COMPONENT+ BUILT-IN TESTING
A TECHNOLOGY FOR TESTING SOFTWARE COMPONENTS[1]

Mariusz Momotko, Lilianna Zalewska
Rodan Systems S.A.
(Mariusz.Momotko, Lilianna.Zalewska}@rodan.pl

Abstract. How to test interaction between software components? Is it possible to test such components in a standard way during run-time? An attempt to answer such questions in a systematic way is the C+ BIT technology developed in the EU Component+ project. This paper presents the framework of the technology and its preliminary evaluation done in a software experiment within the Component+ project. In this experiment Rodan Systems together with other four NAS partners verified effectiveness and usefulness of the technology.

1. Introduction

Component-based development is widely expected to revolutionize the way in which software is developed, deployed and maintained [8]. Instead of continually developing all software from scratch, as is largely the case today, organizations will be able to generate new applications simply by assembling prefabricated components. Moreover, the prefabrication of components need not always be performed in house, but may be realized by third party component vendors selling commercial components on the open market. Thus, the vision of component-based development is to bring software engineering more in line with other engineering disciplines where assembling new products from standard parts is the norm.

This model of software development presents some challenges, however. With traditional development approaches, the bulk of the integration work is performed in the development environment, giving engineers the opportunity to pre-check the compatibility of the various parts of the system, and to ensure that the overall deployed application is

[1] This work was supported by the European Commission project Component+, project no. IST-1999-20162.

working correctly. In contrast, the late integration implied by component assembly means there is often little opportunity to verify the correct operation of applications before deployment time. Although component developers may adopt rigorous test methodologies, with non-trivial software components it is impossible to be certain that there are no residual defects in the code - formal proof or 100% test coverage are not viable options in most practical cases. Compilers and configuration tools can help to some extent by verifying the syntactic compatibility of interconnected components, but they cannot check that individual components are functioning correctly (i.e. that they are semantically correct), or that they have been assembled together into meaningful configurations (i.e. systems). As a result, components that may have behaved correctly in the sanitary condition of the development-time testing environment, may not behave so well when deployed in a system where they have to compete with other (third party) components for resources such as memory, processor cycles and peripherals.

Realizing the ultimate vision of component-based development is therefore contingent on individual components having the built in ability to check their respective deployment environments, and systems of components having the built in ability to check that they are collectively behaving as expected. Only then will the true benefits of the "plug and play" vision promised by component-based development become a reality ([1]). The Component+ BIT technology directly addresses this need by extending the component model to incorporate in-situ, run-time tests that can be performed without manual intervention (www.component-plus.org).

The structure of document is as follows. In the first part the framework of the Component+ BIT technology is given. This framework defines the C+ BIT architecture, specifies the types of the BIT components and describes two main techniques of the technology, that is contract testing and quality of service (QoS) testing. In the second part, description of the technology validation software experiment is given. This description includes specification of the experiment pre-requisites, the experiment itself, and its qualitative as well as quantitative results. The technology framework as well as its validation have been done within the Component+ EU project.

2. C+ BIT framework

In the first phase of the project the five original project partners (The Swedish Institute of Production Engineering Research; Southampton Institute; Fraunhofer Institute for Experimental Software Engineering; PHILIPS Semiconductors Systems Laboratory, Southampton; Fundacion LABEIN, Bilbao; Engineering Ingegneria Informatica S.p.A, Itally; University of Pau, France) developed the framework of the C+ BIT technology. This technology proposes two techniques to test software components: contract testing and quality of service (QoS) testing ([2]). Both techniques are described in the next sections.

2.1. Contract testing

The relation between two software components in a system is supported by a contract in which each party's obligations and rights are defined [5]. Pre-conditions and post-conditions are associated with the operations a server offers. A client calling an operation has the obligations to respect the pre-conditions and the server has the responsibility to respect the post-conditions of a called operation. Both client and server considered as components have invariants ruling the overall interaction.

Definition: *Built-in contract testing represents an in-built verification mechanism to check whether an acquired server is correct according to its client's expectation or whether a client provides the correct operational environment for an acquired server. That is it verifies that client and server abide by their contracts.*

From the definition follows that contract testing basically concerns two pieces of software and their mutual interaction. It further follows that testing of the contract between components has two fundamental aspects:

- To verify that a server implements the semantics of what its clients have been developed to expect.
- To verify that a client behaves according to what a server has been developed to expect.

