UML-based Qualification of COTS Components

Franck Barbier

PauWare Research Group
BP 1155
64013 Pau CEDEX, France
Franck.Barbier@PauWare.com

Abstract. There is a dilemma between, on the one hand, easily and straightforwardly acquiring and integrating external, prefabricated, cost-effective, plug & play components and, on the other hand, the strong need for assessing and possibly accepting these components so that their incorporation into in-house products is not the source of any damage. This paper supplies some technical leads based on the concomitant and coherent use of the Unified Modeling Language (UML) and the programming mechanism of reflection. As an illustration, a concrete approach developed with the Component+ project is described. This approach relies on the Built-In Test (BIT) technology that aims to guide COTS component design that facilitates qualification.

Keywords: Componentware, Quality analysis and evaluation, UML.

1 Introduction: Reusing COTS Components

The increasing availability of COTS components over the Internet makes the structuring of an e-market now credible and realistic [1]; that is to say, the access to rich sites/repositories of varied components with dedicated search engines, elaborate and useful documentation that helps to foster the acquisition and integration of COTS software in in-house products. Such experiments already exist (see for instance www.ecots.org) but deficiencies still remain in the sense that COTS components are only exposed through descriptive and intrinsic attributes (OS compatibility, editor, general-purpose category as multimedia for instance…). Even if this information is important, there are, however, no accurate and reliable data about the way by which such external entities may be technically and rigorously incorporated into larger systems. As stressed by Szyperski *et al.*: "Components are for composition" [2]. This means that different properties have to be known; for instance, how to setup an acquired component according to the deployment environment in which it is executed? Will this acquired component behave the same in

both collaboration pattern *X* and in collaboration pattern *Y*? More generally, how to assess, under different reuse angles and perspectives, COTS components that in essence keep potential reusers in a state of distrust. In this scope, qualification amounts to checking COTS components in deployment environments instead of, in development environments only. This calls for special testing properties that may be used within a deferred phase by reusers in order to put into practice qualification.

More generally, many factors slow down the rapid development of a COTS component marketplace. One of these factors is the well-known NIH (Not Invented Here) syndrome. In order to transform foreign components into trustworthy components, developers need methods and tools that give them a high level of understanding. The need for such comprehension may, however, be in conflict with the generally close nature of COTS products. Too much derived knowledge may indeed be incompatible with vendors' constraints, as for instance, the protection of intellectual rights or the respect of security rules.

A paradox is thus clear and evident. In contrast to home-made software, COTS components are encapsulated software units and, as such, hide many parts of their implementation. Even if this is safe with regard to the principles of modularity, high cohesion and low coupling, we expect to determine how these external entities may behave in users' deployment environments which are often very different from vendors' development environments. The overall challenge of COTS components may thus be simply and typically summarized as follows: Moving up from sorcery (Fig. 1) to applied science by instrumenting and supporting the COTS component acquisition process, from taking possession to intensive and recurrent reuse via in-depth evaluation.

Fig. 1. COTS Component Integration: Sorcery?

In Fig. 1, the software architect, Merlin l'Enchanteur, tries with the assistance of his development team to evaluate the possible incorporation of a recently acquired COTS component into his famous COTS-based product: "The Cooking-Pot".

Trying to solve the problems above mentioned, section 2 of this paper discusses, in an independent way, three technical directions: UML and executability in modeling and

reflection in programming. We show in this section that an interesting synergy between these three elements is possible. Section 3 presents the BIT technology and a concrete environment for putting into practice this technology. Section 4 briefly describes a Web context in which components may be evaluated and thus qualified. This happens before section 5 in which we conclude and provide perspectives.

2 Prerequisites for COTS Component Assessment

Our leitmotiv is that COTS components must be designed for assessment. This responsibility falls to vendors in providing facilities that realistically permit, and, in the best case, favor any late *in-situ* evaluation (*i.e.*, that relative to the users' operating contexts). From an economical point of view, COTS component providers that organize their supply based on such a principle should have a better market positioning and an enhanced credibility. From a technical point of view, we here sketch the main software techniques that are necessary, but possibly not sufficient, for designing more "autonomous" components.

