2009 16th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems

A Systematic Process for Implementing Gateways for Test Tools

Jessica Díaz, Agustín Yagüe and Juan Garbajosa
Universidad Politécnica de Madrid - Technical University of Madrid (UPM)
E.U. Informtica Ctra. Valencia Km. 7 E-28031 Madrid, Spain
Email: {yesica.diaz,agustin.yague,juan.garbajosa} –at– upm.es

Abstract

Test automation is facing a new challenge because tools, as well as having to provide conventional test functionalities, must be capable to interact with ever more heterogeneous complex systems under test (SUT). The number of existing software interfaces to access these systems is also a growing number. The problem cannot be analyzed only from a technical or engineering perspective; the economic perspective is as important. This paper presents a process to systematically implement gateways which support the communication between test tools and SUTs with a reduced cost. The proposed solution does not preclude any interface protocol at the SUT side. This process is supported using a generic architecture of a gateway defined on top of OSGi. Any test tool can communicate with the gateway through a unique defined interface. To communicate the gateway and the SUT, basically, the driver corresponding to the SUT software interface has to be loaded.

1. Introduction

Nowadays, many of the common devices such as home devices or automotive electronics include embedded systems. These systems more and more often interoperate with information systems, resulting in what sometimes is called as systems of systems. At the same time more and more often happens that these systems may communicate through Internet enabling the vision of an Internet of Things (IoT). The Internet of Things concept [1] initially focused on RFID technologies and their applications, but later Smart Embedded Devices and Sensor Networks have entered the scene and can be considered as part of the IoT [2]. Smart Objects term is sometimes used to include RFID, Smart Embedded Devices, and Sensor Networks [3]. To support IoT testing, test tools must be capable to interact with systems under test made of ever more heterogeneous smart objects. The interaction between test tools and SUTs has been solved from technical and engineering point of view but current solutions lack of the required generality and scalability to test any SUT. For example, the TTCN-3 [4] standard, and tools such as FIT [5], FitNesse [6], Easyaccept [7], and TOPEN [8], [9] test any kind of software using façade components to support the interaction with the SUT; however these façade components are specific for each SUT. In fact, the problem was discovered when a test tool built in-house, TOPEN [8], had to be connected over and over again with different SUTs, and the cost of this connection was higher than the adaptation of TOPEN to a new domain itself. Therefore, the motivation of this work is the lack of general purpose solutions to support the interaction between test tools and SUTs. A general solution could include flexible adapters or gateways or, even better, the gateway and the process to produce or to adapt the gateway in a systematic way.

Services engineering and Service oriented architecture (SOA) have become essential drivers to integrate smart objects into the IT-landscape. Traditionally, in home and industrial automation environments, communication between the individual system components or devices is supported by a central system in a hierarchical network. From centralized to distributed systems, there is an evolution where all devices would publish their capabilities in the form of high-level services, i.e., known commonly as device-level SOA [10]. At present some kind of convergence is taking place in the direction of web services, but several other non-service oriented interfaces are alive and widely used being necessary to maintain the compatibility with those systems that do not support high-level services. Home automation is an example of the integration of different smart devices, networks, and services. So, a wide range of technologies live together in home automation such as HAVi [11], LonWorks [12], Konnex [13], X.10 [14], or HomePNA [15], and others as Bluetooth [16], UPnP [17] or even conventional web services. Though it can be expected that in the future interfaces become more and more uniform or standardized. That is, at present, though it may be possible to add a connector built-in with web-services for any device or application, this has to be produced for each device each time [3]. Therefore the main issue related to the interaction problem between test tools and SUTs is not only technical or engineering but basically economic.

This paper presents a process to systematically implement gateways which support the interaction between test tools and SUTs with a reduced cost. To achieve this goal the process uses a generic and configurable architecture of a gateway hosted on top of an OSGi (Open Service Gateway initiative) Platform [18][19]. The design guidelines for such

978-0-7695-3602-6/09 $25.00 © 2009 IEEE
DOI 10.1109/ECBS.2009.40

a gateway were presented in [20]. Basically, the architecture has two well defined interfaces: the interface to the test tool and the interface to the SUT (interfaces have been described break down in section 3.3). The interface to the test tool is fixed and public so that any test tool can get access to the gateway in the same way. The interface to the SUT focuses on devices and device drivers. The interaction between the gateway and a SUT device simply requires that the adequate driver for the SUT device interface is loaded in the OSGi Platform. The drivers can be loaded in runtime thanks to OSGi capabilities. The proposed solution does not preclude any interface protocol at the SUT side.

As the gateway architecture variability is identified, the process requires only to modify this variability part for each new SUT device. Once a device is identified, what could be understood as service discovery, the test tool can be configured based on the information obtained from the service (operation) offered by this device. The described approach simplifies tremendously the process of interaction problem between test tools and SUTs. The validation of the proposed solution was performed using TOPEN, a domain-oriented acceptance testing environment built in-house.

