
Evolutionary Testing in Component-Based Real-Time System Construction

Hans-Gerhard Groß

Nikolas Mayer

Fraunhofer Institute for Experimental
Software Engineering

Sauerwiesen 6
D-67661 Kaiserslautern, Germany;

{grossh,mayern}@iese.fhg.de

Abstract

Execution time analysis is an essential verification activity during real-time system development. This activity can be performed dynamically through optimisation-based analysis techniques such as evolutionary testing. Evolutionary testing is already successfully used under the traditional procedural development paradigm. This paper is an initial attempt in making evolutionary testing also applicable under the more recent object-oriented, component-based software development paradigm. Here, the application of evolutionary testing is more difficult compared with the traditional procedural development approaches because component-based architectures are inherently encapsulated, and often represent state machines. This work proposes a solution that makes evolutionary testing available in component-based real-time system construction. It is based on built-in testing interfaces and on the execution of an object's invocation history through evolutionary testing.

1 INTRODUCTION

One of the most important motivations for the application of component-based software engineering techniques in practice is that new applications can be created with significantly less effort than in traditional approaches, simply by assembling the appropriate prefabricated parts. In popular computer terminology this is captured by the “plug and play” metaphor. As soon as the relevant parts have been “plugged” together, they should be able to “play” with each other in the resulting system. However, contemporary component technologies are still some way

from realizing this vision, especially when component-based real-time applications are considered.

With traditional approaches, the bulk of system integration work is performed in the development environment, giving engineers the opportunity to pre-check, the compatibility of the various parts of their system. This ensures that the overall application is working correctly. In contrast, the late integration implied by the assembly of readily deployable components, means there is little opportunity to verify the correct functional and non-functional operation of the resultant application collectively before deployment in the run-time environment. Although, component developers may adopt rigorous test methodologies and deliver fault-free components, there is no guarantee that an assembly of such components will be free of residual defects. In fact, even totally fault-free components may cause failures if they are connected to the wrong kind of components at deployment time. Compilers and configuration tools can help by verifying the syntactic compatibility of interconnected components, but they cannot check that individual components are functioning correctly in the scope of the application. This means that they are semantically correct (functionally as well as non-functionally). As a result, components that may have behaved correctly under the sanitary condition of the development-time testing environment may not behave so well when deployed in the actual run-time environment.

This is even more apparent when non-functional requirements are considered, for example the compliance of the application to a real-time schedule. Such real-time requirements are not only affected by individual components in a component-based real-time system, but also by the entirety of all components that make up an application. Each individual component may have a well-defined timing behaviour on a particular platform. However, this is completely changed if it is plugged to other

components that collectively implement functionality of a real-time application. The timing behaviour of each possible combination of a component's feasible usage profiles in respect to other components must therefore be verified when the compliance to the timing schedule of such a system is tested. Clearly, this cannot be done a priori for each component since the developer of that component can never anticipate its usage in a particular context. Timing verification can only be performed when components are assembled and put together into a new configuration. Although, the effort involved in plugging components together may be relatively small, therefore, the effort involved in verifying that the resulting assembly of components works as expected, and shows the expected run-time behaviour, may be much greater. The savings that are promised by component-based development may thus be wiped out by the extra effort needed at integration and deployment time to verify that the resultant application is acceptable.

This paper is an attempt to lay down the foundations of an architecture and a methodology that permit the dynamic, optimisation-based timing analysis of real-time applications made up of prefabricated, readily available and deployable parts. The introduced work draws its ideas from evolutionary testing technology, which is currently successfully applied to non-component architectures [3, 4, 5, 6, 7] and new advances in the testing methodology of component-based systems [2].

The following section introduces the terminology of evolutionary testing as optimisation-based timing analysis technique, and the problems that inherently encapsulated, object-oriented software components pose to testing. Section 3 proposes a component architecture with auxiliary testing interfaces that may help to overcome these problems. Section 4 discusses how the technology may be used in larger-scale applications and identifies and resolves the problems that this technique is creating. The outcome is an approach to testing the timing behaviour of component-based real-time systems that is based on executing and optimizing the invocation history of an object's transactions. Section 4 also illustrates the technique with a small example application. Finally, section 5 summarizes and concludes the paper.