2.1 The Unified Modeling Language v. 2.0

Since the UML is an agreed and widespread industrial standard and, taking into account the large and significant enhancements in the forthcoming[1] ver. 2.0 [3-4] that are in particular dedicated to component specification [5], we request from vendors the availability of a more or less formal and detailed specification of their products. Built on a XML DTD (a.k.a. XMI), any UML model may then be easily shared and exchanged through the Internet. Moreover, commercial Web sites may directly benefit from this format to structure and maintain their component repositories.

At this time, few downloadable COTS components are documented by means of the UML or, if some of them are documented, then one may wonder how UML models may make an impact on the component purchasers' assessment process. We have at one's disposal interesting results that establish, for instance, the way by which test material can be derived from UML models in general and from UML State Machine Diagrams in particular [6]. However, if only Class Diagrams and/or Component Diagrams are available, how is it possible to capitalize on these kinds of models when the evaluation is done at deployment time? To sum up, we look for an environment in which models "persist at runtime" so that reusers are able to compare specifications with their own observations about component and assembly behaviors.

Following this line of reasoning, open source or binary COTS components built from UML specifications must offer dedicated services in their provided interfaces that are

1 Accepted but not yet fully formalized and stabilized at the time of writing.

specifically made for evaluation. We mean here that if a component state machine diagram, for instance, imposes that, after a *R* request occurring under a precise set of conditions *C*, the component reaches the *S* state[2], then the reuser must be equipped with inherent tools to check such a formalized behavior in her/his own operating environment. *C* corresponds to the component's initial state(s) (before the arriving of *R*) and to local parameters, namely transition guards. Boolean values of these guards may thus vary if they refer to environment property values (*e.g.*, connections to resources). Again, even if the component's vendor guarantees such a behavior in her/his development context, no proof exists about the fact that the said behavior will occur in the reuser's deployment context.

2.2 Executability

Pragmatism and efficiency that prevail in industry put testing before everything else in search of a COTS component evaluation method [7]. Testing is unfortunately not the panacea: "The dominant software testing theories and methods are based upon "white box" testing that assumes the program code or other detailed representation of the software module to be available. (This is generally untrue of commercial, off-the-shelf (COTS) software and much legacy software.)" [8]. This viewpoint is confirmed in [9]. The author observes that testing COTS components in other contexts than those of component builders, is crucial and mandatory. Moreover, the author raises the problem of considerable (and thus expensive) testing efforts that are peculiar to COTS components. Again, this results from the close nature of COTS software that is not prepared for testing.

In relation with the prior observation (Section 2.1) stating that UML is a good candidate for component specification, we consider executability [10] as promising for component and assembly behavior verification & validation. In UML, the absence of formal bases prevents us from "checking" models. The expression "model checking" is here used as it is commonly understand in the area of mathematics-centered specification methods *(e.g.*, Z, VDM, B). The difference between executing a model and checking a model is the fact that executable modeling languages emphasize animation or simulation [11] through instances (or elements) instead of reasoning about abstractions, *i.e.*, types (or sets). The process of verification & validation is thus by definition incomplete. Such an approach is common in scenarios (a.k.a. UML Sequence Diagrams) in which the chronology of message exchanges is represented between instances playing special roles, and thus does not apply for all of the instances of a type.

2 Multiple concurrent states are also acceptable.

2.3 Reflection

Reflection is the ability in object-oriented programming languages to dynamically (at runtime) have information on objects. This notion is also comparable to that of metadata in databases for instance. In Java, the *java.lang.reflect* sub-library offers all of the necessary predefined classes to obtain metadata on objects. It also presents a secure protocol to activate objects (invocation of methods) based on the computed meta-information. The idea of a secure protocol relates to the fact that, for instance, private methods cannot be run. In contrast to C++ which has a limited reflection support (RTTI standing for Run-Time Type Information), the C# programming language supports an enhanced reflection API, as does Java. More generally, the strong relation between distributed programming and components surely makes reflection a first-class programming trend for the future. In current practice, technological component models such as that of Enterprise JavaBeans are grounded on reflection. Furthermore, reflection has also been proved mature and relevant for testing software components in general [7]. To sum up, reflection cannot be ignored in modern component-based development.

