Proceedings

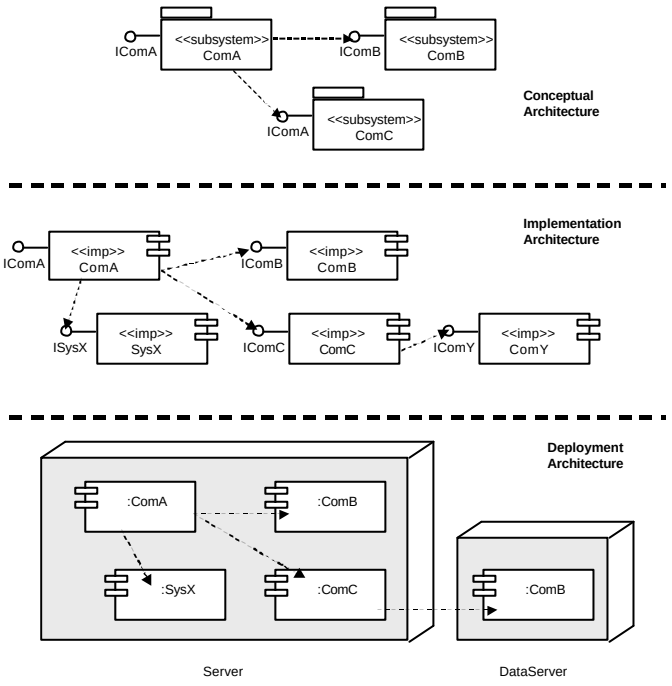
Workshop on

Component-based Software Engineering

COMPOSING SYSTEMS FROM COMPONENTS

Editors:

Ivica Crnkovic, Stig Larsson and Judith Stafford



9th IEEE Conference and Workshops on

Engineering of Computer-Based Systems

April 8-11 2002 at Lund University, Lund, SWEDEN

Contents

Workshop on Component-Based Software Engineering: Composing Systems from Components

Ivica Crnkovic
Mälardalen University, Sweden
ivica.crnkovic@mdh.se

Stig Larsson
ABB, Sweden
stig.bm.larsson@se.abb.com

Judith Stafford
Software Engineering Institute, USA
jas@sei.cmu.edu

1. Introduction

Component-based Software Engineering (CBSE) is concerned with the development of systems from reusable parts (components), the development of components, and system maintenance and improvement by means of component replacement or customization.

Building systems from components and building components for different systems requires established methodologies and processes not only in relation to development/maintenance phases, but also to the entire component and system lifecycle including organizational, marketing, legal, and other aspects. In addition to objectives such as component specification, composition, and component technology development that are specific to CBSE, there are a number of software engineering disciplines and processes that require methodologies be specialized for application in component-based development. Many of these methodologies are not yet established in practice, some have not yet been developed.

The progress of software development in the near future will depend very much on the successful establishment of CBSE; this is recognized by both industry and academia. The growing interest in CBSE is also reflected in the number of workshops and conferences with CBSE tracks [2-5].

2. The workshop

The goal of this workshop is to bring together researchers and practitioners to share experience and research results, both of works in progress and practical experience, on topics relevant to building systems from components. Systems attributes in relation to component attributes and the composition process are the primary subjects of the workshop.

Suggested areas of interest include, but are not restricted to:

- Software architecture as related to CBSE
- Analysis/design methods for building component-based systems
- Selection/evolution criteria for components and assemblies of components
- Predictability of component compositions
- Configuration management of components and component compositions
- Verification of systems based on component attributes

The workshop will open with a statement defining the goals and objectives of the workshop, followed by a presentation by a guest speaker from industry, reporting on an industrial experience of using a component-based approach to system development. The workshop will continue as a combination of presentations of the most interesting and relevant papers and resultant discussions. The workshop will focus on the discussions pertaining to topics addressed by the highest percentage of accepted papers.

3. References

[1] Bachman, et. al. , K. C., Technical Concepts of Component-Based Software Engineering, report CMU/SEI-2000-TR-008, Software Engineering Institute, Carnegie Mellon University, 2000.

[2] 4th and 5th ICSE Workshops on CBSE: Component Certification and System Prediction, Benchmarks for Predictable Assembly, http://www.sei.cmu.edu/pacc

[3] 27th and 28th Euromicro Conferences: CBSE track, http://www.idt.mdh.se/ecbse

[4] First International Working Conference on Component, http://swt.cs.tu-berlin.de/cd02/

[5] ICSR7 2002 Workshop on CBD Processes, http://www.idt.mdh.se/CBprocesses

Re-thinking the Development Process of Component-based Software

Antonia Bertolino, Andrea Polini
IEI-CNR, Area della Ricerca di Pisa, Italy
{bertolino, a.polini}@iei.pi.cnr.it

Abstract

This paper contribution to the ECBS workshop is a position statement that a wide gap exists between the technologies for Component-based Software Engineering and the scientific foundations on which this technology relies. What is mostly lacking is a revised model for the development process. We very quickly outline a skeleton for re-thinking the models that have shaped the software production in the last decades, and we start to make some speculations, in particular for what concerns the testing stages. As a working example, we take in consideration the Enterprise Java Beans framework. However, our research goal is to draw generally valid conclusions and insights.

1. Position statement

Since its early moves in the 60's, the history of software engineering has seen on the user's side the progressive growth of expectancies and reliance placed on the software services, and on the producer's side a strenuous attempt to master the consequent escalation of products dimensions and complexity.

To make software production more predictable and less expensive the research efforts have been driven by the two keywords of "discipline", in the form of process models to control the development cycle, and of "re-use", favouring the adoption of OO paradigms. However, despite the efforts, the implementation of a new system "from scratch" involves each time long development times, high production costs and difficulties in achieving further evolution and adaptations to new demands.

A component-based approach to software engineering, similarly to what is routine practice in any traditional engineering domain, seems to provide finally "the solution" to all problems inherent in traditional methods, and we assist today to a sort of revolution in the ways software is produced and marketed.

In Component-based Software Engineering (CBSE), a complex system is accomplished by assembling simpler pieces obtained in various manners. In principle, CBSE perfectly combines the two leading SE principles of "discipline" and "re-use": in fact, only forcing a rigorous discipline on how components are on one side developed, and on the opposite side utilized, a component-based system can be successfully obtained. Moreover, in CBSE re-use of components is one of the leading concerns, and is pursued since the early inception phases.

Ideally, by adopting a component-oriented approach, production times can be reduced, more manageable systems can be obtained, and, above all, such assembled systems can be easily updated by substituting one or more elements in the likely event that future market offerings provide functionalities deemed better than those of the components currently implemented in the system.

It is our concern, though, that current results are not sufficient: the rapid technology advances (e.g., .Net, EJB) are not backed by adequate parallel progress on the theoretical side. In the absence of a reference scientific framework, the proposed technological solutions appear fragmented and unrelated, and their adoption remain difficult and expensive. A software developer is provided with technologies to use and combine components, but is puzzled by the proliferation of partial solutions: a paradigm in which to use them, and criteria to follow in the selection of components and frameworks, are lacking. Paradoxically, the technologies are there, but the conceptual foundations to employ them must still be built.

It is clear that component-based software production requires a major and urgent revision of both the processes and the methods to be adopted in the development of software products. The classical life cycle models are no longer adequate, and also the professional figures that are involved in the software production and business change.

To see why, and what need to be done on the research side, we make some speculations in the following sections. To make the discussion more concrete we specifically focus this position paper on the testing stage and on the EJB framework. However, it is our future research aim to revisit the various stages of the traditional development process, and to develop concrete example within EJB as well as in other popular frameworks.

2. Considerations on the development process for the component-based age

The "standard way" in software production is a phased model in which essentially a phase starts where the previous one finishes. Let us sort out for instance what is typically found in the Table of Content of a traditional textbook in software engineering. There will certainly be a chapter dealing with the requirement analysis stage, a following chapter dealing with design, a chapter dealing with verification and testing, and finally a chapter dealing with maintenance, plus a part putting all these pieces together within a coherent process model. How well and how much does this base structure, that came out from decades of progress, fits within CBSE? The answer is obviously not so well and not so much.

The point is that, even though iterations and concurrent activities may be foreseen among the phases, a "partial order" is always imposed or assumed between the various stages above mentioned. Considering the opportunity of using components requires a totally different process that permits to manage the "non-determinism" introduced by the new approach. We bring in this notion of a non-deterministic process to highlight that, in this context, the various development activities are no longer carried out in any necessary sequence. In fact in the early phases of the development you cannot know if you will find the components already implemented or will have to develop them internally. Also, the specification of the overall architecture may depend on the adoption of certain components. Then, in a certain sense, we need generic process models that can account for the different consequences induced by the use of components produced externally or internally and that establish some "synchronization" points among all the involved stakeholders.

Besides, it is generally recognized that a condition to increase the adoption of components is to design components "for reuse", and therefore to produce adjustable components not too much shaped to fit within a specific context. That is right, but it guarantees only a part: the possibility. For successfully achieving reuse in practice, it is necessary not to early commit to a fixed system architecture independently from its constituent components, but to consider the components features as well since the early specification and design stages. In this sense we think to an incremental process, whose various phases are concurrent activities focused on recovering and tailoring components or groups of components.

More specifically, in the development of a component-based application we must initially focus on identifying that or those components that provide the basic functionalities. That is to say, we must elicit the functional and non functional requirements for these "basic" components. When candidate components are found, we can test them, against the specified requirements, and choose the best for our objectives. After having identified the first basic components we can go towards the expansion of the application functionalities, in several directions, and look for new components. The specifications for the new searched components must now derive from considerations that include the features of the components already acquired. This cycle is repeated until all the application functionalities are covered.

Perhaps sometimes the search task, for a component, can fail. In this case you can choose to implement the component or you can reduce the required functionalities and retry the search.

Obviously this iterative search-and-refine process is a preliminary idea yet, and it does not want to be complete or definitive, it wants only to illustrate a possible path. In the next section we concentrate the attention on a particular point of the picture showed above, and explain in more detail the testing phase as we imagine it might be expanded in the component development model.

As said, we focus our investigation within the EJB architecture, which has been conceived as a component-based technology to develop server-side applications, particularly in the commercial domain. The EJB platform specification was defined by Sun [1], which has also implemented a reference realization that is freely available for download from the Sun web site [2].

The EJB architecture relies on a complex middleware that manages all the aspects relative to concurrency, security, persistence, and distribution. The management of this complex task by the middleware permit the implementation of simpler components and reduce the risk of error, then the amount of testing.

3. Revising the testing process: a proposal

The distributed component approach makes many traditional testing techniques inadequate or inappropriate, and thereby calls for defining new processes, methods and tools to support testing activities. Weyuker [3] claims that in a component approach the testing performed by the component developers is insufficient to guarantee the component behaviour in new contexts and then underlines the necessity of a retesting made by the component user.

Regarding the costs of production, the advent of true CBSE presupposes the creation of a components market that can make it economically viable to develop software pieces for subsequent assembly. The success of the component approach to development requires therefore thinking in terms of system families, rather than single systems. Consequently, testing procedures must also be refocused: rather than on the definition and maintenance of test suites for single applications, attention must be directed to the development of test patterns for product families. The need for Software Architecture models in the development of component systems is widely recognized [4]. In the stages of testing, such formal

models can also be used to generate test cases, either automatically or assisted in some way.

One further complicating factor of the testing activities is represented by components whose source code is unavailable. Such components, in fact, require verification, not only that the features declared by the producer are fulfilled as expected, but also that no undeclared hazardous features are present.

The practical approach that we are going to illustrate seems to be well shaped to the component-based production, and maybe it can reduce the problems mentioned above. It originates from the considerations made in the previous section and is strongly based on the use of the *reflection* feature [5] of the selected language; for this reason the easy choice for us was the Java language.

In accordance with the process model sketched in the previous section, we suppose to have a first phase in which we establish the features that a certain component must have. In our framework this specification must be given in the form of a "virtual component" codified as a class, henceforth named `Spy`, whose required interfaces are established (so the methods and relative signatures). The only duty of every method of this class is to pack the parameters and invoke the method `executeMethod(String name, Object[] param)` of a `Driver` object (that we will illustrate afterwards), passing also to the latter its own name.

From this specification, we can put at work several teams with two different targets:

1. Developing test cases from the specification. If there are more than one team on this target, each of them can focus its attention on a particular feature;
2. Searching suitable components in the organization repository or on the market.

The test cases will be developed on the basis of the methods defined in the class `Spy`, and in a preliminary version the test cases are progressively numbered, for example, `TestCase7`, and each will form a class. All the test cases classes must be collected in a package together with the `Spy` class. Obviously the generated tests are functional/black-box and independent from a real implementation.

The test case and the `Spy` classes must extend respectively the abstract class `TestCase` and `InformationSwap`, both contained in the package `it.sssup.testing`. These classes contain methods that permit to set objects for the re-addressing of method invocation.

The searching of a suitable component is not a trivial task, in fact a real component can look very differently from that defined by the `Spy` class. In particular, we can list five different levels of accordance that, anyhow, guarantee the possible usefulness of a component in the particular application:

1. the methods name are different, but the related names have equal signatures.
2. as above, but with different parameters order
3. virtual methods have less parameter (we must set default values for the real parameter)
4. the parameters have different types, but we can make them compatible, through suitable transformations
5. the functionality of one virtual method is provided collectively by more than one method.

It is however indispensable that these differences are overtaken and for this reason we require that the searching team draw up an XML file to be used by the `Driver` object to drive the testing. In fact after the test packages are developed and at least one component is identified, a team can start the testing of it to verify that it is really compliant with the specifications.

To clarify we can provide a simple example on how we think the approach could work. The example is only declarative and obviously trivial, but we think it can be useful for the purpose.

Suppose that an Italian software house needs a simple software component to manage a bank account, and for this purpose it codifies the following `Spy` class:

```
package bankaccount.test;
import it.sssup.testing.*;
public class Spy extends InformationSwap{
    …
    public void versamento(String cod,int sum){}
    public void prelievo(String cod,int sum){}
    public int bilancio(String cod){}
}
```

From this `Spy` class, the testing teams can produce the test case class as below:

```
package bankaccount.test;
import it.sssup.testing.*;
public class TestCase6 extends TestCase{
   public runTest(){
       int before=spy.bilancio("123");
       spy.versamento("123",500);
       spy.prelievo("123",300);
       if (spy.bilancio("123")!=(before+200)){
           System.out.println("KO");
       } else { System.out.println("OK"); }
   }  }
```

In the meantime let us assume that the searching team has found a suitable component, but with different method names (deposit, withdrawal, balance) and also with different parameters order. This team produces the corresponding XML file that specifies the mapping from the virtual object to the real object.

Within the EJB framework, then, we can run the following client, passing to it the name of the package containing the test and the name of the XML file.

```
import it.sssup.testing.*;
public class ClientEJB {
  public static void main(String[] args) {
    try {
      Context initial = new InitialContext();
      Object objref =initial.lookup(
```

```
                "java:comp/env/ejb/TrivAcc");
        AccHome home =
          (AccHome)PortableRemoteObject.narrow(
                        objref, AccHome.class);
        Driver dr =
               new Driver(args[0],args[1],home);
        dr.execuTests();
      } catch (Exception e) {}
}
```

Obviously the core of the approach is the package `it.sssup.testing` that contains the specifications of the class `Driver` and of the two abstract classes `InformationSwap` and `TestCase`, that must be extended by, respectively, the `Spy` and the test case classes. The scope of `Driver` is to re-direct the invocation of the virtual methods in `Spy` to the real methods in the component, based on the information contained in the XML file. It is important to note that, in our framework, the implementation of `Spy`, of test cases `and` of test client classes is sufficiently simple and must follows the various specification above outlined.

This model is particularly suited to the context of a complex middleware, such as EJB, because it might solve many questions relative to component integration. In the EJB framework the testing can be performed running a simple tester client. EJB advantage is a strong standardization, or, said in other terms, the "discipline" that we mentioned above, which is the basic philosophy of EJB. Each user-developed bean must comply to the "bean-container contract", which imposes the realization of precise interfaces.

4. Research directions

The component-based approach opens up several new areas for research. Before all, to permit the growth of CBSE it is necessary to realize more suitable development environments. A first effort in this sense can be found in [6], where seven principal features that a development environment must satisfy are also identified.

A component-oriented world then calls for determining methodologies that can allow component builders and users to agree on the tasks to be carried out by a given component. Research in this field suggests that a component must be endowed with a series of additional information (apart from that making up its interface) that allows it, in a certain sense, to be framed semantically. This information can be used by the customer in the different phases of a development cycle [7], [8]. This line of investigation is particularly important in relation with our approach, mainly regarding the searching task. We have already outlined the difficulties concerning this task; it is desirable, then, to identify information that must reside in the specifications and in the component definition, and that can aid the searching team.

Also in the perspective of establishing an agreement between the customer and the seller, it has been investigated the opportunity that a "certification authority" is established [9]. The goal of this organization is to certify components submitted by the developers. Perhaps, also in this context the approach above depicted can be useful. In fact, the SCL (Software Certification Laboratories [9]) can define "virtual standard components" and provide, for them, benchmarks for several contexts in the form of a package containing the `Spy` and the test cases classes. The developers can then verify their components against these tests, after downloading the package and compiling the XML file. Perhaps this "modus operandi" can simplify the standardization in the production of components. In fact the SCL could define classes of components in the form of the functionality that they must provide.

Regarding more specifically the approach depicted, two directions mainly emerge as possible lines of investigation. The first is a more conceptual work, and is referred to the necessity to develop and clarify in more detail the various phases of the incremental approach. In particular we need to establish methods for extracting test case from the specifications. Besides, by way of real case studies, we want to value the real benefits that the proposed approach can produce in the component-based production.

The second line of investigation, instead, is more practical and concerns the development of tools that assist the different teams implied in the testing activities above mentioned. We refer to the development of tools to aid the drawing up of the XML file, for the searching phase and for test cases extraction.

5. References

[1] B. Shannon, "Java™ 2 Platform Enterprise Edition Specification" http://java.sun.com/j2ee/download.html

[2] J2EE reference implementation. http://java.sun.com/j2ee/sdk_1.3/index.html

[3] E.J. Weyuker, "Testing Component-Based Software: A Cautionary Tale", *IEEE Software*, Sept./Oct. 1998, pp. 54-59.

[4] D. Garlan, "Software Architecture: a Roadmap", in A.Finkelstein (Ed.) *The Future of Soft. Eng.*, ICSE 2000.

[5] The Java Tutorial, Reflection, http://java.sun.com/docs/books/tutorial/reflect/index.html

[6] C. Lüer and D. Rosemblum, "WREN – An Environment for Component-Based Development", in Proc. ESEC/FSE 2001, ACM Sigsoft Vol. 26, N.5, September 2001, pp. 207-217

[7] A. Orso, M.J. Harrold, and D. Rosenblum, "Component Metadata for Software Engineering Tasks", *EDO2000*, LNCS 1999, pp. 129-144.

[8] J.A. Stafford and A.L. Wolf, "Annotating Components to Support Component-Based Static Analyses of Software Systems", Proc. the Grace Hopper Celeb. of Women in Computing 2001.

[9] J. Voas, "Developing a Usage-Based Software Certification Process", *IEEE Computer*, August 2000, pp. 32-37.

Test Reuse in CBSE Using Built-in Tests

Jonas Hörnstein, Håkan Edler
IVF Industrial Research and Development Corporation
Mölndal, Sweden
{jonas.hornstein, hakan.edler}@ivf.se

Abstract

Component-based software engineering (CBSE) is expected to drastically reduce the time spent on developing software through the use of prefabricated components. However, some of the time gained on reusing components instead has to be spent on testing that components work as specified the new environment. The Component+ project aims at solving this by using built-in tests. This paper presents an architecture for the integration of built-in tests in software components that makes it possible to reuse tests and hence minimize the time spent on testing.

1. Introduction

The vision of Component-based Software Engineering (CBSE) is to allow software systems to be assembled from reusable components. This vision has already been realized in other engineering disciplines like mechanical and electronic, while software systems are still largely built from scratch every time.

CBSE is expected to drastically reduce the time spent on developing software as the components can be prefabricated in-house or even bought from a third party vendor on the open market. However, CBSE also introduces some challenges that must be solved in order to realize the true benefits of software reuse. One of these challenges is to verify that the components fit in the new environment when they are reused. Traditionally the different software parts were integrated within the development environment to form a single application. Hence, the compatibility of the different parts and the functionality of the resulting application could be validated at development time.

Components on the other hand, are designed to be reused in a variety of applications and by other people than those who developed the components. Even though the component developer can, and should, test their components thoroughly at development time they have no possibility to verify that the component will work when it is deployed in a new environment. Each time the context in which the component is placed changes, the component has to be tested again.

