Incorporation of test functionality into software components

Franck Barbier, Nicolas Belloir and Jean-Michel Bruel

LIUPPA, Université de Pau et des Pays de l'Adour
BP1155, F-064013 Pau cedex, France
{barbier, belloir, bruel}@univ-pau.fr,
URL: http://liuppa.univ-pau.fr

Abstract. COTS components trustworthiness is a key issue to be addressed within the field of component-based software engineering. This problem relies on the duality between development and deployment. COTS components vendors may prove varied properties for their components but purchasers may want to validate these properties in different execution environments. Built-In Test is thus the ability to endow components with extra functionality in order to develop in-situ tests. This paper stresses a Java library that supports Built-In Contract Testing. Complex component behaviors are ruled and observed based on states and reactivity to client requests. A large component consisting in a Programmable Thermostat illustrates the Built-In Contract Testing technology and the offered Java library.

1 Introduction

Component-based software engineering (CBSE) is currently the most recommended technique in terms of reuse and of reduction of development costs. The development of applications in this context is based on the principle of composition of components interacting together. In particular, this composition is carried out by connecting component interfaces that provide services, with clients requiring these services. However, composition of components remains syntactical and thus raises many problems. Among possible investigations, let us mention semantic interoperability [1], composability [2], prediction of assembly behaviors [3].

Experience shows that, whatever the level of certification/quality of service announced by a component, it is fundamental to give to its customers the possibility of testing it in situation in its new environment (which is called "run-time testing" in the remainder of this paper). In this context, it is necessary for a component to be able to show that it behaves according to its specification in the final phase of development (contract testing), or in the phase of deployment (quality of service testing). In this purpose, we have developed a Java library that implements these principles. In order to attenuate component integration, we advocate a special design technique for COTS components based on complex state machines (vendors side) and/or some customisation (users side) that adapts and allows to plug a given COTS component into our library. One recurrent feature of COTS components is the obstacle to access source code. In the Java context, we thus use the reflection capabilities of this language to permit dynamic access at run-time to internal properties. Test protocols are especially written once for all and greatly help the way COTS components may be acquired, assessed and finally (re)used.

We present in this article the main orientations of our approach, and we illustrate the use of our library. Thus, in section 2 we present the general context of components testing. Our approach is illustrated by means of a concrete example in section 3. Finally we conclude in section 4 and present some perspectives.

2 Component testing

As for electronic components, software components "are integrated" in software architectures without modification. In some extreme cases, components can even be incorporated into applications at run-time (such as in CORBA platforms). This kind of development results in increasing the importance of the assembling and the validation phases compared to the implementation phase. The developer thus builds the tests that are suitable for unit testing. The customer, on the other hand, generally does not test the component itself (it is supposed to be validated, even certified, by the supplier) [4]. It is nevertheless essential to test its integration in real execution contexts (e.g., communication with other components, appropriateness of interfaces, fault tolerance, conformance to specified behavior). Thus we are not actually and directly interested in testing a component. We want to provide the component with a certain degree of functionality allowing to assess its behavior in its deployment environment.

Previous works in the field of Built-In Test (BIT) rather deal with self-test [5] (i.e., automatic triggering of tests), or with the improvement of tests set definition [6]. Approaches close to ours is that in [7]. They define the notion of testable component that supports remote testing based on a generic testing interface and an architectural model. We enhance such an approach in dealing with the deep and detailed internal behavior of components within tests, allowing contract and quality of service testing.

