Component Behavior Prediction and Monitoring through Built-In Test

Franck Barbier
LIUPPA, Université de Pau
BP 1155
64013 Pau CEDEX, France
Franck.Barbier@univ-pau.fr

Nicolas Belloir
LIUPPA, Université de Pau
BP 1155
64013 Pau CEDEX, France
Nicolas.Belloir@univ-pau.fr

Abstract

Real-time systems or safety-critical applications require high-confidence software components. Component behavior prediction refers to the ability to check, even certify, component specification conformance at development time. Complementarily, odd and varied execution contexts linked to the idea of deployment impose extra checking when components are deployed. This paper on purpose proposes incorporation of test into components. Components that own states and complex dependencies between these states, are methodically specified in using UML Statechart Diagrams. Code is next derived in order to verify at development time component functioning in relation to specification. At deployment time, facilities are offered for (re-)configuring components to carefully fit specific runtime environments. Built-In Test (BIT) material may thus be optionally generated in components to capture execution conditions and to make possible component behavior adjustments. All of these principles are powered by means of the BIT/J dedicated Java library that is presented and illustrated in the paper.

1. Introduction

Components trustworthiness is a key issue to be addressed in the field of component-based software engineering. This expectation relies on the well-known dichotomy between development and deployment [1]. Using formal verification & validation techniques for components and assemblies, supposes however the existence of mathematical specifications of component system behaviors. Precise models of execution environments are also needed. Unfortunately, such models are hard to construct or to find.

In contrast, built-in test singles out empirical checking of components in the sense that test material may be customized, deployed, as well as ruled at runtime. Built-In contract testing is a specialized approach in which component states and their complex relationships play an important role in contract expressions [2]. Moreover, monitoring test code cannot be ignored since critical "normal" behavior phases cannot tolerate concurrent test execution [3].

We discuss in this paper a framework that includes a library used for endowing components with built-in test code. To that extent, Section 2 exposes the general idea and principles of built-in test while Section 3 precisely states how built-in contract testing can be exercised. Section 4 closes the paper in coping with an example that shows how the BIT/J Java library supports BIT.

2. Built-In Test

Built-In test is defined by Binder in [4] as follows: "**Built-in test** refers to code added to an application that checks the application at runtime." This idea is not new in electronics for instance. Built-In Self-Test or BIST concerns electronic components that may be individually tested in an automated way. Regarding software, Binder also notices that assertion checking is a straightforward support for BIT. Assertions in the Eiffel programming language [5] is a reputable native mechanism to incorporate invariants, pre-conditions and post-conditions into code. This leads to the expression: "design by contract". Other advanced and dedicated assertion languages exist, as for instance the Sugar language [6].

2.1. Built-In Testing Technique

We propose in this paper an enhanced approach developed within the Component+ European project (www.component-plus.org). We initially observe that assertions are essentially based on class/component instance states. For an object like a stack for instance, a pre-condition (i.e. a contract) to "pop" is "not empty". However, components are in most cases not fine-grained software entities, but sizeable elements that own numerous states, as well as complex dependencies between them (e.g. concurrency, nesting, elaborate event processing). Moreover, components have a required interface [7] that corresponds to what is required by the component in order to properly function. States of components are thus partially composed of those of their subparts, i.e. subcomponents that are fully encapsulated (i.e. unshared) and play a significant role in the implementation of the wholes they belong to. In other words, wholes are components that are visible by clients. Client components ground their requests on the provided interface of wholes that may delegate jobs to subparts.

Success (i.e. contract respect) thus hinges on accurate behavior phases or instants that together relate to wholes and parts [8]. In this scope, at development time, contracts are expressed via state combination based on appropriate operators (*and* for instance meaning concurrency). We thus have built-in test to predict component behavior, as well as assembly behavior. Contracts may indeed be violated in certain assembly patterns while they may work in other circumstances. Hissam *et al.* in [9] discuss prediction capability design via the expression: “prediction-enabled component technology”.

At deployment time, contracts are *a priori* removed from components in software releases because of realism. Test code first causes overheads: memory overload in embedded systems for instance. Next, test code execution cannot occur at any time and therefore, must be monitored to work consistently and concomitantly with “functional” (a.k.a. “normal”) code. However, contracts may be certified in given environments while they may fail in others. Furthermore, contracts cannot be only based on states but have to take into account execution conditions that change from one operating environment to another one.

The built-in testing technique proposed in this paper tries to go beyond these limits. Apart from hard performance constraints, we keep contracts in components. We also monitor (i.e. trace, diagnose and possibly assess/measure) component behavior through built-in test. We call that Quality of Service (QoS) testing in the rest of the paper. Figure 1 sketches the overall spirit of our approach.