The remainder of the paper is structured as follows. Background and related work are analyzed in section 2. The systematic process to define and implement a gateway is described in section 3. Section 4 presents a study case. Finally, some conclusions and future work are presented in section 5.

2. Background and Related Work

The communication problem between test tools and systems under test (SUTs) could be faced from an interoperation perspective. Service-orientation is growing up and is becoming a predominant approach to enable software interoperability for heterogeneous complex systems [21], [22]. Examples such as Device Profile for Web Services (DPWS) [23], OSGi, Java Intelligent Network Infrastructure (JINI) [24], and Universal Plug and Play (UPnP) [17] show the current trend. One approach to solve a rigid schema for the interoperation between test tools and SUTs is based on a middleware network to which SUT and application gateways can connect using web-services. The problem is that plugins for different protocols have to be implemented as needed [25]. Though this solution is scalable, each plugin have to be implemented separately in a non-systematic way. And this is the situation that, precisely, is to be avoided.

Cumulus [26] introduces a service-oriented architecture facilitating maintenance and administration of a distributed customized middleware for Web services applications according client interoperability requirements, this means, clients can use middleware as services. However, this solution is not always supported by sparse embedded devices, e.g. home automation devices are featured by sparse resources and low processing power and solutions as web services are unsatisfactory because they put high demands on the embedded devices [27]. Other approaches have been driven to service-oriented infrastructures based on the Device Profile for Web Services (WS-DP or DPWS) [28]. DPWS favours the adoption of the SOA paradigm in the embedded-device supporting the integration of device-provided services in enterprise-wide application scenarios. This solution is based on peer-to-peer interactions between devices-level SOA connected over a common network infrastructure using IP-based network protocols. A first step towards DPWS adoption is the implementation of middleware components as a bridge between manufacturers' native code, usually proprietary, and Web services. At present this issue has not still achieved.

Nowadays the OSGi Platform has attracted the interest of numerous researchers. OSGi is an initiative focused on the interoperability of applications and services based on its component integration platform (Service Platform) providing a service-oriented, component-based environment. OSGi provides loosely coupling to components, scalability, portability, and the capability to add, remove or modify dynamically services without significant effort or disrupting operations. Modular development and hot service deployment are key characteristics for our commitment to OSGi technology. It is being used to support the implementation of distributed services and components [29]. Since OSGi appeared in 1999, it has been seen as one of the alternatives to support residential gateways. Many of these works are focused on eHome services [30], [31], eHealth services [32], or vehicular services [33], providing the users with an abstract view of the system. These approaches are particular examples for the OSGi functionality and, unlike our approach, are domain specific and do not provide such flexible interoperation infrastructure to communicate external tools with complex systems. However this, an analysis of OSGi shown us the possibility of taking it as a basis for our objective. The key issue was to notice that OSGi could be used to define an architecture that could be viewed as having a front end and a backend, with well identified variability points.

A first step in this direction was presented in [20]. This paper means one step ahead adding the definition of a systematic process to adapt the gateway for testing any system. The ability to implement highly adaptable software components is one way to capitalize on the commonality within software families [34]. So, the adaption process of the generic gateway improves the productivity achieving high levels of reuse versus current solutions based on specific gateways. A major issue, however, is the management of variability. For this, formal methods for representing and managing variability are required [35].

3. Gateway systematic implementation process

3.1. Gateway abstract model

The gateway model has to consider how the interaction between test tools and SUTs is done. An analysis allowed to understand that acceptance test cases definition, though systems are from very diverse domains, has a lot in common as far as the basic operation concerns. This means, from the perspective of a test engineer that interacts with the SUT, it is possible to identify some common and domain-independent patterns; these patterns support the system operation. That is the case of message exchange of commands/responses, and event publish/subscribe pattern. A semi-automated procedure for identifying these patterns is being defined; a first step has been already described in [36].

As a result, the gateway abstract model is structured into two components, a frontend and a backend, with a well defined internal interface. Also, the gateway presents two external interfaces: one supports the interaction with test tools (frontend) and other supports the interaction with SUTs (backend). Figure 1 depicts the abstract model of the gateway as well as its interaction with test tools and systems under test.

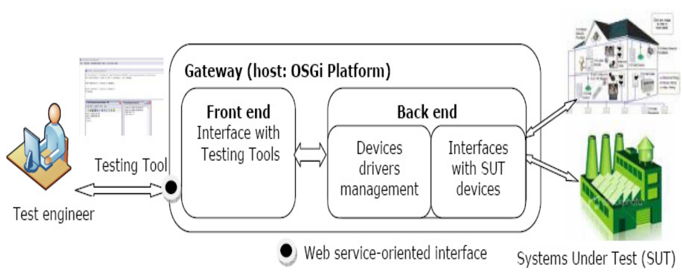

Figure 1. Gateway abstract model

The *frontend* provides a fixed and public web interface for test tools that any test tool will use. We have validated this interface with TOPEN but any test tool could connect to the gateway using this interface. So the test cases, defined from one of these test tools, should be defined in terms of:

1) Mapping between physical devices and test tool logical entities, i.e., represented in a gui.
2) Test command execution requests.
3) Event notification subscription requests.
4) Finally, actions to start and stop listening event notifications.