The benefits of CBSE are therefore dependent on how much work that is needed in order to verify that the components work correctly in their deployment environment. If the components cannot be applied without extensive rework or retesting in the target domains, the time saving becomes questionable [1]. Unfortunately components exhibit certain characteristics that make them difficult to test. They often contain state variables which means that the result obtained when calling an operation is dependent on the history of previous calls. Moreover components have to be considered black boxes as they are often delivered as binary code where the internal mechanisms are unknown to the user. The same holds for electronic components, which are literally delivered as black boxes and the same problem of low testability has been identified for complex integrated circuits. In order to increase the testability of these components they often have built-in tests (BIT).

BIT can also be used to increase the testability of software components [2]. Some work has been done in order to apply BIT to software components. Wang et. al. [3] put the complete test suite inside the components. This way the tests are constantly present and reused with the component. While this strategy might seem attractive at first sight, it is generally not a feasible solution. The tests are constantly occupying space while most tests will only be used once when the component is deployed. Another approach to BIT has been taken by Martins et. al. [4]. They put a minimal number of tests, like assertions, inside the components, which are reused together with a test specification. However, specific software has to be used in order to transform the test specification into real tests.

This paper presents a flexible architecture for the integration of BIT in software components, which makes it possible to reuse tests without additional software. The architecture was developed within the European project Component+ [5].

The rest of the paper is organized as follows. In section 2 we discuss how verification and validation of software is affected by the introduction of CBSE, and why we have to build in test facilities in order to solve the problems

arisen. In section 3 we explain the Component+ architecture that allows for tests to be built in. Section 4 gives specific examples on how the architecture can be used in order to reuse tests. Conclusions and future work are discussed in section 5.

2. Verification and validation of components

Verification and validation (V&V) of software has always been an important and difficult task in software engineering. With the introduction of CBSE it has grown even more important, and unfortunately also more difficult since the components have to deal with a large number of contexts that are essentially unknown when the components are developed. The final testing cannot be done until the component is deployed, i.e. taken in use, in the target domain.

This is challenging, since the testing then has to be performed by other people than those who developed the component. During development time white box testing can be used, which takes the internal structure of the component under consideration. However, one of the main ideas behind CBSE is that the internals of the components are hidden once they are developed. Therefore techniques for black box testing have to be used. Testing components using only traditional black box techniques is difficult though. Most components encapsulate both data and functionality and are therefore state machines. Since traditional black box testing techniques are only concerned with detecting failures at the border of the component, they will not detect errors in the internal state of a component. The difference between errors and failures can be described as [6].

Error: *Manifestation of a fault in a system*
Failure: *Deviation of the delivered service from compliance with the specification*

Errors may only show up as failures after certain sequences of calls or after repeatedly executing the same test case, hence the difficulty to find them with traditional techniques. Built-in testing is aimed at overcoming this problem.

2.1. Built-in tests

Since built-in tests are put inside the components they are able to detect errors. This is important since errors can be hidden for long time before they are exposed as failures. Techniques that can be used to find errors in software components include assertions [7][8] and control flow checking [9][10].

It is also possible to include test cases that verify the component's functionality. However, to thoroughly test a component, lots of test cases have to be included in the component. This add a lot of overhead to the component, while many of the test cases, like most functional tests, does not add any value to the component once executed in the context where the component has been deployed. Significantly less over-head has to be added to the component if the test cases are placed outside the component and the built-in tests provide the information needed in order to test it. Since most components are state-machines, they are typically tested using state-based testing. Doing this effectively without being able to set and read the states is a practical impossibility [2]. Built-in tests can be used for setting and reading the component's state.

If these built-in tests are provided through well-defined interfaces, as in the Component+ architecture, they significantly increase the testability of the components.

2.2. Built-in testing

To be used for verification and validation the tests built-in have to be executed. A test is the execution of one or many test cases. For state-based testing, a test case consists of.

1. The initial state of the component
2. Test data that will be used when calling the component's operations
3. The expected output value and the expected final state according to the specification.

Built-in testing makes use of the built-in tests in order to set the component to an initial state before calling its operations, and to verify that the component ended up in the expected final state without any errors.

3. Component+ BIT architecture

The Component+ architecture is built from three types of software components: BIT components, Testers, and Handlers. In order to define those we first define what we mean by a component. In Component+ we have chosen to base our component model on the KobrA development model [11]. The component definition given below is therefore taken from KobrA, which in turn has been adapted from the definition in [12]:

Software component:
A software component is a unit of composition with contractually specified interfaces and context dependencies only. A software component can be deployed independently and is subject to composition by third parties.

This component model does not put any constraints on the technology used to develop the component. Technologies like CORBA, EJB, and COM, all fit in the architecture, and even traditional APIs can be viewed as components.

3.1. BIT components

In the Component+ architecture, the component under test must be a BIT component, i.e. have built-in test mechanisms.

BIT component:

A BIT component is a software component with built-in test mechanisms, which are provided through one or more interfaces, BIT interfaces.

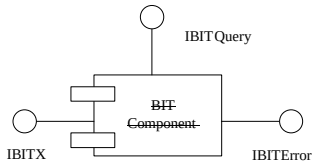


Figure 1. BIT component

In figure 1, a BIT component and its provided BIT interfaces are shown. IBITQuery is the only mandatory interface and has to be provided by all BIT components. It is used to determine what test facilities a component supports. The actual tests are provided by a number of interfaces called IBITX, where X can be either some standardized interface for a given test, such as for deadlock testing and timing testing, or some tailor-made interface for that specific BIT component. IBITError provides information about detected errors.

3.2. Testers

To be useful, the BIT components have to be connected to some other component that exercises the built-in tests and evaluates the information provided by the BIT interfaces. Such a component is called a tester component.

Tester component:

A component that uses one or more BIT interfaces.

It is the tester that does the actual testing and verifies that the component is working according to the specification. In the same way as there exists both standardized and tailor-made BIT interfaces, there also exists both standardized and tailor-made Testers. Standardized testers are used to detect, for example, deadlock situations. It is important that these are standardized since they need information from many different BIT components, which could be developed by independent vendors. Other testers, such as testers for state-based testing has to be customized in order to fit the state-model of a specific BIT component.

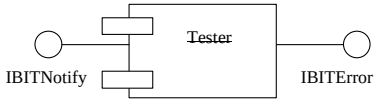


Figure 2. Tester component

A tester component, figure 2, provides IBITNotify to which BIT components can report for example state changes, and IBITError that provides information about detected failures.

Since the Testers are separated from the BIT components they can be changed at any time in order to best suite the context in which the component operates, and various Testers can be combined in order to obtain a more thorough testing.

3.3. Handlers

The architecture also includes handlers. They provide an IBITErrorNotify, figure 3, to which both BIT components and Testers can signal errors. Handlers do not contribute to the verification and validation, but can be used to obtain fault-tolerant systems.

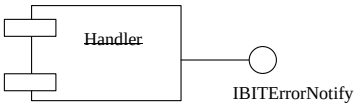


Figure 3. Handler component

4. Test reuse

The introduction of BIT components, with their increased testability is an important step towards reducing the time spent on testing. However, it is by reusing testers, as well as the BIT components, that the true benefits are accomplished.

We feel that there is a need for reusing tests at several stages of the component's life cycle, when the component is deployed, during normal execution, and in maintenance.

4.1. Test reuse at deployment-time

In the typical scenario for CBSE a component can be bought off-the-shelf and inserted into a new system. That way the same component can be used in a variety of systems, thus realizing the vision of code reuse. In the Component+ project, this vision is extended to include also the code used to test the components. A BIT component is therefore delivered together with one or more suitable testers that can be used to verify that the component is able to correctly provide its services in its deployment environment.

The relationship between components can be formalized as a contract, expressing the rights and obligations for the involved components [7]. The vendor of the BIT component should supply a tester that checks that the BIT component is able to abide by its side of the contract after it has been deployed in the new environment, figure 4. The tester typically does some kind of state-based testing, and the IBITContract therefore provides means to set and read the component's state.

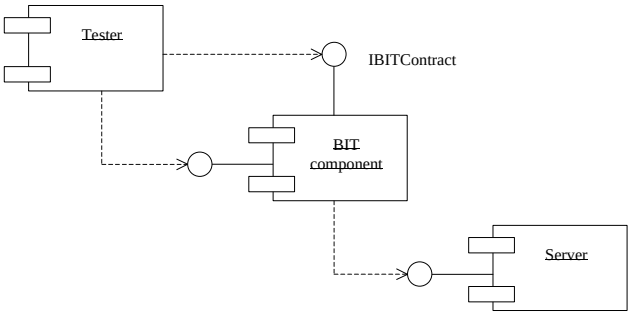


Figure 4. Contract testing

4.2. Test reuse at run-time

At run-time, it is difficult to use complete test cases without interfering with the normal execution. Instead the built-in test mechanisms are used to monitor the execution and to signal detected errors.

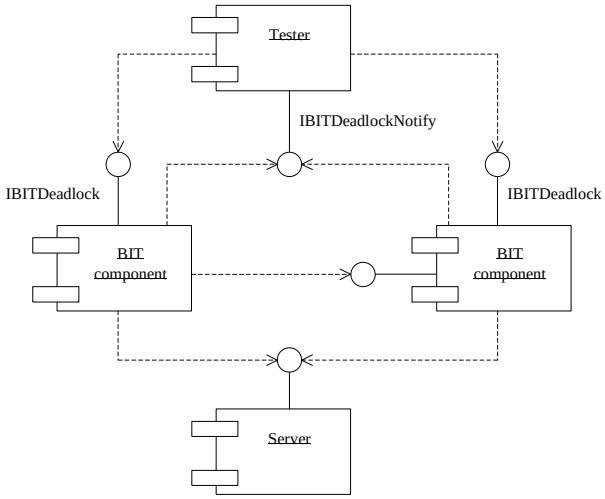


Figure 5. Deadlock testing done at run-time

Apart from the test mechanisms built-in, which are automatically reused with the component, there is a possibility to reuse testers that perform some kind of monitoring on system level. These testers gather information from several BIT components in order to detect for example deadlocks, figure 5. Standardized testers for deadlock testing and timing testing are being developed within the Component+ project.

4.3. Test reuse at maintenance

If the developer changes the BIT component, e.g. to remove some fault or add some functionality, the component has to be tested again. This can be done with the same tester as is delivered with the component, or even with a white-box tester as the testing is done by the developer. This is identical to today's regression testing.

5. Conclusions and future work

The architecture proposed by Component+ offers the possibility to reuse tests for CBSE. This might lead to substantial timesaving when assembling a system from components. Future work within the Component+ project includes trying the technology in industry and to measure the impact on development time.

6. Acknowledgement

This work has been done within the Component+ project (IST-1999-20162) and all partners within the consortium have contributed to the result.

7. References

[1] Guindi, D. S., Ligon, W. B., McCracken, W. M., Rugaber, S., "The impact of verification and validation of reusable components on software productivity", *Proceedings of the Twenty-Second Annual Hawaii International Conference on System Sciences*, 1989, pp. 1016-1024

[2] Binder, R. V., "Design for Testability in Object-Oriented systems", *Communications of the ACM*, Vol. 37, no. 9, September 1994, pp. 87-101

[3] Wang, Y., King, G., Fayad, M., Patel, D., Court, I., Staples, G., Ross, M., "On Built-in Test Reuse in Object-Oriented Framework Design", *ACM Journal on Coputing Surveys*, Vol. 32, No. 1, March 2000

[4] Martins, E., Toyota, C. M., Yanagawa, R. L., "Constructing Self-Testable Software Components", *Proceedings of the 2001 International Conference on Dependable Systems and Networks*, Göteborg, Sweden, July 2001, pp. 151-160

[5] EC IST-1999-20162, Component+, *www.component-plus.org*, February 2002

[6] Laprie, J. C., (ed.) *Dependability: Basic Concepts and Terminology*, Springer-Verlag, Wien, 1992

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

[8] Binder, R. V., *Testing Object-Oriented Systems: Models, Patterns and Tools*, Addison Wesley Longman, 1999

[9] Mahmood, A., McCluskey, E. J., "Concurrent Error Detection Using Watchdog Processers – A Survey", *IEEE Transactions on Computers*, Vol. 37, no 2, 1988

[10] Miremadi, G., Karlsson, J., Gunneflo, U., Torin, J., "Two Software Techniques for On-line Error Detection", *Twenty-Second International Symposium on Fault-Tolerant Computing*, 1992

[11] Atkinson, C., et. al., *Component-Based Product Line Engineering with UML*, Addison-Wesley, 2001

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

Architecture Trade-off Analysis and the Influence on Component Design

Iain Bate and Neil Audsley
Department of Computer Science
University of York, York, YO10 5DD, UK.
{iain.bate, neil.audsley}@cs.york.ac.uk

1 Introduction

The production and assurance of systems that are safety-critical and/or real-time is recognised as being costly, time-consuming, hard to manage, and difficult to maintain. This has lead to research into new methods whose objectives include:

- Modular approaches to development, assurance and maintenance to enable:
  - Increased reuse;
  - Increased robustness to change and reduced impact of change.
- Integration strategies that allow systems to be procured and produced by multiple partners, and then efficiently integrated;
- Ways of determining the approach likely to be the "best" (the best can only be found with hindsight);
- Techniques for identifying and managing risks.

Many of the component-based engineering techniques are considered relatively mature for developing dependable components and ensuring correctness across their interfaces when combined with other components, e.g. approaches based on rely-guarantees [1]. This paper addresses the following key remaining issues:

- how the system's objectives should be decomposed and designed into components (i.e. the location and nature of interfaces); and
- what functionality the components should provide to achieve the system's objectives.

The paper develops a method for:

1. *derivation of choices* – identifies where different design solutions are available for satisfying a goal.
2. *manage sensitivities* – identifies dependencies between components such that consideration of whether and how to relax them can be made. A benefit of relaxing dependencies could be a reduced impact to change.
3. *evaluation of options* – allows questions to be derived whose answers can be used for identifying solutions that do/do not meet the system properties, judging how well the properties are met and indicating where refinements of the design might add benefit.
4. *influence on the design* – identifies constraints on how components should be designed to support the meeting of the system's overall objectives.

Our approach satisfies the objectives by building on existing approaches, i.e. Goal Structuring Notation (GSN) which is used for safety arguments [2], and UML which is used for modelling systems [3]. However, a key fact is that the methodology proposed is not dependent on the specific techniques advocated. The approach only needs a component -based model of the system's design with data flow coupling between the components and there being a way of reasoning about properties (and their inter-relationships), constraints, and assumptions associated with the coupling. (A coupling is considered as a connection between components.) The approach we are proposing would be used within the nine-step process of the Architecture Trade-Off Analysis Method (ATAM) [4]. The differences between our strategy and other existing approaches, e.g. ATAM, include the following.

1. the techniques used in our approach are already accepted and widely used (e.g. nuclear propulsion system and missile system safety arguments) [2], and as such processes exist for ensuring the correctness and consistency of the results obtained.
2. the techniques offer strong traceability and the ability to capture design rationale.
3. information generated from their original intended use can be reused, rather than repeating the effort.
4. the method is equally intended as a design technique to assist in the evaluation of the architectural design and implementation strategy as it is for evaluating a design at a particular fixed stages of the process.

The method is described further in section 2. This is followed by a demonstration of the approach through the case study presented in section 3. Section 4 considers how the architecture trade-off analysis can be used to influence the way in which components are designed.

2 Trade-Off Analysis Method

2.1 Overview of the Trade-Off Analysis Method

Figure 1 provides a diagrammatic overview of the method. Stage (1) of the trade-off analysis method is producing a model of the system to be assessed. This model should be decomposed to a uniform level of abstraction. Currently our work uses UML for this purpose, however it could be applied to any modelling approach that clearly identifies components and their couplings. Arguments are then produced in stage (2) for each coupling to a corresponding (but lower so that impact of later choices can be made) abstraction level

than the system model. (An overview of GSN is given in section 2.2.) The arguments are derived from the top-level properties of the particular system being developed. The properties often of interest are lifecycle cost, dependability, and maintainability. Clearly these properties might be broken down further, e.g. dependability may be decomposed to reliability, safety, timing etc.. In practice, the arguments should be generic or based on patterns where possible. Stage (3) then uses the information in the argument to derive options and evaluate particular solutions. Part of this activity uses representative scenarios (e.g. what happens when change X is performed) to evaluate the solutions. The use of scenarios is not discussed in this paper.

Based on the findings of stage (3), the design is modified to fix problems that are identified – this may require stages (1)-(3) to be repeated to show the revised design is appropriate. When this is complete and all necessary design choices have been made, the process returns to stage (1) where the system is then decomposed to the next level of abstraction using guidance from the goal structure. Components reused in other context could be incorporated as part of the decomposition. Only proceeding when design choices and problem fixing are complete is preferred to allowing trade-offs across components at different stages of decomposition because the abstractions and assumptions are consistent easing the multiple-criteria optimisation problem.

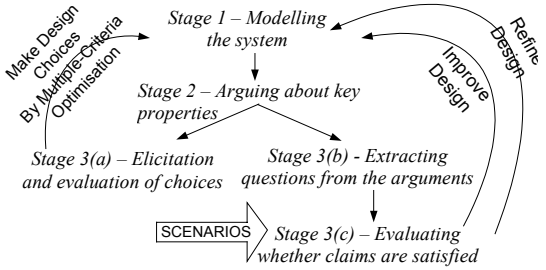


Figure 1 - Overview of the Method

2.2 Background on Goal Structuring Notation

The arguments are expressed in the GSN [2] that is widely used in the safety-critical domain for making safety arguments. In brief, any safety case can be considered as consisting of requirements, argument, evidence and definition of bounding context. GSN - a graphical notation - explicitly represents these elements and (perhaps more significantly) the relationships that exist between these elements (i.e. how individual requirements are supported by specific arguments, how argument claims are supported by evidence and the assumed context that is defined for the argument).

The principal symbols in the notation are shown in Figure 2 (with example instances of each concept). The principal purpose of a goal structure is to show how **goals** (claims about the system) are successively broken down into sub-goals until a point is reached where claims can be supported by direct reference to available evidence (**solutions**). As part of this decomposition, using the GSN it is also possible to make clear the argument strategies adopted (e.g. adopting a quantitative or qualitative approach), the rationale for the approach (**assumptions**, **justifications**) and the **context** in which goals are stated (e.g. the system scope or the assumed operational role). Further details are found in [2].

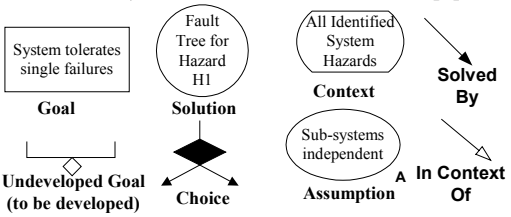


Figure 2 - Principal Elements of GSN

3 Case Study – Simple Control System

The example being considered is a continuous control loop that has health monitoring to check for whether the loop is complying with the defined correct behaviour (i.e. accuracy, responsiveness and stability) and then takes appropriate actions if it does not.

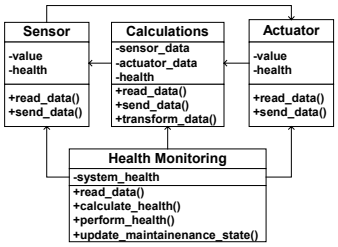


Figure 3 - Class Diagram for the Control Loop

At the highest level of abstraction the control loop (the architectural model of which is shown in Figure 3) consists of three elements; a sensor, an actuator and a calculation stage. It should be noted that at this level, the design is abstract of whether the implementation is achieved via hardware or software. The requirements (key safety properties to be maintained are signified by **(S)**, functional properties by **(F)** and non-functional properties by **(NF)**, and explanations, where needed, in italics) to be met are:

- the sensors have input limits (**S**) **(F)**;
- the actuators have input and output limits (**S**) **(F)**;
- the overall process must allow the system to meet the desired control properties, i.e. responsiveness (dependent on errors caused by latency **(NF)**), stability (dependent on errors due to jitter **(NF)** and gain at particular frequency responses **(F)**) [6] (**S**);
- where possible the system should allow components that are beginning to fail to be detected at an early stage by comparison with data from other sources (e.g. additional sensors) **(NF)**. Early recognition would allow appropriate actions to be taken including the planning of maintenance activities.

In practice as the system development progresses, the

component design in Figure 3 would be refined to show more detail. For reasons of space only the *calculation-health monitor* coupling is considered.