2 EVOLUTIONARY TESTING AND COMPONENTS

Evolutionary Testing (ET) applies evolutionary algorithms to typical software testing problems [6, 7]. The target is to find optimal tests for a given test criterion. Dynamic timing analysis can be defined as software testing with the violation of the timing schedule as test criterion. This, in fact, represents a typical optimisation problem that can be tackled by an evolutionary search process. The optimisation parameters of this process are represented by the parameters of the test cases. This is a

vector that defines an input scenario for the task under test. The cost function is determined by the time it takes to execute the task with a given input combination. The evolutionary algorithm realizes an optimisation over a task's input parameter combinations, and this optimisation is guided through the time it takes to execute the task with the given set of inputs.

Evolutionary testing is already successfully applied to dynamic execution-time analysis problems. This is demonstrated in the literature, for example [3, 4, 5, 6, 7]. However, the technique has not yet been applied to timing analysis of object-oriented, component-based real-time systems. The essential difference between the traditional procedural development paradigm and the more recent object-oriented development paradigm lies in how data and functionality are treated. Whereas the procedural approach propagates a strict separation of data and functionality, the object or component paradigms encourage the exact opposite. This is the combination and encapsulation of data and functionality. This causes a fundamental difference in how modules of the two kinds are treated during testing. In a strict sense, "traditional" procedures are tested from outside their encapsulation boundary, which means that they are executed with some input, whereas objects must be tested from within their encapsulation boundary. Though, this only applies if the smallest test unit is considered and the object is state-based. Sole functional components are not state-laden and represent a collection of procedures that can be tested from outside their encapsulation boundary. All other state-laden objects are different as explained in the following.

The smallest test unit is a single procedure in the "traditional" development paradigm. It is a class that means an encapsulated assembly of attributes and methods, in the object-oriented paradigm. Of course, in the first case, one could think of a module comprising an assembly of different functions which all have access to global parameters. This would be representing a module as encapsulating some coherent functionality. In fact, this is how some object-oriented languages are actually implemented. However, the object paradigm abides by this principle in a much more natural and consequent way, in that a class has contents and a name space, it encapsulates some coherent functionality, and it denotes a single abstraction; hence the class is the smallest test unit in object-oriented development.

This difference in organisation has a significant bearing on the definition of tests for each paradigm. A test for a simple procedure comprises

- some input, this is a list of attributes that the tested procedure accepts as input;
- some event, this is the name of the procedure that will be tested;

- some output and behaviour, this is the returned result of the function's calculation and observable behaviour (e.g. exhibited timing behaviour);
- some expected output and behaviour according to the specification, this is what the procedure is expected to calculate for the given set of inputs, and the expected defined behaviour (e.g. timing according to the timing schedule).

Dynamic timing analysis concentrates on the exhibited timing behaviour, rather than the calculated results.

In contrast to simple procedural testing, object testing typically considers also state-transition testing. Here, a test case is also defined through the internal states of the component. It therefore comprises

- some input plus some initial state of the object, this is a state for which the event will be valid (preconditions). The state is defined through the combination of the internal attributes of the object;
- some event, this is the name of the tested operation of the object. This event will typically cause a state-transition, and it will put the object into a final test state.
- some or no output and behaviour plus some final state. The fact that no output may be generated for an object is quite typical. Many objects are accumulating the effects of external stimuli internally without any noticeable effect outside the encapsulation boundary.
- some expected output and behaviour plus some expected final state according to the state model and the specification (post-conditions).

This difference in testing has an essential bearing on how dynamic execution time analysis must be organised and applied in an object-oriented, component-based architecture. All internal state variables are by definition hidden to outside entities of an object. This also includes the test software, in this case represented by the evolutionary algorithm. States can only be accessed and changed through the provided functional interface of the class, and not through some global variables as it is the case in procedural development. In general, only a distinct history of interface operation invocations will define the combination of the internal attributes that makes up an internal state.

3 EVOLUTIONARY TESTING APPLIED TO COMPONENT-BASED REAL-TIME APPLICATIONS

The fact that object-oriented entities are inherently encapsulated, and they also represent state machines, creates a fundamental difficulty for the application of evolutionary testing. In order to generate worst-case

timing behaviour for some operation of an object, an evolutionary algorithm must not only optimise and provide the input parameter values for the operation, but additionally, it must optimise and provide appropriate initial states from which the event will be triggered. The execution time of an operation is defined through the values of the internal state variables plus the input values of the operation. The evolutionary algorithm must set the values for the internal state variables from outside the object's encapsulation boundary. This would require all internal state variables to be made publicly available to external clients of the component.

