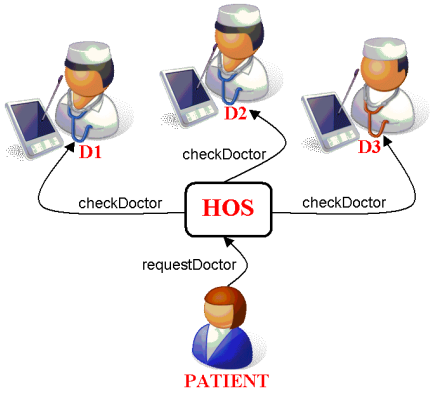


Figure 1. The Example Scenario

Furthermore, the latency of a doctor's response may be also influenced by the network connectivity between the HOS and the doctor (for example at a certain point in time, a doctor can be far away from an access point). In order to verify the actual QoS level that the HOS can deliver in such varying real-life conditions, it is fundamental to be able to model and measure the influence of all the above factors.

In the rest of the paper, we consider the interactions among a patient, an HOS and the services offered by three different doctors' PDAs (i.e. D1, D2, D3 in Fig. 1). The services provide the same functionalities, but the QoS agreements between the HOS and each of them are different. The HOS interface exposes a *requestDoctor* operation, while all the services exported by the doctors expose the *checkDoctor* operation. When the HOS's *requestDoctor* operation is invoked by a patient, the HOS invokes the *checkDoctor* operation on D1, D2, D3 successively.

To illustrate our approach, we suppose that the web service implementing the HOS is under development. Furthermore, we put ourselves in the role of the team that develops the HOS and needs to test the implementation in an environment that is as close as possible to the run-time environment. As developers, we would like to reduce the effort to be spent in developing a testbed for the HOS.

The PUPPET tool as presented in [4] addresses this issue by automatically generating stubs that simulate the required external services that for different reasons could not be available during the testing phase. Such stubs are capable of reproducing both the functional and extra functional behavior of the missing services, and are ready-for-use in mocking up the QoS testing environment. Nevertheless, the previous version of the tool did not consider that the mobility of the services in the testbed may affect the QoS experienced at the port of the SUT.

In the following of this paper we present an extended version of PUPPET that can emulate also the mobility of the devices on which the stubbed services are deployed. In this way, our approach is able to derive realistic values for the QoS offered by the HOS to the patients, by capturing the effects of mobility and context (i.e. mutual position) on service latency and reliability.

3 Puppet

In this section, we introduce PUPPET illustrating first the main characteristics of the approach and then the logical architecture of the implemented tool. As described above, the idea of PUPPET is general and could be applied to any instantiation of the Service Oriented Architecture (SOA). However, the current implementation focuses on the Web Services technology.

3.1 PUPPET: The Approach

In SOAs, services collectively interact to execute a unit of programming logic [3]. Service composition allows for the definition of complex applications at higher levels of abstractions. Nevertheless, since services are always part of a larger aggregation, their executions often rely on the interaction with other/external services.

In such a cooperating scenario, let us consider the example of a service provider who develops a composite service `SUT`, which is intended to interact with several other existing services (e.g. the services `S1`, `S2`, and `S3` deployed on PDAs `D1`, `D2`, and `D3`). In general, we can suppose that the service provider needs to test the implementation of `SUT` but he/she does not own or control the externally invoked services: for example interactions may have a cost that is not affordable for testing purposes, or the external services are being developed in parallel with `SUT`.

The approach proposed by PUPPET is to automatically derive stubs for the externally accessed services `Si` from published functional and extra functional specifications of the external services. PUPPET generates an environment (the services stubs) within which the composite service can be run and tested (see Fig. 2).

While various kinds of testbed can be generated according to the purposes of the validation activities, PUPPET aims specifically at providing a testbed for reliable estimation of the exposed QoS properties of the `SUT`. Concerning the externally accessed services, PUPPET is able to automatically derive stub services that expose a QoS behavior conforming to the extra functional specifications such as agreements among the interacting services.

Once the QoS tested is generated, the service provider may test the `SUT` by deploying it on the real machine used at run-time. This would help in providing realistic QoS measures preventing the problem of recreating a fake deployment platform; in particular, the QoS evaluations will also take into account the other applications running on the same machine that compete for resources with the service under test (it is worth noting that handling this case would be extremely difficult using analytical techniques).