Stage 2 is concerned with producing arguments to support the meeting of objectives. The first one considered here is an objective obtained from decomposing an argument for dependability (the argument is not shown here due to space reasons) that the system's components are able to tolerate timing errors (goal **Timing**). From an available argument pattern, the argument in Figure 4 was produced reasoning that *"Mechanisms in place to tolerate key errors in timing behaviour"* where the context of the argument is *health monitor* component. Figure 4 shows how the argument is split into two parts. Firstly, evidence has to be obtained using appropriate verification techniques that the requirements are met in the implementation, e.g. when and in what order functionality should be performed. Secondly, the health monitor checks for unexpected behaviour. There are two ways in which unexpected behaviour can be detected (a choice is depicted by a black diamond in the arguments) – just one of the techniques could be used or a combination of the two ways. The first way is for the *health-monitor* component to rely entirely on the results of the internal health monitoring of the *calculation* component to indicate the current state of the calculations. The second way is for the *health-monitor* component to monitor the operation of the *calculation* component by observing the inputs and outputs to the *calculation* component.

In the arguments, the leaf goals (generally at the bottom) have a diamond below them that indicates the development of that part of the argument is not yet complete. An argument is complete when all leaves have been fully developed such that they are terminated by solutions. The solutions are typically requirements for the evidence to be provided. The evidence provided is normally quantitative in nature, e.g. results of timing analysis to show timing requirements are met.

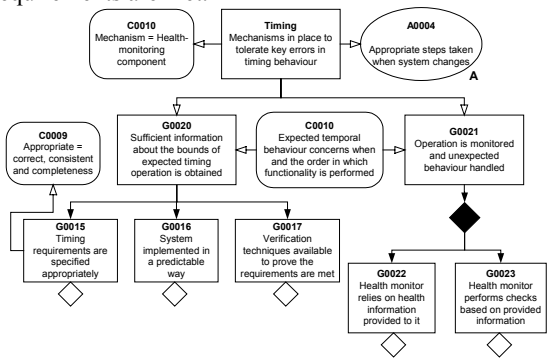


Figure 4 - Timing Argument

Next an objective obtained from decomposing an argument for maintainability (again not shown here due to space reasons) that the system's components are tolerant to changes is examined. The resultant argument in Figure 5 depicts how it is reasoned the *"Component is robust to changes"* in the context of the *health-monitor* component. There are two separate parts to this; making the integrity of the calculations less dependent on when they are performed, and making the integrity of the calculations less dependent on the values received (i.e. error-tolerant). For the first of these, we could either execute the software faster so that jitter is less of an issue, or we could use a robust algorithm that is less susceptible to the timing properties of the input data (i.e. more tolerant to jitter or the failure of values to arrive).

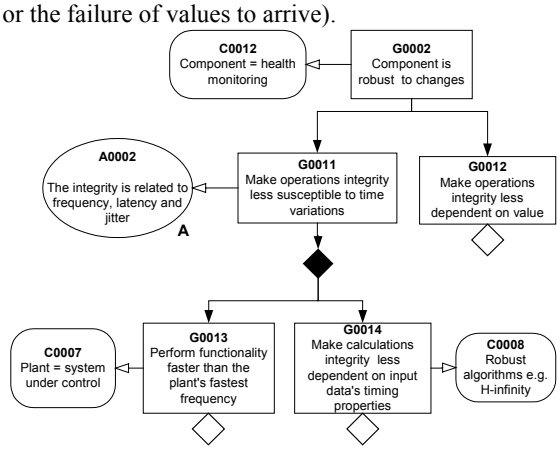


Figure 5 – Minimising Change Argument

The next stage (stage 3(a)) in the approach is the elicitation and evaluation of choices. This stage extracts the choices, and considers their relative pros and cons. The results are presented in Table 1.

Content	Choice	Pros	Cons
Goal G0021 - Operation is monitored and unexpected behaviour handled	**Goal G0022 -** Health monitor relies on health information provided to it	Simplicity since health monitor doesn't need to access and interpret another component's state.	Can a failing/failed component be trusted to interpret error-free data.
	Goal G0023- Health monitor performs checks based on provided information	Omission failures easily detected and integrity of calculations maintained assuming data provided is correct.	Health monitor is more complex and prone to change due to dependence on the component.
Goal G0011 - Make operations integrity less susceptible to time variations	**Goal G0013 –** Perform functionality faster than the plant's fastest frequency.	Simple algorithms can be used. These algorithms take less execution time.	Period and deadline constraints are tighter. Effects of failures are more significant.
	Goal G0014 - Make calculations' integrity less dependent on input data's timing properties.	Period and deadline constraints relaxed. Effects of failures may be reduced.	More complicated algorithms have to be used. Algorithms may take more execution time.

Table 1 - Choices Extracted from the Arguments

Stage 3(b) then extracts questions from the argument

that can then be used to evaluate whether particular solutions (stage 3(c)) meets the claims from the arguments generated earlier in the process. Table 2 presents some of the results of extracting questions from the arguments for claim **G0011** and its assumption **A0002** from Figure 5. The table includes an evaluation of a solution based on a PID (Proportional Integration Differentiation) loop.

Question	Importance	Response	Design Mod.	Rationale
Goal G0011 - Can the integrity of the operations be justified?	Essential	More design information needed	Dependent on *response* to questions	N/A
Assumption A0002 - Can the dependency between the operation's integrity and the timing properties be relaxed?	Value Added	Only by changing control algorithm used	Results of other trade-off analysis needed	N/A

Table 2 – Evaluation Based on Argument

Table 2 shows how questions for a particular coupling have different importance associated (e.g. *Essential* versus *Value Added*). These relate to properties that must be upheld or those whose handling in a different manner may add benefit (e.g. reduced susceptibility to change). The responses are only partially complete (design modification and rationale not at all) for the solution considered due to the lack of other design information. As the design evolves the level of detail contained in the table would increase and the table would then be populated with evidence from verification activities, e.g. timing analysis.

4 Influence on Component-Based Design

The content of the arguments presented in section 3 can be used to influence the way components in the system are designed and the way in which the architecture is decomposed. This section discusses the influences from some of the goals and in doing so demonstrates the links between the architecture trade-off analysis and component-based design.

From Table 1 it can be seen that some of the choices that need to be made about individual components are affected by choices made by other components within the system. Two cases of influence are given below:

1. *On Component's Functionality* – In Figure 5 goal **G0014** leads to a design option of having a more complicated control algorithm that is more resilient to changes and variations in the system's timing properties. However goal **G0014** is in opposition to goal **G0023** from Figure 4 since it would make the health-monitoring component more complex.
2. *On Abstractions and Interfaces* – Goal **G0021** in Figure 4 leads to a choice over where *health monitoring* functionality is situated. These are; entirely in the *health monitor* component, or partially in the *calculation* component and the rest in the *health monitor* component. The choice alters the abstractions and interfaces between the two components since all relevant data needs to be passed between the components if the *health monitor* component is entirely responsible. In contrast if it is only partially responsible, then a health level would be passed and maybe some data to allow limited validation to be performed in the *health monitor* component. The choice therefore affects the components' design as well as how achievable objectives such as reuse and maintainability are.

Other choices made may not influence the abstractions and interfaces but may affect the components' design. This can be demonstrated through the choice originating from goal **G0011**. Independent of how calculations are performed, the health monitoring is still based on whether the control loop meets the requirements given in section 3. This requires data concerning current sensor inputs and actuator outputs to be passed from the calculation components to the health monitoring. With this data it can be checked whether the inputs and outputs are within limits as well as determining the responsiveness and stability criteria are being met [6]. Hence the abstraction and interface is not affected, but the design of the calculation component and the checks performed are affected.

5 Conclusions

This paper has addressed a method to support architectural design and implementation strategy trade-off analysis, one of the key parts of component-based development. Specifically, the method presented provides guidance when decomposing systems so that the system's objectives are met, deciding what functionality the components should fulfil in-order to achieve the remaining objectives, and showing how this influences the design of components.

Further work could include performing different case studies, to show how argument and design patterns can be used to increase the efficiency of applying the technique, to understand better the relationship between system architecture and component design, and to establish a means by reusing existing work for performing the multiple-criteria optimisation.

6 References

[1] B. Meyer, *Applying Design by Contract*, IEEE Computer, 25(10), pp. 40-51, October 1992.

[2] T. Kelly, *Arguing Safety – A Systematic Approach to Safety Case Management*, DPhil Thesis, YCST-99-05, Department of Computer Science, Univ. of York, 1998.

[3] B. Douglass, *Real-Time UML*, Addison Wesley, 1998.

[4] R. Kazman, M. Klein, P. Clements, *Evaluating Software Architectures – Methods and Case Studies*, Addison Wesley, 2001.

[5] J.-C. Laprie, *Dependable Computing and Fault Tolerance: Concepts and Terminology*, in Proceedings of the 15th International Symposium on Fault Tolerant Computing (FTCS-15), pp. 2-11, 1985.

[6] R. Harbor and C Phillips, *Feedback Control Systems*, 4th Edition, Prentice Hall, 2000.

The Future of Component-Based Development is Generation, not Retrieval

Hans de Bruin Hans van Vliet
Vrije Universiteit, Amsterdam
Mathematics and Computer Science Department
De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands
e-mail: {hansdb,hans}@cs.vu.nl

Abstract

Component-Based Development (CBD) has not redeemed its promises of reuse and flexibility. Reuse is inhibited due to problems such as component retrieval, architectural mismatch, and application specificness. Component-based systems are flexible in the sense that components can be replaced and fine-tuned, but only under the assumption that the software architecture remains stable during the system's lifetime. In this paper, we argue that systems composed of components should be generated from functional and non-functional requirements rather than being composed out of existing or newly developed components. We propose a generation technique that is based on two pillars: Feature-Solution (FS) graphs and top-down component composition. A FS-graph captures architectural knowledge in which requirements are connected to solution fragments. This knowledge is used to compose component-based systems. The starting point is a reference architecture that addresses functionality concerns. This reference architecture is then stepwise refined to cater for non-functional requirements using the knowledge captured in a FS-graph. These refinements are the architecture-level counterpart of aspect weaving as found in Aspect-Oriented Programming (AOP).

1 Introduction

Component-Based Development (CBD) is concerned with the development of systems from reusable parts, that is, components. Unfortunately, CBD has not lived up to its expectations yet (see for instance [1, 3, 12]). To be successful with CBD as a reuse technology, components have to be retrieved from a kind of repository based on a set of criteria to judge whether components are fit for the job or not. The criteria not only include functional requirements, but non-functional requirements, such as performance and footprint, as well. The current state of affairs is that not many components can actually be reused. The problem is partially caused by incomplete component specifications that do not tell the whole story. Yet another cause is architectural mismatch [10]; a component may function perfectly well in one setting, but may fail in a different setting due to making (possibly undocumented) assumptions on the environment. Despite these problems, we believe that there are more fundamental problems to be solved before components can be reused. True, we have no difficulty with reusing domain independent components such as user interface and database connectivity components. However, it is less obvious to reuse application specific components, not only across application domains, but in the same application domain as well. Typically, components are too rigid as far as its functional and non-functional properties are concerned to be adapted to (slightly) different settings.

For the aforementioned reasons, some researchers and practitioners do not longer view CBD as a means for reuse (see for instance [5]). They view CBD as a development method for constructing flexible software in which the main driver is change. A system should be designed in such a way that components exhibit cleanly defined interfaces and that they do not depend strongly on the support offered by surrounding components. If designed as such, components are eligible for change by means of component replacement or customization. Unfortunately, CBD is not the final answer for managing change. The underlying assumption is that a software architecture, composed of components, can be developed that remains relatively stable during the lifetime of a system. This assumption is not

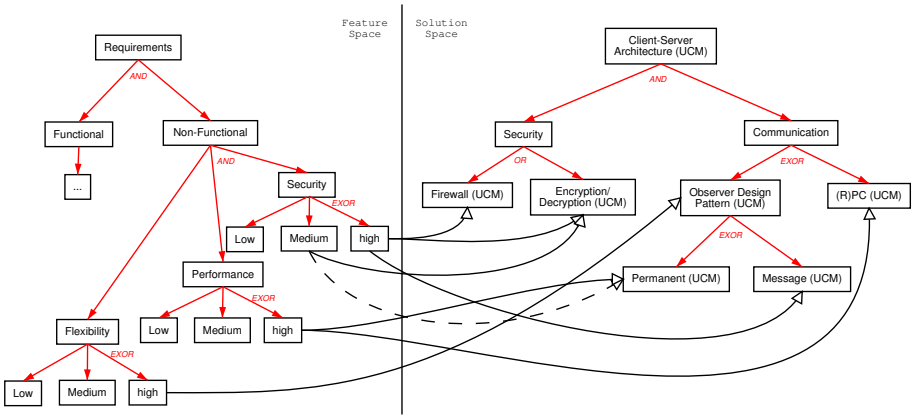


Figure 1. Feature-Solution graph for the Client-Server system.

valid in general. Systems evolve over time, and many times in rather unpredictable ways. At some point in time, we might find that a software architecture can not longer accommodate evolutionary changes, and hence the CBD approach falls in pieces. Also, some changes are not localized, but affect multiple components. Examples of such crosscutting effects are typically caused by imposing new or changed non-functional requirements on a system. Consider, for example, shifting the emphasis from performance to footprint, or vice versa. This will likely have an impact on a large number of components, and might even require a different software architecture.

We have reached the conclusion that CBD does not deliver its promises of reuse and flexibility. This leaves us with the question which alternative methods and techniques can be used instead. Our answer is that component-based systems should be generated. We are not alone in this view, generative programming techniques are gaining more and more interest these days [6]. We propose a generation technique that generates systems from functional as well as non-functional requirements. The generation technique technique is based on two pillars:

Feature-Solution (FS) graphs. A FS-graph captures architectural knowledge in the form of desired features (e.g., functional and non-functional requirements) and solutions that realize these features (e.g., architectural and design patterns).

Top-down component composition. We envisage a quality-driven approach to generating component-based systems in which the FS-graph plays a crucial role. This process ia akin to the process described in [2]. The first step in this process is the derivation of a reference architecture that meets the functional requirements set. Next, the attention focuses on non-functional requirements by iteratively applying known design solutions as codified in the FS-graph. Typically, this requires several iterations. These iterations might also involve backtracking steps because we usually have to deal with conflicting requirements.

The impact of quality aspects, such as performance and security, is typically not restricted to a single component only, but may affect a large number of components. In a FS-graph, we can capture all the knowledge that is required to refine multiple components simultaneously in a consistent manner. This type of refinement may be called Aspect-Oriented Programming (AOP) [11] at the architectural level.

2 Component Generation Techniques

In this section, we discuss the generation techniques in more detail. At the heart of the iterative, quality driven approach for generating component-based system is the FS-graph. Consider as an example a Client-Server (CS) system in which a client component requests a server component to perform one of its duties. A FS-graph for the CS system is shown in Figure 1. Two spaces are recognized in the FS-graph. The Feature (F) space contains the requirements, whereas the Solution (S) space contains solutions addressing these requirements. Features as well as solutions are decomposed in AND-(EX)OR decomposition trees. An AND

decomposition of a node in either the feature or the solution space means that all its constituents must be available, an OR requires an arbitrary (≥ 0) number of constituents, and an EXOR requires precisely one constituent. The key idea is that a feature in the F-space may select a solution in the S-space as defined by directed selection links between nodes (indicated by a solid line). It is also possible to explicitly rule out a particular solution. This is done by connecting a feature to a solution with a *negative* selection link (indicated by a dashed line).

In the example, we focus on non-functional requirements, in particular flexibility, security and performance requirements. If a high flexibility level is desired, the FS graph dictates that we should use the Observer design pattern, because of its properties of reducing the coupling between peers and supporting multiple observers. On the other hand, if we want high performance, the FS graph selects a direct invocation style in the form of (remote) procedure calls. It is interesting to observe that a high level of flexibility and a high level of performance cannot be obtained simultaneously since these requirements select solutions that rule out each other, as implied by the EXOR decomposition of the communication node. Thus, a FS graph contains trade-off information as well. Typically, several design process cycles are required to arrive at a design that satisfies all non-functional requirements. The process is shown in Figure 2 and is described in detail in [8].

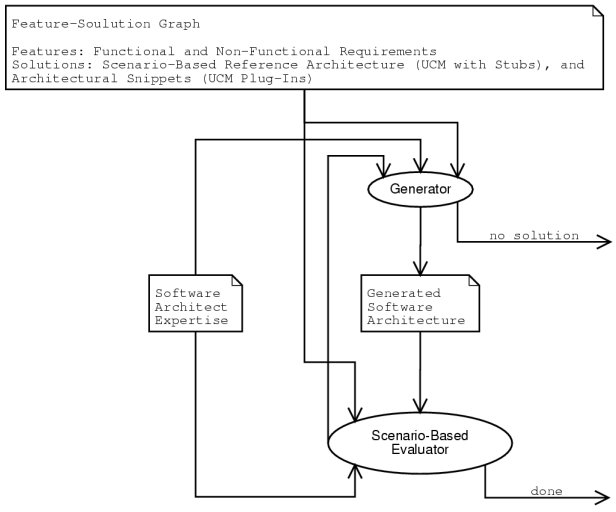


Figure 2. The process of generating and evaluating architectures.

The generator generates the components on the basis of requirements and architectural solutions captured in the FS-graph. The generator uses the FS-graph to refine the given CS architecture. Next, the generated system is evaluated against all functional and non-functional requirements set. Now suppose that a certain requirement has not been met in the current system. By consulting the FS-graph, we might come up with several solutions that can be applied to remedy the shortcoming. Thus, in principle, the outcome of the evaluation phase can be used to drive the architecture generation process in the next iteration. That is, the generator selects a solution and then generates a refined system, which is evaluated in its turn. This process is repeated until all requirements are met or we run out of potential solutions.

We use Use Case Maps (UCM) [4] for representing component-based systems. UCM is a diagrammatic modeling technique to describe behavioral and, to a lesser extent structural, aspects of a system at a high level of abstraction. UCM provides stubs (i.e., the hooks or variability points) where the behavior of a system can be varied statically at construction time as well as dynamically at run time. The advantage of UCM is that it focuses on the larger, architectural issues, and its support of plug-ins and stubs, both static and dynamic. Further details can be found in [9].

3 Component Generation in Practice

We are currently putting our approach in practice in the QUASAR (QUAlity-driven Software Architecture) project. The goal is to generate real systems in real application domains using the techniques described in this paper. Although it is too early to draw definite conclusions, the approach looks promising.

As a starting point, we use a reference architecture that is composed of components that implement the required functionality. The components are identified through a domain engineering process capturing the commonalities and variabilities of a domain. What sets our approach apart from a standard CBD approach is that we do not treat a component as a black-box that can be adapted to a certain extent. Instead, we use a grey-box approach [7] in which the behavior of components is described in terms of UCMs with stubs. The stubs provide the means to refine components recursively. The FS-graph plays a central role in this process. It contains all the knowledge that is required to tailor components in order to meet non-functional requirements such as performance and security requirements. A refinement operation is not necessarily restricted to a single component only but may crosscut

several components. The knowledge of how and where the system has to be adapted can be captured in a FS-graph as is illustrated in Figure 1. Effectively, the FS-graph ensures that all refinements needed to satisfy particular requirements are effectuated.

Our current experiences show that the goal of generating component-based systems from functional and non-functional requirements is not (yet) feasible. The limitations of our approach are the following. A fairly developed reference architecture is needed before the iterative generation process can start. In order to arrive at a reference architecture, obviously some design decisions need to be made. In principle, the generation process should be a closed-loop process without requiring human (software architect) intervention. This is hard to achieve since the evaluation of certain quality attributes requires an expert eye.

4 Concluding Remarks

CBD as a black-box assembly technique is in our opinion a dead-end street, because of its inherent problems of lack of reuse and flexibility. Instead of composing a system out of existing or newly developed components, we propose to compose a component-based system from functional and non-functional requirements and design solution fragments captured in a FS-graph. In this approach, reuse assets stem from two sources. Firstly, from a domain engineering process we can identify a set of reusable (grey-box) components. Secondly, reusable solution fragments (e.g., architectural and design patterns) can be connected to domain and application specific functional and non-functional requirements in a FS-graph. These solution fragments can then be used for component adaptation.

In conclusion, the keywords for characterizing the CBD future are domain engineering and component generation/adaptation. The FS-graph and the top-down composition techniques described in this paper show how this future can be realized, although there is plenty of room for improvement.

References

[1] Paul G. Basset. *Framing Software Reuse: Lessons from the Real World*. Prentice Hall, Upper Saddle River, New Jersey, 1996. Yourdon Press.

[2] Jan Bosch. *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach*. Addison-Wesley, 2000.

[3] Alan W. Brown and Kurt C. Wallnau. The current state of CBSE. *IEEE Software*, 15(5):37–46, September 1998.

[4] R.J.A. Buhr. Use Case Maps as architecture entities for complex systems. *IEEE Transactions on Software Engineering*, 24(12):1131–1155, December 1998.

[5] John Cheesman and John Daniels. *UML Components: A Simple Process for Specifying Component-Based Software*. Object Technology Series. Addison-Wesley, Reading, Massachusetts, 2000.

[6] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, Reading, Massachusetts, 2000.