Reflection is in fact a powerful programming technique that can be used by COTS component vendors to develop assessment-specific interfaces. Implementations of these interfaces can explore and deliver any internal data required by reusers to obtain an in-depth understanding. Moreover, configuration actions should be possible to set up a given component so that it may function in its target environment better. Indeed, metadata, in particular, aims at making components aware of their surroundings.

3 Component+

This section concretely addresses the issues listed in previous sections in presenting the Component+ European project (IST-1999-20162, www.component-plus.org). The COTS component facet of this project is more specifically discussed in [12]. Another important issue addressed in this project is contract testing [13]. In this paper, we single out the concrete outcomes of this project, namely the Built-In Test technology and, more precisely, a Java framework that supports this technology.

3.1 Built-In Test

"**Built-in test** refers to code added to an application that checks the application at runtime." [14]. By analogy to Built-In Self-Test that is peculiar to hardware components, the idea behind BIT is to encapsulate and to keep test code[3] in COTS components, and

3 Overheads caused by BIT is out the scope of this paper. Further details can be found from: www.component-plus.org.

consequently equip them with a testing interface (Fig. 2) in order to have deployment-oriented facilities for assessment. In Fig. 2, the Component+ approach views a component as having three kinds of provided interfaces: functional which is common, testing and configuration.

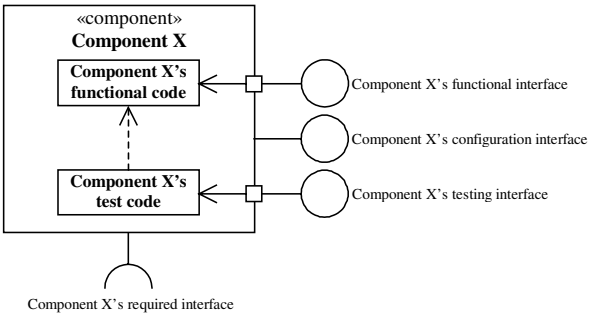

Fig. 2. Canonical structuring of an ordinary component (UML 2.0 formalism)

In the scope of COTS components, reusers have in principle (encapsulation preservation) no way for accessing, and thus for changing, the test code: it is written once and for all thanks to the reflection capabilities of Java. It is thus unique and the same for all components. Customization occurs through, and only through, the possible implementation (a predefined default implementation exists) of three services declared in a Java interface called *BIT_testability*:

```
public interface BIT_testability {

    void invariant() throws
    Statechart_invariant_exception;

    void precondition(String action) throws
    Statechart_precondition_exception;

    void postcondition(String action) throws
    Statechart_postcondition_exception;

    // different other things here

} // BIT_testability
```

In this code, the *invariant()* method for instance, is supposed to analyze any data relating to the user's operating environment and thus to return the acceptance or the rejection of a test case (*Statechart_invariant_exception* raising is used to indicate abnormal deployment conditions). Many other testing tools (test case objects, test scenario objects and tester objects) are offered; these are presented in [12].