To make such test possible in a dynamic environment test software has to be built into the components, both the server and the client. In addition separate testers can be developed together with the component. As with built in tests such testers should preferably be developed before or in parallel with the component. An investment in the verification infrastructure in early phases of the development of a component pays off already in the later phases. When the component is reused the return on investment is still greater. The developer using the component can check its behavior and the component can check if it is deployed in an inappropriate environment.

A vendor can offer separate testers for components with BIT functions. If well documented such testers give a user of the component a good picture of the intended use of the component. Tester can do lighter or heavier test of the component and the choice of testing level represents a point of variability in the tester. This variability can be handled through the configuration interface. ([3])

The interaction between a server and its clients is in principle asymmetric: A client knows which server it connects to and which services its asks for while a server normally does not know its clients. Hence the importance of testing the contract when a server is acquired by another component.

2.2. QoS testing

The component-based software development paradigm adopts the approach that software systems are built from aggregates of software "black boxes" that are connected in some way. The use of a component is based on a two-way contract; the component guarantees to

conform to its specification provided the surrounding system conforms to its side of the contract. Whilst testing verifies (to some extent) that the component conforms to its side of the contract, there is normally no mechanism to confirm that the system conforms to its side of the contract. Therefore there is a requirement to check that the integrating system is conformant to the component contract. Quality of service testing is concerned with verification of the interaction between a component and the rest of the system in which it is deployed. Quality of service testing is not only concerned with the functions and behavior the component offers but also its quality attributes and the surrounding system. Since usage varies over time (and in real-time systems depends on asynchronous events), correct component usage must be continually verified.

Definition: *QoS testing comprises a set of techniques, interfaces, and instrumentation, which facilitates, in whole or part, the continuous verification of component and system behavior and the localization of defects.*

The focus is on those aspects of component correctness, suitability for its purpose, and quality attributes that cannot be fully established during the component development cycle. Normally, a component will be tested by the development team in some kind of test environment but there will be no mechanisms to verify the behavior of the component once it is instantiated into the target system. Quality of service testing aims to address this issue by including permanent instrumentation of the component with mechanisms that can verify correct behavior in the target environment throughout the lifetime of the component.

Most software products have residual faults when they are deployed. Given good models of defects in software systems it is possible to devise built-in checking mechanisms to detect errors that occur when residual faults are triggered.

Components executing in a system compete for shared resources with other components. Conflicting demands are difficult to detect during development and cannot be detected by a component itself. Only the system can detect them during usage. BIT in the components can be of great use to ensure the reliability of the system.

Debugging is the art of finding faults once they have manifested themselves through errors and failures in the system. Debugging is hard in real-time systems. Functions for logging and tracing are of great help and can be realized as BIT functions.

There are a number of testing functions that can verify the correct behavior of a system during usage. They can also increase the reliability of a system and help a developer to find defects.

2.3. Common BIT architecture

The C+ BIT architecture identifies four types of components: BIT components, Testers, Handlers and System constructor [9].

BIT components are the components that can be placed under test or provide information for use by testers. BIT components may implement internal tests, or they may simply provide information that is acted upon by external testers. The tests and test information can be appropriate for contract testing, performed at (re)configuration times, or

QoS testing, performed during on-going execution while the system is fulfilling its normal obligations.

Testers collect information from BIT components and use that information in tests. For contract testing, they essentially are responsible for running self-contained tests on individual components, while for QoS testing the tester will usually collate information from several BIT components to obtain meaningful test data.

Handlers are components handle the results of tests. The Component+ BIT architecture allows for testing, but does not specify how the results of tests should be handled. The appropriate actions will be highly dependent on the system. So specific handlers are not defined by the architecture, but the interface through which they are notified of the test results is specified. Since BIT components may incorporate their own tests, handlers may interface to testers or directly to BIT components.

System constructor is a component responsible for the instantiation of (high level) BIT-components, testers, and handlers, and their interconnection.

The BIT architecture specifies a number of interfaces that BIT-components, testers, handlers and system constructors must provide. In the case of handlers and system constructors, however, much is left to the system designer as error processing is application specific and the method of system construction (i.e. component instantiation and association) varies. The BIT approach does not advocate the use of any particular component architecture, nor is it confined to any one implementation language. Only those aspects that must be specified for compatibility are specified, leaving sufficient flexibility and extensibility for wide application. This typically means only the definition of public interfaces, data types, and constants.