[7] Hans de Bruin. A grey-box approach to component composition. In Krzysztof Czarnecki and Ulrich W. Eisenecker, editors, *Proceedings of the First Symposium on Generative and Component-Based Software Engineering (GCSE'99), Erfurt, Germany*, volume 1799 of *Lecture Notes in Computer Science (LNCS)*, pages 195–209, Berlin, Germany, September 28–30, 1999. Springer-Verlag.

[8] Hans de Bruin and Hans van Vliet. Scenario-based generation and evaluation of software architectures. In Jan Bosch, editor, *Proceedings of the Third Symposium on Generative and Component-Based Software Engineering (GCSE'2001), Erfurt, Germany*, volume 2186 of *Lecture Notes in Computer Science (LNCS)*, pages 128–139, Berlin, Germany, September 10–13, 2001. Springer-Verlag.

[9] Hans de Bruin and Hans van Vliet. Top-down composition of software architectures. In Per Runeson, editor, *Proceedings of 9th International Conference and Workshop on the Engineering of Computer-Based Systems (ECBS'2002), Lund, Sweden*, pages 1–10, April 8–11, 2002.

[10] David Garlan, Robert Allen, and John Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE Software*, 12(6):17–26, November 1995. Carnegie Mellon University.

[11] Gregor Kiczales, John Lamping, Anurg Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In M. Askit and M. Matsuoka, editors, *Proceedings of 11th European Conference on Object-Oriented Programming (ECOOP'97), Finland*, volume 1241 of *Lecture Notes in Computer Science (LNCS)*, pages 220–242, Berlin, Germany, June 9–13, 1997. Springer-Verlag.

[12] Hafedh Mili, Ali Mili, Sherif Yacoub, and Edward Addy. *Reuse-Based Software Engineering: Techniques, Organization, and Controls*. John Wiley and Sons, New York, 2002.

Using Component Composition for Self-customizable Systems

Ioana Şora Pierre Verbaeten Yolande Berbers
Katholieke Universiteit Leuven, Department of Computer Science
Celestijnenlaan 200A, 3001 Leuven, Belgium
Ioana.Sora@cs.kuleuven.ac.be

Abstract

Self-customizable systems are equipped with mechanisms to automatically adapt themselves to a set of user requirements or to their environment. We address this customization problem through component composition.

Our approach is based on hierarchically decomposed component systems, deploying composed components as a means of abstracting details. Composition is performed in a stepwise refinement manner, which allows to handle the complexity of the system and to realize very fine-tuned compositions even when composition decision is made automatically. The composition strategy is driven by anonymous dependencies established between components by their requirements. Our goal is to perform unanticipated customizations with as few user interventions as possible.

We evaluate and prove our composition approach by building customized network protocols.

1.Introduction

Component-based development is a proven approach to manage the complexity of software and its need for customization. The self-customization problem is a complex one, raising research questions about: the factors that trigger customization, how the composition decision is made and what infrastructure is needed to implement the changes. The focus of this paper is on the decisional question: what components will be deployed and what collaborations will be between them. This decision must be automatically made by a composition strategy implemented in the system.

The issue here is to be able to do as much unanticipated customizations as possible while still guaranteeing a correct composed system. Many approaches for automatic synthesis address only layered architectures for their simplicity [1], [2]. This limitation does not allow a fine tuned composition of more complex systems. Other approaches [3] rely on the criteria-driven selection of a right implementation for the defined components of a system. This limits the possibility of unanticipated customization.

We address the problem of the composition of a whole system according to a set of requirements by dividing it into subproblems of layered compositions on each flow of the system. In our approach, fine-tuned compositions and managing the complexity of the system are possible by deploying composable components. Our composition strategy is driven mainly by the anonymous dependencies established between components by their requirements. Unanticipated customizations are feasible in this strategy.

We evaluate and prove our composition approach by building customized network protocols. We integrate an automatic composition module in DiPS (Distrinet Protocol Stack framework) [4], a component framework for developing open protocol stacks, that ensures the needed infrastructure for composition.

The remainder of this paper is organized as follows. The next section presents the architecture and component model. Section 3 covers our correct composition strategy. We illustrate our approach by a protocol stack composition in Section 4 and summarize our results in Section 5.

2. Component model

2.1. Architecture

We work with a *flow-based architecture* using fine-grained components as the basic building blocks. A system is defined by a number of *flows* on which components are plugged one after the other. Components communicate by means of an *anonymous interaction model*, which limits mutual communication dependencies between components and allows individual components to be reused.

In our model, a system is built from components and connectors. Components may be *simple* or *composed*. A simple component is the basic unit of composition that is responsible for a certain behavior, and has one input and one output port. Composed components appear as a grouping mechanism and may have several input and output ports. The internal structure of a composed component is aligned on a number of *flows* connecting input with output ports.

Each component provides a set of services and may require that a certain set of services is available for it. When composing services together, the composed component gains own added value, and, as a whole, provides new services at a higher abstraction level than its parts.

2.2. Component descriptions

A component is described by a set of *ports* for the interaction with the rest of the system (*input ports* and

output ports) and a specified functionality. The functionality of a component is described by a set of *provides-requires clauses. Requires* clauses are associated with ports. In the case of simple components, p*rovides* clauses are associated with the component as a whole. In the case of composed components, *provides* clauses can also be associated with ports, reflecting the internal structure of the component. The composed component as a whole is always defined by its own *provides* clause, which expresses the higher-abstraction-level features gained through the composition of the subcomponents. The vocabulary used to describe the own *provides* of a composed component is distinct from the vocabulary deployed for describing the *provides* of its subcomponents. This abstraction definition must be done by the designer of the composed component. In the case of composed components, the existing *flows* must be specified, to allow establishing input-output relationships.

By default, it is sufficient that requirements are met by some components that are present in the flow connected to that port. One can specify *immediate* requirements, which apply only to the next component on that flow.

In our approach, the provides-requires clauses are expressed through lists of *properties*. A *property* is a rather abstract feature (a name). This enables the definition and implementation of our composition algorithm in a domain-independent manner. It is possible that a property is further specified with a list of subproperties, which describe fine-tuning attributes of the global property.

2.3. Structure of composed components

We use hierarchical relationships between components as a means of structuring and of providing fine-grained composition. In our model, the composed components do not have a fixed implementation. In this approach lies a powerful part of the customization capability: the full implementation of the component will be composed as a result of customization options. Fixed elements are these specified in the component description (ports, internal flows, general provides-requires clauses) and a set of *structural constraints*. These constraints *do not* fully determine a structure. The structural constraints comprise: *basic structural constraints*, *structural context-dependent requirements for components* and *inter-flow dependencies*. The basic structural constraints must be specified by the developer of the composed component. These describe the minimal properties that must be assembled on certain flows for the declared *provides* of the composed component to emerge and may define a "skeleton" of the composed component. The structural context-dependent requirements express requirements of other components when deployed here as subcomponents. These requirements will be added by the developer of these subcomponents. The inter-flow dependencies specify relationships between the flows.

We consider as an example the REL component, a component that realizes a reliability protocol. It has two flows, corresponding to the downgoing and upgoing paths through the protocol stack. Different types of reliability may be realized, but the REL component has a set of structural constraints imposed by its basic functionality. The basic functionality that contributes to all reliability protocols is quite simple: in order to recover from packet loss, the sending part will resend the data until an acknowledgement from the receiver has arrived. The basic structural constraints thus state that on the downgoing flow a retransmission strategy has to be provided, followed by a headerconstruction. On the upgoing flow, there has to be a headerparsing, a dispatching element that routes differently data and feedback packets, creating a flow ramification, and on these two flows, there has to be an acknowledgement receiving respectively an acknowledgement sending. These basic structural constraints imply a "skeleton" like that depicted in figure 1.

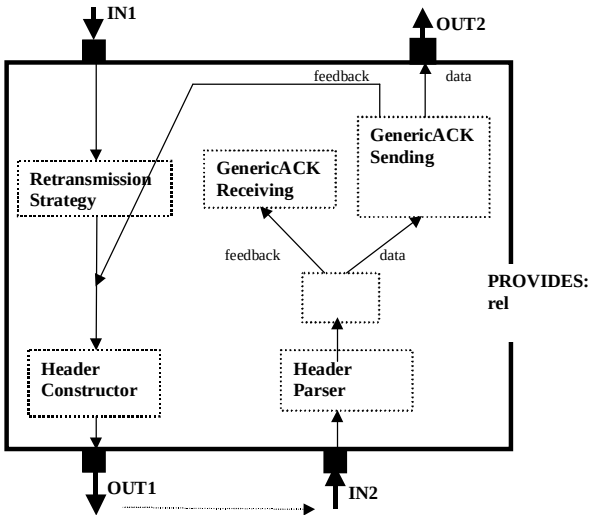


Figure 1. Example: basic structural constraints represented as a composed component skeleton

Between the two flows, the downgoing and upgoing flow, there is a "continuation" relationship. Over these basic structural constraints, a series of customizations will be possible. Different retransmission and acknowledgement strategy components will actually get deployed. For example, one choice can be to use positive or negative acknowledgements - that will reflect in deploying an `ACKSending` or a `NAKSending` component.

An important strength of our approach is that we do not limit the customization of composed components to filling in the given structure with right implementations. It should be possible that new components, which can provide further enhancements or customizations for the composed component, are discovered. The insertion of these new components is permitted anywhere on the existing flows, as long as their component descriptions do not contradict existing requirements (structural constraints of the composed component or requirements of the already present components on that flow).

In the case where new components are defined and implemented, there might appear situations where the existing requirements (own requirements of component and structural constraints of composed component) are not enough to exclude bogus compositions (are not able to

prevent the new component to be placed in inappropriate places). In this case, the provider of the new component must also specify a set of structural requirements to be added to the structural constraints of the composed components in which this new one could be deployed.

3. Correct composition

Our criteria for a correct composition is "matching all *requires* clauses with *provides* clauses, on all flows in the system". This criteria is used as well for validating a composition as for determining the right composition of a system with a set of given desired properties.

We introduce the mechanism of *propagation of requirements* as an essential element of our strategy. This mechanism works like delegating the responsibility for requirements posed to a component to other components. In a composition where a simple component B is connected to an outgoing port of component A while not fulfilling (all) requirements associated with that outgoing port of A, these unfulfilled requirements are virtually propagated to the outgoing port of B. In the case of composed components (with multiple input and output ports), the propagation of requirements follows the "intracomponent pathways" [5] originating in that input port. The internal flows of the composed component are used to determine which output is really affected by one particular input. A similar phenomenon of propagation appears in the case of composing components. The *requires* and the *provides* clauses of the subcomponents are propagated to the external ports of the composed component. We define a propagation mechanism in the context of components, similar to the propagation mechanism defined by Perry [6] in the context of formal description of program modules.

The automatic composition problem is the following: given a set of requirements describing the desired system, and a component repository that contains descriptions of available components, the composition process has to find a set of components and their interactions to realize the desired functionality. For our composition strategy, the requirements describing the desired system have to be expressed as *sets of required properties*, defined in the same vocabulary as used for the component descriptions. Rather than enumerating desired system properties, requirements should be expressed in a sufficiently high abstraction level domain specific language. In [7] we presented our view on deploying a *translation layer* for user requirements as a front-end tool for the composition module, as an essential element for the practical success of an automatic composition approach.

The composition results through stepwise refinements as depicted in Figure 2. After a composition process has determined that it wants a certain component type in place, a new composition problem may be launched for composing the internal structure of that component.

The overall building process is driven by the requirements. The required properties for the system are put on the main flow of the system and propagated from that point on, while adding components. The addition of new components on the flow occurs according to the current requirements, which are those propagated from the initial requirements together with those of the new introduced components. When a requirement has subrequirements (like `REQ p1 WITH p11, p12`) then a component found to provide p1 will have to be fine-tuned, so that its internal structure is compliant to the set of subrequirements p11, p12. A component is selected for the solution if it matches at least a subset of the current requirements. A solution is considered complete when the current requirements set becomes empty. It is possible that for certain sets of requirements no solution can be found.

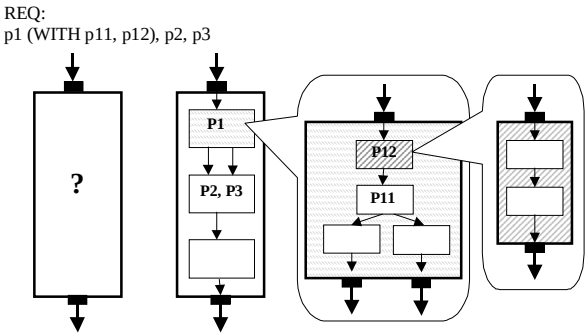


Figure 2. Composition in successive refinement steps

4. Case study

We have applied our composition approach to customize network protocol stacks. Since currently protocol stacks operate in various contexts, it is not possible to know the required properties of a stack in advance, so stack configuration mechanisms are needed.

We designed and integrated an automatic composition module into DiPS (Distrinet Protocol Stack framework) [4], a component framework for developing open protocol stacks. DiPS ensures the run-time support for dynamic protocol stack changes and provides the infrastructural support for the composition of layers and components.

The following example illustrates our composition approach. An application needs a reliable communication link for multimedia transmissions. This translates into the global requirement `REQUIRES rel (WITH MultimediaRel), transp, non_local`.

A stack, which has as structural constraints the existence of two flows, a downgoing and upgoing path, will be constructed as a sequence of layers, following the requirements along the downgoing flow. A similar example is discussed in [7], but considering only coarse-grained components. We have shown there how propagation of requirements leads to the selection and ordering of the components on one flow. The composition of the stack could result in two solutions, TCP on IP or REL on UDP on IP, both combinations providing reliable transport.

In the case study presented in this paper we go an important step further: the reliability property has to be fine-tuned, i.e. for multimedia transmissions, which requires a special retransmission policy. This fine-tuning is not

possible when composing only monolithic coarse-grained components. Our current work permits fine-tuning through the deployment of composable components. The REL component will be composed according to the requirement MultimediaRel applied over its structural constraints. The TCP reliability retransmission strategy is not suitable, thus TCP on IP will be rejected.

For the composition of the REL component (figure 3), the MultimediaRel requirement is forwarded to the downgoing flow of the component, leading to the selection of the `MultimediaRelStrategy` component for providing the right retransmission strategy. The component `MultimediaRelStrategy` requires further support for readjustment of the retransmission timeout – this leads to inclusion of a `RoundTripTimeCalculator`, placed, according to its own and structural requirements, on the upgoing flow. The `RoundTripTimeCalculator` needs time stamps to be attached on its incoming flow – so a `TimeStampAttacher` component is placed on the downgoing flow after the retransmission strategy. Acknowledgement sending and receiving has to be handled, according to the skeleton of the composed component. Since no preference for the acknowledgement strategy exists, positive acknowledgements are chosen (the `AckReceivingUnit` and `AckSendingUnit` components). `AckSendingUnit` is a generic composed component that has to be composed. A Filter is needed, and component `NextSequenceFilter` will be chosen, since it is compatible with multimedia retransmission strategy on its incoming flow.

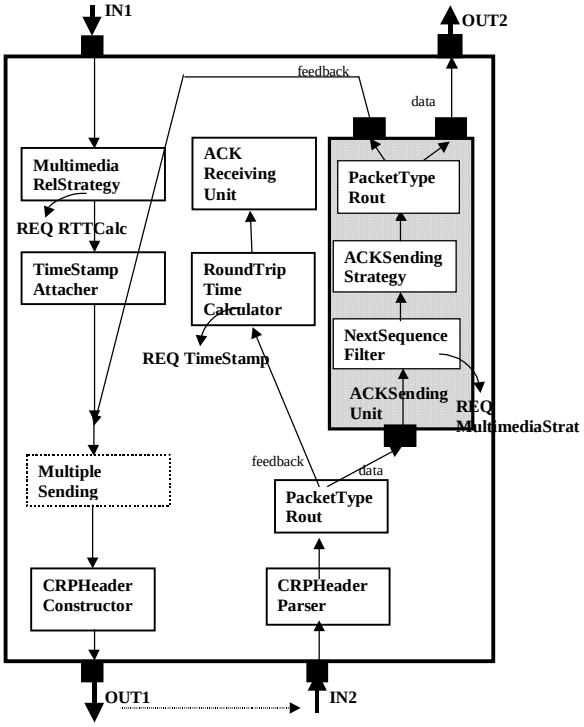


Figure 3. Composition of a reliability layer component

To make it clearer how our approach may handle unanticipated customizations, suppose that a new component, `MultipleSending`, is developed and could be used to enhance the performance of the Reliability layer. The requirements of this component impose that it is used on an outgoing flow of a retransmission strategy. This implies that, when multiple sending is required, such a component is deployed like in figure 3.

The reuse of existing components is possible in different contexts. For example, a `Fragmenter` or a `RoundTripTimeCalculator` may be used in different contexts, use guided by the structural constraints of the components where they are going to be deployed.

5. Conclusions

This paper addresses automatic composition of component-based systems. We discuss our composition strategy, that finds the set of components and their interactions to compose a system starting from a given set of required properties for the system.

Our ongoing work, presented in this paper, is generalizing our composition approach for layered systems presented in [7] to cope with more general flow-based architectures and to permit very fine-tuned customizations of large systems. Another strength of our approach is that it is open to unanticipated customizations, being able to include new component types in compositions, along with the reuse of existing components.

Using the networking domain as application domain example, we illustrate how our composition strategy works on building a protocol stack that complies to client-specific requirements. Developing a prototype that integrates an automatic composition module into the DiPS [4] component framework, we have validated our approach.

6. References

[1] D. Batory, G. Chen, E. Robertson, T. Wang, "Design Wizards and Visual Programming Environments for GenVoca Generators", *IEEE Transactions on Software Engineering*, Vol.26, No. 5, May 2000.

[2] K. Czarnecki, U. Eisenecker, "Synthesizing Objects", in *Proceedings ECOOP'99, Lecture Notes in Computer Science 1628*, Springer, Lisbon, Portugal, June 1999, pp. 18-42.

[3] E. Truyen, B.N. Joergensen, W. Joosen, "Customization of Object Request Brokers through Dynamic Reconfiguration", in *Proceedings TOOLS Europe 2000*, June 2000, France.

[4] F. Matthijs, "Component Framework Technology for Protocol Stacks", PhD Thesis, Katholieke Universiteit Leuven, December 1999.

[5] J.A. Stafford, A.L. Wolf, "Architecture-Level Dependence Analysis for Software Systems", *International Journal of Software Engineering and Knowledge Engineering*, Vol.11, No.4, August 2001, pp. 431-452.

[6] D.E. Perry, "Software Interconnection Models", *Proceedings of the 9th International Conference on Software Engineering*, Monterey CA, March 1987, pp. 61-69.

[7] I.Şora, F.Matthijs, Y.Berbers, P.Verbaeten, "Automatic composition of systems from components with anonymous dependencies specified by semantic-unaware properties", in *Proceedings TOOLS EE 2001*, Sofia, Bulgaria, March 2002.

Case Study: A Component-Based Software Architecture for Industrial Control

Frank Lüders
ABB Automation Technology Products
frank.luders@mdh.se

Andreas Sjögren
Mälardalen University
andreas.sjogren@mdh.se

Abstract

When different business units of an international company are responsible for the development of different parts of a large system, a component-based software architecture may be a good alternative to more traditional, monolithic architectures. The new common control system, developed by ABB to replace several existing control systems, must incorporate support for a large number of I/O systems, communication interfaces, and communication protocols. An activity has therefore been started to redesign the system's architecture, so that I/O and communication components can be implemented by different development centers around the world. This paper reports on experiences from this effort, describing the system, its current software architecture, the new component-based architecture, and the lessons learned so far.

1. Introduction

Increased globalization and the more competitive climate make it necessary for international companies to work in new ways that maximize the synergies between business units around the world. Interestingly, this may also require the software architecture [1] of the developed systems to be rethought. In a case where different development centers are responsible for different parts of the functionality of a large system, a component-based architecture may be a good alternative to more traditional, monolithic architectures, usually comprising a large set of modules with many visible and invisible interdependencies. Additional, expected benefits of a component-based architecture are increased flexibility and ease of maintenance [2,3].

This paper reports on experiences from an ongoing project at ABB to redesign the software architecture of a control system to make it possible for different development centers to incorporate support for different I/O and communication systems. The remainder of the paper is organized as follows. In Section 2, the ABB control system is described with particular focus on I/O and communication. The software architecture and its transformation are described in more detail in Section 3. In Section 4, we analyze the experiences from the project and try to extract some lessons of general value. Section 5 reviews some related work in this area. Section 6 present our conclusions and outlines future work.