In our view, contract testing thus investigates whether a component deployed within a new runtime environment is likely to be able to deliver the services it is contracted to deliver. The approach essentially makes some well-defined states of the component accessible to clients of a component. This allows a client component to place an acquired component into a defined state, to invoke one or more of its services, and then to verify that the correct result and state are reached before accepting the component as “correct”. Such contract tests are typically performed when a system of components is (re)-configured, and usually occur infrequently at one or two well defined instants during a system’s running lifetime.

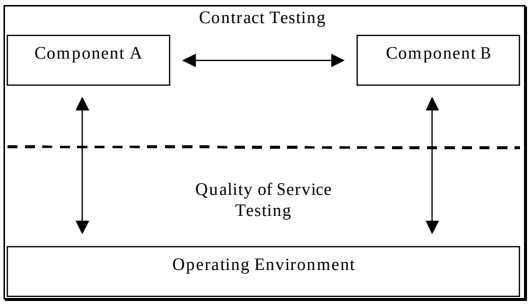


Figure 1. Contract Testing versus QoS Testing

QoS testing concentrates on continuous verification of component/assembly behavior while it is fulfilling its normal obligations. Two main categories of errors are considered: those arising from residual defects within a component, and those arising from erroneous component interactions which may lead to system level failures. Some of these situations can be detected within the component. In these cases, the component is instrumented with internal test code which offers continual verification, and supplies special entry points which allow external entities to invoke and control the tests. In other cases, such as process deadlock, the error cannot be identified within a single component but depends on a combination of components. To cover these situations, the component also offers interfaces through which “inaccessible” information can be extracted. This information is used by external entities for test purposes. We emphasize contract testing in this paper. QoS testing is detailed in [10].

2.2. Built-In Testing Process

Built-In Test is an early concern in the software development process compared to “traditional” test. As it happens, recent utilizations and observations in the Component+ project show that components aim to be equipped with BIT code as soon as they are constructed. As such, their specification has to be done accordingly. As shown later, components without accurate specifications may however become BIT components in the scope of the BIT/J library and Java. More precisely, we use the Java reflection support (*java.lang.reflect* package) that favors component introspection. Formal specifications based on state machines remain however a more powerful way of running BIT.

Thus, in the design phase of the development process, thoughts on how BIT components interact together lead to view software architectures differently. Moreover, tester components are not parts of applications. Hence, we have also to pay great attention to how to separate the normal application and the testing one while perfectly ensuring their mutual collaboration. This problem is going to be solved through the use of a Java standard relating to distributed component and application management (Section 4.2).

Finally, we may note that BIT greatly facilitates by anticipation component integration problems. From recent experience, we observe that BIT design investment at the beginning of the software development process shortens the integration phase.

3. Built-In Contract Testing

In component-based development, integration testing is a source of huge challenges. Interactions of individual pairs of components especially need to be semantically tested [11]. Component interfaces are in fact the basic support for checking compatibility between components. However, interfaces syntactically describe potential exchanges. This does not ensure that the provided and required interfaces of two different components match in the semantic sense, nor that their full range of mutual collaboration possibilities are attained. In this scope, Heiler notices in [12] that: "*Semantic interoperability* ensures that these exchanges make sense – that the requester and the provider have a common understanding of "the meanings" of the requested services and data."

The objective of built-in contract testing is to check that the environment (server component or operating environment) of a component meets its expectations. For example, if a component C is a client of another component S, the purpose of the test software built into C is to check S by invoking its services and to verify that they behave individually and collectively as expected. Since the tests present in a client assess the behavior of a server, they essentially evaluate the semantic compliance of the server to the "clientship" contract. While most contemporary component technologies enforce the syntactic conformance of a component to an interface, they do nothing to enforce the semantic conformance. Built-In contract testing offers a feasible and practical approach for validating the behavioral semantics of components. This semantics may be formalized with modern modeling languages. Concrete concerns lead us to use Harel's Statecharts [13] that are part of UML [14].

In our framework, we impose a minimal set of features and conditions to make components testable. An introspection mechanism which provides access to and information about components is required.