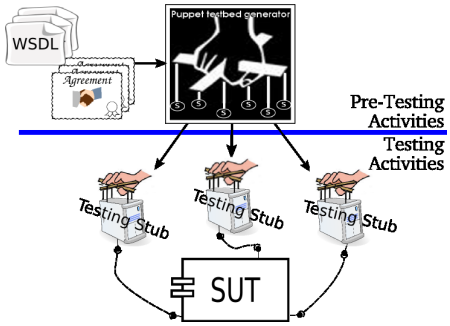


Figure 2. General idea of PUPPET

3.2 PUPPET: The Tool

The logical architecture of PUPPET is structured in layered modules, whereby each module plays a specific role within the stub generation process (see Fig. 3).

The most external module focuses on converting the abstract part of a WSDL representation into a collection of Java classes and interfaces. In PUPPET such transformation is performed exploiting the Apache-Axis *WSDL2Java* utility [15]. For each generated set of Java classes, the *WSDL-2Java* tool generates also an "axis-dependent" deployment specification. (i.e. the WSDD files [15])

Assuming that a service is characterized by means of some extra functional properties, the second module of the PUPPET generation process fills the stubs with the code simulating the extra functional behavior. We assume that the desired QoS properties of a service are expressed according to the WS-Agreement specification [9].

For each kind of term in the agreement, PUPPET implements a specific interpretation by means of a given operational semantics. Specifically, PUPPET interprets the QoS statements into a parametric portion of Java code [4]. This can be a quite complex and effort-prone task, but given a specific set of QoS properties, the mapping of the related concepts has to be done only once and for all. During the generation of the code emulating the QoS properties, the WSDD specifications are exploited in order to link the proper Java files.

The stubs developed thus far include the set of operations they export and the emulation code for the extra functional behaviors as specified in the WS-Agreement. Next, PUPPET can includes a module to link each stub with code emulating the supposed functional behavior [4].

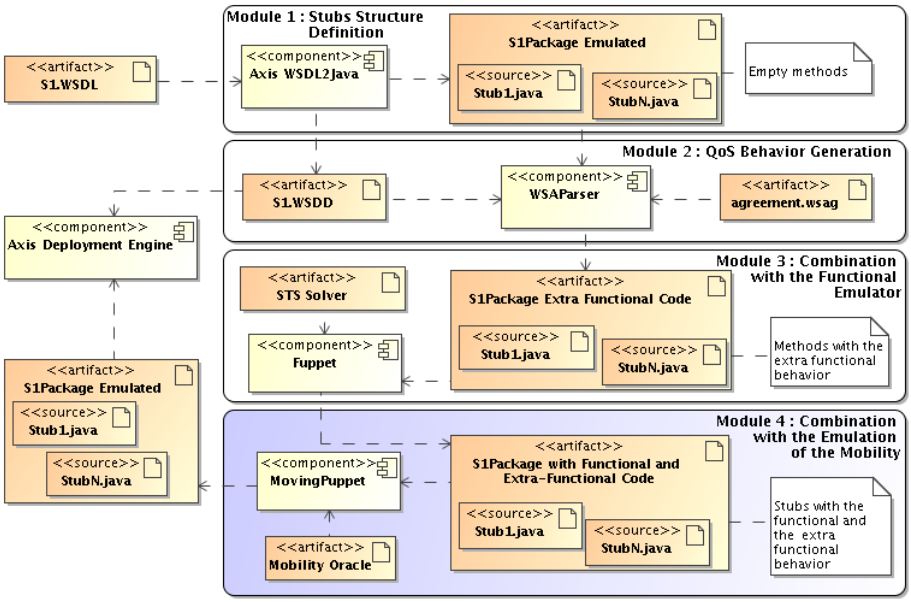


Figure 3. PUPPET **Logical Architecture**

The functional behavior of a service is modeled using an automata model called Symbolic Transition System (STS) [7]. PUPPET inserts into the stubs parametric code able to wrap an STS simulator [2]. Specifically, for each invocation to a service the stub can call the STS simulator package, choose one of the possible functionally correct results, and send it back to answer the service client request. The STS simulator can keep track of the symbolic states in which the STS can currently be. Thus, to supply the emulation of the functional behavior, PUPPET would demand that the external services carry on the STS specification corresponding to their provided interface.