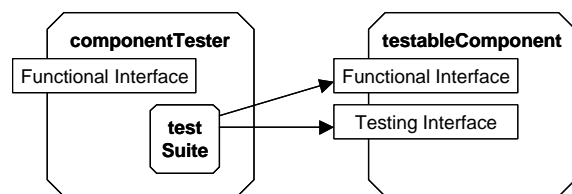


Figure 1: Simplified set up for contract testing.

However, most components will not exhibit their internal state attributes. Especially vendors of commercial components will be reluctant to publish any internal component implementation. Dynamic execution-time analysis that is applied to component-based real-time systems is therefore relying on a specific architecture. This architecture draws its concepts from the built-in contract testing technology developed within the EC Component+ project [2]. Contract testing augments a component with an auxiliary interface that is used to control the initial and final internal states of a component defined through the pre- and post-conditions of each individual test. This set up is depicted in Figure 1. Every testable component provides an auxiliary public interface that comprises operations for internal state setup and state validation. A test suite that is contained in a tester component will use for each individual test

- the testing interface to set the component into the initial state defined through the preconditions of the test,
- the functional interface in order to execute the tested operation with the provided input,
- the testing interface again to check the final state according to the required post-conditions of the operation.

Evolutionary testing requires yet another of these testing interfaces to enable the needed controllability for performing dynamic execution time assessment. This interface can be accessed by an external client of the component, in this instance it is the test software in form of the evolutionary testing algorithm. This uses the evolutionary testing interface to set the internal state

variables of the tested component to the values that are generated within the evolutionary process. The normal functional interface is used to execute the tested operation and provide its input parameter sets generated by the evolutionary algorithm.

Figure 2 illustrates this infrastructure. In this instance, the testing infrastructure extends the functionality of the component. This way of realizing testability is advantageous for a class that will be deployed without the auxiliary testing infrastructure (e.g. in embedded systems).

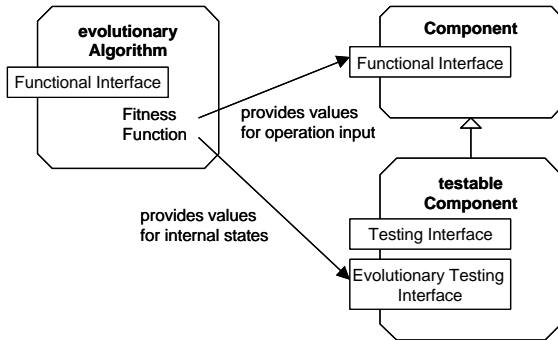


Figure 2: Testing infrastructure for evolutionary testing.

4 SCALABILITY OF THE APPROACH

The previous section has introduced the notation of an evolutionary testing interface with which a component may be augmented so that the evolutionary testing process gains full control over the tested entity. This also includes the component's internal states, without having to expose its internal implementation. The described infrastructure only focusses on execution time verification of one single top-level component. However, component-based real-time applications are typically assemblies of multiple components that collectively realize functionality of the overall system. This means that components, in order to fulfill their own obligations towards their clients, often rely upon acquiring server components that realize part of their functionality. In practice, this means that high-level user functions may be subdivided into transactions that spread over many components in an application.

4.1 PROBLEMS WITH TESTING INTERFACES

If the top-level component contains and relies upon other server sub-components, the evolutionary testing infrastructure can be extended, so that the top-level component exhibits the evolutionary testing interfaces for all of its contained sub-components. This is illustrated in Figure 3. Each contained sub-component can pass the reference of its evolutionary testing interface up to the next component in the nesting hierarchy. This permits the evolutionary algorithm outside the encapsulation boundary of the

component assembly to “see” and access all evolutionary testing interfaces that are contained in the nested hierarchy. Such a combination of multiple components is considered as a single abstraction whose internal state can be set through the externally visible evolutionary testing interfaces of the individual nested components. The tested transaction is executed through the externally visible functional interface of the top-level component. This transaction may be implemented through operations of subsequent server components, so that the execution may spread through the entire component assembly. The technical realization of such a set up with externally visible testing interfaces is feasible for most contemporary component technologies and languages.