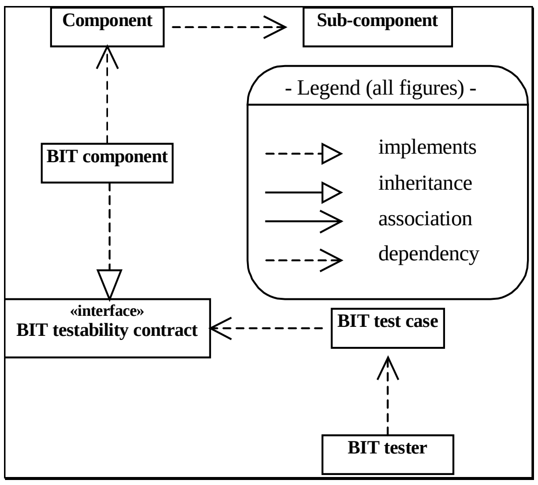


Figure 2. Core of Built-In Contract Testing

Figure 2 describes the core framework for built-in contract testing. A BIT component compensates for a component that may be a COTS component. In this latter case, contracts do not *a priori* exist inside the component. They are thus actually incorporated into the BIT component. Testing the BIT component is equivalent to testing the source component.

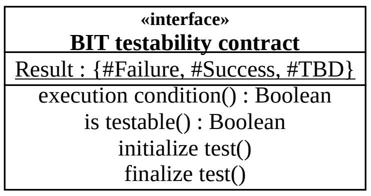


Figure 3. Testability Contract

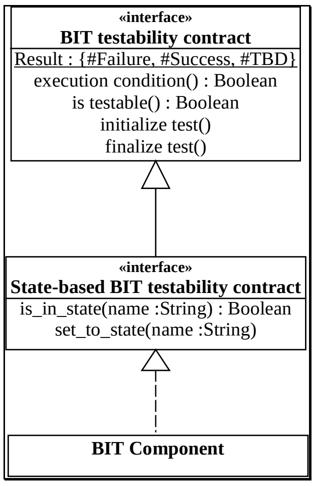
Besides, the BIT component has to inform testers of deployment constraints. The BIT component thus strongly depends upon the *BIT testability contract* predefined interface (Figure 3). The *initialize_test* and *finalize_test* operations allow to setup and to restore the tested component before, respectively after, a test execution. Arbitration or conflict resolution with "normal" code progress in particular occur within these stages. In the same line of reasoning, the *is_testable* function determines whether a specific status of the component precludes for testing. For example, a component may run a data acquisition parallel process embodied by an associated *Running data acquisition* concurrent state. We thus need to write:

```
public boolean is_testable() {
       return   !   is_in_state("Running   data
acquisition"; // not testable in this state
}
```

As for the *execution_condition* service, it may report special contexts linked to the deployment environment.

Finally, execution results are readable by means of three symbolic (use of # in UML) values of the *Result* attribute (see Figure 3). Sometimes, a non deterministic result is tolerated (*TBD* standing for *To Be Determined*).

As shown in the Java code above, we need accurate knowledge on the inside of the tested component and for that, an extended version of the *BIT testability contract* interface is often needed (Figure 4). In the same way, elements in Figure 4 add new services to the component's regular functional interface and serve contract testing purposes. Indeed, components are state machines and require state transition testing. Before a test can be executed, the tested component must be brought into the initial state which is required for a particular test. After test case execution, the test must verify that the outcome (if generated) is as expected, and that the tested component resides in the expected final state. Figure 4 shows an appropriate interface called *State-based BIT testability contract* from which a BIT component can be linked to.



«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

Figure 4. Extension of *BIT testability contract* with *State-based BIT testability contract*

4. Illustration

In order to illustrate the applicability of the concepts presented above, we have developed a Java library whose overall architecture is presented in Figure 5. Built-In contract testing 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, we just cope with assessment of computation results, as well as management of execution environment phenomena and faults in general. 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 elements call on Harel's formalism, an underlying library (*Statecharts* package: top-left of Figure 5) is reused, i.e. the two following classes: *Statechart* and *Statechart monitor*. *BIT state* and *BIT state monitor* are BIT-oriented specializations of the *Statecharts* package in order to create a connection with *State-based BIT testability contract*, *State-based BIT test case* and *State-based BIT tester*.