While the *backend* has been designed to systematically integrate smart devices based on any interface as well as conventional web services. Implementing the gateway basically consists in identifying drivers for a given interface and installing them into OSGi Platform. These drivers are shared by the OSGi community. Obviously, in case a driver is not available it would have to be written.

The systematic process proposed to implement gateways for test tools uses this abstract model as starting point. The process is described in the next section.

3.2. Systematic process overview

The systematic process distinguishes two types of elements: devices and the drives. Devices represent those components that a SUT is made of. Drivers are software components that support the interface with the device, i.e., device native code. Basically the systematic process has three parts: (i) device register, (ii) driver register, and (iii) test tool interface implementation. The process exploits the commonality across the gateways encouraging the systematization of the adaption process of the generic gateway for testing any system. This process is a sequence of seven well defined steps:

1) Identify the devices of which SUT is made of.
2) Check those devices registered in the gateway.
3) Register in the gateway those non-registered devices.
4) Identify the device drivers. An appropiate driver is needed for each software/hardware interface supported by the system under test.
5) Check those drivers registered in the OSGi Platform.
6) Install and register in the OSGi Platform those non-registered drivers.
7) Implement a web client interface to support the communication between the test tool and the gateway.

The gateway architecture is the basis to provide the systematic process for adapting the gateway to the SUT. Since variability has been perfectly identified, changes on the SUT are integrated into the gateway systematically.

3.3. Gateway architecture bundles

In this work the objectives of the gateway architecture are: (i) to facilitate that the development of the gateway can be performed systematically, (ii) to provide a uniform service oriented interface to the test tool, and (iii) to integrate the interface heterogeneity of system devices. All these items are addressed to reduce the impact of designing a specific gateway for each specific SUT -in general- or device -in particular-.

The architecture supports service, message and event-based interactions and has been structured in four layers, as it has been shown in figure 2. One of these layers is the OSGi Platform, in particular an OSGi Platform implementation called Knopflerfish.

Each functionality is described in the architecture by one or more components (or OSGi bundles). Figure 3 shows breaks down the gateway components architecture in terms of an UML component diagram.

The **drivers management** function is represented by the bundles Driver Service, Driver Factory and Driver (see

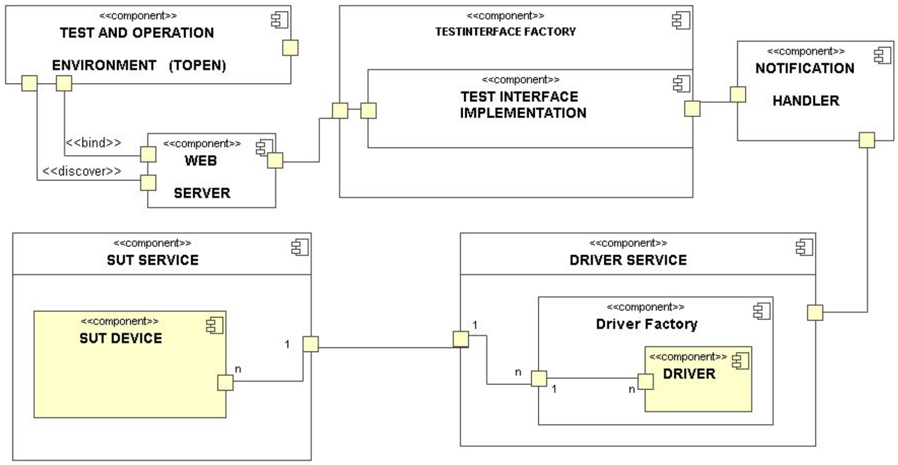

Figure 3. Component diagram

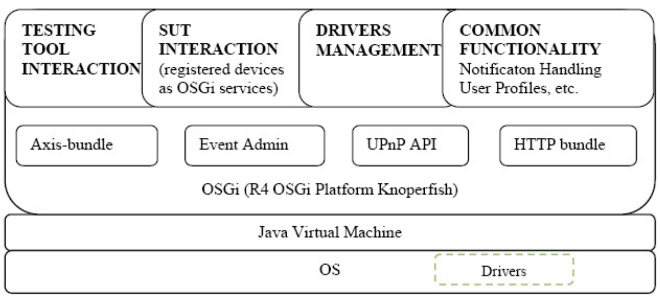

Figure 2. Architecture Layers

Figure 3). These bundles are in charge of managing the different drivers that support the device interfaces. For each device interface (supported by a specific technology) Driver Service implements a driver's factory. For example: UPnP, Bluetooth, Sockets, etc.

The **SUT interaction** is represented by the bundles SUT Service, and SUT Device (see Figure 3). These bundles represent the physical structure of the SUT, i.e., registered devices as OSGi services, and manage the binding between devices and their specific drivers. The SUT Service uses the services offered by a specific driver and exports the commands, responses, and notifications supported by the device/SUT. For example, the devices of a Home Automation System (eg. a heating system or a TV) support commands (eg. turn on or shut down) and may notify an operation failure.