However, the fact that the internal states of sub-components can be set externally through the evolutionary testing interfaces permits that internal state variables may be set to values that cannot possibly be generated by the client that uses such a sub-component. The evolutionary testing process may in fact generate a semantic mismatch between the overall behaviour of a component as a stand-alone entity and the behaviour of the same component in the context of a distinct client. A client's usage profile of an embedded server component may not permit certain states of the server to be reached in this particular context, although this server may well permit these states in a different context. The evolutionary testing process that externally sets the states, “knows” nothing about the dependencies between the embedded clients and servers, or their context of usage.

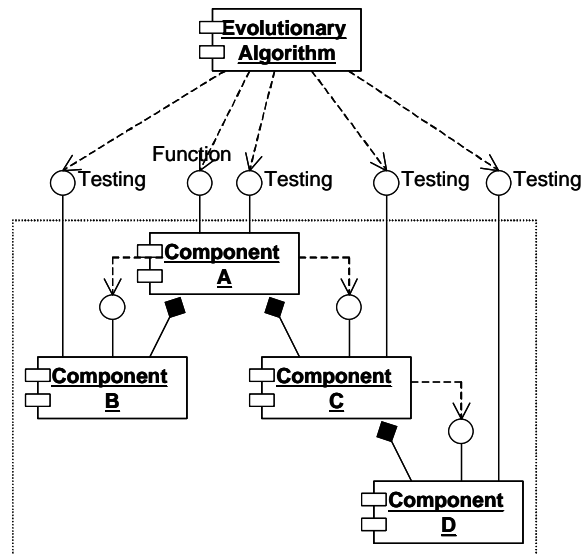


Figure 3: UML style component diagram for an evolutionary testing set up with nested components.

Such dependencies are typical for aggregates of components, and they represent a filtering effect that is increased toward deeper nesting levels in the containment hierarchy of the component assembly. This filtering effect

must be considered in the design of an architecture for evolutionary testing, if it is applied to object-oriented real-time applications.

4.2 DEMONSTRATOR APPLICATION

To demonstrate the importance of the filtering effect in component-based systems, we consider a part of a sample telecommunication system (figure 4).

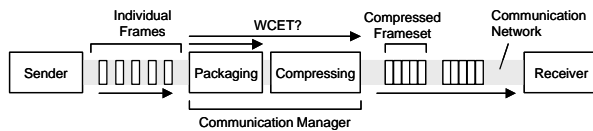


Figure 4: Sample communication system.

Within this system, picture frames produced by a sender are grouped in framesets (of size 5), compressed and transmitted to a receiver. For this discussion, the packaging of the five incoming frames into a frameset and subsequent compression of the frameset define a single transaction. This system is outlined in Figure 4. In order to investigate the worst-case execution time (WCET) of this transaction or of its parts, a detailed description of the system is necessary (see Figure 5).

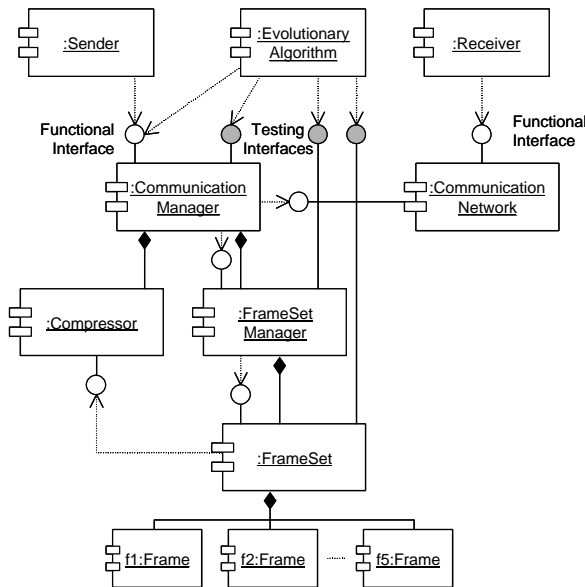


Figure 5: Component architecture of sample communication system.

