Testing Reusable Software Components from Object Specification

Muthu Ramachandran
School of Computer Science and Maths
University of Portsmouth
Portsmouth PO1 2EG
United Kingdom

Abstract
Most Consumer Electronics products today contain complex embedded software. We believe a component-oriented approach is an ideal way to handle the diversity of software complexity. Our earlier work on reusable components has addressed the development of the Koala component model has been developed (as an outcome of the ESPRIT project ARES) to address reuse with concept of late binding [1]. This work has been carried out as a part of the research project on ‘Testing Reusable Software Components’. Our approach to testing components was based on the principles of testing from object models. Therefore we have decomposed a COM-like component into OO models. Therefore we are able to generate a volume of key test cases to study boundary value testing and analysis on component interfaces, which is the key to achieve testability and reusability of software components.

Introduction
In consumer electronics products such as TVs and VCRs, software roughly doubles in size every two years. Control functions become more complex, signal processing tasks shift from hardware to software, software intensive features emerge (Electronic Programming Guide), products are integrated (TV-VCR, Internet, Web-TV, DTV, etc.) [3]. Components need to be well modeled, specified, designed, and tested before they can be reused in an application. Components should be explicitly designed for reuse. Traditionally components are often specified in one or more of the following representation(s):

- Component notations
- Formal/semi-formal notations
- Higher level specification languages
- Graphical representation of the source code

Visual notations are useful to understand component connections/compositions. Therefore our intention was to use UML notations to represent component interfaces as modeled in koala components which are COM-like components [1]. This is mainly for conducting boundary value testing on interfaces. Testing during system modeling phase can help prevent errors being introduced during final stages. To this end tools like StP/T (an integrated test tool with CASE) can check, verify, and generate test cases from object models. General guidelines on testing reusable software have been identified to endure testability during the process of component development.

The Koala Component Model (KCM) has been used as a framework to implement reusable components for consumer products. This work investigates techniques on how to test generic components. In Koala, a component is described as a self-contained asset (modeled as ICs) which communicates through interfaces. A component provides functionality through interfaces, and in order to do so it requires functionality through interfaces. The detailed description of the koala model can be found in [1].

Teletext acquisition component is shown in Figure 1, which has been chosen to illustrate different modeling techniques. The component CtxtAcquisition provides facilities for parsing and decoding into pages of EUR-
WST and US-WST teletext packets. Each interfaces contain a pages of datasheet description making testers and product developers hard to understand its purpose in a short span of time during product development and testing.

The above model is fine when the component developers are expert software designers/reuse engineers who may have good understanding of koala components and reuse. However, this has become difficult for test engineers to understand each component in short span of time during product testing and development. Hence why we re-engineered koala components using object models. Also our aim was to conduct automated testing using StP/T which is a tool that can generated test cases from object models.

**Modeling components as objects for automated testing**

The visual notations and diagrams play an important role in any modeling – it is the glue that holds the process together. It serves as the language for communicating decisions that are not obvious or cannot be inferred from the code itself. UML is a language of visual notations used to specify, visualise, and document the artifacts of an object-oriented system under development [2]. UML has also emerged as the industrial de-facto standard. It represents the unification of the Booch, OMT, and Objectory notations, as well as the best ideas from a number of other methodologies.

Figure 2 illustrates modeling in UML for the same component that is shown in Figure 1. The object view of the component model is for two main reasons, a) to conduct automated boundary value analysis testing on data types, values, ranges, etc., and b) it integrates documentation with modeling in order to simplify navigation and understanding.

The following is the simple algorithm used when producing the object view of the component model:

- Component name becomes an object class (For example, CtxtAcquisition)
- Define all the parameters and interfaces (including provides and requires) as object attributes/data types. This is mainly for generating test cases with StP/T, which requires all data types to be declared within the object specification.
- Try and define all the constants and range values in attribute definition part of the object class. In this example, eur-wst: Nat8 = 2..41 and all the interfaces are defined.
- Produce all the provides interfaces (of the Koala component) as interface object and define aggregation relationships (is-part-of). Example, all the provides interfaces IduWst, ItxtDecode, and IsoftwareInit are defined as interface objects.
- Produce all the requires interface as interface object and define associatively relationships (composed-of). In this example, all the requires interfaces Ipagerwst, Ipacket830, ITxtPlatformDiv, and IpagerDisplay are defined as interface objects.
- Follow steps 2 & 3 above for each interface objects in order to complete the definition of attributes and functions/operations.
Prerequisite keep all the interface data sheets handy to fill in the details.
Requires interfaces are connected with dotted lines

In the following section we discuss some of the test case generation strategies to make use of the object models described in this section.

**Testing reusable components**

Testing components remains a challenging area of research. One of the main objectives to validate and test against component properties that are adopted during the design. Here koala components are developed reuse with late binding in mind.

Higher productivity gains can be achieved with generic components. It is harder to test a generic component that consists of a number of generic parameters. Therefore we need to test for all possible applications through customisation. Figure 3 shows a generic model of a component test suite where components can be customised through link modules, test cases can be generated, and applications can be built. Without some guidelines on how to test and reuse components, we may not have the confidence to reuse a component. How should the component be re-tested, and if necessary be modified? The following are the test techniques that can be adopted when testing reusable software components. We believe the work described here is applicable to any component model not only to koala components.
**Automated test case generation from object models**

At the component level, StP/T can generate test cases for each interface and with different combinations. These are arranged as test units, which allows testers to choose test units as desired. This leads to establish well tested components, which is one of the basic criteria of a reusable component. Test during modeling helps to verify both syntax and semantics of the models and generates a vast number of test cases but without test oracles which need to be hand coded. StP/T does generate templates for building test oracles and assertions. Existing work in this area is encouraging [3-4]. Figure 5 shows how Black box testing has been conducted using StP-Test tool.

Chart 1 illustrates the volume of test cases generated for each individual interface for the component (CTxtPageStore). As shown in this chart, we can find test cases from 35 for the smallest interface (an interface that has minimum number of parameters and functions, in this case Ipacket0) to 410 test cases for the complex interface (an interface that has a maximum number of parameters and functions, IpageAccess).
Depending on the type of test phase, one can choose test cases based on test case selection criteria. Figure 6 provides a guideline on test case selection against component characteristics. These are testability, observability, reusability, adaptability, interoperability, etc. When testing reusable component one of our main aim is to identify how well that component satisfy defined reuse characteristics. During this phase you are not testing their functionality.

Figure 6 Testing against component characteristics
For each type of system or requirements, one can choose one of these characteristics to test against and hence helps to release a product or a system.

**Conclusion**
Complexity of the embedded software products growing exponentially and hence the demand for reusable software components to meet growing market demands. We have seen a number test strategies and methods
that can be applied to test for reuse thus achieving testability. However the most important aspect is to identify appropriate specification techniques so that reusability and testability characteristics can be specified along with the functionality during early in the development. Issues on testing to make sure reusability remain an active and challenging research area. We believe the techniques and models described here are applicable across applications and general principles on component design.

Acknowledgement
The author would like to thank test members at Redhill, Tim Trew, Graham Thomason, and Ron van Ommering at Nat Lab, NL.

References
4. R. M. Poston, Automated testing from object models, a special issue on Object-Oriented Software Testing, Communications of the ACM, Vol.37, No.9, September 1994.