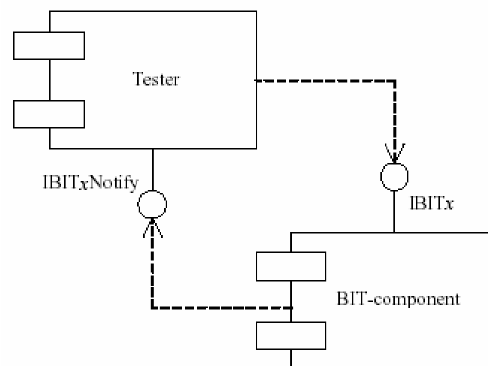


Figure 1. IBITx and IBITxNotify interfaces

In general, information from a BIT-component can flow in two ways. First, an external entity, such as a tester, can access the information via methods of the appropriate BIT interface, in a polling mode. Second, a callback mechanism can be established, where appropriate, so that external entities are notified when events of interest occur. At least one, and possibly both, modes of operation are supported by each BIT interface. By convention, interfaces to test services provided by a component are named *IBITx*, where x is the type of test catered for. Where an external tester is involved, and communication is required from

component to tester, the corresponding interface on the tester is named *IBITxNotify* (see Figure 1). As an example, to facilitate deadlock testing, components may provide an *IBITResource* interface. A deadlock tester provides the *IBITResourceNotify* interface, through which a component can notify the tester of events of interest, in this case thread and resource creation/destruction, and resource requests, allocations and releases. Similar relationships may exist between, for example, handlers and BIT-components, and handlers and testers. There can be any number of test interfaces, to address a wide variety of problems, including, but not limited to; deadlock testing, timing verification and performance profiling, memory management, code integrity, data integrity, and trace facilities.

Whilst any number of test interfaces may be specified, four "primary" interfaces are defined which form the foundation of the architecture. The primary interfaces are:

- *IBITQuery*: allows an external entity to query the availability of specific test services and, if present, to acquire a handle to the corresponding *IBITx* interface.
- *IBITError*: provides the means for error propagation. The *IBITError* interface can be queried to determine the error status, or can be configured to notify a corresponding *IBITErrorNotify* interface of the occurrence of specified error events
- *IBITErrorNotify*: provides a mechanism for error notification, as typically provided by handlers so that they can be informed by BIT-components and testers when specified error events occur.
- *IBITRegister*: provides the mechanism for associating BIT-components with testers and handlers. This interface provides methods for notifying the creation and destruction of BIT-components. Through this interface, a specific "system constructor" can be created for the instantiation and connection of BIT components.

Table 1 defines which interfaces are mandatory on which BIT artefacts. Note that, since the *IBITError* interface supports polling as well as a callback mechanism, *IBITErrorNotify* is not mandatory, but would typically be provided by most handlers. The BIT-component can be configured by a handler to utilize its *IBITErrorNotify* interface if provided. Note also that a BIT-component is defined as such by the provision of the *IBITQuery* interface - it does not have to provide any other test facilities. Thus, compatibility with the BIT architecture is easily achieved. The provision of a pair of corresponding interfaces, such as *IBITError* and *IBITErrorNotify*, would be typical in a BIT environment, as some flexibility is necessary in the way in which information is distributed. Support for both polling and callback modes of operation (i.e. pulling and pushing of data) offers the flexibility required to meet the demands of a wide spectrum of application areas.

Table 1. Mandatory BIT interfaces

Element	IBITQuery	IBITError	IBITErrorNotify	IBITRegister
BIT-component	mandatory	typical		
Tester	mandatory	mandatory		mandatory
Handler	mandatory		typical	mandatory
System constructor				mandatory

2.4. Example configuration

Figure 2 illustrates an example BIT system configuration comprising three BIT-components (one of which is a child of a higher level BIT-component) with resource monitoring support, an external deadlock tester, a handler, and a system constructor, showing the interfaces involved. Associations related to the *IBITQuery* interfaces are not shown for clarity – these associations are established during registration via the *IBITRegister* interfaces of the handler and tester. The secondary interface, *IBITResource*, supports resource and thread event notifications, and can communicate with an *IBITResourceNotify* interface as defined in [10].