In order to assure that the compression and transmission only takes place if the frameset is completely filled with frames, the *FrameSetManager* controls the *FrameSet* as shown by the state diagram depicted in Figure 6. The *FrameSetManager* continuously accepts frames in the empty and waiting state until the frameset, to which the frames are dumped, is filled up and throws an *FS_FullException*. This exception triggers the compression of the frameset, and its completion sets the *FrameSetManager* to

the ready state. Furthermore, the *CommunicationManager* is notified by an *FS_ready* signal that the frameset is then ready to be sent. After the frameset has been sent (as indicated by a *FS_SENT* signal), the *FrameSetManager* is set back to empty state.

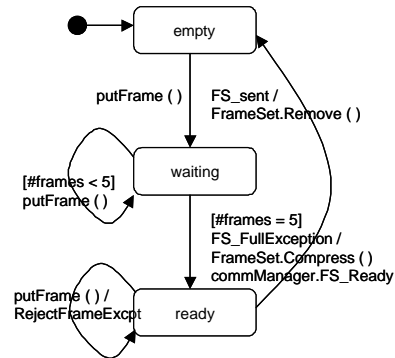


Figure 6: State diagram of *FrameSetManager*.

The behaviour of the *FrameSet* is specified through the state diagram depicted in Figure 7.

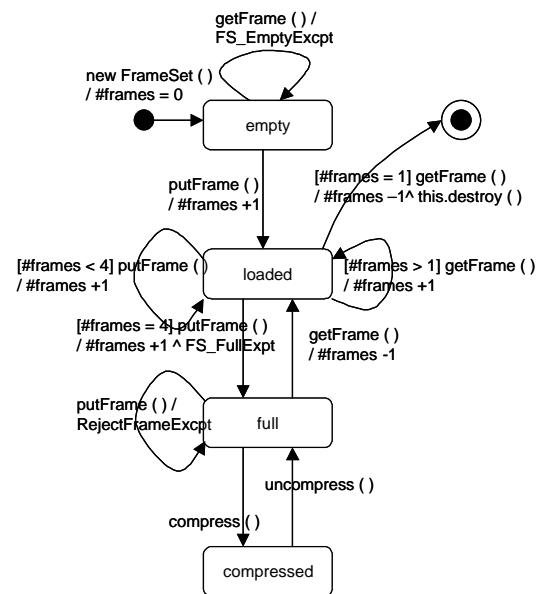


Figure 7. State Diagram of *FrameSet*

The state diagram of the *FrameSet* component shows that the compression of the frameset will trigger a transition from the full to the compressed state. The *compress()* method is only invoked on the *FrameSet* by *FrameSetManager*, if an *FS_FullException* was received.

In order to test the worst-case execution time of the transaction, the testing interfaces of the *CommunicationManager*, *FrameSetManager* and *FrameSet* are must be made accessible to the *EvolutionaryAlgorithm* component. The testing interface of the *CommunicationManager* simply makes the *FS_ready* signal readable to

the evolutionary algorithm. In other words, the frameset was filled and compressed and is now ready to be sent. This is used for measuring the time that the transaction takes until it is completed. The other two testing interfaces provide methods for setting the internal states of *FrameSetManager* and *FrameSet*, so these states can be controlled independently.

A chromosome for the optimization process typically encodes the content of five frames (since in this example a full frameset comprises five frames), the internal state of the *FrameSetManager* and the internal state of the *FrameSet*. As described in section 4.1, testing interfaces allow the independent setting of the internal component states, thereby completely disregarding the filtering effect exhibited by one component using another component in a particular way. This leads to infeasible system states during the optimization process. This is summarized in Table 1 for the presented example.

FrameSet Manager	FrameSet			
	<i>empty</i>	<i>loaded</i>	<i>full</i>	<i>compressed</i>
<i>empty</i>	x	infeas.	infeas.	infeas.
<i>waiting</i>	infeas.	x	x	infeas.
<i>ready</i>	infeas.	infeas.	infeas.	x

Table 1: Infeasible system states that must be considered for optimization.

Table 1 displays the possible combinations of states that may be set through the testing interfaces. However, most of these combinations represent unfeasible states that can never be reached during the normal operation of the component assembly.

4.3 OPTIMIZATION OF THE INVOCATION HISTORY

Infeasible states during the optimization process must be avoided, since this will lead to unrealistic execution times at best, or to system malfunction at worst. Therefore, introducing testing interfaces that would short-circuit component interaction and thus allowing illegal combinations of component states, cannot be allowed.