The new contribution of this paper concerns the module that finally plugs into the obtained stubs the emulation of the mobility. The detailed description of this module is given in Sec 5.

In the end of the generation process, the developers can collect the generated service stubs mocking up the environment they would emulate by deploying the generated stub on any Axis platform.

4 The NS-2 Simulator

The NS-2 simulator is an open source network simulation tool widely used in the networking research community [1]. The first version of the Network Simulator was developed by the Network Research Group at the Lawrence Berkeley National Laboratory (LBNL). The second version (NS-2) is now part of the Virtual InterNetwork Testbed (VINT) project. NS-2 is an object-oriented, discrete-event driven network simulator; it is implemented in C++ and uses OTcl (Object Tool Command Language) for building command and configuration interfaces. It was first aimed at simulating large TCP/IP networks, and further enhanced to integrate wireless extensions like IEEE 802.11, ad hoc networks and, more recently, cellular communications.

In the approach proposed in this paper, we take advantage of a simple functionality of NS-2, that is the generation of a mobility model for the network nodes. For defining a simple movement of nodes we use the setdest tool integrated within NS-2. For example, the following line in the network scenario configuration file means that at 4.5 seconds, the node `1` starts to move toward the location $(210, 16)$ at the speed of $1.5m/s$.

```
...
$ns_ at 4.5 "$node_(1) setdest 210 16 1.5"
...
```

Clearly, more complex or randomly generated movements can be defined in the NS-2 simulator.

Results of NS-2 simulations are shown in a tabular form by means of trace files. In a trace file, a timestamped line is produced for each data packet that travels on the network. In particular, the trace file can report the location of each node at a given time interval (for example each second). Thus, specifying for each node the initial position, and its speed, it is possible to trace the movement of the node during the whole simulation. For example, the following portion of an output trace file reports the x, the y and the z coordinates of the node `1` (i.e. `-Ni 1`) at the seconds 1, 2, and 3 of the

simulation time (i.e. `-t`).

```
...
 ... −t 1.000000000 ... −Ni 1 −Nx 601.29 −Ny 760.77 −Nz 0.00 ...
 ... −t 2.000000000 ... −Ni 1 −Nx 602.57 −Ny 761.54 −Nz 0.00 ...
 ... −t 3.000000000 ... −Ni 1 −Nx 603.86 −Ny 762.32 −Nz 0.00 ...
...
```

For sake of clearness, we report that this file is obtained by specifying an initial position of the (x, y, z) node's coordinates equal to $(600.00, 760.00, 0.00)$ respectively, and imposing a speed of the node of $1.5m/s$.

The NS-2 simulator can be used during the analysis and the design of a service oriented application. As described in Sec. 5, the PUPPET tool uses the trace files produced by the NS-2 simulator as models for the mobility of nodes. Thus, the mobility models generated during the phases of analysis and design can be reused in order to reproduce the considered mobility scenarios also in the testing phase of the `SUT`.

5 The Puppets in Motion

In this section we describe the module that plugs into the stubs code statements emulating services deployed on mobile devices (`Module 4` in Fig. 3). Mobility of devices on which software services are deployed has a significant impact during the assessment of the QoS properties of a `SUT`. In particular, in a mobile environment where actors can move, and services may appear or disappear, it is no longer possible to rely simply on the agreement contracted at the port of each service provider. The mobility of the nodes may in fact give rise to different perceptions of QoS properties even though all the services in the scenario are respecting the contractual agreements. For example, the availability that a client perceives of a service may vary according to their relative position, even though the reliability provided at service-side is always the same.

PUPPET emulates the mobility of the remote services equipping the generated stub with a hook able to interact with a mobility oracle. A mobility oracle is a service that takes in input a mobility model produced by the NS-2 simulator described in Sect. 4. This service exports an operation that taking in input the name of the nodes where two service are deployed, returns an index in $[0..1]$ measuring the damping factor in the QoS levels due to their relative distance. The index is 0 if the nodes are in the same position, else it is 1 if the two nodes (and the services deployed on them) are not visible each other. Values in $(0..1)$ give indications on their relative distance.

PUPPET injects into the generated stub a portion of Java code that interprets the damp index received from the mobility oracle. If the returned damp index is 1, a failure is generated as an exception raised by the platform hosting the Web Service stub. In this manner, the reliability of the `SUT` will depend not only on the reliability exposed in the QoS agreements by the composed services, but also on the availability of the services it composes, as simulated by the mobility model of a given scenario.