2. The ABB control system

Following a series of mergers and acquisitions, ABB now has several independently developed control systems for process, manufacturing, substation automation and related industries. The company has decided to continue development of only a single, common control system for these industries. One of the existing control systems was selected as the starting point. The software has two main parts, the ABB Control Builder, which is a Windows application running on a standard PC, and the system software of the ABB controller family, running on top of a real-time operating system on special-purpose hardware. The latter is also available as a Windows application called the ABB Soft Controller.

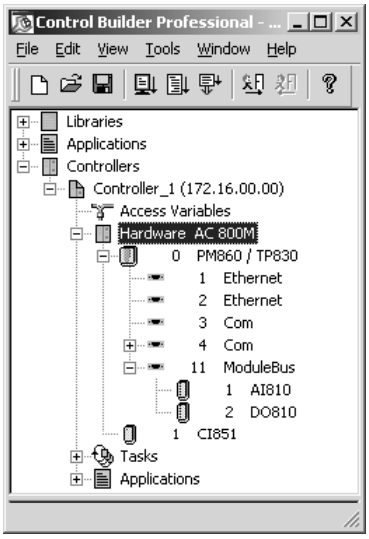


Figure 1. The ABB Control Builder

The Control Builder is used to specify the hardware configuration of a control system, comprising one or more controllers, and to write the programs that will execute on the controllers. When the configuration and the control programs, called a control project, are downloaded to the control system via the control network, the system software of the controllers is responsible for

interpreting the configuration information and for scheduling and executing the control programs. Figure 1 shows the Control Builder with a control project opened. It consists of three structures showing, the libraries used by the control programs, the control programs themselves, and the hardware configuration, respectively. The latter structure is expanded to show a configuration of a single AC800M controller, equipped with an AI810 analogue input module, a DO810 digital output module, and a CI851 PROFIBUS-DP communication interface.

To be attractive in a wide range of industry sectors, the common control system must incorporate support for a large number of I/O systems, communication interfaces, and communication protocols. In the current system, there are two principal ways for a controller to communicate with its environment, I/O and variable communication. When using I/O, variables in the control programs are connected to channels of I/O modules using the program editor of the Control Builder. Figure 2 shows the editor with a small program, declaring one input variable and one output variable. Notice that the I/O addresses for the two variables correspond to the position of the two I/O modules in Figure 1.

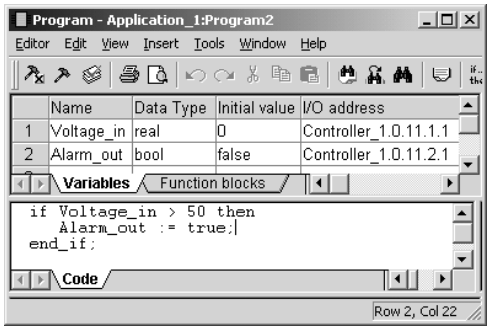


Figure 2. The Control Builder program editor

Variable communication is a form of client/server communication. A server supports one of several possible protocols and has a set of named variables that may be read or written by clients that implement the same protocol. A controller can be made a server by connecting program variables to so-called access variables in the Control Builder. A controller can act as a client by connecting to a server and reading and writing variables via the connection.

3. Componentization

3.1. Current software architecture

The software of the ABB control system consists of a large number of source code modules, each of which are used to build the Control Builder or the controller system software or both. Figure 3 depicts this architecture, with emphasis on I/O and communication. Many modules are also used as part of other products, which are not discussed further here. This architecture is thus a product-line architecture [4], although the company has not yet adopted a systematic product-line approach. The boxes in the figure represent logical components of related functionality. Each logical component is implemented by a number of modules, and is not readily visible in the source code.

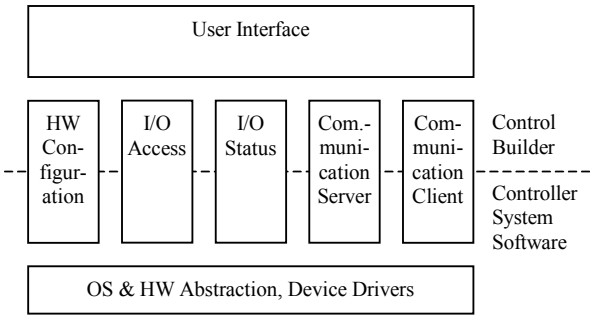


Figure 3. The current software architecture

The main problem with this architecture is related to the work required to add support for new I/O modules, communication interfaces, and protocols. For instance, adding support for a new I/O system may require source code updates in all the components except the User Interface and the Communication Server, while a new communication interface and protocol may require all components except I/O Access to be updated. As an example of what type of modifications may be needed to the software, we consider the incorporation of a new type of I/O module.

To be able to include a device, such as an I/O module, in a configuration, a hardware definition file for that type of device must be present on the computer running the Control Builder. For an I/O module, this file defines the number and types of I/O channels. The Control Builder uses this information to allow the module and its channels to be configured using a generic configuration editor. This explains why the user interface does not need to be updated to support a new I/O module. The hardware definition file also defines the memory layout of the module, so that the transmission of data between program variables and I/O channels can be implemented in a generic way.

For most I/O modules, however, the system is required to perform certain tasks, for instance when the configuration is compiled in the Control Builder or during start-up and shutdown in the controller. In today's system, routines to handle such tasks must be hard-coded for every type of I/O module supported. This requires software developers with a thorough knowledge of the source code. The limited number of such developers therefore constitutes a bottleneck in the effort to keep the system open to

the many I/O systems found in industry. The same is true for communication interfaces and protocols.

3.2. Component-based software architecture

To make it much easier to add support for new types of I/O and communication, it was decided to split the components mentioned above into their generic and non-generic parts. The generic parts, commonly called the generic I/O and communication framework, contains code that is shared by all hardware and protocols implementing certain functionality. Routines that are special to a particular hardware or protocol are implemented in separate components, called protocol handlers, installed on the PC running the Control Builder and on the controllers. This component-based architecture is illustrated in Figure 4. To add support for a new I/O module, communication interface, or protocol to this system, it is only necessary to add protocol handlers for the PC and the controller along with a hardware definition file. The format of hardware definition files is extended to include the identities of the protocol handlers.

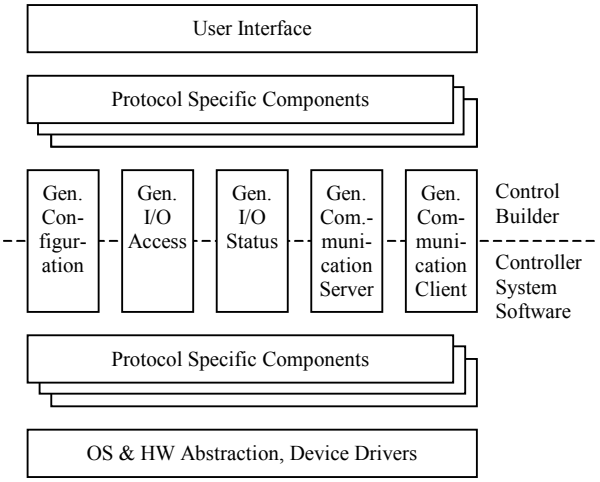


Figure 4. The component-based architecture

Essential to the success of this approach, is that the dependencies between the framework and the protocol handlers are fairly limited, and even more importantly, well specified. One common way of dealing with such dependencies is to specify the interfaces provided and required by each component. ABB's component-based control system uses Microsoft's Component Object Model (COM) [5], since it provides suitable formats both for writing interface specification, using the COM Interface Definition Language (IDL), and for run-time interoperability between components. For each of the generic components, two interfaces are specified: one that is provided by the framework and one that may be provided by protocol handlers. Interfaces are also defined for interaction between protocol handlers and device drivers. The identities of protocol handlers are provided in the hardware definition files as the Globally Unique Identifiers (GUIDs) of the COM classes that implement them.

COM allows several instances of the same protocol handler to be created. This is useful, for instance, when a controller is connected to two separate networks of the same type. Also, it is useful to have one object, implementing an interface provided by the framework, for each protocol handler that requires the interface. An additional reason that COM is the technology of choice is that it is expected to be available on all operating systems that the software will be released on in the future. In the first release of the system, which will be on a platform without COM support, the protocol handlers will be implemented as C++ classes, which will be linked statically with the framework. This works well because the Microsoft IDL compiler generates C++ code corresponding to the interfaces defined in an IDL.

When a control system is configured to use a particular protocol, the Control Builder uses the information in the hardware definition file to load the protocol handler on the PC and execute the protocol specific routines it implements. During download, the identity of the protocol handler on the controller is sent along with the other configuration information. The controller system software then tries to load this protocol handler. If the protocol handler is available an object is created and the required interface pointers obtained. Objects are then created in the framework and interface pointers to these passed to the protocol handler. After the connections between the framework and the protocol handler has been set up through the exchange of interface pointers, a method will be called on the protocol handler object that causes it to continue executing in a thread of its own. Since the interface pointers held by the protocol handler references objects in the framework, which are not used by anyone else, all synchronization between concurrently active protocol handlers can be done inside the framework.

4. Lessons learned

The definitive measure of the success of the project described in this paper will be how large the effort required to redesign the software architecture has been compared to the effort saved by the new way of adding I/O and communication support. At the time of writing, the specification of generic interfaces and the implementation the framework are largely completed, and it seems safe to conclude that the efforts are of the same order of magnitude as the work required to add support for an advanced I/O or communication system the old way, i.e. by adding code to the affected modules. From this we infer that, if the new software architecture makes it sub-

stantially easier to add support for such systems, the effort has been worthwhile. We therefore find that the experiences with the ABB control system supports our hypothesis that a component-based software architecture is an efficient means for supporting distributed development of complex systems.

A lesson of general value is that it seems that a component technology, such as COM, can very well be used on embedded platforms and even platforms where runtime support for the technology is not available. Firstly, we have seen that the overhead that follows from using COM is not larger than what can be afforded in many embedded systems. In fact, used with some care, COM does not introduce much more overhead than do virtual methods in C++. Secondly, in systems where no such overhead can be allowed, or systems that run on platforms without support for COM, IDL can still be used to define interfaces between components. Thus, the same interface definitions can be used with protocol handlers implemented as dynamically linked COM components and statically linked C++ classes or C modules.

Another interesting experience from the project is that techniques that were originally developed to deal with dynamic hardware configurations have been successfully extended to cover dynamic configuration of software components. In the ABB control system, hardware definition files are used to specify what hardware components a controller may be equipped with and how the system software should interact with different types of components. In the redesigned system, the format of these files has been extended to specify which software components may be used in the system. The true power of this commonality is that existing mechanisms for handling hardware configurations can be reused largely as is. The idea that component-based software systems can benefit by learning from hardware design is also aired in [2].

5. Related work

The use of component-based software architecture in real-time, industrial control has not been extensively studied, as far as we know. One example is documented in [6], which describes work not based on industrial experiences, but from the construction of a prototype, developed in academia for non-real-time platforms with input from industry. A research project focusing on tools end techniques for ensuring correct composition of components in embedded systems is described in [7]. An example of a commercial system for component-based development of real-time control systems is ControlShell [8], which supports construction from re-usable components using a graphical editor and automatic code generation.

6. Conclusions and future work

The initial experiences from the effort to redesign the software architecture of ABB's control system to support component-based development are promising. We have already claimed that the experiences recorded in this paper support our hypothesis that component-based software architectures is a good alternative to monolithic architectures for complex systems developed in distributed organizations. It will be a primary goal of our future work to strengthen this claim by presenting data that verifies that the development of I/O and communication support is made substantially easier by the new architecture.

In addition, we plan to study in more detail how non-functional requirements are addressed by the software architecture, since the architecture of a system is often seen as a primary means for meeting such requirements [1]. We will, for instance, look at reliability, which is an obvious concern when externally developed software components are integrated into an industrial system. Other goals are to investigate the additional expected benefits of increased flexibility and ease of maintenance and to compare the performance of the system after the redesign to that of the current system.

7. References

[1] L. Bass, P. Clements, R. Katzman, *Software Architecture in Pracice*, Addison-Wesley, 1998.

[2] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, Addison-Wesley, 1997.

[3] H. Hermansson, M. Johansson, L. Lundberg, "A Distributed Component Architecture for a Large Telecommunication Application", *Proceedings of APSEC '00*, December 2000.

[4] J. Bosch, *Design and Use of Software Architectures – Adopting and Evolving a Product-Line Approach*, Addison-Wesley, 2000.

[5] Microsoft Corporation, *The Component Object Model Specification*, Version 0.9, October 1995.

[6] A. Speck, "Component-Based Control System", *Proceedings of ECBS '00*, April 2000.

[7] T. Genssler, C. Zeidler, "Rule-Driven Component Composition for Embedded Systems", *Proceedings of the ICSE '01 Workshop on CBSE*, May 2001.

[8] S. A. Schneider, V. W. Chen, G. Pardo-Castellote, "ControlShell: Component-Based Real-Time Programming", *Proceedings of RTAS '95*, May 1995.

Using Parameterised Contracts to Predict Properties of Component Based Software Architectures

Ralf H. Reussner
CRC for Enterprise Distributed Systems
Technology Pty Ltd
Monash University
900 Dandenong Road,
Caulfield East, VIC 3145, Australia
reussner@dstc.com

Heinz W. Schmidt
Center for Distributed Systems
and Software Engineering
Monash University
900 Dandenong Road,
Caulfield East, VIC 3145, Australia
hws@csse.monash.edu.au

Abstract

This position paper presents an approach for predicting functional and extra-functional properties of layered software component architectures. Our approach is based on parameterised contracts *a generalisation of design-by-contract. The main contributions of the paper are twofold. Firstly, it attempts to clarify the meaning of "contractual use of components" a term sometimes used loosely – or even inconsistently – in current literature. Secondly, we demonstrate how to deploy parameterised contracts to predict properties of component architectures with non-cyclic dependencies.*

1. Introduction

In the recent past, two successful areas of software engineering, namely software architecture and software components moved closer together. In fact, we believe they are just two sides of the same coin, given the intertwining of architectural and detailed design and the need to connect both system-level and component-level reasoning about software.

One of the major motivations of software architecture, the aim to reason explicitly about extra-functional properties during software-design, benefits from focusing on *component based* software architectures. While current research in that area mostly concentrates on component interoperability checks within software architectures and component adaptation in case of incompatibility, predicting properties of the overall architecture from known component properties has gained attraction [2, 11].

From a conceptual point of view, one can consider software architectures as structuring principles and methods for component assemblies. Software architecture and component (re-)configuration are also closely connected. Based on this strong connection, architecture promises help in compositional reasoning, i.e., predicting system properties and qualities from component properties by using the architectural structuring mechanisms in the reasoning process and without recourse to the component internals.

To be able to predict overall architectural properties, local interaction must be correct in terms of the interface model. This means no errors should occur by calling methods with wrong parameters or by calling methods in the wrong order. Due to that necessity of local correctness, we discuss contracts for components first (in section 3). Such contracts specify conditions for correct local interaction. Beyond traditional contracts, in section 4 we present a generalisation called *parameterised contracts* [7, 6]. These deal with the prediction of properties of composite components based on the properties of their basic components. In addition, parameterised contracts depend on the environment in which components are deployed. Some of the environment properties become parameters to these contracts. The importance of this parameterisation becomes clear, when looking at extra-functional properties like timing behaviour or reliability. Here, the timing behaviour (reliability) of the component clearly depends on the timing behaviour (reliability, resp.) of environmental services used by the component.

2. Example

As an example, Figure 1 shows a composite component (`MobileMailViewer`) which offers the service of displaying mails of various formats on a mobile personal organiser. Internally, the `MobileMailViewer` consists of a `Controler` (handling the selection of mails, connec-

tion to an address book, formatting of strings, etc.) and a `MailServer` (delivering the mails) and a `ViewerSoftwareServer`, which provides the `Controler` with the a viewer appropriate to the format of the actual email. Since memory is limited on mobile devices, we assume the device cannot store viewers for all formats. Also the programmer of the controler cannot foresee all future mail or attachment formats. To the `Controler` component, both servers are

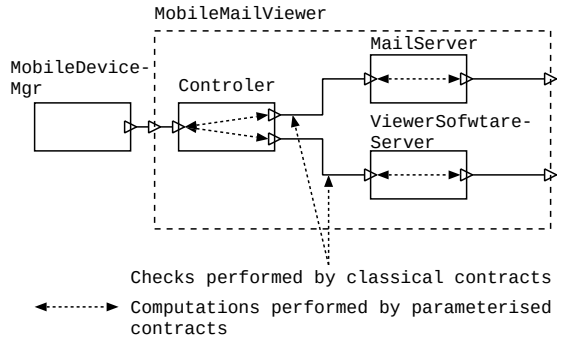


Figure 1. Configuration of a mobile viewer

remote. Nevertheless, the personal manager program on the mobile device considers the `MobileMailViewer` as a single local component.

In our figure, rectangles denote components and triangle denote interfaces. Interfaces are objects themselves not just (meta-)descriptions. Components have two kinds of interfaces: provides- and requires-interfaces. The former describe services offered by a component, the latter services required by the component. Components connected to a components interfaces form the *environment* of the component. In our example, the `Controler` component requires the `MailServer` and the `ViewerSoftwareServer` and offers services to the `MobileDeviceManager`. Although the need of requires-interfaces is obvious for interoperability and substitutability check (and well-known in literature [13, 4]), current component models like Sun's EJB or Microsoft's .NET only contain provides interfaces. (One notable exception is CORBA 3.0).

3. Contractual Use of Components in Software Architectures

Much of the confusion regarding "contractual use" of a component derives from the double meaning of the term "use". It can refer to

1. the *run-time use*, when the methods of a component are called. For example, `displayMessage` of the `Controler` component is used in this way.

2. the *composition-time use*, when a component is placed into a new context or environment. This includes the development time but can also happen when an operational system is reconfigured.

Depending on the above case, contracts play a different role. Before actually defining contracts for components, we briefly review the design-by-contract principle. According to [5, p. 342] a contract between the client and the supplier consists of two obligations for each service or method:

- The client must satisfy the precondition of the supplier.
- The supplier has to fulfil its postcondition, provided the precondition was met be the client.

Each of the above obligations can be seen as the benefit to the other party. In a nut shell:

> If the client fulfils the precondition of the supplier, the supplier will fulfil its postcondition.

It is clear, that a used component plays the role of a supplier. But to formulate contracts for components, we also have to identify the pre- and postconditions and the client of a component.

Considering the run-time use first, the clients of a component C are all components connected to C's provides interface(s). The precondition for run-time use is the collection of preconditions of the component's services. Likewise the run-time use postcondition is the collection of postconditions of the provided services. This is still close enough to the traditional design-by-contract. Hence we consider this kind of contract the *service* contract.

The composition-time use or reuse occurs in the architectural design or reconfiguration of a system. From an architectural viewpoint the component C depends on its environment through its requires interfaces. Its correct functioning does not only depend on the preconditions of its provided services. Hence we regard the the requires interfaces as a kind of abstract preconditions at the component contract level. Similarly, from an architectural viewpoint, the provided services are the promised benefits to the (run-time) user. Therefore, we consider those provides interfaces the postconditions of the component-level contract. From the perspective of logical specifiation the story is a little more complex in that each of these conditions is a conditional specification – for instance, the required environment behaviour is only delivered if the component itself satisfies the preconditions of the requires interface services.

Putting it all together we formulate the architecture-by-contract principle as follows:

> If the user of a component fulfils the components' required interface (i.e., the precondition) by offering the right environment the component will offer its services as described in the provided interface (i.e., its postcondition).

Note that checking the satisfaction of a requires interface includes checking whether the contracts of required services (the service contracts specified in the requires-interface(s)) are sub-contracts of the service contracts stated in the provides interfaces of the required components. The notion of a subcontract is described in [5, p. 573] and generalised in [9] using contravariant typing for methods but importantly including invariants as conditions for distributed and hence typically concurrent environments.

For checking the correct contractual use of the `Controler` component in our example we check whether the services specified in the requires interface of `Controler` are included in the provides interface of `MailServer` and whether the contracts of requires services are subcontracts of provides services. Similarly we check the binding between the other requires and provides interfaces.

More generally, when architecting systems (i.e., introducing new components), we have to check the bindings of their requires interfaces to the used environmental provides interfaces in addition to checking the use of the component's providees interfaces. When replacing a component with a newer one, we not only have to check their contract (i.e., the bindings of their requires-interfaces to the used components, like mentioned above), but also the contracts of the using environmental components (i.e., the bindings from the provides-interfaces), because one has to ensure, that by a replacement non of the existing local contracts have been broken. In our example, if we replace the `Controler` component we have to (a) check the contractual use of `Controler`, i.e., we check the precondition of the `Controler` (i.e., the interoperability with `MailServer` and `ViewerSoftwareServer`), and (b) we have to check whether the precondition of `MobileDeviceMgr` is still fulfilled (i.e., checking the contractual use of `MobileDeviceMgr`).