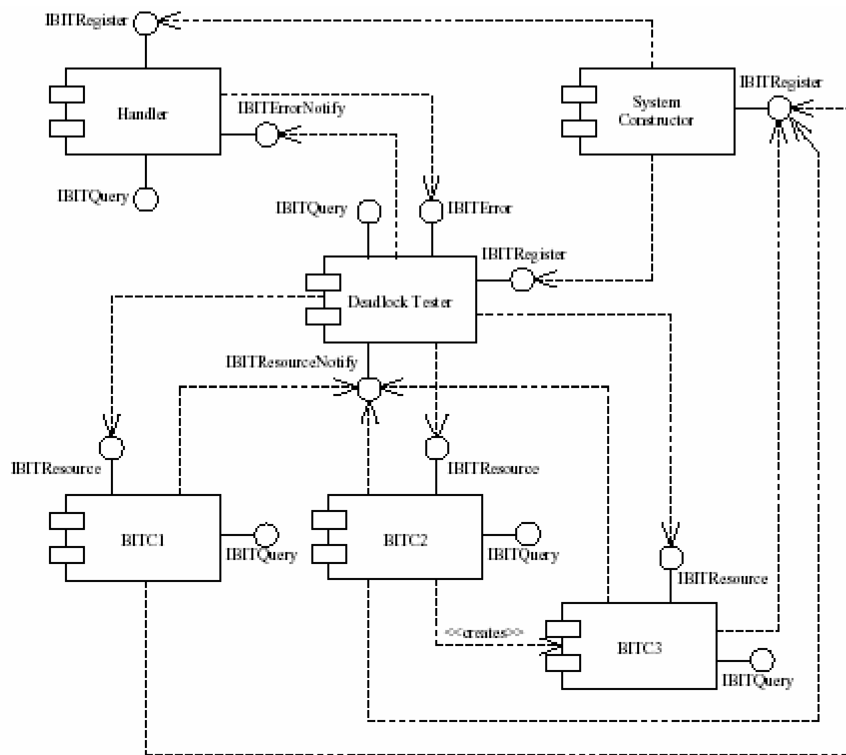


Figure 2. An Example of system configuration

A representative sequence of component interactions is illustrated in Figure 3. In this example, the system constructor is implemented such that it creates the system's initial (top-level) components, including handlers and testers. It maintains an internal table of handles to the *IBITRegister* interfaces of handlers and testers so that they can be notified of the creation of new BIT-components.

On start-up, the system constructor first creates the handler, passing it a handle to the system constructor's *IBITRegister* interface. The system constructor is then called back using the *IBITRegister :: NotifyCreation* method which is passed a handle to the handler's *IBITQueryInterface*. This handle is stored by the system constructor in an internal table. The system constructor then creates the deadlock tester in a similar way, and when called

back, it notifies the handler of the tester's creation. The handler queries the tester for the presence of an *IBITError* interface (which it has, being a compulsory provision on testers). Having acquired a handle to this interface, the handler is able to configure it, using the *IBITError :: SetNotifyInterface* method, so that the tester will notify the handler of error conditions.