When the damp index returned by the mobility oracle is included in $(0..1)$, the Java code injected into the stub behaves emulating an additional latency that is function of the damp index. In this manner, the interactions between services deployed on different nodes that can move in a space, will affect the response time. The delta introduced in the latency is function of the damp index.

Finally, the mobility oracle returns a 0 damp index when two mobile devices are in the same location. In this case, the extra functional behavior defined in the agreement is not modified.

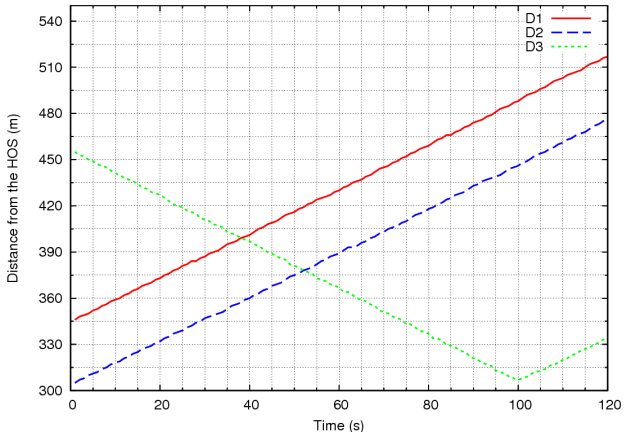
Note that both the functions computing the metric for the visibility and the damping factor can be specifically defined for each scenario. For example, in Sec. 6 we will implement the visibility function in terms of euclidean distance between the two nodes. Two nodes will be visible if and only if their relative distance is below a given threshold. Also, the damping index influencing the QoS agreed by the services will increase linear to the euclidean distance of the nodes.

For the sake of completeness, we note that in this version of PUPPET one only instance of the mobility oracle is shared among all the stubs of a testing scenario.

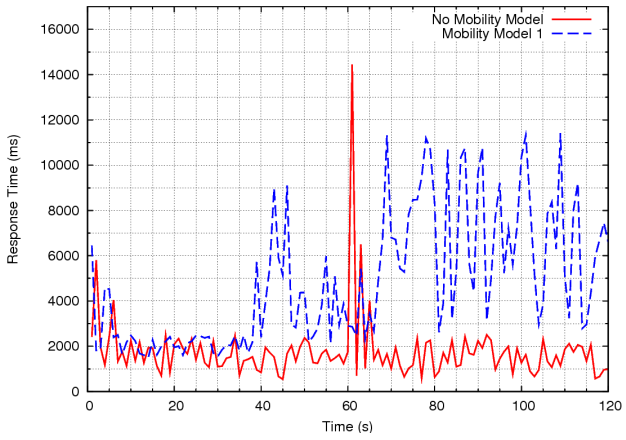
6 Experimental Results

In this section we present some experimentations we made on the motivating example we gave in Sec. 2, in order to illustrate the usefulness of mobility models in the assessment of QoS properties of a composite service. PUPPET was used in order to generate and to mock up testing environments for the chosen `SUT`, i.e. the HOS. Thus, we compare here the QoS properties measured at the port of the HOS assuming different execution scenarios. We assume that across the examined scenarios the composed services always respect the agreed QoS properties; the scenarios vary only in terms of their mobility models.

In particular, referring to the motivating example, PUPPET automatically derived stubs for the services S1, S2, S3, assumed as deployed on the doctors' PDAs D1, D2, D3 respectively. In the following we show how the mobility of doctors influences the HOS's estimated QoS properties. The results are provided in terms of service latency, and reliability of HOS in responding to a patient request. In a first experiment we measured the QoS properties of the HOS when no mobility was associated with the PDA (baseline model referred to as No Mobility). Then we repeated the same test execution adding to the simulated devices in the scenarios differing mobility models, to which we refer as Mobility Model 1, and Mobility Model 2. For illustrative purposes, Mobility Model 1 was defined to show vari-



(a) Distances of the Nodes from the HOS



(b) Response Time of the HOS

Figure 4. Testing with the Mobility Model 1

ations in the latency of the HOS; while Mobility Model 2 was defined to emphasize the variation in the reliability of the HOS.

Note that, in this example, we used only deterministic movements of the nodes. However, as already described in Sec. 4, more complex or randomly generated movements can be used.