There is a range of formalisms used for specifying pre- and postconditions, defining a range of interface models for components (see for extensive discussions and various models e.g., [3, 12, 6]). This leads naturally to different kinds of contracts for components [1].

Another degree of freedom in the abstract principle of design-by-contract and our extension to architecture-by-contract is the time of component deployment. Component contracts as discussed here describe the deployment of components at composition-time. This stresses the importance of contracts which are statically checkable. When a system is architected or reconfigured, errors are common. Therefore, the direct feedback regarding correct component deployment is very helpful in practice, because it can assure the absence of composition errors. In contrast, the run-time checks can detect contract violations at run-time only. In many classes of distributed systems, such late detection of composition errors is unsatisfactory from a quality of service point of view. The problems and costs of late composition error discovery are compounded as the person running the system and triggering the error usually is not the system architect or maintainer, which may now have difficulties reproducing the error or obtaining sufficient information to locate and correct it. Additionally the high costs of hardware component recalls and replacements are well known in other industries. Similarly cost explosions can be expected in a component software industry despite the benefit of electronic recall and delivery.

4. Parameterised Contracts

A component rarely fits directly into a new reuse context. For a component developer it is hard to foresee all possible reuse contexts. Hence, it is also hard for a developer to provide components with reasonable configuration options to fit into future reuse contexts. Coming back to our discussion about component contracts, this means, that in practice one single pre- and postcondition of a component will not be sufficient, because of the following common cases:

1. the precondition of a component is not satisfied by a specific environment while the component itself would be able to provide a meaningful subset of its functionality. In the example, the `ViewerSoftwareServer` might fail or even be absent, but the `Controler` could still present standard text emails, although perhaps not display certain attachments.

2. a weaker postcondition of a component is sufficient in a specific reuse context. For example, the component user might not require all functions. Hence the component will itself require less functionality vis its requires interfaces and hence weaken its component precondition.

To model this we need some sort of adaptive pre- and postconditions. We call these *parameterised contracts* [7, 6]. In case 1 a parameterised contract computes the postcondition dependent upon the strongest precondition guaranteed by a specific reuse context. Hence the postcondition is parameterised with the precondition. In case 2 the parameterised contract computes the precondition dependent upon the postcondition (which acts as a parameter of the precondition). For components this means, that provides- and requires-interfaces are not fixed but are computed to some extent taking into account the reuse context. Hence, in contrast to classical contracts, one can say:

> Parameterised contracts link the provides- and requires interface(s) of the same component (see fig. 1). They range over many possible actual contracts (i.e., ultimately interfaces).

Interoperability is a special case now: if a component is interoperable with its environment, its provides interface will not change. If the interoperability check fails, the parameterised contract tries to compute a new provides interface.

5. Applications of Parameterised Contracts

Like classical contracts, parameterised contracts depend on the actual interface model and should be statically computable. In any case, the software developers do not have to foresee possible reuse contexts but has to provide a bidirectional mapping between provides- and requires-interfaces. For simple interface lists (signatures a la CORBA IDL say), this means, that for each provided service, a list of required external services must be provided by the component developer. When computing the actual provides interface, a service would only be included, if all its required services are provided by the component's environment. If interfaces also describe component protocols, one has to specify a mapping from the provides interface to the requires interface protocol which also identifies the order in which reuquires services are invoked. We have developed tools for such models [6].

For extra-functional properties, the application of parameterised contracts is crucial. For example, one cannot specify the timing behaviour of a software component by some fixed figure. For example the worst-case time of a real-time component is always some function of the time it takes to perform critical system services that are only provided in the deployment environment. The same argument holds for reliability as empirically validated in our recent paper [8].

By connecting parameterised contracts of single components within component architectures and considering critical properties of the deployment environment, one can now compute the overall architectural properties. Our methods are described in [6, 10] in more detail. They necessitate parameterised contracts and are currently limited to non-cyclic architectures (i.e. layers of abstract machines). In our example, we can compute the timing of the composed component `MobileMailViewer` by computing the timing the `Controler` can provide in dependency of the timing `MailServer` and `ViewerSoftwareServer` can provide.

6. Conclusion

This paper discussed contractual usage of software component. We present requires interfaces as precondition of components and provides interfaces as postconditions. Parameterised contracts then link provides and requires interfaces of the same component. They are motivated by the necessity of computing functional and extra-functional component properties dependent upon deployment context. Our methods are supported by an existing tool and discussed using a running example.

References

[1] A. Beugnard, J.-M. Jézéquel, N. Plouzeau, and D. Watkins. Making components contract aware. *Computer*, 32(7):38–45, July 1999.

[2] I. Crnkovic, H. Schmidt, J. Stafford, and K. Wallnau. 4th ICSE workshop on Component-Based software engineering: Component certification and system prediction. In *Proceedings of the 23rd International Conference on Software Engeneering (ICSE-01)*, pages 771–772, Los Alamitos, California, May12–19 2001. IEEE Computer Society.

[3] B. Krämer. Synchronization constraints in object interfaces. In B. Krämer, M. P. Papazoglou, and H. W. Schnmidt, editors, *Information Systems Interoperability*, pages 111–141. Research Studies Press, Taunton, England, 1998.

[4] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *Proceedings of ESEC '95 - 5th European Software Engineering Conference*, volume 989 of *Lecture Notes in Computer Science*, pages 137–153, Sitges, Spain, 25–28 Sept. 1995. Springer-Verlag, Berlin, Germany.

[5] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, Englewood Cliffs, NJ, USA, second edition, 1997.

[6] R. H. Reussner. *Parametrisierte Verträge zur Protokolladaption bei Software-Komponenten*. Logos Verlag, Berlin, 2001.

[7] R. H. Reussner. The use of parameterised contracts for architecting systems with software components. In W. Weck, J. Bosch, and C. Szyperski, editors, *Proceedings of the Sixth International Workshop on Component-Oriented Programming (WCOP'01)*, June 2001.

[8] R. H. Reussner, H. W. Schmidt, and I. Poernomo. Reliability prediction for component-based software architectures. *submitted to Journal of Systems and Software – Special Issue of Software Architecture - Engineering Quality Attributes*, 2002.

[9] H. W. Schmidt. Compatibility of interoperable objects. In B. Krämer, M. P. Papazoglou, and H. W. Schnmidt, editors, *Information Systems Interoperability*, pages 143–181. Research Studies Press, Taunton, England, 1998.

[10] H. W. Schmidt and R. H. Reussner. Generating Adapters for Concurrent Component Protocol Synchronisation. *accepted for the Proceedings of the Fifth IFIP International conference on Formal Methods for Open Object-based Distributed Systems*, Mar. 2002.

[11] J. Stafford and K. Wallnau. Predicting feature interactions in component-based systems. In *Proceedings of the Workshop on Feature Interaction of Composed Systems*, June 2001.

[12] A. Vallecillo, J. Hernández, and J. Troya. Object interoperability. In A. Moreira and S. Demeyer, editors, *ECOOP '99 Reader*, number 1743 in LNCS, pages 1–21. Springer-Verlag, 1999.

[13] N. Wirth. *Programming in MODULA-2*. Springer-Verlag, 3rd Edition, 1985.

Estimation of Static Memory Consumption for Systems Built from Source Code Components

E.M. Eskenazi, A.V. Fioukov, D.K. Hammer, M.R.V. Chaudron
Department of Mathematics and Computing Science, Eindhoven University of Technology,
Postbox 513, 5600 MB Eindhoven, The Netherlands
+31 (0)40 – 247 4449
{ e.m.eskenazi, a.v.fioukov, d.k.hammer, m.r.v.chaudron }@tue.nl

Abstract

The quantitative evaluation of certain quality attributes – memory consumption, timeliness, and performance – is important for component-based embedded systems. We propose an approach for the estimation of static memory consumption of software components. The approach deploys the Koala component model, used for embedded software in TV sets. There are two main parts in the method: specification of the memory demand of components and estimation of memory demand for systems built of these components. The proposed method allows flexible trade-off between estimation effort and achievable precision, yet requiring no changes in the tools supporting the Koala component model. The method may be extensible to include other resource attributes as well.

1. Introduction

Nowadays, component-based engineering [4], [6] actively enters the area of product families for resource-constrained embedded systems. For example, Philips Electronics is deploying a proprietary component model, called Koala [5].

The Koala component model focuses on the following points: (1) diversity handling to build different products from existing components for supporting the development of product families and (2) efficiency in resource-constrained systems, since it is applied to the high-volume electronics domain where product costs are the driving factors.

Koala does not yet provide any explicit mechanism for the quantitative assessment of the memory demand for code and static data. The most challenging issue here is to predict the memory demand for component compositions, given the demands of the constituents. Since our goal is the support of the early product creation phases (feasibility study, architecting etc.), we choose for static evaluation techniques. As we aim at reasoning about quality attributes of Koala components, we concentrate on source code components.

2. Problem analysis

This section describes the objectives of the approach, complications to be tackled, and assumptions made.

2.1 Requirements

We formulated the following requirements for the approach for memory consumption estimation:

1. The approach should be compositional. This means that the memory consumption of the component composition should be expressed in terms of the memory demands of the constituents[1].
2. The approach should be tunable with respect to the estimation accuracy. There is a trade-off between the estimation effort and the accuracy.
3. The approach should support budgeting. It should be possible to take into account the estimates of memory demands for the components that are not developed yet.

2.2 Complications

There are some Koala language-specific features [5] influencing the memory consumption of a component:

1. *Diversity and optional interfaces*. Diversity interfaces are used to tune reusable components for specific needs of product family members. The components can be configured with diversity parameters via diversity interfaces. Optional interfaces contain one Boolean diversity parameter determining whether interface is present in a particular product or not.
2. *Function binding*. It is possible to substitute a function of an interface with an expression (a piece of code) specified in the component description file.

[1] In principle, this requirement should hold for any quality attribute. For more detail, the reader is referred to [1], [2], [3].

Consequently, some C-language expressions are injected into the component code, and prediction of the code size becomes dependent on that.

3. *Interface binding*. During the build process, the Koala compiler monitors the use of the *provides* interfaces of the components. The component is not reachable and excluded from building if none of its *provides* interfaces are connected.

There are additional complicating factors, such as:

1. C Compiler optimizations. Modern compilers can optimize object code to decrease the amount of the required memory. The results of these optimizations may be very context dependent and are consequently hard to predict.
2. Platform dependency. The necessary amount of memory for a component depends on target hardware platform due to bus width and data alignment.
3. Mapping of memory regions. After compiling, the object code can be allocated to different types of memory, e.g. internal ROM (IROM), external ROM (XROM), external RAM (XRAM), etc. The allocation is defined by a locator configuration file. As the Koala compiler cannot access this file, it is more difficult to account for the contribution to a particular memory region. Modification of this file would require complete recalculation of memory consumption for different regions.

2.3 Assumptions

The following assumptions were made:

1. Function binding is ignored.
2. Compiler options are not changed, i.e. compiler optimizations are considered to be fixed.
3. Platform dependencies are not accounted for.
4. The distribution of memory types according to the locator configuration is fixed.

3. Memory consumption model

This section introduces mathematical basis for the approach and describes its implementation within the Koala framework.

3.1 Analytical expression

In general, the size of component code and static data can be calculated by the following formula:

$$\mathrm{size}\left(c,\bar{E}\right)=\sum_{i\in \mathrm{sub}(c)}\mathrm{size}\left(i,F_i\left(\bar{E}\right)\right)+ \\ +\sum_{m\in \mathrm{mod}(c)\wedge \mathrm{reachable}(m,\bar{E})}\mathrm{size}\left(m,\bar{E}\right)+\sum_{s\in \mathrm{rtsw}\left(c,\bar{E}\right)}\mathrm{size}(s,\bar{E}), \quad (3.1)$$

where $\bar{E}$ is a set of interfaces[2] of component *c;* $\bar{E}_i$ is a set of interfaces of sub-component *i;* $F_i : E \to E_i$ is the function that specifies how the interfaces of sub-component *i* are bound within component *c* (this also includes mapping of the diversity parameters of component *c* onto the ones of sub-component *i*); $\mathrm{sub}(c)$ is the set of all the sub-components of *c;* $\mathrm{mod}(c)$ is the set of all the modules[3] of *c;* $\mathrm{reachable}\left(m,\bar{E}\right)$ is a predicate indicating if the module *m* of the component *c* is reachable; $\mathrm{rtsw}\left(c,\bar{E}\right)$ is the set of all the run-time switches[4] of a component *c;* $\mathrm{size}\left(x,\bar{E}\right)$ is the function that calculates the size of a (sub)-component *x*, module *x*, or run-time switch *x*, taking into account the interfaces $\bar{E}$; *i* denotes a sub-component *i* of component *c*, and *m* denotes a module *m* of the component *c*.

Note that formula (3.1) holds both for code and static data size.

3.2 Specification

This section describes a method for the specification of memory demand for a component.

We introduce an auxiliary *provides* interface *IResource* specifying memory consumption of a component (Figure 1). The members of this interface correspond to particular types of memory. Each component is attached with a formula for the estimating the memory size of each type. This formula employs constants, expressions related to Koala features (e.g. diversity parameters), and arithmetic operations.

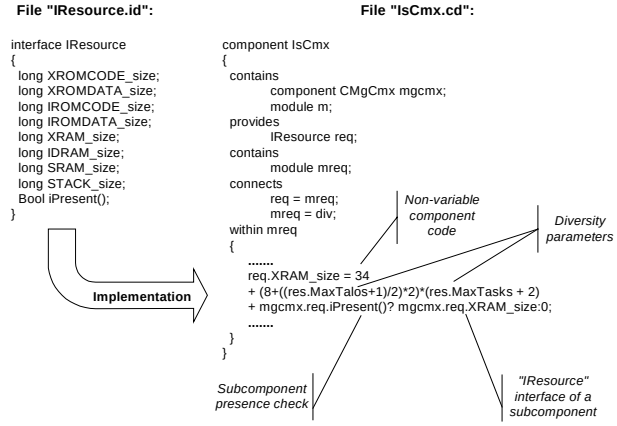


Figure 1. Example of "IResource" interface.

---

[2] $\bar{E}$ denotes the set of diversity, optional and *provides* interfaces. Note that actual dependence on $\bar{E}$ may involve only a subset of $\bar{E}$, e.g. only component's *provides* interfaces.

[3] A module is a code block implementing interface functions [5].

[4] A run-time switch occurs whenever a non-constant expression controls the switch. For more detail, the reader is referred to [5].

The formula for calculation of the sizes is an expression over diversity parameters, optional interface connections, and sizes (similar formulas) of the sub-components. It also can contain some constants for denoting the sizes of the inner modules.

The component "CIsCmx" (Figure 1) includes the sub-component "CMgCmx" and the module "m". The specification of external RAM (XRAM) size consists of the following parts.

1. Contribution of the module "m":

```
34 + (8 + (res.MaxTalos+1)/2)*2)*(res.MaxTasks+2)
```

This formula contains a constant part and a variable part which depends on the diversity parameters *MaxTalos* and *MaxTasks*.

2. Contribution of the sub-component "CMgCmx":

```
mgcmx.req.iPresent() ? mgcmx.req.XRAM_size : 0
```

The expression *mgcmx.req.iPresent()* indicates whether any module of "CMgCmx" is reachable. The module is reachable if the *provides* interface implemented with this module is needed for any other component. If *mgcmx.req.iPresent()* is true, then the size of "CMgCmx" is added to the size of "CIsCmx". For the component "CMgCmx" the similar interface "*req*" is specified, and *mgcmx.req.XRAM_size* provides the size of "CMgCmx" to account for in the formula for "CIsCmx".

When using this specification technique, the memory consumption estimates for component compositions can be calculated automatically by the Koala compiler.

4. Memory consumption estimation

This section describes two approaches for memory consumption estimation, based on the specification technique from the previous section. Both approaches are illustrated with the experimental results.

4.1 Two possible approaches

Three types of components can be distinguished in the Koala model [5]: (1) basic components that do not contain other components, (2) compound components that may contain other components, forming a hierarchy (see Figure 2), and (3) configurations that are top-level components without any *provides* and *requires* interfaces. The configuration is a set of components assembled together to form a product.

For estimation of the component size, two approaches were considered: (1) an *exhaustive* bottom-up approach and (2) a *selective* top-down one. These two approaches trade estimation accuracy against estimation effort.

In the *exhaustive* approach, all diversity and optional *requires* interfaces of all compound and all basic components are taken into account (see Figure 2). The component hierarchy is traversed in a bottom-up way, starting from the basic components up to ones at the defined level of the hierarchy. The formula is constructed for each component, until a formula for the entire configuration is determined.

The *selective* approach deals only with diversity and optional interfaces of the compound components located at some fixed level of the composition hierarchy (see Figure 2, e.g. only components C1, C2, and C3). If it is impossible to obtain a sufficiently accurate formula at this level, then also the sub-components need analyzing and constructing formulas for them. The considered degree of nesting should be as deep as necessary for achieving a sufficiently accurate formula.

The formula for the top-level component is a sum of the formulas of its constituents. To define formulas depending on the diversity parameters and optional interfaces, the investigated component is wrapped with an auxiliary configuration. The formulas can be built in an empiric stepwise way: their extrapolations are obtained by sequential compiling of the wrapper with various values of diversity parameters and different sets of connected optional interfaces. All relevant components[5] contained in the top-level one also need wrapping to construct their own formulas. Building of the formulas can be facilitated by code observation (e.g. when a diversity parameter defines the size of an array).

Note that both approaches support budgeting, i.e. the expected memory demands of non-existing components can be involved into a formula.

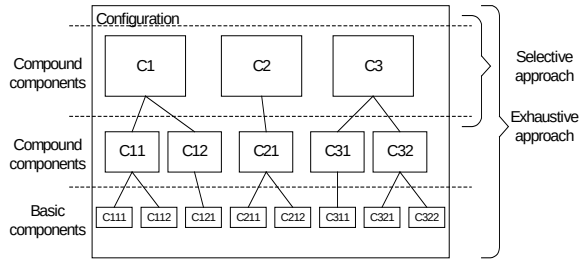


Figure 2. Approaches at different levels of component hierarchy.

The main differences between these approaches are the estimation accuracy and annotation effort.

The exhaustive approach ensures the required level of accuracy for the entire composition if all components are annotated with sufficiently accurate formulas (in the general case). However, this implies huge amount of effort.

The selective approach may not ensure the defined level of accuracy. Achieving the appropriate level of accuracy may require analysis of deeper levels of the component hierarchy, while considering only selected components may reduce the amount of effort needed for annotation.

[5] For the exhaustive approach, all components are relevant.

4.2 Estimation examples

To demonstrate both memory estimation approaches, we applied them to two different component configurations taken from the existing software stack for TV sets. The first configuration consisted of seven components was used for checking the exhaustive approach, while the second one consisted of 22 components was used for checking the selective approach.

The software stack for the case study was implemented for a 16-bit derivative of the popular Intel 8051 micro-controller. This micro-controller differentiates several types of memory. For each type of memory, the estimates were compared with the actual sizes, considering different sets of diversity parameters and connections of different optional requires interfaces (see Table 1 and Table 2).

Table 1. Estimates and actual sizes for exhaustive approach.

Type of memory	Real size (bytes)	Estimated size (bytes)	Relative error (%)
XROM Data	166	166	0,00
XROM Code	19429	19477	0,25
IROM Code	3363	3425	1,80
IROM Data	379	379	0,00
IDRAM	572	572	0,00
SRAM	145	145	0,00
XRAM	2123	2123	0,00

Table 2. Estimates and actual sizes for selective approach.

Type of memory	Real size (bytes)	Estimated size (bytes)	Relative error (%)
XROM Data	12379	12479	0,81
XROM Code	70996	71409	0,58
IROM Code	21353	21401	0,20
IROM Data	1705	1703	0,12
IDRAM	796	796	0,00
SRAM	544	544	0,00
XRAM	84471	84607	0,15

5. Conclusions

We have proposed a method that allows estimating the memory consumption for Koala component compositions. The proposed method is illustrated with examples taken from the existing software stack.

We have described the mechanism for specification of the component memory demand for code and static data. This mechanism employs standard constructions of the Koala component definition language.

The suggested specification mechanism is compositional and hierarchical: the memory demands of a compound component are specified in terms of memory requirements of its constituents, and each component can be used in another context without changing the specification of its memory consumption. When using this technique, the memory consumption estimates for component compositions can be calculated automatically by the Koala compiler.

This mechanism also supports budgeting; i.e. the expected sizes of the components being developed can be incorporated into the specification.

Two approaches for the estimation were proposed: *exhaustive* and *selective*. Each approach was validated with a case study. High estimation accuracy can be achieved for both approaches.

Further research will be directed towards additional validation and generalization of the proposed technique. Firstly, the thorough validation of the approach with more experiments will be performed. Secondly, the possibility to generalize and apply this approach to other component models will be considered. Finally, the different ways to specify the memory consumption and other resource attributes (particularly, in XML-based description language) will be investigated.