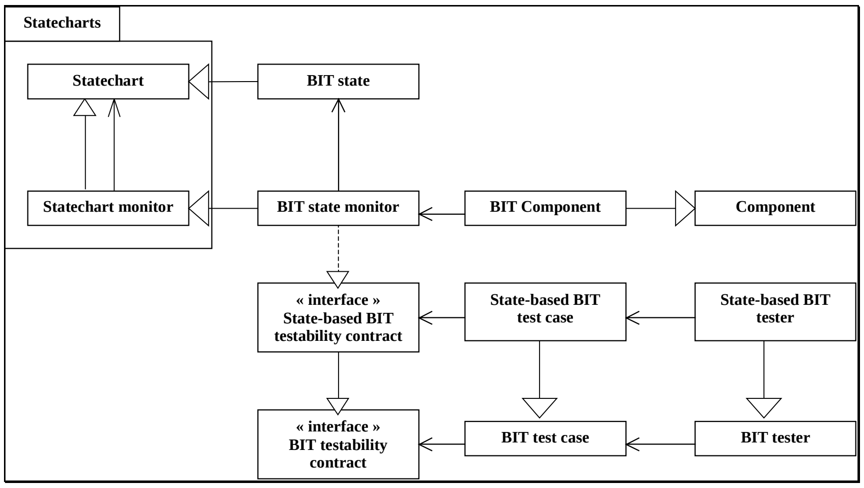
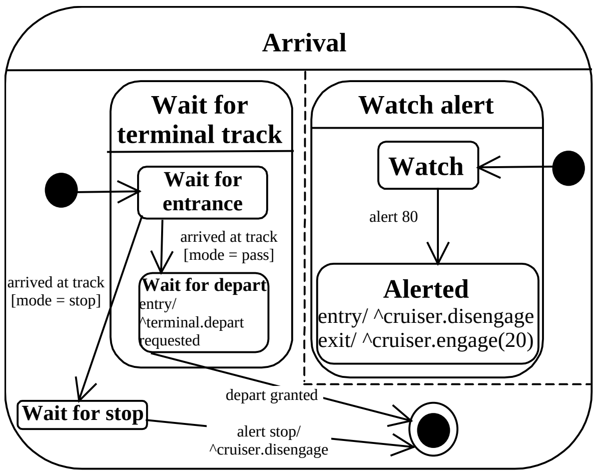


Figure 5. Architecture of the BIT/J Library

4.1. Component Behavior Prediction

Figure 6 is a partial view of a Railcar system component presented in [15].



Arrival
Wait for terminal track
Watch alert
Wait for entrance
Watch
alert 80
arrived at track [mode = pass]
arrived at track [mode = stop]
Wait for depart
entry/ ^terminal.depart requested
Alerted
entry/ ^cruiser.disengage
exit/ ^cruiser.engage(20)
depart granted
Wait for stop
alert stop/ ^cruiser.disengage

Figure 6. Partial Statechart of a Railcar System Component (*Arrival* Substate)

From Figure 6, we are able to construct the UML Component Diagram appearing in Figure 7. This especially allows to exhibit the provided interface of the *Railcar system* component based on the technique stated in [10]. In Figure 7, the provided interface is incomplete since the statechart is itself incomplete in Figure 6.

The following code is therefore generated:

```
public class BIT_railcar_system {
…
protected    BIT_state    Watch,    Alerted,
Wait_for_entrance, Wait_for_depart, Watch_alert,
Wait_for_terminal_track, Wait_for_stop, Arrival;
protected BIT_state_monitor Railcar_system;
…
```

```
Arrival                                       =
(Watch_alert.and(Wait_for_terminal_track).xor(Wa
it_for_stop)).name(“Arrival”);
Railcar_system            =                 new
BIT_state_monitor(Arrival.xor(…));
…
```

The direct substates of *Arrival* are connected together: *and*, *xor* and nesting (realized through assignment) are the operators used.

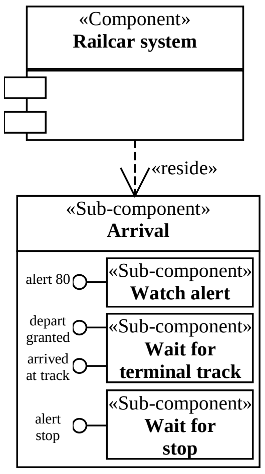


Figure 7. UML Component Diagram Deriving from Figure 6

Event processing is then managed as follows:

```
synchronized public void arrived_at_track() {
 try {
boolean guard = (mode == Pass);
Railcar_system.fires(Wait_for_entrance,Wait_for_
depart,guard);
guard = (mode == Stop);
Railcar_system.fires(Wait_for_entrance,Wait_for_
stop,guard);
// event interpretation cycle:
Railcar_system.used_up_with_recovery();
 }
 catch(StatechartException se) {
```

```
…
 }
}
```

Test scenarios within tester objects can finally be developed:

```
BIT_railcar_system brs;
…
State_based_BIT_test_case     sbbtc     =     new
State_based_BIT_test_case(brs,"arrived_at_track"
,null,null,null);
sbbtc.test("Wait for stop");
// Possible failure:
System.err.println("Interpretation:      "      +
sbbtc.interpretation());
```

4.2. Component Behavior Monitoring

We evoke in Section 3 the four key operations of the *BIT testability contract* interface that help configuring BIT components and ruling tests in optimal conditions. At this stage of the library's development, we focus on the utilization of the Java Management Extensions (a.k.a. JMX) in order to present the testing interface within a Web browser.