As already introduced in Sec. 2, the HOS invokes consecutively the services S1, S2, and S3 looking for the first one that is available. The services S1, S2, S3 behave according to their respective WS-Agreement specifications. In particular, the response times R_1, R_2, and R_3 of the services S1, S2, and S3 are different with $R_1 > R_2 > R_3$. Also we consider that the WS-Agreement specifies that each service guarantees a reliability of max 3 failures in any time window of 1 minute. If no mobility models are used, both the latency and the reliability of HOS descend directly from the QoS provided by the accessed services S1, S2, and S3. We now introduce the mobility models; we assume that two services are mutually visible if the relative distance of the devices on which they are deployed is shorter than 400 ms. Furthermore, we assume that the latency of the interacting services is affected by a factor that is proportional to the distance of the nodes.

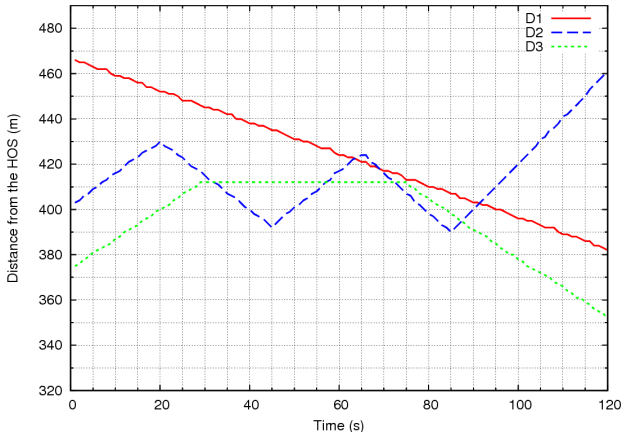
Fig. 4.a plots the distances of the doctors' PDAs from the HOS in the Mobility Model 1. Fig. 4.b depicts the measured response time of the HOS comparing the scenario without mobility models with the scenario with the Mobility Model 1. We can observe that initially the response time of the HOS with Mobility Model 1 is close to the one measured with the No Mobility model: the HOS tries to invoke service S1 which is available and which replies with a response time that is lower than the one of the other services. Around second 40, the service S1 becomes unavailable because of the PDA D1 is assumed to move away from the scope of the HOS device. Then, the HOS invokes service S2 which response time R_2 is higher than R_1. This can be also observed in the plots after second 60, when the response time of HOS grows as the only service available is S3, which has the highest response time. Note that, around second 60, either considering mobility or not, the response time of the HOS increases anyway as the HOS generates an error when none of the services invoked is available. Such failure was due to the simulation of the reliability conditions that the considered agreement includes.

Fig. 5.b, shows the HOS's reliability, computed as the number of failures in the unit of time. In particular, we compare the reliability obtained by considering Mobility Model 2, depicted in Fig. 5.a, in comparison with No Mobility. In both cases, around second 5, we see a low reliability value due to an error occurred in the invoked service S3. On the contrary, in the interval between seconds 20 and 80, while the exposed HOS's reliability is high if no mobility is considered, its value goes well below 100% when the doctors' PDAs move according with the Mobility Model 2. This happens because of within such mobility model a service is not always available in that time interval.

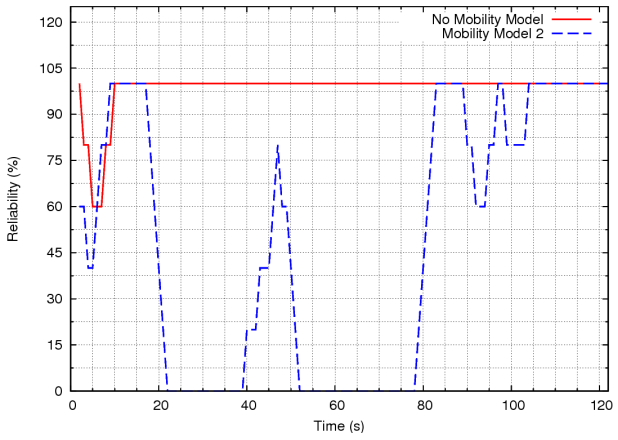
Although preliminary, the above examples show how the new version of PUPPET can be used to realistically test the QoS assessments of composite services in the context of mobile environments.