6. Acknowledgements

We are grateful to Rob van Ommering and Chritiene Aarts for a valuable contribution in this work. We also thank Marc Stroucken for constructive suggestions and critical feedback.

7. References

[1] M.R.V. Chaudron, E.M. Eskenazi, A.V. Fioukov, D.K. Hammer. A Framework for Formal Component-Based Software Architecting. In Proceedings of Specification and Verification of Component-Based Systems Workshop, OOPSLA Conference 2001, Tampa, USA, October 2001.

[2] D. K. Hammer and M.R.V. Chaudron, Component Models for Resource-Constraint Systems: What are the Needs?, Proc. 6th Int. Workshop on Object-Oriented Real-Time Dependable Systems (WORDS), Rome, January 2001.

[3] D.K. Hammer, Component-based architecting for distributed real-time systems: How to achieve composability?, in Mehmet Aksit (ed.), Software Architectures and Component Technology, Kluwer, 2002

[4] G.T. Leavens, M. Sitaraman, Foundations of component-based systems, Cambridge University Press, 2000.

[5] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee, The Koala Component Model for Consumer Electronics Software. Computer 33, 3 (2000), pp 33-85, 2000.

[6] C. Szyperski, Component Software: Beyond Object-Oriented Programming, Addison-Wesley, 1998.

The Representation of Component Semantics: A Feature-Oriented Approach

Yu Jia† Yuqing Gu‡
Institute of Software, Chinese Academy of Science
Beijing(100080), China.
Email: †jia_yu@263.net ‡ guyq@sinosoftgroup.com

Abstract: *In this paper a semantic model for component is proposed which is structured in three parts called Domain Space, Definition Space and Context Space. We also argue that the feature-oriented method is an effective and practical approach to fulfill the semantic model.*
Keywords: *CBD, Semantics, Feature-Oriented*

1. Introduction

In CBD, one of the most critical issues is how to evaluate the reusability of a component [1, 2]. Generally speaking, the reusability of components comprises of two aspects - the syntax and the semantics. Recently, the amount of efforts focuses on the syntactic reusability to achieve connection other than the semantic reusability to describe the function and extra-function of components. For instance, CORBA CCM [5] cannot well exhibit the services it provides. We think such a drawback will greatly hinder the engineering practice in CBD.

Different from the traditional developing paradigms, in CBD the user requirements (viz. problem domain) need to be directly matched to the third party components (viz. solution domain), which reduces some traditional developing phrases for increasing efficiency and quality. Ideally, if there exists a semantic representation approach which has an invariable form in both the problem domain and the solution domain, the difficulties in model converting activities via semantics (including components retrieval, adaptation, composition etc.) will be overcome radically. In fact, the Feature-Oriented Reuse Method (FORM) [4] is regarded as such a satisfying and promising approach [9].

In FORM a logical entity called *feature model* is suggested to "develop domain architectures and components." However, we think that the feature model can be a good means to describe the component semantics if properly analyzed and designed. So this paper is to use and extend FORM addressing component semantics. We name our approach as *FORM for Component Semantic* (FORM/CS).

The kernel of FORM/CS is the theory of two models, the semantic model of component, which is composed of domain, definition and context; and the feature model, which is well organized into a hierarchy called *Feature Space*. This paper is arranged as follows: Section 2 and 3 discuss the component model and the semantics model respectively. Section 4 is to define the feature-oriented component semantics by Feature Space. Finally, a semantic stream in the CBD process is illustrated in principle of FORM/CS.

2. The Conceptual Model of Component Semantics

In the research community of software reuse, "3C model" [2, 8] is a generally accepted reusable component model, which separates the component into three distinct facets as *concept*, *content*, and *context*. However, the 3C model is not formally and well defined. Hence, below a comprehensive and concrete 3C model is represented in notation of the formal specification method - language Z [7]; then the semantic model of component is given based on it.

DEFINITION 2.1 (*Component Model*) A component is a identifiable software unit in an explicit context with contractually specified semantic interfaces that are reasonable in a domain as well as syntactic interfaces that are supported by component frameworks.

$$Component == CONCEPT \times CONTENT \times CONTEXT \quad (2.1)$$
$$CONCEPT == SEM\_INTERFACE \times SYN\_INTERFACE \quad (2.2)$$
$$CONTENT == IMPLEMENTATION \quad (2.3)$$
$$CONTEXT == SEM\_CONFIG \times SYN\_CONFIG \times DOMAIN \quad (2.4)$$

Where, (2.1) *CONCEPT* : "Abstract functionality that the component provides."; *CONTENT* : "The implementation of that abstraction."; *CONTEXT* : "What is needed to complete the definition of a concept or content within a certain environment." [8]

(2.2) *SEM_INTERFACE*: The set of semantic interfaces that describe the functional and extra-functional properties of components; *SYN_INTERFACE*: The set of

connecting interfaces that are programmed in frameworks such as CORBA IDL, COM IDL or EJB;

(2.3) *IMPLEMENTATION*: The set of executable code units.

(2.4) *SEM_ CONFIG*: The variability of *SEM_INTERFACE* depending on context; *SYN_CONFIG*: The variables of *SYN_INTERFACE* determined by context; *DOMAIN*: The specific application domain in which the concept is defined.

The component model DEFINITION 2.1 is compatible with the industrial standard (referring to *SYN_INTERFACE*); and also the *DOMAIN* consists with FORM to create the Domain-Specific Software Architecture (DSSA). In fact, the standard DSSA is shared by all components. So, *DOMAIN* is separated from component as a standard concept set. When describing component semantics, it is necessary to refer the *DOMAIN* it belongs to.

DEFINITION 2.2 (*Semantic Model*) The component *semantics* is the meaning and use of components in perspective of domain-specific service in the real world.

$$Semantics == DOMIAN \times SEM\_INTERFACE \times SEM\_CONFIG$$

In DEFINITION 2.2 the component semantics is appropriately constructed not only naturally deriving from the Component Model but also deliberately considering the reasonability, completeness and feasibility of semantic representation. *SEM_INTERFACE* is the set of service instances contracting between the providers and the consumers of the component. All concepts in the contracts are defined in *DOMAIN* with some context-dependent parameters presenting in *SEM_CONFIG*. The emphasis of "real world" indicates that the semantics should be captured directly by business meanings.

3. The Feature-Oriented Component Semantics

3.1 Feature

The *feature* is not a novel concept in computer science. Many fields (e.g. Pattern Recognition, CAD ect.) use feature-like concepts in similar methodologies to solve different problems.

DEFINITION 3.1 (*Feature*) Features are the constructing units of component semantics, as well as the ontology of domain knowledge in real world.

$$Feature \cong [\ id : IDENTIFIER;\ interpret : IDENTIFIER \leftrightarrow \mathbb{P} ONTOLOGY \mid \forall id_1, id_2 : IDENTIFIER \bullet interpret(id_1) = interpret(id_2) \Rightarrow id_1 = id_2\ ]$$

Where, *IDENTIFIER*: The set of feature names; *ONTOLOGY*: The set of domain knowledge to describe the feature, mostly expressing in natural languages; *interpret*: A function to map the *id* to the meaning of features in the natural language.

A *feature item* is the instance of a feature normally belonging to the basic data types (e.g. integer, string, boolean etc). Given a set *FEATUREITEM* of feature items and a function: $instance : Feature \rightarrow FEATUREITE\ M$, a Feature is *satisfiable* if following proposition is true: $\exists x : FEATUREITEM \bullet x = instance(Feature)$.

3.2 Feature Space

DEFINITION 3.1 (*Feature Space*) A Feature Space Ω is the architecture of component semantics formed by features *fea* and feature relations *rel*.

$$\Omega \cong [\ fea : Feature; rel : Feature \times Feature\ ]$$

We have identified four general relation types underlying the Feature Space as follows: *Aggregation relationship*, *Generalization relationship*, *Dependency relationship* and *Association relationship*

Although the universal feature relations far exceed the expressive ability of AND/OR graph, it might as well informally depicting Ω as tree structure called *feature tree* to make use of the visual property of tree. Considering hierarchical structure formed by aggregation or generalization relationship, when discarding dependency and association relationships between features within the same level (viz. *horizontal* relations), Ω is the tree with the *Feature* data structure as its nodes. The leaves denote *atomic features* while non-leaves denote *compound features*. The root of the tree is called *root feature*.

The instantiation of Ω is defined as:

$$\omega \cong [\ f : FEATUREITEM; r : FEATUREITEM \times FEATUREITEM \mid instance^{\sim}(f) \in \Omega \wedge x\ r\ y \Rightarrow instance^{\sim}(x)\ \ \Omega.rel\ \ instance^{\sim}(y)\ ]$$

EXAMPLE 3.1 A university courses arrangement (see Figure 1). The Feature Space is defined as follows:

Features:

$UDC == \{University, Departments, Courses\}$

Relations:

$includes == (University, Departments)$

$provides == (Departments, Courses)$

Instance functions:

$Univ == \{(University, "xyz")\}$

$Dept == \{(Departments, "Language"), (Departments, "Computer")\}$

$Crs == \{(Courses, "Mathematics"), (Courses, "Chinese"), (Courses, "Database")\}$

Feature items:

$UDCItem == \{("xyz", "Language", "Computer", "Mathematics", "Chinese", "Database")\}$

Relation instances:

includes == {("*xyz*","*Language*"),("*xyz*","*Computer*")}
provides == {("*Language*","*Chinese*"),("*Language*",
"*Mathematics*"), ("*Computer*","*Database*"),("*Computer*",
"*Mathematics*")}

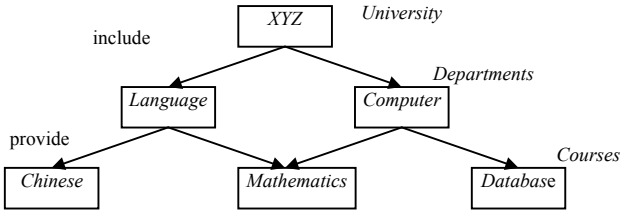


Figure 1 The Feature Space in Tree

4. The Feature-Oriented Representation of Component Semantics

In this section a concrete semantic model will be discussed in principle of FORM/CS. In order to distinct it from the general model, we rewrite the form of DEFINITION 2.2 to be what is called d^2c semantic model:

$$d^2c == \Omega_{dom} \times \Omega_{def} \times \Omega_{con}$$

Where Domain Space $\Omega_{dom} == DOMAIN$, Definition Space $\Omega_{def} == SEM\_INTERFACE$ and Context Space $\Omega_{con} == SEM\_CONFIG$.

Domain Space is the product of Domain Engineering, which represents the commonality and variability in Feature Space to specify the DSSA of the software families.

DEFINITION 4.1 (Domain Space) The Domain Space Ω_{dom} is a sound and complete Feature Space that expresses the knowledge for a specific domain.

Definition Space specifies the semantics for an individual component. FORM/CS is a kind of descriptive semantics, which declares the intension of the functional and extra-functional properties of a component without concerning the implementation and the state transition.

DEFINITION 4.2 (Definition Space) The Definition Space Ω_{def} is an instance set of Ω_{dom} that expresses the service provided by a component.

The component semantics is possibly influenced by the context when an individual component is integrated into an application. The Context Space is what expresses the variability of an individual component when adapting to the context.

DEFINITION 4.3 (Context Space) The Context Space Ω_{con} is a collection of configurable features and feature relations that represent the variable parts of the component semantics. They are set by context.

Considered only semantic analysis, the process of CBD is a serial of operations to compose, decompose and modify the Feature Spaces [6]. Figure 2 (on end of next page) illustrates a simplified CBD process framework that shows the semantic stream.

A. Domain Analysis: All activities in Domain Engineering to create a Ω_{dom} for a specific domain. Ω_{dom} is stored into a Component Depository as a widely accepted standard.
B. Component Development: Either legacy or new software is appropriately wrapped with semantic interface according to the industrial standards. The semantic interface comprises the fixing function/extra-function as Ω_{def} and variable as Ω_{com}. The certified components are also stored into component repository.
C. Retrieval: Requirements in form of Feature Space are used to match the components in repository according to the semantic similarity. A group of available component candidates is acquired. In retrieving the reasoning techniques are utilized.
D. Evaluation: Various aspects are considered including technical or non-technical factors. After analyzing and evaluating every candidate, a most suitable component is selected.
E. Adaptation: The satisfied component are modified, that is, the Ω_{def} is changed according to specific requirements and the context parameters are attached to Ω_{com}. Note that the Ω_{con} is instantiated to be ω_{con} and after adapting, the Ω_{def} may be changed.
F. Composition: The qualified component is integrated into an application. The semantic consistency should be checked in the application via some reasoning techniques.
G. Running: The component runs in the application. There is no change in semantics.
H. Evolution: The practice-tested component is wrapped again according to industrial standards, and recycles into the component repository. Note the Ω_{con} is abstracted and separated again for changeable parts of the component.

5. Conclusions

The radical source of difficulty in component reuse may be the comprehension gap between the component providers and consumers in different contexts [2]. FORM/CS is one of the promising methods addressing this problem in theory and practice. However, our research just begins. There still exist lots of issues for

further investigation. For example, how to obtain and select features in domain analysis; how to decrease the complexities of times and space about Feature Space.

References

[1] Martin Blom, Eivind J. Nordby. "Semantic Integrity in Component Based Development". Project Report, Mälardalen University, Sweden, March 2000.

[2] Stephen H. Edwards. "Toward A Model of Reusable Software Subsystems". In: Steve Philbrick and Mark Stevens, eds. *Proceedings of the Fifth Annual Workshop on Software Reuse*, Larry Latour, Oct 1992.

[3] Yu Jia, Yuqing Gu. "Representing and Reasoning on Feature Architecture: A Description Logic Approach". *Workshop on "Feature Interaction in Composed Systems"*, ECOOP 2001.

[4] Kang, K.; Kim, S.; Lee, J.; Shin, E.; & Huh, M. "FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures". *Annals of Software Engineering* 5, 5 (September 1998): 143-168.

[5] Object Management Group (OMG). "Components FTF Edited Drafts of CORBA Core Chapters", Document Number ptc/99-10-03, URL: http://www.omg.org>, 1999.

[6] John Penix, Phillip Baraona, Perry Alexander. "Classication and Retrieval of Reusable Components Using Semantic Features". In *Proc: 10th Knowledge-Based Software Engineering Conf.*, Boston, MA: IEEE Comp. Soc Press, November 1995. 131-138,

[7] Mike Spivey. The Z Notation: A Reference Manual. *Prentice Hall International Series in Computer Science*, 2nd edition, 1992.

[8] Will Tracz. "Implementation working group summary". In: James Baldo ed. *Reuse in Practice Workshop Summary*, Alexandria, VA, April 1990:10-19

[9] C. Reid Turner, Alfonso Fuggetta, Luigi Lavazza, and Alexander L. Wolf. "A Conceptual Basis for Feature Engineering", *Journal of Systems and Software*, Vol. 49, No. 1, December 1999, pp. 3-15.

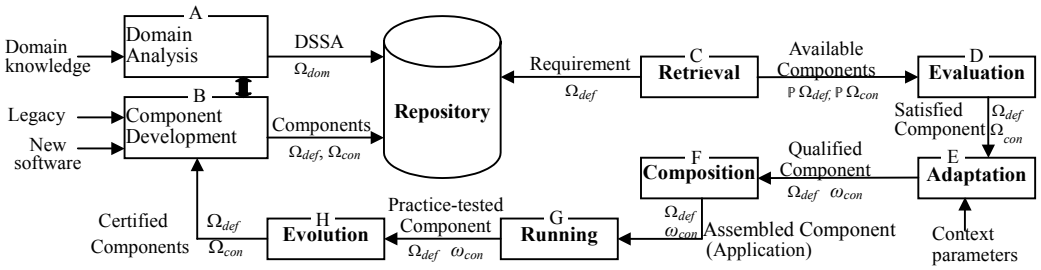


Figure 2 The Semantic Stream in CBD Process

A Component-based Environment For Distributed Configurable Applications

Ahmed Saleh
University of Westminster, UK
saleha@wmin.ac.uk

George R. Ribeiro-Justo
Cap Gemini Ernst & Young[1], UK
George.Justo@capgemini.co.uk

Stephen C. Winter
University of Westminster, UK
wintersc@wmin.ac.uk

Abstract

One of the basic requirements for distributed applications to run under different working environments is to be flexible, configurable, portable and extensible. Using the current development techniques independently falls short in supporting most of these requirements due to complexity of their integration and the conflict of their objectives. In this context this paper describes an integrated environment based on an interface description language called NCSL, an architecture description language called NADL, and a supporting management system composed of a component-based framework and an event management system that facilitate the process of developing and managing distributed configurable applications based on their non-functional requirements (NFRs).

1. Introduction

While computing power and network technology have improved dramatically over the past decade, the design and implementation of complex distributed configurable applications remain difficult and time-consuming. Also, the need for considering distributed applications' non-functional requirements (i.e. performance, reliability, security, etc.) has added further complexity to the process of developing these applications. Using traditional development techniques often result in static and difficult to understand applications that do not address the user requirements. Also, due to the evolving nature of distributed systems' environments, applications that can tolerate the continuous upgrade of such environments are often developed on a per application basis.

Component-based frameworks have emerged as the new technology that can facilitate the development of distributed applications through reusable components. As its name suggests, a component-based framework is a collection of software components that have been developed independently but can interact and collaborate with each other to support the development of a group of applications or solve a particular type of problems. Unfortunately, constructing distributed configurable applications from pre-existing reusable components of such frameworks cannot be achieved without understanding the structure and functionality of these components. Despite some successful attempts, most of the current frameworks rely on providing reusable components that can be plugged in together in different configurations to build up the applications, but are not able to tackle the problem of design reuse, where the entire structure/architecture of the application can be reused to build new applications. Furthermore, very few frameworks have addressed the problem of integrating the non-functional requirements of the application's services due to the difficulties of representing and controlling such requirements at run-time.

This paper describes an integrated environment for supporting the development and control of distributed configurable applications through a collection of distributed components that collaborate within a specific configuration to satisfy both the developer and environment requirements. This environment is based on a framework of distributed reusable components called FRODICA (Framework for Distributed Configurable Applications). Each constituent component of the framework should have a well-defined interface that has been defined by the NCSL language (Non-functional Component Specification Language) that describes the components' functional and non-functional requirements to enable their interaction regardless to their implementation details. The components' interaction and the configuration itself is defined by an architecture description language called NADL (Non-functional Architecture description Language), which defines the architectural structure of the application and its run-time constraints, and the rules of selecting/integrating different components according to the application's NFRs,.

2. Related Work

Many researchers have investigated the development of component-based frameworks to support the construction of configurable applications in a distributed context. For example, C++CL [1] is an OO (object-oriented) framework for developing reconfigurable distributed systems. It is based on the CL model where an application is divided into two sets of components: tasks and configurations. The computation is usually performed by

[1] The views and conclusions contained in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied of Cap Gemini Ernst & Young.

tasks that can interact with each other via local ports. The configuration is the part of the program where the system structure is specified and controlled. This consists of defining task instances, connecting them and managing their execution. C++CL is considered as a real attempt to create an object-oriented framework for developing dynamic distributed software architectures. However, it does not support the definition of NFRs at any stage of the development process.

The Aster project is another attempt based on matching the NFRs of an application with the NFRs of selected components and connectors manipulated by the Aster framework [3]. This matching process results in generating a customized middleware that provides the NFRs of the application. Although the Aster framework proved to be efficient in implementing several transactional and non-functional properties, it does not cover all concepts of software architecture (e.g. connectors, ports, etc.) only components and some basic connectors are supported. In addition, it does not address the problem of managing NFRs during run time.

Unlike Aster, the QuO (Quality Objects) framework [4] is an integrated environment for developing distributed applications with QoS requirements. Its main idea is based on the notion of *contracts*, *delegates* and *system condition objects* that negotiate an acceptable region of QoS prior establishing a connection between a client and a server. When both client and server agree upon a specific region, the connection is established and the QoS level is monitored for further developments. Although QuO offers more flexibility than other frameworks, it depends heavily on CORBA IDL to provide its code generator with the appropriate interface, ORB proxy and ORB before generating the executable code of the system. In addition, QuO only concentrates on the structure of the components and their QoS, but does not address the global architecture of the application and its NFRs.

3. The FRODICA Framework

As mentioned in the introduction, the key point to facilitate the development of new applications from pre-existing reusable components is to understand the structure of these components and how they interact. Taking this into consideration, FRODICA [6] has been developed as a four-tier framework that can reside above the operating system and below the application layer. The layering approach adopted by FRODICA categorises the components into four separate layers according to their functionality and complexity. In this context, components of top layers can extend/customise the functionality of the corresponding lower-layer components in order to tailor the topmost-layer components to suit individual distributed applications.