An expected benefit is remote testing. Based on the general-purpose COTS component idea [15], we may imagine that purchasers remotely access, assess and next buy components over the Internet. Vendors provide then benchmarks grounded on the BIT philosophy. We nevertheless want, *in essence*, to let the purchasers develop their *own* sets of tests located on their own site. An other expected advantage (not yet implemented in BIT/J) is thus the externalized execution of test code. At this time, we have no separation between test code and normal code, even if execution conflicts are carefully managed. We thus intend to remotely monitor BIT component behaviors which seems for us a more suitable approach in the spirit of a COTS component marketplace.

5. Conclusion

Online controlled verification of components aims to go beyond proof correctness of component models. Deployment factors greatly influence such a research trend and direction. This paper together proposes a built-in testing approach and a concrete tool in order to reach such an objective. Built-In contract testing is a technique further detailed and explained in the paper by means of the example of a Railcar system component. This example favors the understanding of the BIT/J Java library that effectively and efficiently supports BIT.

Limitations of what is proposed are linked to Java. We use the *java.lang.reflect* package within BIT/J in order to dynamically manipulate component features. We in particular do not need an access to source code which is satisfactory with regard to the COTS component spirit. We currently investigate C++ components by using the RTTI (Run-Time Type Information) programming interface. We also think about component platforms, namely .NET, through appropriate component languages, e.g. Eiffel. One of our key goal is indeed the use of languages that offer reflection characteristics, i.e. dynamic access and management of components at runtime in order to make BIT credible. The more obvious and tangible perspective of our work is to view BIT as a basic technique to create and to leverage a component marketplace over the Web.

References

[1] Crnkovic, I., Hnich, B., Jonsson, T., and Kiziltan, Z., 2002. Specification, Implementation, and Deployment of Components, *Communications of the ACM*, 45(10), pp. 35-40.

[2] Jézéquel, J.-M., Deveaux, D., and Le Traon, Y., 2001. Reliable Objects: Lightweight Testing for OO Languages. *IEEE Software*, July/August, pp. 2-9.

[3] Wang, Y., King, G., Fayad, M., Patel, D., Court, I., Staples, G., and Ross, M., 1998. On Built-In Tests Reuse in Object-Oriented Framework Design, *ACM Software Engineering Notes*, 23(4), pp. 60-64.

[4] Binder, R., 2000. *Testing Object-Oriented Systems – Models, Patterns, and Tools*, Addison-Wesley.

[5] Meyer, B., 1997. *Object-Oriented Software Construction*, 2nd Edition, Prentice Hall.

[6] IBM, November 2000. *Guide to Sugar Formal Specification Language,* Version 1.3.1. IBM Haifa Research Laboratory, Israël.

[7] Szyperski, C., 1998. *Component Software – Beyond Object-Oriented Programming*, Addison-Wesley.

[8] Barbier, F., 2002. Composability for Software Components: An Approach Based on the Whole-Part Theory, proceedings of *The 8th IEEE International Conference on Engineering of Complex Computer Systems*, Greenbelt, USA, IEEE Computer Society Press, December 2-4, pp. 101-106.

[9] Hissam, S., Moreno, G., Stafford, J., and Wallnau, K., 2001. *Packaging Predictable Assembly with Prediction-Enabled Component Technology*, Technical Report CMU/SEI-2001-TR-024, Carnegie Mellon University, Pittsburgh, PA.

[10] Barbier, F., 2003. *Business Component-Based Software Engineering*, Kluwer.

[11] Gao, J., Gupta, K., Gupta, S., and Shim, S., 2002. On Building Testable Software Components, proceedings of *The 1st International Conference on COTS-Based Software Systems*, Orlando, USA, Lecture Notes in Computer Science #2255, Springer, February 4-6, pp. 108-121.

[12] Heiler, S., 1995. Semantic Interoperability, *ACM Computing Surveys*, 27(2), pp. 271-279.

[13] Harel, D., 1987. Statecharts: a visual formalism for complex systems, *Science of Computer Programming*, 8, pp. 231-274.

[14] Object Management Group, 2001. *OMG Unified Modeling Language Specification*, version 1.4, Needham, MA.

[15] Harel, D., and Gery, E., 1997. Executable Object Modeling with Statecharts, *IEEE Computer*, July, pp. 31-42.

[16] Wallnau, K., Hissam, S., and Seacord, R., 2002. *Building Systems from Commercial Components*, Addison-Wesley.