7 Related Work

The testing for QoS evaluation is approached in many works [6, 17] by addressing the derivation of a testing suite representative of the real usage scenario. In this direction, techniques considering context-changes in context-aware applications are recently developed [11, 16, 20]. In particular, the authors of [20], propose an approach for improving the test suite for ubiquitous applications. They aim at identifying context changes that can occur and affect the ap-



(a) Distances of the Nodes from the HOS



(b) Measured Reliability of the HOS

Figure 5. Testing with the Mobility Model 2

plication's behavior at any time during the execution, then they dynamically direct the application execution towards the generated context sequences.

Another research area focuses on the ad-hoc development of testing environments aimed at emulating mobility for evaluating performances of specific protocols and applications for mobile networks. Examples of such testing environments are presented in [13, 18, 19].

In this paper, we work at application level and we are interested in the automatic generation of testbed for QoS validation of context-aware service oriented systems. In particular, we focus on context changes due to movement of networked nodes. With reference to the generation of testbeds for validation of QoS, a work presenting some similarities with the approach presented here is [10] in which a performance testbed generator for service-oriented systems is presented. The main difference with our proposal is that the system proposed in [10] makes no use of any kind of contract or agreement specification, differently from the proposed approach which is based on QoS agreements as described in [4].

The interesting issue addressed in this paper, not approached in [4], is how the location changes of network nodes on which the interacting services are deployed, affect the QoS of the SUT. Indeed, the extra functional correctness of a service running on a networked device not only depends on its internal execution environment but also the external environments provided by the network where it is deployed. As amply discussed in the paper, the movement of nodes can change dynamically the interacting environment for the SUT.

Mobility in service-oriented testing research is still an open issue. Many works on mobile computing [8, 12, 14], propose solutions of logical mobility involving software, such as mobile code and mobile agents, that move between different servers and may use different sets of services on each of them. In particular, the author of [14] proposes a testing framework, called the Flying Emulator, where the physical mobility is emulated through logical mobility, by means of mobile agents.

As in our work, the main intent in [14] is the testing of a software in a mobile environment. Specifically, the author defines a framework that is able to migrate the SUT through different interconnected networks where various kind of services could be available. Thus, [14] assumes that the implementations of all the services are available at testing time. However, as we discussed, in SOA 3^{rd}-party services may not always available for testing purposes. Differently, PUPPET includes the automatic generation of the stubs for these kind of services.

In this paper we consider a mobility model generated by NS-2 simulator and used by our testbed to emulate the mobility of the remote services. However, our approach does not impose any restriction in the definition of the mobility models. In [5] the authors propose the Connectivity Trace Generator (CTG). CTG infers the mobility models from the traces of movements collected from scenarios already "in use". Such traces can be used as mobility test cases. In particular, as future work we could integrate the two approaches using the output traces from CTG in order to define the movements of the nodes in NS-2.

8 Conclusions

When dealing with mobile environments, the extra functional properties of a composite service may depend not only on the QoS features offered by the accessed services, but also on the configuration and on the movement of the nodes that host such services.

PUPPET is an environment for the automatic generation of stubs simulating the behavior of external services invoked by a SUT. The service developer can test the SUT,

without having to access the real surrounding services. In fact, it is often not possible to actually use third-party services for testing purposes (e.g. services not available, usage costs, unwanted side effects).

Nevertheless, an effective testbed generator that realistically emulates the runtime environment has to capture the mobility of the nodes. To achieve this, we assume that during the analysis and the design of a service-oriented application, the mobility models of the most interesting scenarios are specified. Each of such mobility models describes how the devices that host the relevant services are supposed to move at runtime.

The contribution of this work is an extension of PUPPET that includes a module for the emulation of device mobility. Specifically, the PUPPET tool links each generated stub with a mobility oracle. The mobility oracle determines the effects on the QoS properties emulated by the stubs as a function of the relative distance between nodes where the stubs are deployed.

Thus, in addition to the models describing the functional and the extra functional behavior of each service that is generated, PUPPET takes into account mobility models obtained with the NS-2 network simulator.

The NS-2 simulator is an open source network simulation tool widely used in the analysis and the design of protocols and applications for interacting network nodes. In the approach proposed in this paper, we used NS-2 only to simulate the movement of the nodes in a given space. As a future work, we intend to extend the integration between PUPPET and the NS-2 simulator by exploiting the more complex and advanced modeling functionalities of NS-2 in order to give a more precise characterization of the effects of mobility on QoS and to better address the testing of adaptable context-aware applications.