The **notification handling** is represented by the bundle Notification Handler (see Figure 3). Notification Handler manages notifications and alarms associated to certain device events such as device discovery or operation failures. The physical system produces notifications whereas the SUT Service bundle consumes the notifications and spread them to the test tool through the Test Interface bundle.

And finally, the **testing tool interaction** is represented by the bundles Test Interface Factory, Test Interface Implementation, and Web Server (see Figure 3). Web Server manages the interaction between test tools and the gateway using Web Services. Test Interface implements the accessible SOAP services such as commands sending, response receiving, or managing notifications by test tools. Therefore any test tool will get access to the gateway through this uniform interface which provides the methods needed to execute a test case (see Code 1):

Code 1 Gateway-TestTool interface methods

```
mapping(logicalID, physicalID)
executeTestCommand (commando)
suscribedToNotifications(notification)
startListeningNotifications
stopListeningNotifications
```

The *mapping* method allows the association between a device, that a test engineer may test from the graphical user interface of a test tool, and the physical device (identified by a IP address, a urn [1], etc.). When the test engineer runs a test case on a system under test, the *executeTestCommand* method runs a command on a device. The *suscribedToNotifications* method supports the handling events from system under test, such alarms, errors, etc. The *startListeningNotifications* method allows the asynchronous reception of events (subscribed previously) from the system under test, so that the test tool is able to detect operation failures (alarms, critical states, etc.). Finally the *stopListeningNotifications* method stops the event reception. Any test tool can access the gateway through a web service client by means of this interface.

1. Uniform Resource Name

3.4. Variability modelling

From this architecture, it is possible to conclude that the functionalities *interaction with a test tool* and *notification handling* are commonalities of the gateway, independently of the SUT domain, but the components that implement the functionalities *drivers management* and *interaction with the SUT* will be replaced depending on the SUT devices that will be teseted. As a result, these components will be part of the variability; this variability must be managed, such that a variability point can be extended according to the device driver. This driver corresponds to the interface. This commonality and variability can be modeled using product-line techniques [37], [38]. Below, a formalized variability expression for a generic gateway is presented using a notation that defines all possible configurations for adapting the gateway to different devices in different domains: a home automation system and a slot machine system. The notation supports dependencies and constraints (*requires* clause) between variable features. Group cardinalities indicate an exclusive (*one-of* clause) choice or non-exclusive (*more-of* clause) choice:

```
  all(WebServer,
 TestInterface,
 NotificationHandler,
   one-of (more-of (printer, mobile, hifi-system, TV),
           more-of (PLC, slot-machine))
   one-of (more-of (UPnP, Bluetooth, HomePNA, HomeRF),
           more-of (sockets, Modbus, CANBus, Profibus)),
)
printer requires UPnP
mobile requires Bluetooth
hifi-system requires (one-of (HomePNA, HomeRF))
PLC requires sockets
slot-machine requires CANBus
```

3.5. Gateway construction process

Following the systematic process defined in section 3.2, the adaptation process of a gateway to test a particular device requires the implementation of three only components:

(i) A new Driver bundle (defined in section 3.3) must be implemented for each new existing device interface in the system under test. Driver bundles implement the *Driver Service interface*. The complexity of these bundles depends directly of the complexity of the technology supported by the interface.

(ii) A driver factory has to be implemented for each driver but it consists in a few code lines that support the factory pattern.

(ii) For testing a new SUT (e.g. a home automation system), a new SUT Service bundle (defined in section 3.3) has to be implemented. Its implementation is quite simple: the SUT Service bundle implements the *SUT service interface* and requires a few lines of code to support the descriptions of the devices of which SUT is made of. Initially the SUT could be made of zero, one, or more devices. Devices can be added to the system in a dynamic way since the *dynamic-device* bundle support dynamic registration of devices. If the Driver bundle for a particular device has already been installed, the *dynamic-device* bundle registers this device sending the test tool a notification of a new connected device. If the Driver bundle for this particular driver has not been installed, this Driver bundle must be installed manually in the OSGi platform that hosts the gateway. It is important to emphasize that this process does not require to stop the gateway execution.

Commands and notifications supported by the devices are inputs for the gateway. That is, the gateway receives the specific operations/services descriptions (operational interface) supported by each specific device (figure 4). The Driver bundle parses the operational interface definitions (e.g. WSDL) generating an output in a standard format (see Code 2). It was decided that the use of XML scripts was the right approach in terms of simplicity and flexibility. The operational interface definition is an XML string stream with several attributes, method descriptions and notifications which the test engineer could monitor, execute or subscribe to.

Code 2 Operational interface definition

```
<actionList>
<action>
  <name>SetTime</name>
  <argumentList>
  <argument><name>NewTime</name></argument>
  <return><name>Result</name>  </return>
  </argumentList>
</action>
</actionList>
```

Therefore, new devices are integrated systematically because to add a new device basically implies to register the new device and to load a new driver in OSGi Platform.