3.2 COTS Component Behavior Prediction Based on UML State Machine Diagrams

The availability of a UML-based behavioral specification (a statechart) for an individual component makes possible its deferred evaluation and high-level understanding under "true" runtime conditions. More precisely, and in the spirit of COTS systems, vendors that might produce such costly documentation, should have a significant return on investment. For that, the Component+ Java framework includes an appropriate support for statechart implementation. Additional services of the *BIT_testability* interface (code below) allow to monitor all of the inside of a component. For instance, the *verbose()* method supports a complete diagnosis of possible failures that may occur between two run-to-completion cycles of the component's statechart.

```
public interface BIT_testability {

    // different other things here

    String current_state();

    boolean in_state(String name);

    void to_state(String name) throws
    Statechart_functional_exception;

    String verbose();

} // BIT_testability
```

Again, a generally satisfactory default implementation exists for *BIT_testability* but most of the time, other interesting configuration operations may be added and implemented, as, for instance, a "reset "action that puts a component in a given state after a failure (see code below).

3.3 Example

For illustration purposes, we reuse a case study named Railcar System [**Erreur ! Source du renvoi introuvable.**] (Fig. 3). We show a Railcar component statechart in Fig. 4. A Railcar component instance communicates with Terminal component instances and a Control center component instance in order to synchronize arrivals and departures with passengers getting on and of.

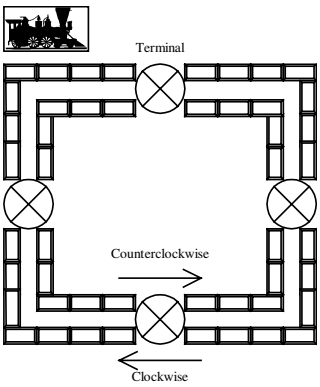

Fig. 3. Railcar System

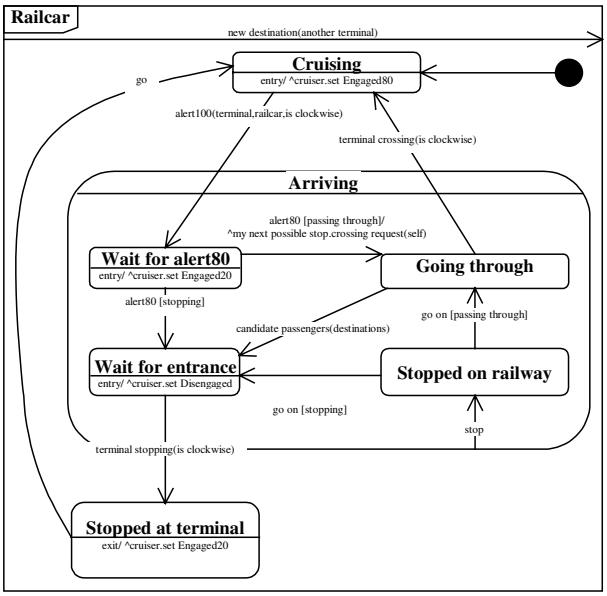

Fig. 4. Statechart of a Railcar component (UML 2.0 formalism)

By means of the offered Java library, the implementation of the Railcar component that conforms to an Enterprise JavaBean is as follows:

```
public class RailcarBean implements
SessionBean,BIT_testability {
```

First, a default implementation of the *to_state(String name)* configuration method exists:

```
public void to_state(String name) throws
Statechart_functional_exception

{state_machine().to_state(name);}
```

Next, a possible implementation of an additional *reset()* configuration method may have the following look:

```
public void reset () throws
Statechart_functional_exception

{to_state("Cruising");}

// other default implementations here
```

Finally, an example of service in the functional interface (*alert80()* request in Fig. 4) is implemented as follows:

```
public void alert80() throws
java.rmi.RemoteException,Statechart_exception {

boolean stopping =
_destination_board.contains(_my_next_possible_stop.getHand
le());

boolean passing_through = ! stopping;

_Railcar.fires(_Wait_for_alert80,_Wait_for_entrance,stoppi
ng);

RailcarRemote[] args = new RailcarRemote[1];

args[0] = (RailcarRemote)this._ejb_context.getEJBObject();
```

The use of the Java reflection API (invocation of the *crossing_request(RailcarRemote railcarRemote)* function) occurs as follows:

```
_Railcar.fires(_Wait_for_alert80,_Going_through,passing_th
rough,_my_next_possible_stop,"crossing_request",args,State
chart_monitor.Reentrance);
```