2.1 Built-In Contract Testing

We consider a COTS component as an aggregate of sub-components that are implementations of several operations that it provides. In Java, such a component is realized through a class that possesses fields whose type is that of these sub-components, recursively. Figure 1 describes the micro-architecture in which an anonymous COTS component is connected with the predefined classes of our library. A BIT component is built such that it "acquires" all of the properties of "Component". In Figure 1, a UML dependency is used in order to defer the choice of an adequate programming mechanism (inheritance is often useful). The BIT testability contract interface, detailed in section 3, is a set of operations that are systematically used within BIT test case, itself systematically used by BIT tester. BIT test case is a Java class that has frozen test protocols, namely "initialize the test", establish "execution conditions", "get results and/or failures" and "finalize the test". Any BIT component may customize (i.e. override) these basic actions in taking care, opportunistically, of the property values of "Component". BIT tester allows to develop test scenarios: sequences, expected results, aborting/completion policies, etc. A great benefit of this approach is that most of the test stuff is not dependent upon the specificity of the evaluated COTS component. For instance, competing components that may be bought in order to satisfy very special requirements, can be compared based on the same test framework. Moreover, the test stuff is actually built in "BIT component". BIT component (instead of Component) may be deployed in order to measure the quality of service at run-time. Notice however that in this case, one has to pay attention to performance and resource burden: BIT component creates some overload compared to Component. A drawback of our approach is that sub-components are fully encapsulated entities, and as such, analysis and diagnoses are difficult to determine at a deep level of aggregation. We have thus extend the library to cope with nested and concurrent states of components, in order to supply a better access to the internal of a component.

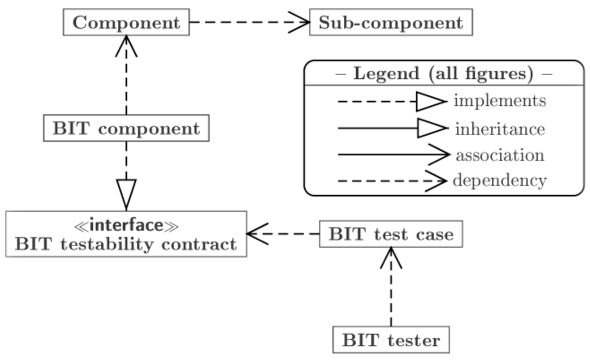


Fig. 1. Dependencies in Built-In Contract Testing

2.2 State-based contract testing

Figure 2 shows an extended interface called State-based BIT testability contract from which a BIT component can be linked to. The white arrows with dotted line is the Java "implementation" relation between classes and interfaces. This second way of running Built-In Contract Testing is more coercive in the sense that a state machine is need for the tested COTS component. While such a specification is common in real-time systems for instance, one cannot always expect that. Some reengineering work is thus sometimes required to extract a behavioral specification from an existing component.

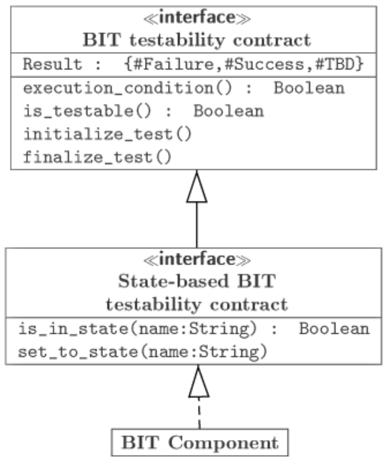

≪interface≫
BIT testability contract
Result : {#Failure,#Success,#TBD}
execution_condition() : Boolean
is_testable() : Boolean
initialize_test()
finalize_test()
≪interface≫
State-based BIT testability contract
is_in_state(name:String) : Boolean
set_to_state(name:String)
BIT Component

Fig. 2. Extension of the BIT/J library for state-based contract testing

3 A concrete example of the BIT/J library usage

In order to show the applicability of the concepts presented above, we first present a Programmable Thermostat component. Next, we discuss the construction of the corresponding BIT component, while insisting on a precise step-by-step process.

3.1 The BIT/J library

Figure 3 is a complete overview of the BIT/J library. Built-In Test can first be simply carried out by means of the three main elements named BIT testability contract, BIT test case and BIT tester. In this case, BIT just copes with assessment of computation results, execution environment and faults. In the more complicated case, three equivalent state-oriented facilities are required: State-based BIT testability contract, State-based BIT test case and State-based BIT tester. Since these three last one use Harel's formalism called Statecharts [8], a underlying sub-library ("Statecharts" package: top, left of Figure 3) is reused (i.e. Statechart and Statechart monitor). BIT state and BIT state monitor are contextual specializations of the sub-library in order to create a connection with

State-based BIT testability contract, State-based BIT test case and State-based BIT tester.