4. Case Study

As mentioned in the Introduction, the validation of the proposed solution is performed using a particular test and operation environment developed in-house called TOPEN [8]. This section introduces a particular implementation of the gateway for testing a home automation system implementing only the variation points.

4.1. The Test Tool

TOPEN provides mechanisms for the definition and execution of operation and test cases through a domain specific language. This means that TOPEN supports test cases specification through a domain-specific language in the direction

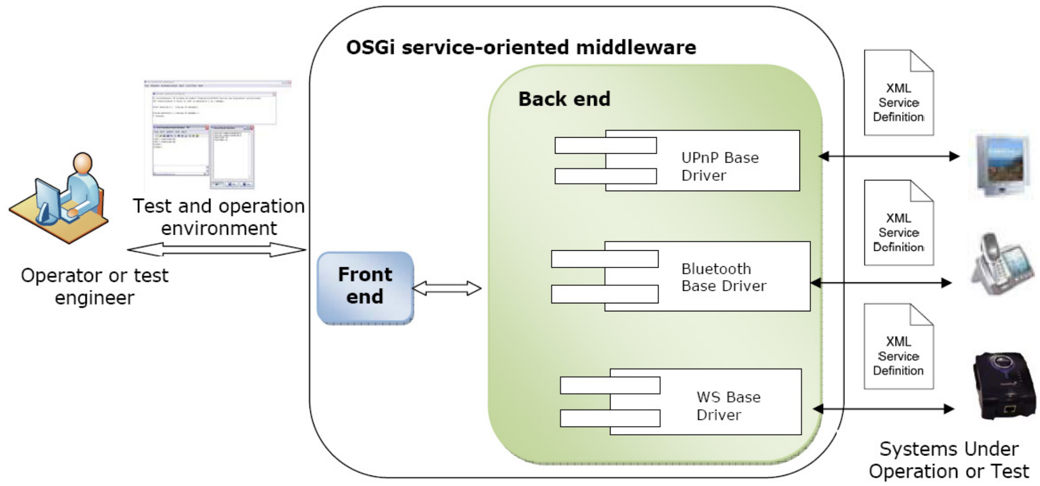

Figure 4. Deployment of the gateway architecture overview

of [39], [40]. The TOPEN architecture is independent of the application domain; however a TOPEN product is specific for a particular domain. However there still exist a component that must be re-implemented when the system domain changes or when a new devices is added. The referred component (gateway) implements the communication with the SUT. The proposed gateway architecture was designed to reduce the effort required to adapt this gateway to new system domains or new devices.

4.2. A home automation system

In a home automation system, an operator could start and stop a device, set the value of an attribute or get the value of an attribute. Asynchronously, the home automation system could notify the completion of an operation such as the electric lighting, the temperature change of the air-system etc., or a stranger presence through security camera. TOPEN manages all these notifications and warns the operator about these events.

The implementation of this study case has been achieved through a simulator that implements the behavior and the software interface. So, the validation of the proposed gateway is performed using a simple but complete *UPnP ricecooker* simulator developed in-house following the specification available in [41].

From the Gateway-SUT interaction point of view, the variation points are deployed in the gateway in terms of bundles that can be registered or unregistered from the OSGi service registry at any time. As it will be mentioned in section 3.3 the Driver bundle is domain specific, and therefore a new Driver bundle must be implemented: *UPnP Driver bundle* implements the Driver Service interface and exports specific services for an UPnP device (see Code 3). The Driver bundle must be installed in the OSGi platform. Once the UPnP ricecooker is networked, the gateway will be able to register the ricecooker.

Code 3 Driver bundle implementation.

```
class UpnpDriver implements DriverServiceInterface, Runnable {
  private boolean notify=true;
  public UpnpDriver(DeviceListener deviceProvider) {
    deviceProvider.setDriver (this);
    ref = DriverManagerActivator.getContext().
            getServiceReference(EventAdmin.class.getName());
    eventAdmin = (EventAdmin) DriverManagerActivator.
            getContext().getService(reference);
  }
  public String sendCommand(String command) {
    Device dev =deviceProvider.search (uid);
    [...]
  }
  [...]
}
```

From the TOPEN-Gateway interaction point of view, a first step implies setting explicitly the relationship between a device (SUT) model, represented graphically at TOPEN GUI, and a physical device. This mapping intends to establish the correspondence between logical devices (devices that can be observed by the tester at the TOPEN GUI) and physical devices. This process requires a binding with the *mapping* service (see Code 4) with the parameters: “myricecooker” and “urn:schemas-upnp-org:service:ricecooker:1”. TOPEN may send the ricecooker a command for setting the ricecooker mode and then a command for checking that the mode is the expected mode. This process requires that TOPEN executes a call to the *executeTestCommand* service, defined in section 3.3, with the parameter “send set ricecooker mode fast” (see Code 4). Then TOPEN calls to the *executeTestCommand* service with the parameter “send get ricecooker mode” (see Code 4). TOPEN will look forward to the ricecooker responses. Also, TOPEN may send the ricecooker a command for waiting the warmlamp lighting (see specifications available in [41] specification). This process requires that TOPEN executes a call to the *suscribedToNotifications* service, defined in section 3.3, with the parameter “warmlamp ON” (see Code 4). Then TOPEN executes the *startListeningNotifications* service for listening notifications from the riceccoker, and may execute the *stopListeningNotifications* service for stopping the notifications reception. These examples show that the same calls are required if the ricecooker is supported by UPnP, X.10, Bluetooth, or any other protocol.