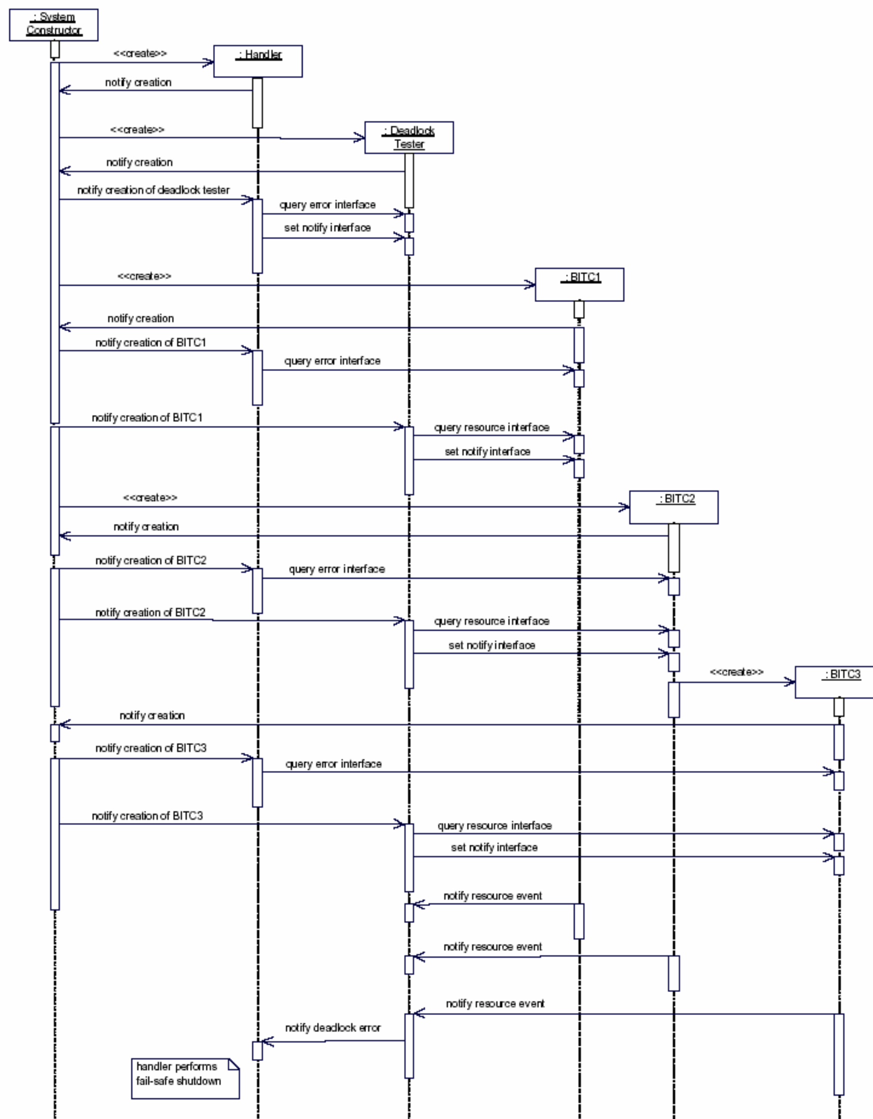


Figure 3. Sequence diagram of representative component interactions

The system constructor then creates BIT components BITC1 and BITC2, again passing a handle to its *IBITRegister* interface. The handle is stored by each BIT-component for

future use. The system constructor is then called back by the BIT-components, which in turn notifies the handler and tester of their creation (it could have done this directly, since it created them, but this provision is made primarily for components lower in the hierarchy). In this example, the BIT components only support deadlock testing (via the *IBITResource*) interface and consequently have no *IBITError* interface. The handler therefore ignores them. The tester, however, queries their services via the *IBITQuery* interface and confirms the presence of *IBITResource* (the only interface of interest to this particular type of tester). Their resource interfaces are then configured to notify the tester of resource related events, such as resource allocation and release, via the tester's *IBITResourceNotify* interface.

Whilst the communication between the system constructor and the initial (high-level) components that it creates might at first seem excessive or unnecessary, using this generalised method of component creation automates the registration process, ensuring that all system level components are aware of the existence of new BIT components not created directly by the system constructor. In this example, BITC2 creates a child component, BITC3, passing the handle it stored when BITC2 was created. BITC3 uses this to notify the system constructor of its creation, which then proceeds, as before, to notify the tester and handler of the existence of this new component. In this example, the handler does not find an *IBITError* interface, but the tester finds an *IBITResource* interface and configures it as previously described. Thus, the connectivity illustrated in Figure 6 is achieved.

During execution, resource related events are notified to the deadlock tester, which responds to each event by determining whether a deadlock condition exists (e.g. by constructing and reducing a Resource Allocation Graph, RAG). When this occurs, the handler is notified of the error via the *IBITErrorNotify* interface. The handler can then take whatever application specific error processing is required. This example shows a number of resource notifications, and the subsequent identification and handling of an error event by some form of fail safe shutdown.

3. Technology evaluation – EC experiment

The second phase of the Component+ project was focused on the technology evaluation. In this part Rodan and four other partners (Technical University of Tallin; 4DSoft, Hungary; University of Rouse, Bulgaria; ISoft, Bulgaria) from Newly Associates States (NAS) validated and verified practically the C+ technology. This work was based on comparison of the quality, development and maintenance cost for non-BIT and BIT components. Both qualitative as well as quantitative results were collected. The main aim of the experiment was to answer two basic questions: if the C+ BIT technology may be applicable and whether it is useful to apply.