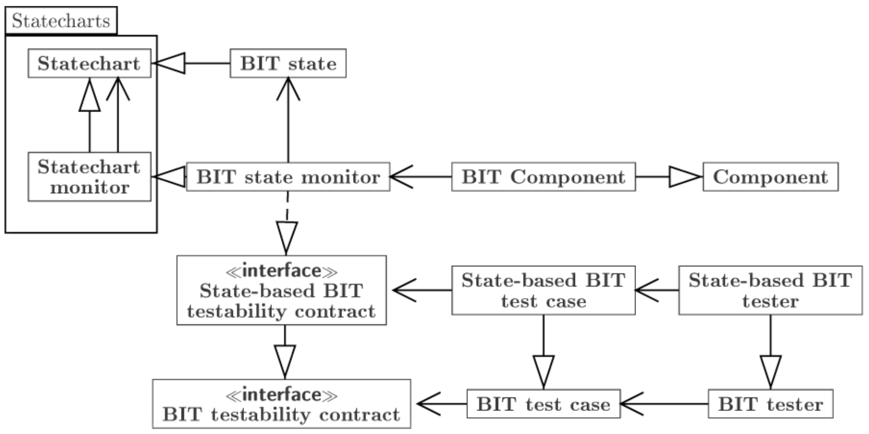


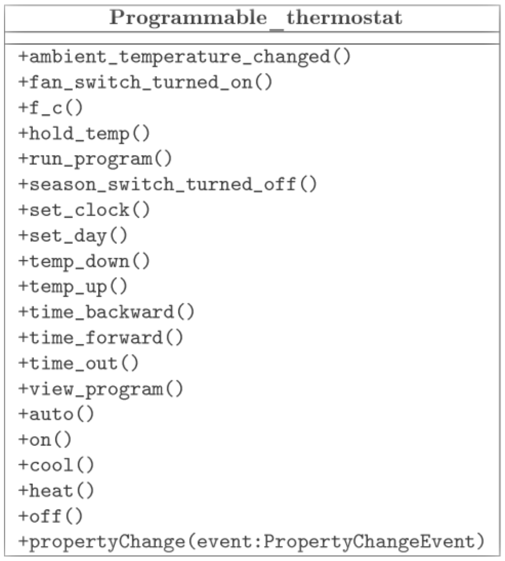
Fig. 3. Architecture of the BIT/J library

3.2 The Programmable Thermostat Component

We give here the specification of the interface provided by a Programmable Thermostat component (see Figure 4). The operations in the interface are requested by clients. They mask the inside of the Programmable Thermostat component. Traditional testing is thus confined to activating operations while intermediate hidden results, states and possible faults have to be known to assess deployement compliance.

3.3 The BIT implementation of the Programmable Thermostat Component

The realization of a BIT component and its evaluation occur within four phases. The first one can have different shapes according to the component nature: whether it was derived "from scratch", i.e. conceived since the beginning as a BIT component, or whether it was built from an existing component. Our method first consists in linking the BIT part to the original component. The second step consists in implementing the behavioral specification in the BIT part. The third step consists in providing an implementation for the interface relating to the standard testability contracts. Lastly, at the time of the fourth step, a given customer of the BIT component defines specific test cases handled by testers.



```
+ambient_temperature_changed()
+fan_switch_turned_on()
+f_c()
+hold_temp()
+run_program()
+season_switch_turned_off()
+set_clock()
+set_day()
+temp_down()
+temp_up()
+time_backward()
+time_forward()
+time_out()
+view_program()
+auto()
+on()
+cool()
+heat()
+off()
+propertyChange(event:PropertyChangeEvent)
```

Fig. 4. Programmable Thermostat interface

Step 1 - linking the BIT part to the original component. There are several ways for bounding a component to its BIT incarnation. Firstly, the BIT part can inherit from the component. This method is interesting because it allows to directly manipulate the component itself. Indeed, it is then possible, for example, to have access to protected attributes and operations, in order to setup the component in a particular state before execution of a specific test. That also allows to directly describe its behavior in terms of its attribute values if the component was initially designed to support the BIT technology. Secondly, the component can be included as a particular field value in its BIT version. The major advantage is encapsulation respect since the protected attributes and operations are then not activable. In the remainder, we use the first approach.