Code 4 TOPEN invocation of the gateway services.

```
// Make a service
service = new CommunicatorATTimplServiceLocator();
// Now use the service to get a stub
port = service.getremoteGateway();
urn = "urn:schemas-upnp-org:service:ricecooker:1"
answer = port. mapping("myriceccoker", urn);
param = "send set ricecooker mode fast";
answer = port.executeTestCommand(param); [...]
param = "send get ricecooker mode";
answer = port.executeTestCommand(param); [...]
notification = "warmlamp ON";
answer = port.suscribedToNotifications(notification);
answer = port.startListeningNotifications();
[...]
answer = port.stopListeningNotifications();
}
```

An example of a test for a ricecooker is shown in the figure 5. Figure 5 shows a test procedure in a window at the upper right corner of the display. TOPEN compiles and executes each test procedure, and sends the command to the ricecooker through the gateway. The results of the execution are shown in a window at the upper left corner of the display. Figure 6 shows the sequence diagram for the get temperature command, get time command, etc. in detail. For example, the wait command acts on the Notification Handler to identify which notifications, relevant to its execution, must be sent to TOPEN. The end of the wait command will be indicated by a notification; once the notification is sent, TOPEN can continue the execution of the rest of the commands.

5. Conclusions and Further Work

This work has highlighted the need for a systematic process to build gateways that connect test tools and complex SUTs, such as those available nowadays. These SUTs may be belong to any of the multiple existing domains, either service oriented, including different interfaces, or not, but the gateway should support all the cases to be really useful to the test engineer. This systematic process is supported by a generic architecture, as a natural approach for the gateway design. This generic architecture is described. A key issue is that a uniform, service oriented, interface between for the tool to access the gateway has been specified. The implementation of a gateway from this architecture following the described process is possible without a significant effort.

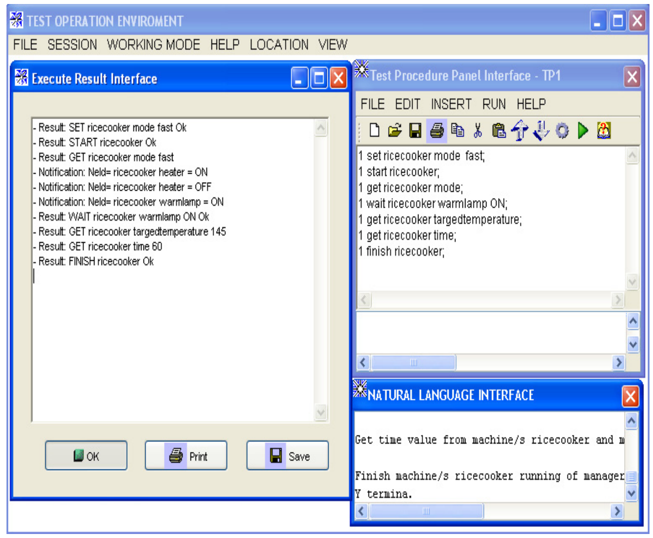

Figure 5. TOPEN GUI for the rice cooker.

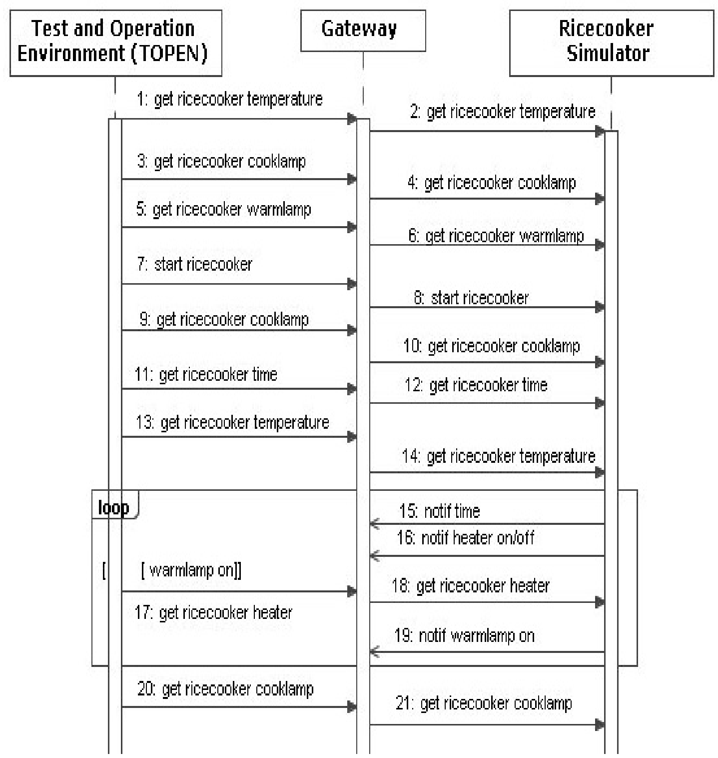

Figure 6. Communication protocol.