An alternative approach for testing the worst-case execution time in component-based real-time systems is to augment the chromosome with the component's invocation history. This means that a transaction is considered as a history of events that bring the component assembly into the distinct state for which we expect significant timing behaviour. In other words, for each invocation that is encoded in the chromosome, the system is executed, leading to state changes in sub-ordinate components of the assembly. In this way, the internal states are not set through the testing interface but through the normal functional interface that the component is readily providing. For the example application this means that a chromosome represents five invocations of the *putFrame* method with the five frames that are eventually

stored in the frameset. The evolutionary algorithm therefore optimizes the contents of the five frames. Consequently, the test environment has only external access to the component assembly, so that sub-components are only set to feasible states that the usage profile of their context permits. The transaction candidates that will be subject to evolutionary testing are determined through the behavioural model of the tested entity.

In the example, we expect the worst-case timing behaviour of the *FrameSetManager* component when the last of the five frames is dumped into the frameset thereby triggering the compression. This represents a particular path through the manager's behavioural model (Figure 6) that will be assessed through an evolutionary test.

Method	Execution Time	Additional time
<i>putFrame</i>	time for storing a frame	compression time
<i>putFrame</i>	time for storing a frame	
<i>putFrame</i>	time for storing a frame	
<i>putFrame</i>	time for storing a frame	
<i>putFrame</i>	time for storing a frame	

Table 2: Scenario with the longest expected execution time.

Individual calls to the *putFrame* method will only reveal the timing for storing a single frame in the frameset. In contrast, the combination of five consecutive invocations of that method, and this in fact maps to the full state in Figure 6, will reveal the worst case timing behaviour, since the last call includes the compression algorithm (Table 2). The worst-case timing of the *putFrame* method is therefore only revealed if the component is in the full state (i.e. when it holds five frames). The execution time for the compression depends upon the contents of the individual frames. In order to optimise the previously described scenario, the chromosome will comprise the data for the picture frames. For a test, it is sub-divided into the separate frames to be used as input to the sequence of *putFrame* method invocations. In other words, the fitness function manages the separation of the chromosome into five consecutive operation calls.

This methodology works for sending the information that is encoded in the chromosome into the tested unit. However, the resulting execution time (objective fitness) of a method invocation cannot be passed back to the evolutionary algorithm through the encapsulation boundary without a testing interface. This means that the testing interface of the *FrameSetManager* must be implemented in a way that it can take the timing of the tested transaction store it, and notify the *Evolutionary-Algorithm* component of its termination. This mechanism is realized through a notification interface as part of the testing interface. It is implemented and connected to the testing component as described in [2].

4.4 FINAL ARCHITECTURE AND METHOD

Worst-case execution time checking of the considered component (*FrameSetManager*) comprises that every individual operation invocation is executed and its timing assessed. The items that have to be considered are defined in the functional interface specification, and the behavioural model (Figure 6). The behavioural model represents the individual states and transitions that collectively define the execution time of a transaction. Every path of the behavioural model maps to one specific test, and thus to one single optimization process. The individual optimization processes that must be performed are determined by the state-transitions as summarized in Table 3.

state transition	event chain	chromosome
empty to waiting	putFrame(frame_1)	frame_1
waiting to waiting ($f < 5$)	putFrame(frame_1)	frame_1
	putFrame(frame_2)	frame_2
waiting to ready ($f = 5$)	putFrame(frame_1)	frame_1
	putFrame(frame_2)	frame_2
	putFrame(frame_3)	frame_3
	putFrame(frame_4)	frame_4
	putFrame(frame_5)	frame_5
ready to ready ($f > 5$)	putFrame(frame_1)	frame_1
	putFrame(frame_2)	frame_2
	putFrame(frame_3)	frame_3
	putFrame(frame_4)	frame_4
	putFrame(frame_5)	frame_5
	putFrame(frame_6)	frame_6

Table 3: Invocation histories to be tested according to the behavioural model of the *FrameSetManager* component in figure 6.