Step 2 - Description of the behavior with a state machine. To describe the component behavior, it must be specified via a state machine that conforms to Harel's formalism (Figure 5). Each state is an attribute of the BIT component whose type is BIT_state (or Statechart). A unique special attribute is also implemented within the BIT component whose type is BIT_state_monitor (or Statechart_monitor). Below, we have the declaration of two states: Run and Hold :

```
protected Statechart _Run;
protected Statechart _Hold;
```

```
//...
protected Statechart_monitor _Programmable_thermostat;
```

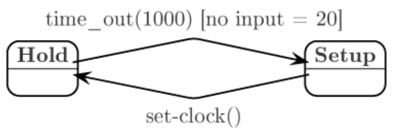


Fig. 5. An extract of the Programmable Thermostat state machine

Below, instantiation of simple states occurs. It is also possible to build complex (i.e. nested) states, recursively. In order to connect them together, the *and* (concurrency between states) and *xor* (common exclusiveness between states) operators are used. As for nesting, it is managed through assignment (e.g. _Operate is assigned with an expression mixing its sub-states).

```
// declaration of the state machine in the BIT component
_Run = new BIT_state("Run");
_Hold = new BIT_state("Hold");
//...
```

```
// complex BIT_state
_Operate=((_Run).xor(_Hold)).and(...)).name("Operate");
```

It is possible to associate an operation with an incoming (or an outgoing or an internal) transition. This operation is then automatically triggered. *These operations can be those of sub-components.* This permits the correlation between the BIT component states and the activation of sub-components. This also permits the trace of execution paths in case of faults at a deep level of nesting.

```
// Automatic dynamic access to the "set_time" internal
// action through the "time_out" operation belonging
// to the BIT component's interface
  _Operate = (((_Run)).name("Operate")).internalAction(
    "time_out", this,"set_time",null);
//...
```

Once all of the states defined, the state machine of the BIT component must be declared and instantiated:

```
// State machine
  _Programmable_thermostat =
    new BIT_state_monitor((_Operate).xor(_Setup)...);
```

Finally, the initial states of the state machine must be established. In our example, the initial states are Hold and Current_date_and_time_displaying.

```
  _Hold.inputState();
  _Current_date_and_time_displaying.inputState();
```

Step 3 - Definition of the testing interface. The BIT/J library provides general-purpose operations which may be overloaded to pay attention to contextual phenomena (see “execution_condition()" for instance in Figure 2). These operations are supported by two entities of the library. The first is the BIT_testability_contract interface. It allows to develop *traditional* contracts. In contrast, the is_in_state() and set_to_state() operations are members of the State_based_BIT_testability_contract interface and rule the setup of states before a test (see below), and the evaluation of target, expected or unexpected, states after a test.

```
// In the BIT Component
public boolean Configuration_1() throws ... {
   set_to_state("Hold");

   // execution of the "set_clock()" operation
   set_clock( );
}
```

Step 4 - Instantiation and execution of the test case objects. The tester instantiates a BIT component. Then, it can run the test sets that embody a given policy. For that, it instantiates test cases (i.e. instances of BIT_test_case and/or State_based_BIT_test_case) by specifying operations that have to be run. The global test process relies on the test() operation and result/fault interpretation depends upon the interpretation() operation. The code below shows the realization of a test concerning the set_clock() Programmable Thermostat's operation. The expected result is the arrival at the *Setup* state.

```
// bc is a BIT component
bc = new BIT_programmable_thermostat (temperature);

// Put the BIT component in a specific state before the test
bc.set_to_state("Hold");