The interface between the tool and the gateway, regarded as frontend, implements the services to interface test tools while the backend integrate heterogeneous smart devices in dynamic complex systems notifying asynchronously a new device discovery. Through a device operation XML description, the backend provides the frontend with the commands and notifications that a test engineer (through a test tool) could execute and subscribe to. The adaptation process of the gateway for testing new devices requires that some well identified bundles, representing device drivers, are added to the gateway. That is variability is perfectly identified and managed.

As future work two directions are being followed. The first concerns the consolidation of the interface between the tool and the gateway. The objective would be to standardize it, so that a test tool output could be expressed in terms of this interface services. Second, the current design of the gateway supports several SUTs devices, but the current implementation only one. This implementation will be extended to support several devices within the same gateway.

Acknowledgment

The work reported here has been partially sponsored by the Spanish "Ministerio de Educación, Ciencia y Cultura" within the OVAL/PM TIC2006-14840 project, "Ministerio de Industria, Turismo y Comercio" within the FLEXI FIT-340005-2007-37 (ITEA2 6022) and UPM under their Researcher Training program. Authors are indebted to Jennifer Pérez and Pedro P. Alarcón for their comments.

References

[1] ITU, "ITU Internet Reports 2005: The Internet of Things," International Telecommunication Union (ITU), Geneva, 2005.

[2] A. Reinhardt, "A machine-to-machine internet of things," in *Business Week*, April 2004.

[3] L. M. S. de Souza, P. Spiess, M. Koehler, D. Guinard, S. Karnouskos, and D. Savio, "Socrades: A web service based shop floor integration infrastructure," in *Internet of Things 2008 Conference*, Zurich, Switzerland, 2008.

[4] TTCN-3, "Etsi european standard (es) 201 873-1." *TTCN-3: Core Language. Version:3.2.1, web http://www.ttcn-3.org/StandardSuite.htm*, 2005-06.

[5] FIT, "Fit acceptance testing framework," web http://fit.c2.com. [Online]. Available: http://fit.c2.com

[6] Fitnesse, "Fitnesse," web http://www.fitnesse.org. [Online]. Available: http://www.fitnesse.org

[7] J. P. Sauvé, O. L. A. Neto, and W. Cirne, "Easyaccept: a tool to easily create, run and drive development with automated acceptance tests," in *AST '06: Proceedings of the 2006 international workshop on Automation of software test*. New York, NY, USA: ACM Press, 2006, pp. 111–117.

[8] J. G. B. Magro and J. Perez-Benedi, "A software product line definition for validation environments," in *SPLC Confernce 2008: Software Product Lines Conference*, 2008.

[9] P. Alarcon, J. Garbajosa, A. Crespo, and B. Magro, "Automated integrated support for requirements-area and validation processes related to system development," *Industrial Informatics, 2004. INDIN '04. 2004 2nd IEEE International Conference on*, pp. 287–292, June 2004.

[10] I. Delamer and J. Lastra, "Loosely-coupled automation systems using device-level soa," *Industrial Informatics, 2007 5th IEEE International Conference on*, vol. 2, pp. 743–748, June 2007.

[11] HAVi, "Home audio video interoperability," Web http://www.havi.org. [Online]. Available: http://www.havi.org

[12] Echelon-Co., "Introduction to the lonworks platform: An overview of principles and practices v2.0," 2001.

[13] Konnex, "Knx specification, version 1.1," Diegem, 2004.

[14] X.10, Web http://www.x10.org. [Online]. Available: http://www.x10.org

[15] HomePNA, web http://www.homepna.org. [Online]. Available: http://www.homepna.org

[16] Bluetooth, Web http://www.bluetooth.com. [Online]. Available: http://www.bluetooth.com

[17] UPnP, "Universal plug&play." Web http://www.upnp.org. [Online]. Available: http://www.upnp.org

[18] OSGi, "Osgi alliance," Web http://www.osgi.org/Main/HomePage. [Online]. Available: http://www.osgi.org/Main/HomePage

[19] OSGiAlliance, "Osgi service platform core specification." *The OSGi Alliance Release 4, Version 4.0.1*, July 2006.

[20] J. Diaz, A. Yagüe, P. P. Alarcon, and J. Garbajosa, "A generic gateway for testing heterogeneous components in acceptance testing tools," in *ICCBSS '08: Proceedings of the Seventh International Conference on Composition-Based Software Systems (ICCBSS 2008)*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 110–119.