3.1. Pre-conditions

In order to assure credibility of the experiment a set of its pre-conditions was made ([6]). The mentioned conditions concerned three areas: software development process, development team and specific of the developed components. There was an assumption that

all the mentioned areas but the C+ BIT technology itself were stable and would not be changed.

Three NAS partners developed two versions of the same components: non-BIT and BIT ones. Such approach resolved the problem of function equivalence of the components (the same functionality) but introduced high risks for the experiment reliability in the area of the development team (i.e. how to avoid communication between two teams and how to assure similar level of their competence).

Two remain NAS partners, also Rodan, decided to use the results of developing former versions of their components (i.e. versions had developed before the C+ project started) and to compare them with the new ones (equipped with C+ BIT technology). For these former versions appropriate software metrics has already been collected. In addition, there was no problem with different competence of the project team since the new version was implemented by the same team (experience). This approach, however, assumed that the functional complexity of the both versions was similar.

Rodan Systems carried out the experiment for two software components: OfficeObjects® Workflow Manager and OfficeObjects® Role Manager.

OfficeObjects®WorkFlow Manager (OOWM) manages business processes in organisations. It is responsible for definition, execution as well as monitoring of workflow processes. From end users' point of view, OOWM manages user tasks according to the workflow processes definition and presents them in the users' task lists (or to-do lists). The algorithms to assign users to tasks are focused on effectiveness and efficiency and therefore take into account users' capabilities and their current task loads. In addition, these algorithms leave also some space to rather seldom business situations (through so-called 'ad-hoc' user assignment). To assure appropriate level of supervision, business processes owners are provided with interfaces for process monitoring and possibility of implementation of corrective actions, if necessary.

OfficeObjects®Role Manager (OORM) is a component to manage users, their groups and privileges and to model organisational structure. This quite frequent functionality is present in virtually all information management applications. Rodan's ambition, however, is to develop extremely generic and flexible module being able to satisfy even most demanding requirements of potential customers. If we succeed, the module will be exploited in all applications developed by Rodan. The component is supposed to replace currently used component (OfficeObjects®MZU) that has been deployed within all OO Portal's applications. Since the requirements for the new component significantly differs from the requirements for MZU and the planned implementation environment is different, we treat the OORM development as a completely new product (rather than an increment of the existing product).

3.2. Experiment

The experiment was carried out by Rodan as follows ([7]). In the first stage the BIT architecture has been designed This architecture included:

- two BIT components: OOWM and OORM – BIT versions,

- two contract Testers: OOWM Tester and OORM tester that simulated the set of possible contracts between these components and other components,
- two QoS testers: Timing tester (developed by Phillips) and Task Assignment Tester,
- a Handler based on Log4J package and developed by IVF.

For every BIT components functional and testing interfaces have been specified. All of the testing interfaces have been compliant with BIT interfaces defined in the C+ BIT Vade Mecum [10]. For OOWM Tester 18 test scenarios and 70 test cases were prepared. For OORM Tester 7 test scenarios and 30 test cases were prepared.

In the next stage the detailed architecture of the every component has been designed. Implementation phase included development of all the mentioned Rodan components.

3.3. Results

At the end of the first phase of the C+ project, the original partners summarized the qualitative results of application of the C+ BIT technology. These results were updated and extended after the second project phase, carry out by the NAS partners. In addition, during the second phase a set of basic metrics for software development process (both BIT and non-BIT) were collected. These metrics were then used to carry out a quantitative validation of the C+ BIT technology. The most important qualitative as well as quantitative results achieved by Rodan are presented in the following sub-sections.

3.3.1. Third party companies may prepare their own contract tests

The software companies that use third-party software components are able to define their own test cases and test scenarios and execute them using their own BIT Testers. The vendors of those components may also support them with standard, open-source Testers which may be adopted or extended of additional test cases and test scenarios.

3.3.2. Shortening test cases by using the BIT getState operations

Designing contract tests we observed that extracting results of individual test cases using the functional interfaces may be arduous and introduce additional defects into the Tester itself. For example, verification of the state of a process instance (i.e. its three attributes) required extraction of the full information about the process written in XML. This operation was executed in almost every test case. Implementation of a specialized *getProcessState* function to get state of the process instance allowed us to decrease drastically the size of test cases/scenarios and, implicitly, reduce the chance of introducing additional defects to our Testers.