```

```
// Definition of the test case. The third argument
// is the expected result
   sbbtc1 = new State_based_BIT_test_case (bc, "set_clock",
    null, null, "Setup");

// Execution of the test case
sbbtc1.test();

// Get the test case result
System.err.println("Interp: "+ sbbtc1.result());
```

Quality of service analysis. Below is an example of malfunctioning that the BIT technology may help to detect. The Programmable Thermostat is designed for automatically be forced in the Hold state if previously in Setup and no action performed for 20 seconds. We simulate a defect by means of a undesired delay. Different computer operating systems may lead to different results for specific test.

```
/* Instantiation of the BIT component */
public boolean Configuration_2 () throws ... {

    set_to_state("Setup");

    /* Starting of a timer for 20 s */
    try {
        Thread.sleep(20000);
    } catch(Exception e){...}

    /* Test if the BIT component is in the Hold State */
    if (is_in_state("Hold"))...
    else...
}
```

4 Conclusion

Individual comprehensive behaviors of COTS components often remain buried within their hidden part. Formal behavioral specifications is a first step to make trustworthiness effective but is insufficient. Built-In Test is another step to create confidence. Vendors may equip their COTS components with test stuff in order to run it online, especially within unknown (at development time) execution environments. We enhance in this paper Meyer's "design by contract" idea in allowing first the description of assertions (pre-conditions, post-conditions and invariants) based on complex nested, and sometimes concurrent, states of components that largely hinge on sub-components. Next, we ground our approach on

Java reflection capabilities that ensure that test protocols are written once and for all. In such a context, properties of components are dynamically accessed at run-time. Test stuff is launched within test case objects subject to interfaces embodying testability contracts, are implemented by BIT components. They are in fact two ways for working: either vendors adhere to our component design technique or purchasers need to build BIT components from ordinary components. In the first case, providers add value to their COTS components by delivering customizable testing functionality. In the second case, components have to be adapted in order to be plugged into the BIT/J library. Limited to the Java world, our approach is however a concrete support for the notion of Built-In Test that has been defined more theoretically in some other papers. The Programmable Thermostat component is in this respect, a representative example. We measure for instance the timer event quality of service that may greatly vary from an environment to another one. Finally, we also supply a step-by-step component design process that is based on rationality. The most significant perspective is nowadays the submersion of the BIT/J library in the JMX (Java Management eXtensions) framework. BIT components may thus be evaluated through Web browsers. We intend in a near future to develop remote testing as part of a project that intends to offer component acquirement facilities on the Web.

References

1. Heiler, S.: Semantic Interoperability. ACM Computing Surveys **27** (1995) 271–273
2. Meijler, T.D., Nierstrasz, O.: Beyond Objects: Components. In: Cooperative Information Systems – Trends and Directions. Academic Press, San Diego, CA (1998) 49–78
3. Crnkovic, I., Schmidt, H., Stafford, J., Wallnau, K.: Anatomy of a Research Project in Predictable Assembly. Fifth ICSE Workshop on Component-Based Software Engineering - White paper (2002)
4. Harrold, M.J.: Testing: A Roadmap. In: ICSE - Future of SE Track. (2000) 61–72
5. Wang, Y., King, G., Patel, D., Court, I., Staples, G., Ross, M., Patel, S.: On Built-in Test and Reuse in Object-Oriented Programming. ACM Software Engineering Notes **23** (1998) 60–64
6. Jézéquel, J.M., Deveaux, D., Le Traon, Y.: Reliable Objects : Lightweight Testing for OO Languages. IEEE Software **18** (2001) 76–83
7. Gao, J., Gupta, K., Gupta, S., Shim, S.: On Building Testable Software Components. In: Proceedings of First International Conference on COTS-Based Software Systems, ICCBSS 2002. Lecture Notes in Computer Science 2255, Orlando, FL, USA, Springer LNCS (February 4-6, 2002) 108–121
8. Harel, D.: Statecharts : a visual formalism for complex systems. Science of Computer Programming **8** (1987) 231–274