[21] M. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann, "Service-oriented computing research roadmap," in *Technical report/vision paper on Service oriented computing European Union Information Society Technologies (IST)*, 2006.

[22] M. Papazoglou, *Web Services: Principles and Technology*. Prentice Hall; 1 edition (September 23, 2007), 2007.

[23] S. Chan, D. Conti, C. Kaler, T. Kuehnel, A. Regnier, B. Roe, D. Sather, J. Schlimmer, H. Sekine, J. Thelin, D. Walter, J. Weast, D. W. D. Whitehead, and Y. Yarmosh, "Devices profile for web services," *Microsoft Developers Network Library*, Feb. 2006. [Online]. Available: http://specs.xmlsoap.org/ws/2006/02/devprof/devicesprofile.pdf

[24] Jini, "Jini specs. and api archive." web http://java.sun.com/products/jini/. [Online]. Available: http://java.sun.com/products/jini/

[25] V. Miori, L. Tarrini, M. Manca, and G. Tolomei, “An open standard solution for domotic interoperability,” *Consumer Electronics, IEEE Transactions on*, vol. 52, no. 1, pp. 97–103, Feb. 2006.

[26] E. Wohlstadter, S. Tai, T. Mikalsen, J. Diament, and I. Rouvellou, “A service-oriented middleware for runtime web services interoperability,” *Web Services, 2006. ICWS ’06. International Conference on*, pp. 393–400, Sept. 2006.

[27] E. Zeeb, A. Bobek, H. Bonn, and F. Golatowski, “Lessons learned from implementing the devices profile for web services,” *Digital EcoSystems and Technologies Conference, 2007. DEST ’07. Inaugural IEEE-IES*, pp. 229–232, Feb. 2007.

[28] H. Bohn, A. Bobek, and F. Golatowski, “Sirena - service infrastructure for real-time embedded networked devices: A service oriented framework for different domains,” *Networking, International Conference on Systems and International Conference on Mobile Communications and Learning Technologies, 2006. ICN/ICONS/MCL 2006.*, pp. 43–43, April 2006.

[29] F. Cuadrado, J. Duenas, J. Ruiz, J. Bermejo, and M. Garcia, “An open source platform for the integration of distributed services,” *Advanced Information Networking and Applications - Workshops, 2008. AINAW 2008. 22nd International Conference on*, pp. 1422–1427, March 2008.

[30] M. Kirchhof and S. Linz, “Component-based development of web-enabled ehome services,” *Personal Ubiquitous Comput.*, vol. 9, no. 5, pp. 323–332, 2005.

[31] M. Ditze, G. Kamper, I. Jahnich, and R. Bernhardi-Grisson, “Service-based access to distributed embedded devices through the open service gateway,” *Industrial Informatics, 2004. INDIN ’04. 2004 2nd IEEE International Conference on*, pp. 493–498, June 2004.

[32] P. de Toledo, M. Galarraga, I. Martinez, L. Serrano, J. Fernandez, and F. Del Pozo, “Towards e-health device interoperability: The spanish experience in the telemedicine research network,” *Engineering in Medicine and Biology Society, 2006. EMBS ’06. 28th Annual International Conference of the IEEE*, pp. 3258–3261, 30 2006-Sept. 3 2006.

[33] C. Pinart, I. Lequerica, I. Barona, P. Sanz, D. Garca, and D. Snchez-Aparisi, “Drive: a reconfigurable testbed for advanced vehicular services and communications,” in *1st Workshop on Experimental Evaluation and Deployment Experiences on Vehicular networks (WEEDEV). In Proc. TRIDENTCOM08*, Innsbruck, Austria, 2008.

[34] I. S. staff, “Product line engineering: The state of the practice,” *IEEE Softw.*, vol. 20, no. 6, pp. 52–60, 2003.

[35] A. v. Deursen and P. Klint, “Domain-specific language design requires feature descriptions,” *Journal of Computing and Information Technology*, vol. 10, no. 1, pp. 1–17, 2002. [Online]. Available: http://www.cwi.nl/ arie/papers/fdl/fdl.pdf

[36] P. P. Alarcón and J. Garbajosa, “Identifying application key knowledge through system operations modeling,” in *IEEE ICCI*, 2007, pp. 246–254.

[37] J. Bosch, *Design and Use of Software Architectures Adopting and Evolving a Product-Line Approach*. Addison-Wesley, 2000.

[38] P. Clements and L. Northrop, *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.

[39] H. Wu and J. Gray, “Automated generation of testing tools for domain-specific languages,” in *ASE ’05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*. New York, NY, USA: ACM, 2005, pp. 436–439.

[40] ——, “Testing domain-specific languages in eclipse,” in *OOPSLA ’05: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. New York, NY, USA: ACM, 2005, pp. 173–174.

[41] T. Vinh, “Cosmic-ffp, 2000, case study, rice cooker, version 2.1, january 14, semrl, uqam,.” [Online]. Available: http://www.gelog.etsmtl.ca/cosmic-ffp/casestudies/version2.1ricecookercasestudy.pdf