3.3.3. BIT setState operations may be used to test time constraints

Testing the OOWM component, we had to simulate violation of time constraints (negative test cases). Since it was not possible to change the system date on the server (the server was used by several other project at that time), we developed a *setProcessState/setActivityState* functions that changed the process and its activities deadlines artificially and forced time constraints violation. This function has been added to the test interface of the OOWM component because it changed the internal state of the process instance or its activities.

3.3.4. Automation of contract testing may reduce time to carry out test scenarios

The effort for designing test cases and test scenarios was extended of developing the mentioned Testers but it was compensated by drastically reduction of the effort of carrying out the tests. For example, for the OOWM component the latter effort was decreased 16 man-hours for non-BIT version to 0,5 man-hours for BIT version. It should be noted however that this feature is connected with automation of all testing techniques not only contract testing.

3.3.5. Early thinking about testing – less cost for detection and repairing of faults

The C+ BIT technology requires early design of test cases and test scenarios. This feature enabled us to see our components from other perspective (i.e. testing one) and to analyse its functionality once more. Thanks to this approach we were able to detect several serious design defects in our components.

3.3.6. Better detection of defects related to a bad system configuration

We observed that one of the main advantages of the C+ BIT technology is decreasing effort connected with component deployment. Thanks to contract testers we were able to verify easily whether all of the requirements for the components environment (i.e. other components, system variables, resources, etc.) are satisfied for a given installation.

For OOWM, installation and deployment for non-BIT version at one of our customer was 40 man-hours while for BIT version at a new customer in a different system was 18 man-hours. In our opinion, it is very likely, that applying the C+ BIT technology may reduce deployment time for components by half.

3.3.7. QoS testing – a way for verification of complex algorithms

QoS testers may also be used to verify complex algorithms, since usually verification is much less complex than the algorithm itself (e.g. Hanoi Tower algorithm may be verified by a program with linear complexity).

3.3.8. An independent integrity tester

QoS testing gives an opportunity to develop an independent integrity tester, independent in the sense of different database management systems (DBMSs). Before, Rodan developed such testers using native DBMS mechanism such as constraints, triggers and stored procedures. The maintenance cost for such testers regarding many DBMSs was quite large. In the C+ project we developed an integrity tester that has been used for all DBMSs supported by out systems. Our maintenance cost has been reduced significantly (the number of testers was decreased five times).

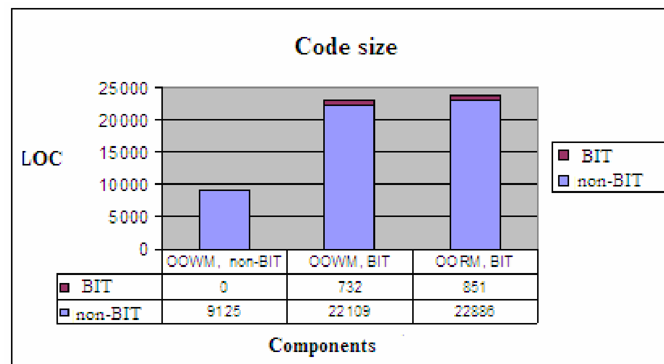
3.3.9. QoS Timing tester as a way to check that something is wrong with the system

Using the Timing tester we were able to check that the system services are not responding on time. Since we set large value for maximal duration of the component services, we were sure that if a given duration constraint was not satisfied, it was very likely that the service hanged.

Such information was crucial, when our company has had very short time to react for system defects. As soon as a duration constraint was violated we were immediately informed that there are some problems with our system and therefore we were able to verify and repair them before they were formally reported by our customer.

3.3.10. The C+ BIT technology does not increase the size of the code significantly

Usage the C+ BIT technology did not increased the size of our components significantly. For OOWM and OORM components their size was increased of about 700 lines, which was about 6-7% of their original size.



	OOWM, non-BIT	OOWM, BIT	OORM, BIT
BIT	0	732	851
non-BIT	9125	22109	22886

Figure 4. Comparison of the size for non-BIT and BIT components

On the contrary, the size of contract Testers was significant – about 10-15% of the size of the BIT components. It was mainly before of large number of test scenarios provided by these testers.

The size QoS testers was very small because we used already developed testers from other partners (and just to adopt them) or they were very simple (Task Assignment Tester).