The *EvolutionaryAlgorithm* component is configured to perform the four optimization processes outlined in Table 3 in a sequential order. Each represents a search process on its own with a specific chromosome with one, two five and six frames to be optimized respectively. Each search process owns a fitness function that invokes the event chain on the tested object, thereby transforming the generated chromosomes into the respective input parameter sets for the tested methods. The component under consideration must be able to measure the time from the start of the event to its termination and notify the calling component that it is finished. This is realized through the testing interface. Hence, this interface does not contain operations to set internal state information (because this may generate infeasible states), but to extract the timing information from the tested component. Additionally, it provides a mechanism to switch into testing mode, in this case, this corresponds to setting the timer for each method invocation. The outcome of the evolutionary testing process is a number of latencies that represent the execution times for the *putFrame* operation, according to the state from which it issued: empty, waiting and ready. This is for testing one single operation invocation. On a

higher component nesting level, this can also be extended to measure the execution time for a complete event chain. In this case, the measurement of the execution time is taken after performing a complete sequence of operation calls, that represents a full transaction.

5 CONCLUSIONS

The work presented in this paper creates the basis for using evolutionary testing technology in component-based software construction. This comprises an architecture and a method. We have introduced the methodology of optimisation-based timing analysis as an application to object-oriented (and consequently component-based) real-time systems. We outlined an architectural solution that opens up inherently encapsulated entities such as objects or components to the application of evolutionary testing, and this fully addresses the fact that objects and components also represent state machines. An initial attempt to address this was the introduction of a state setting interface that may realize an access infrastructure to permit external control of the internal state variables of tested components. However, we have identified the difficulties that this approach creates for component assemblies. This is that it does not consider the filtering effect on client-server relations of sub-ordinate components. We have proposed a feasible solution that sets the internal states in a natural way through the existing functional interface. This new approach does not optimise the state information directly, but indirectly through the individual input vectors according to the defined invocation history. The specification of these scenarios is derived from the component's behavioural model, and follows a defined process. The testing interface of this new applicable solution is only providing timing information, and it is also used to configure the testing procedure. It does not, however, provide any means to set the internal states of the component.

In the traditional procedural paradigm, evolutionary testing approaches are already successfully used for determining the best- or worst-case task execution times, although they have not yet penetrated the real-time community as one would expect. We feel that extending this technology to more recent object-oriented and component-based real-time development activities can generate significant momentum for the interest in these techniques. This is particularly important since dynamic and automatic verification technologies will be much more sought after in future highly re-configurable "plug, test and play" applications that current component technologies envision. We believe that only through the introduction of suitable and powerful verification mechanisms can the full vision of object technology and component-based development become a reality. The lack of suitable configuration and verification methodologies is actually where most problems are buried in this field.

Acknowledgments

This work is partly supported through the German Federal Department of Education and Research under the MDTS project acronym and through the EMPRESS project under the European ITEA framework programme.

References

1. C. Atkinson, et.al (2001). Component-based Product Line Engineering with UML. Addison-Wesley, London.
2. Component+ Project (2001). Built-In Testing for Component-Based Development. EC IST 5th Framework Programme IST-1999-20162 Technical Report (www.component-plus.org), November 2001.
3. J. Hunt (1995). Testing Control Software using a Genetic Algorithm. Engineering Applications of Artificial Intelligence, 8(6).
4. B.F. Jones, H.H. Sthamer, X. Yang, D.E. Eyres (1995). The automatic generation of software test data sets using adaptive search techniques. In: 3rd International Conference on Software Quality Management, Seville, Spain.
5. F. Müller, J. Wegener (1998). A comparison of static analysis and evolutionary testing for the verification of timing constraints. In: 4th IEEE Real-Time Technology and Applications Symposium, Denver.
6. J. Wegener, H.H. Sthamer, B.F. Jones, D.E. Eyres (1997). Testing Real-Time Systems using Genetic Algorithms. Software Quality Journal 6(2).
7. J. Wegener, M. Grochtmann (1998). Verifying timing constraints of real-time systems by means of evolutionary testing. Real-Time Systems 3(15).
8. A. Watkins (1995). A tool for the automatic generation of test data using genetic algorithms. In: Proceedings Software Quality Conference, Dundee, Scotland.
9. Xanthakis, Ellis, Skourlas, LeGall, Katsikas (1992). Application of genetic algorithms to software testing. In: 5th International Conference of Software Engineering, Toulouse, France.