The *run_to_completion()* method moves forward the statechart of a Railcar component:

```
_Railcar.run_to_completion();

} // alert80() method
```

```
} // RailcarBean class
```

4 Web-Based COTS Component Evaluation

Programming components so that they are equipped with self-management features (configuration and testing interfaces as well as implementations of these interfaces relying on reflection and a UML State Machine Java engine) is not sufficient. One also needs a canonical environment in which evaluation may occur in a systematic and standardized way. We here use a standard in the domain of network/software administration and management which is called JMX (standing for Java Management eXtensions). This Java product allows the viewing and manipulation of components as MBeans (*i.e.*, Manageable Beans). In short, the advantages resulting from JMX are:

- The clear distinction between the usual functional interface and the less common testing and configuration interfaces. These last two may appear in Web browsers (Fig. 5). The functional interface may be activated via a network, a GUI, another thread or simply another component instance as planned in the overall management application architecture;
- The possibility of remote testing. In the spirit of a Web marketplace, one may thus imagine online testing facilities for potential COTS component buyers;
- The simulation of UML models (executability) whose feedbacks (observations and results in deployment environments) may be contractually compared to the vendors' original specifications;
- The administration and monitoring of components that are aided by intrinsic capabilities of JMX. Nevertheless, the organization of the inside of a component by means of a statechart and, the controlled access/manipulation of this statechart, generate enhanced administration and monitoring possibilities.

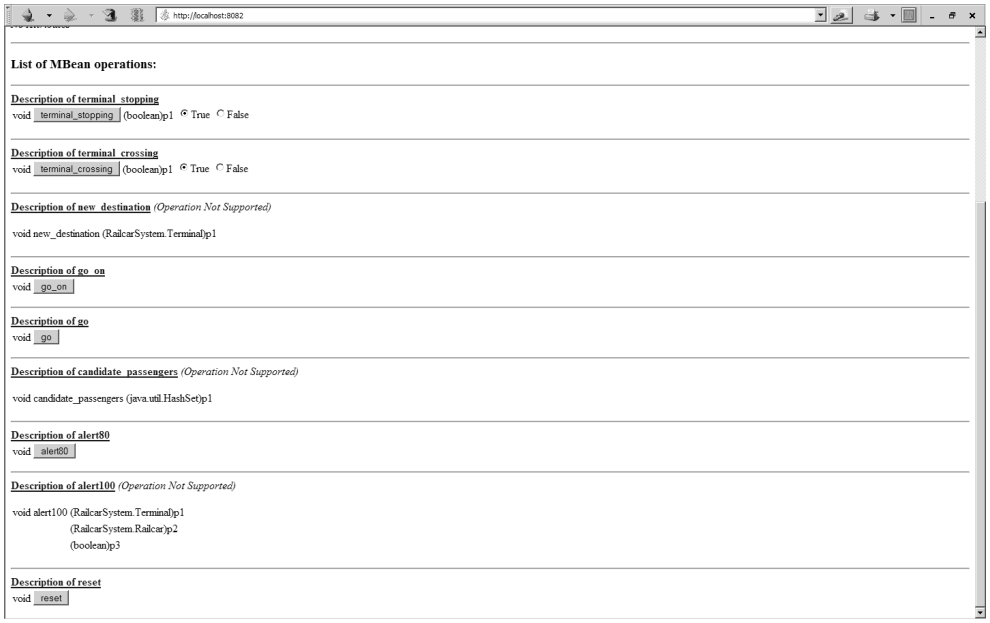

Fig. 5. JMX-based testing and configuration of a Railcar component instance

5 Conclusion

We propose in this paper a different, even innovative, assessment approach for COTS components that is based on the UML 2.0 and the notion of executability. We on purpose briefly describe a Java library named *PauWare.Statecharts* that instruments COTS component evaluation by means of UML State Machine Diagrams. The reflection programming mechanism is intensively used in this library which comes from the Component+ project.