Acknowledgements. The authors wish to thank Andrea Polini for his important contribution to the research on PUPPET. This work was supported in part by the PLASTIC Project (EU FP6 STREP n. 26955) and in part by the TAS^3 Project (EU FP7 CP n. 216287).

References

[1] The network simulator NS-2 homepage. http://www.isi.edu/nsnam/ns/.

[2] Plastic tools homepage. http://plastic.isti.cnr.it/wiki/doku.php/tools.

[3] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services–Concepts, Architectures and Applications*. Springer–Verlag, 2004.

[4] A. Bertolino, G. De Angelis, L. Frantzen, and A. Polini. Model-based Generation of Testbeds for Web Services. In *Proc. of the 20th IFIP Int. Conference on Testing of Communicating Systems (TESTCOM 2008)*, LNCS. Springer Verlag, 2008. – to appear.

[5] R. Calegari, M. Musolesi, F. Raimondi, and C. Mascolo. CTG: a connectivity trace generator for testing the performance of opportunistic mobile systems. In *Proc. of the 6th joint meeting ESEC/FSE*, pages 415–424, September 2007.

[6] G. Denaro, A. Polini, and W. Emmerich. Early performance testing of distributed software applications. In *Proc. of the 4th International Workshop on Software and Performance (WOSP)*, pages 94–103, January 2004.

[7] L. Frantzen, J. Tretmans, and T. Willemse. A Symbolic Framework for Model-Based Testing. In *Formal Approaches to Software Testing and Runtime Verification – FATES/RV 2006*, number 4262 in LNCS, pages 40–54. Springer, 2006.

[8] A. Fuggetta, G. P. Picco, and G. Vigna. Understanding code mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, May 1998.

[9] Global Grid Forum. *Web Services Agreement Specification (WS–Agreement)*, version 2005/09 edition, September 2005.

[10] J. Grundy, J. Hosking, L. Li, and N. Liu. Performance engineering of service compositions. In *Proc. of International Workshop on Service-oriented Software Engineering (SOSE)*, pages 26–32, May 2006.

[11] H. Lu, W. K. Chan, and T. H. Tse. Testing context-aware middleware-centric programs: a data flow approach and an RFID-based experimentation. In *Proc. of the 14th International Symposium on Foundations of Software Engineering*, pages 242–252, 2006.

[12] G. C. Roman, G. P. Picco, and A. L. Murphy. Software engineering and mobility: A roadmap. In *Proc. of the Conference on The Future of Software Engineering (ICSE)*, pages 241–258, June 2000.

[13] S. Sanghani, T. Brown, S. Bhandare, and S. Doshi. EWANT: the emulated wireless ad hoc network testbed. In *Proc. of Wireless Communications and Networking Conference (WCNC)*, volume 3, pages 1844–1849, March 2003.

[14] I. Satoh. A testing framework for mobile computing software. *IEEE Transactions on Software Engineering*, 29(12):1112–1121, December 2003.

[15] The Apache Software Foundation. *Axis User's Guide*. http://ws.apache.org/axis/java/user-guide.html.

[16] T. H. Tse, S. S. Yau, W. K. Chan, H. Lu, and T. Y. Chen. Testing context-sensitive middleware-based software applications. In *Proc. of the 28th International Computer Software and Applications Conference*, pages 458–465, 2004.

[17] E. J. Weyuker and F. I. Vokolos. Experience with performance testing of software systems: Issues, an approach, and case study. *IEEE Transactions on Software Engineering*, 26(12):1147–1156, December 2000.

[18] Y. Zhang and W. Li. An integrated environment for testing mobile ad-hoc networks. In *Proc. of 3rd International Symposium on Mobile ad Hoc Networking & Computing (MOBIHOC)*, pages 104–111, June 2002.

[19] J. Zhou, Z. Ji, and R. Bagrodia. TWINE: A hybrid emulation testbed for wireless networks and applications. In *Proc. of the 25th International Conference on Computer Communications (INFOCOM)*, pages 1–13, April 2006.

[20] Z.Wang, S. Elbaum, and D. S. Rosenblum. Automated generation of context-aware tests. In *Proc. of the 29th International Conference on Software Engineering (ICSE)*, pages 406–415, May 2007.