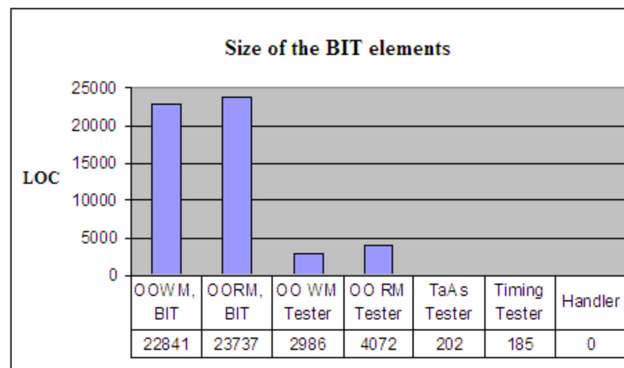


Figure 5. Comparison of the size of BIT elements

3.3.11. Reusability of QoS testers

QoS testers are focused on testing a given service for many different components (also from different vendors) and may be used in many different systems. For example, so far Rodan used QoS Timing tester in two systems: OfficeObjects® Portal (during the C+ project) and OfficeObjects® DocMan (after the C+ project).

3.3.12. The C+ BIT technology - development effort

In our opinion, the C+ BIT technology should not increase significantly the overall development effort for large components (15-20 KLOC). As it was shown in the case of Rodan, the additional effort was about 2-3% of the overall development effort. On the other hand, as it was shown by ISoft ([4]), for small-size components (1-2 KLOC), the overall effort my increase after applying BIT of 60%.

3.3.13. Software quality

Assumption that the C+ BIT technology increases software quality has already been partially verified. Partially, because the deployment phase of the developed components has not been finished yet and the number of defects found is still not the final one.

However, Rodan observed that in OOWM none of the critical defects was reported at the deployment phase so far. In addition, other NAS partners reported greater number of defects found in BIT versions during testing that in non-BIT ones. Hopefully, it may be connected with the less number of remaining defects in the BIT components.

Considering problems existing with the C+ BIT technology it should be underlined that so far there is **no mechanism to test the technology**. It is especially important if many test cases and test scenarios are considered. **Timing tester has to be used cautiously**. If to many services are monitored and the Timer resolution is too small (i.e. frequency of doing verification that checks if the monitored services have been finished), it may cause **serious reduction on the system performance**. The solution of this problem is to monitor only selected services and to decrease frequency of the Timer resolution for checking duration constraint.

4. Summary

This paper presents the framework of the technology and its preliminary evaluation done in a software experiment within the Component+ project. In this experiment Rodan Systems together with other four NAS partners verified effectiveness and usefulness of the technology.

On the base of our results we think that this technology is applicable, and, if it is used carefully, it is worthwhile to apply. So far we use this technology to test four of our components and we plan to use it for the other Rodan's components.

References

[1] Jones C., *Software Quality. Analysis and Guidelines for Success*, International Thomson Computer Press, 1997.

[2] Edler, H., Hörnstein, J., *Test Reuse in CBSE Using Built-in Tests*, Workshop on Component-based Software Engineering, Composing systems from components, Lund, 8-11 April 2002.

[3] Gross, H., G., *Built-In-Test Vade Mecum – Part II, Built-In Contract Testing: Method and Process*, Fraunhofer Institute Experimental Software Engineering Germany, Component+.

[4] Hörnstein, J., *Assesment and evaluation addendum*, D6.2Ad, Component+, 2002.

[5] Meyer, B., *Object-oriented software construction*, Prentice Hall, 1997

[6] Momotko, M., Nowicki, B., *Testowanie wbudowane. Europejski eksperyment walidacyjny*, 4th Polish National Software Engineering Conference, Poznań, , Poland, Oct 2002.

[7] Momotko, M., Zalewska, L., *Rezultaty zastosowania technologii C+ BIT do testowania komponentów programowych*, 5th Polish National Software Engineering Conference, Szklarska Poręba, Poland, Oct 2003.

[8] Szyperski, C., Component Software: Beyond Object-*Oriented Programming*, Addison-Wesley 1999.

[9] Vincent, J., *Built-In-Test Vade Mecum – Part I, A Common BIT Architecture*, Bournemouth University, UK, Component+, 2002.

[10] Vincent, J., *Built-In-Test Vade Mecum – Part II, Interface Specification:Types, Syntax and Semantics*, Bournemouth University, UK, Component+, 2002.