Apart from technical concerns, convincing COTS component vendors to adopt the Component+ approach stumbles over the strong difficulty of elaborating specifications (using UML or not using UML). Economically, vendors will probably agree about the proposed approach whether they will be able to increase their business in comparison with their competitors. To continue the paper's discussion, one may notice that there are several industrial experiments (available from: www.component-plus.org) aimed at further validating the BIT technology.

Regarding the technical status of the Component+ Java framework, most of the current investigations are concerned with component assemblies and, more precisely, how to test

and dynamically (re)-configure a system of components. The concurrent use of a component instance through both its testing interface and its functional interface is sometimes subject to side effects. The study of a test case object may for example show a "strange" (*i.e.*, unexpected) result (Java exception, abnormal reached state…) while any further analysis demonstrates that the crossing of normal and test requests is the source of problems, leading in fact to the conclusion that the component behaves correctly. For a system of components, the flow of input events is multiple. Even if observations are possible, diagnostics and interpretations are more unsteady.

Concerning perspectives of our research work, all of the programming framework has been adapted for the administration of software components running and deployed in wireless and mobile devices. First results show that our Java engine is suitable for these systems but new types of failures peculiar to these systems are now investigated.

Acknowledgments

The work presented in this paper has been partially funded by the European Union within the Component+ IST project (IST 1999-20162).

References

1. Overhage, S., and Thomas, P.: CompoNex: A Marketplace for Trading Software Components in Immature Markets, proceedings of Net.ObjectDays 2003, Erfurt, Germany, September 22-25, (2003) 145-163
2. Szyperski, C., Gruntz, D., and Murer, S.: Component Software – Beyond Object-Oriented Programming, Second Edition, Addison-Wesley (2002)
3. Object Management Group, UML 2.0 Infrastructure Specification, OMG Adopted Specification ptc/03-09-15, September 2003
4. Object Management Group, UML 2.0 Superstructure Specification, OMG Adopted Specification ptc/03-08-02, August 2003
5. Bock, C.: UML 2 Composition Model, Journal of Object Technology, 3(10), (2004) 47-73
6. Briand, L., Di Penta, M., and Labiche, Y.: Assessing and Improving State-Based Class Testing: A Series of Experiments, IEEE Transactions on Software Engineering, 30(11), (2004) 770-793
7. Polini, A.: Testing Component-Based Software Systems, Ph.D. dissertation, Pisa University, November 2004
8. National Coordination Office for Information Technology Research and Development, HIGH CONFIDENCE SOFTWARE AND SYSTEMS RESEARCH NEEDS, January 2001
9. Weyuker, E.: Testing Component-Based Software: A Cautionary Tale, IEEE Software, 15(5), (1998) 54-59
10. Mellor, S., and Balcer, S.: Executable UML – A Foundation for Model-Driven Architecture, Addison-Wesley (2002)
11. Ermel, C., and Bardohl, R.: Scenario animation for visual behavior models: A generic approach, Software and Systems Modeling, 3(2), (2004) 164-177

12. Barbier, F.: COTS Component Testing through Built-in Test in Testing Commercial-off-the-shelf Components and Systems, Beydeda & Gruhn (Eds), Springer, (2005) 55-70
13. Atkinson, C., Groß, H.-G., and Barbier, F.: Component Integration through Built-in Contract Testing in Component-Based Software Quality: Methods and Techniques, Cechich, Piattini & Vallecillo (Eds.), Lecture Notes in Computer Science #2693, Springer, (2003) 159-183
14. Binder, R.: Testing Object-Oriented Systems – Models, Patterns, and Tools, Addison-Wesley, 2000
15. Harel, D., and Gery, E., 1997. Executable Object Modeling with Statecharts, IEEE Computer, 30(7), (1997) 31-42
16. Kreger, H., Harold, W., and Williamson, L.: Java and JMX – Building Manageable Systems, Addison Wesley, 2003