The *communication layer* of FRODICA is the lowest layer of the framework, which is responsible for handling the low-level communication protocols of the system. This layer is mainly concerned with carrying out all the underlying message passing, naming services, binding and data marshalling between distributed components. Accordingly, this layer comprises all platform-dependent software (i.e. libraries and interfaces) required to perform such communications.

The *general-purpose layer* is the middleware layer of the framework that deals with low-level system operations. The main objective of this layer is to hide the platform-specific software and hardware complexity from upper layers, hence provide platform independent environment for system developers to create their applications. This layer acts as the bridge between the application layer and the underlying technology infrastructure. It accommodates a number of management and general-purpose components that provide the basic requirements to build distributed configurable applications.

The *application-oriented layer* is concerned with putting together all the standard services required for supporting the development of an integrated distributed configurable application. Components of this layer are extensively used by system developers in creating their applications, and therefore, they tend to provide the most basic services for developing distributed applications, together with a well-defined interfaces and a clear extensibility methods to enable their use without exploring the complexity of lower layers' components.

Finally, the *specific-application layer* is the topmost layer that comprises the components, connectors and interfaces needed for running a specific application. In this layer, system developers can create their own new components, extend or specialise lower layers' components to build their applications.

4. NCSL Language

The NCSL is a component specification language based on Java. It provides a set of tools for the description and deployment of distributed components, taking into consideration the restrictions and constraints (i.e. non-functional requirements) imposed by the system/developer on these components' services. At the design stage, components are described with the help of a configuration language that defines the internal specifications of each component in terms of the services provided/required by the component, as well as the non-functional requirements associated with each service. The compilation of NCSL into the framework implementation language is achieved via a separate compiler called `NcslToJava`, which examines the validity of the component's interface description and generates an

executable code in the form of Java and XML files. Subsequently, the generated interface will be used by the NADL language (explained at the next section) to identify components' functional and non-functional properties required for configuring distributed applications at run time.

NCSL currently supports three types of non-functional attributes:

- Performance: The performance is defined in terms of average time (measured in millisec) to perform a service.
- Reliability: The reliability is measured in terms of the MTTF (mean time to failures).
- Availability: The availability is measured in terms of the average time to restore (MTTR—mean time to restore) a service after a failure. It is a function of MTTF and MTTR.

The above words are regarded as keywords in NCSL. NCSL also provides the concept of NFR expressions that are Boolean and conditional expressions combining non-functional attribute keywords and their values. For example, a service is required to provide a 'performance = 500 Kb/sec and reliability > 500 mesc'. An example of NCSL illustrated in Figure 1.

```
interface GoldBranch {
              // provided services //
   provide float checkBalance (int customerID,
                               int customer PIN);
       support { performance &&  // supported NFRs
                 availability } ;
              // required services //
   require float getBalance (int customerID,
                         string customer name) ;
  with {performance >= 500 Kb/sec && // required NFRs
            availability >= 500 mesc } ;
   ……
```

Figure 1: The NCSL specifications for a Bank component

To reduce overheads, a component is not required to compute all non-functional attributes, when they are not related to any NFR, but only those critical ones. In this case, NCSL contains a 'support' clause that indicates which non-functional attributes are computed by the component. Remember that the interface corresponds to a contract, therefore if a component supports a non-functional attribute, as described in more details later, the environment and an ADL (Architecture Description Language) script can query the value of that non-functional attribute at runtime.

NCSL adopts the same concepts of ACME (An Architecture Description Interchange Language) [2] in assigning general non-structural information to each architectural entity (i.e. component, connector, port, etc.) to describe its run time behaviour. However, NCSL goes further by defining a set of s to each service supported/required by each one of these entities.

5. NADL Language

Current ADLs allow system developers to integrate heterogeneous software components in a homogeneous way, define and locate distributed components across the network, and adapt their behaviour according to their design preferences. This kind of features is described as the functional requirements of the system. Most ADLs fall short, however, in providing support for the NFRs of the system, which describe its constraints and run-time behaviour. This is due to the fact that they hide the details necessary to specify, measure and control such requirements, and hence provide little support for building systems that can adapt to different levels of QoS.

Incorporating NFRs in the design of the system requires the ADL to specify constraints for the QoS properties of the required and provided services of each component. Also, it requires matching techniques for determining whether a service satisfies non-functional requirements and what are the consequences if a component fails to satisfy the desired non-functional requirements.

As a language that supports the description of re-configurable distributed system according to both their functional and non-functional properties, NADL provides special constructs to deal with NFR description and management. An NADL description (Fig 2) is made of two main sections: a configuration section where components are selected according to their services and their NFRs, and a reconfiguration section where reconfiguration actions are taken, depending on the failure or changes of NFRs. NADL also allows the system developer to define environment specific properties that must be satisfied by all components running the application. For example, a component must run within a specific type of operating system or over a machine with certain memory specifications. These properties enable the system developer to refine his selection to identify components that are more specific.

The key constructor of NADL is the concept of NFR expressions that are extensions of those used in NCSL. In NADL, NFR expressions may contain services from different components while in NCSL they refer to the NF attributes of a specific service. For example, the expression below defines that the service *video* provided by component *comp1* should support availability above 500 msec and at the same time, the *sound* provided by component comp2 should perform above 900 Kb/sec:

```
comp1.video.availability >500 msec
&& comp2.sound.performance >900 Kb/sec
```

The NADL selection of components is based on their interfaces, which already specify their NFRs. After the system identifies possible candidates components, the

configuration can be defined. In general, the selection should be the minimum requirement of the system. During the configuration, it is then possible to define further constraints depending on the candidate components that have been selected. In addition, the architect can specify global constraints relating the various components.

The configuration is built by using the typical ADL constructs such as connect, and start. Observe that NADL also uses the concept of default connectors, which are implemented by the supporting middleware. For instance, in the case of Java components communicating using RMI (Remote Method Invocation), it is possible to connect the components directly by using an RMIConnector default connector. After the configuration has been successfully built, the reconfiguration section specifies conditions for monitoring and managing the configuration. This is done using *when* clauses similar to those used during the configuration. The *when* clauses are evaluated sequentially and the first one that satisfies the corresponding reconfiguration block is triggered. During the reconfiguration, components and connectors can be connected or disconnected, and new components and connectors can be selected to satisfy the architecture NFRs.

```
Application : Bank {
 select {
  component: Comp1 { interface: MainBank ;
   location: remote (osiris.cpc.wmin.ac.uk) ;
   properties: { (getBalance.performance >= 500kb/s ||
         checkBalance.availability  >= 5000 msec };};
 connector: Conn1 { interface: GoldConnector ;
  properties: {dataStream.availability >= 800 msec &&
               dataStream.reliability > 750 msec };
                  // End Properties //
           } ; } ;    // End Conn1 // End select //
 constraints: { Comp1.Performance >= 4000 Kb/sec ;
       propertiesCheckupRate >= 4000; }; // Rate of
                              //checking NFRs in msec//
 implementation: {Bank.Platform = java; //App platform
       Bank.OS = Unix;};//OS for running the app //
 configuration: { conf1: when (select) ;
   do (connect Comp1.getBalance To Conn1.dataStream;
      connect Comp3.withdrawCash To Conn2.dataStream);
conf2: when (Comp3.checkBalance.availability <600 ms);
   do (wait (3000);
        reselect;) } ; // Repeat 'select' process
 reconfiguration: {
   when (Comp1.getBalance.performance <5000 Kb/sec ||
         Comp1.getBalance.availability < 5000 msec);
     do ( start ;
            suspend ;
            stop Comp3.checkBalance ;
            stop Comp3.withdrawCash ;
            resume ;
            end) ;
    }; } // End reconfiguration // End Application //
```

Figure 2: The NADL specifications for a Banking Application

NADL also provides the concept of (global) constraints, which define an NFR invariant for the architecture. The constraint is revaluated after every reconfiguration. Observe that, since NADL is service-driven, reconfiguration is carried out at service level, which means that during reconfiguration the whole component is not affected but only those services involved. Further more, component instances offering a service may run longer than a particular application. This means that existing component instances can be shared by different configurations. The architect may decide whether to use a fresh instance of a service or an existing service.

6. Conclusion

The environment outlined in this paper showed how possible it is to extend existing IDLs and ADLs to support the management of NFRs. It has also demonstrated importance of considering the distributed applications' NFRs at the early stages of the design in order to ease their management and control at run-time. Although, we have decided to build our own management service but there is no reason why the management system could not use services of a middleware such as [5], which supports QoS management. We see these two technologies as complementary rather than competing.

Also, the environment outlined in this paper showed that the combination of software architecture with object-oriented frameworks and language mechanisms can lead to the development of a new generation of well-structured distributed applications that can be easily configured to adapt with different working environments.

7. Reference

1. Justo, G. R. R. and Cunha, P.R.F.: "*An Architectural Application Framework for Evolving Distributed Systems*", Journal of Systems Architecture, Special Issues on New Trends in Programming and Execution Models for Parallel Architectures, Heterogeneously Distributed Systems and Mobile Computing, Vol. 45, No. 15, Sep. 1999.
2. Garlan, D., Monroe, R. and Wile, D.: "Acme: An Architecture Description Interchange Language". Proceedings of CASCON, Nov. 1997.
3. Issarny, V. and Bidan, C.: Aster: A CORBA-Based Software Interconnection System Supporting Distributed System Customization. *In Proceedings of the 3rd International Conference on Configurable Distributed Systems (ICCDS'96).* Mayland, USA, May 1996.
4. Loyall, J., Bakken, D., Schantz, R., Zinky, J., Vanegas, R., and Anderson, K.: "QoS Aspect languages and Their Run-time Integration". Proceedings of the 4th Workshop on Languages, Compilers and Run-time Systems for Scalable computers (LCR), Pennsylvania, USA, 1998.
5. Koh, F. and Yamane, T.: Dynamic resource management and automatic configuration of distributed component system. In *Proceedings of the 6th USENIX COOTS*, Jan 01.
6. A. Saleh and G. R. Ribeiro Justo. A configuration-oriented framework for distributed multimedia applications. *In Proceedings of the Fifteenth Symposium on Applied Computing (SAC200)* Italy, ACM Press, March 2000.

Quality of Service Specification in Dynamically Replaceable Component Based Systems

Dr. Ian Oliver
Nokia Research Center
Itämerenkatu 11-13
Helsinki, Finland
ian.oliver@nokia.com

Abstract

When working with embedded environments that can automatically download components on as as-needed basis it is necessary to ensure that we do not place too much stress (CPU overload, Memory overload etc) on the system in order to achieve optimal performance for the user.

In order to facilitate this one must incorporate quality of service information into the components and perform suitable tests upon this information in order to decide whether to download the component or not. One issue here is how is this information presented, stored and what information should be carried by the component. There are also issues with what the information means and from where it is collected.

In this position paper we describe our initial efforts in specifying the quality of service information and also explore some of the implementation issues we have found.

1 Introduction

This paper describes an approach that we are investigating for the management of quality of service parameters in downloadable component based systems.

We have implemented the ideas expressed in [1] by way of specifying the component characteristics in a textual form and then using various analysis techniques to test this data against current system performance within a given component framework.

2 Architecture Overview

We use an architecture where a system may use a number of components providing specialist facilities, eg: Video CODECs, in an on-demand environment. A limiting factor is that in some cases the platform on which we may be working is of limited processing power and other resources, eg: memory. In this case it is necessary to ensure that the system can accept any given component without compromising the current level of service.

At the highest level of abstraction the quality of service framework consists of three main components:

- Admission Test
- Quality of Service Manager (QoSManager)
- Resource Manager

The QoSManager may have a number of admission tests and resource manager components associated with it.

The QoSManager has the responsibility for interacting with the component to be downloaded, the admission

test and the resource managers. It also makes the decision whether a component is downloaded or not by performing a decision based upon the results of the admission tests. The level of sophistication of the QoS-Manager may vary depending upon the system from a simple Yes/No decision test to one that is capable of load balancing and optimising the system performance.

The admission tests are responsible for processing particular sets of QoS data. For example one may have an admission test for memory usage where the current memory consumption is checked against what is required. Also possible is a more complex test for CPU utilisation in real-time systems based upon rate monotonic analysis [2]. We currenly have three return values for admission tests: Yes, No and Unsure - the latter relating to situations where the test produces an inconclusive result, eg: a CPU utilisations above the RMA utilisation bound but less than 100% utilisation.

The resource monitors are responsible for collecting data about particular aspects of the system, for example, memory usage, CPU usage etc. Again the sophistication of the resource monitors may vary depending upon the needs and capabilities of the system.

3 Component Download Overview

A Component provides a set of functionality, for example a video player for a certain CODEC. Each component contains a Quality of Service Specification detailing what resources that component requires the levels of performance that the component needs and the requirements (eg: memory, CPU etc) for those levels of performance.

When a component download is initiated it is necessary to ascertain whether there are enough system resources available to execute that component with a given level of quality of service. For example in case of a video player the component would specify a quality of service that makes watching video tolerable to the user, for example 1-5 frames per second would be considered the absolute minimum, while 25 or more frames per second be considered ideal.

This information is initially downloaded and the system through some form of quality of service mechanism. This mechanism would then analyse that data with respect the the current (and average) system performance and the currently loaded components.

Given a suitable outcome from this analysis the component would then be downloaded. Obviously if the outcome is negative then the component download will be refused. However we can consider a third situation where the analysis is incomplete or inconclusive. How we proceed in these situations would be dependent upon the sophistication of the quality of service mechanism.

The general download situation can be seen in the UML sequence diagram [3] in figure 1 and described below:

1. Request from the component the quality of service specification
2. Obtain current system performance information from the relevant resource monitoring facilities
3. Analyse the components of the quality of service specification and return a result based upon that analysis.
4. The analysis machines perform some calculation and return a result based upon that calculation
5. A decision for download is made upon those results:
   (a) If the decision is Yes then component download proceeds
   (b) If the decision is No or Unsure then the next quality level will be read. If no new quality level is available then the component download is terminated.

Depending on the sophistication of the decision algorithms, it is possible that the system may to alter the performance levels of existing components in the system to accommodate the new component at the highest performance level possible. One heuristic we

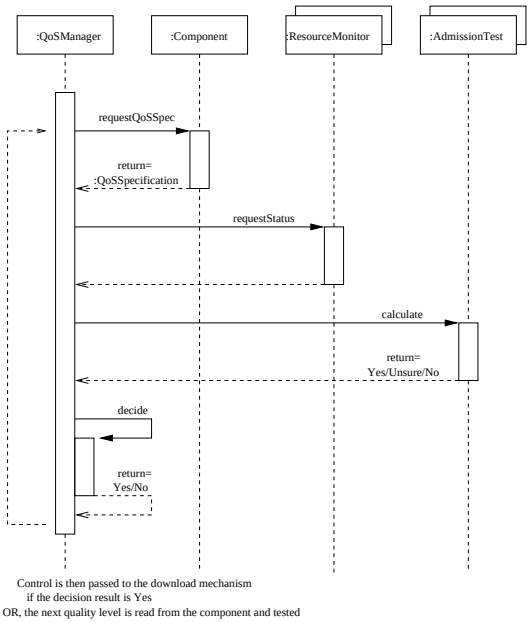


Figure 1. Component and Quality of Service Subsystem Interactions

have applied here is that the latest component to be downloaded probably has the highest priority from the user's perspective. One may then over time attempt to equalise the service levels of all the components in order to balance the performance of the components running in the system.

4 Quality of Service Specification

To afford interoperability between components and the QoS system we have used - and are developing further - a standardised way of communicating the service level information between the individual subsystems. Currently this is implemented using simple text strings. A model of the QoS specification can be seen in the class diagram in figure 2

A QoS Specification is made up of a number of individual levels - level one being the highest quality and lesser quality levels follow henceforth. Within each level we define the quality parameters for certain criteria.

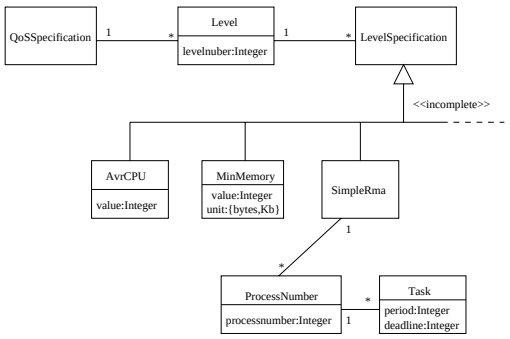


Figure 2. Quality of Service Specification

We must also apply a number of rules to this specification, primarily stating that the level numbers must start at one and increase by values of one, ie: 1,2,3...

Each individual criteria may have its own consistency rules, for example we may have the rule[1]:

context AveCPU
inv: **self**.$value \geq 1$ **and self**.$value \leq 100$

Again similarly for the MinMemory resource specification and correspondingly more complex rules for the SimpleRMA resource specification.

It must also be ensured that a level one quality of service specification is for a higher level of quality than a level two specification, similarly for two and three and n and n+1.

An example specification some some component may be:

```
Level 1:
    AVECPU { 22% }
    SIMPLERMA { 1: P=50, D=10 2: P=100, D=5}
    MINMEMORY { 15Kb }
Level 2:
    AveCPU { 15% }
    RMA { 1: P=50, D=10 2: P=100, D=5}
```

From the above we can see that this component to achieve its level one quality of service proposal it re-

[1] Written using OCL [3]

quires that the system provide on average 22% of CPU time to the component and that for a RMA analysis the component has two main processes one of which could be called with period 50ms and deadline 10ms and the other 100ms and deadline 5ms in the worst case situation.

5 Obtaining Quality of Service Values

So far we have outlined the architecture, download procedure and quality of service specification. It however is necessary to obtain the values and the criteria for writing the quality of service specification. This unfortunately does prove problematical.

In [4] describes the situation where performance analysis is made upon a component and concludes with the fact that obtaining this data is difficult and in some cases may actually be impossible. We take the view that it is always the case that some generalisations and guesses can be made.

Because generally we work with embedded systems, parameters such as memory consumption are *easily* calculated for a given component. Average CPU utilisation and RMA figures are more problematical especially when dealing with a component that can run across many different platforms of varying capability. One solution we have investigated is that the components will have to be tested and from these testing and simulation runs we can obtain performance figures. If we also include information about what platforms the components have been tested on then it is possible to extrapolate that data in order to refine the quality of service values. For example if on a 200MHz system a certain component requires on average 20% CPU resources, then on a 400MHz system this figure will be approximately half. The work described in [1] has also investigated this.

These are broad and in some cases naïve generalisations but they are at present a good enough base-line for calculating these figures. As more data is collected upon component performance then we can both refine the quality of service values and also the methods for calculating and analysing those figures. Work is currently under way on investigating this, for example we have test-bed environments simulating component download running on both Linux and Solaris machines with various capabilities.

6 Conclusions and Future Work

Obviously the system and method described in this paper are in an early state of development, however we have proved that the ideas do work at least in demonstration environments (eg: Linux/Solaris host, Java executables). The current performance would of course not be acceptable in a *real* real-time system.

A number of issues do need to be resolved such as increasing the efficient of the system so that its performance does not impact greatly on the user-side of the system. This is especially the case when this is implemented in a real-time or embedded system and the consequences of are described in [4].

Also we must seek to improve the set and defintion of the quality of service attributes and parameters. This is something that we can only refine by experience while testing these kinds of systems. We have found so far that generally we are getting reasonable results from the work described here.

References

[1] A. Alonso, I. Casillas, and J. A. de la Puente. Dynamic replacement of software in hard real-time systems: Practical assessment. In *Proceedings of the 7th Euromicro Workshop on Real-Time Systems (EUROMICRO-RTS'95)*, 1995.

[2] M. H. Klein, T. Ralya, B. Pollak, R. Obenza, and M. G. Harbour. *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems*. Kluwer Academic Publishers, 1993.

[3] Object Management Group. *OMG Unified Modelling Language Specification (Action Semantics)*, version 1.4 (final adopted specification) edition, January 2002. OMG Document Number ad/02-01-09.

[4] M. Sitaraman, G. Kulczycki, J. Krone, W. F. Ogden, and A. L. N. Reddy. Performance specification of software components. In *Proceedings of the Symposium on Software Reusability, SSR'01, Toronto, Canada*, 2000.

Software Component Deployment in Consumer Device Product-lines

Ronan Mac Laverty, Aapo Rautiainen, Francis Tam
Nokia Research Center, Helsinki
{ronan.maclaverty,aapo.rautiainen,francis.tam}@nokia.com

Abstract

Effective deployment of components is imperative for consumer device manufacturers; these must utilize the resources available optimally. For single systems this is a standard software engineering problem, but for product-lines new techniques must be devised. These are needed to allow component reuse while minimizing the overhead from cross product components. To achieve this a prototype for a tool to automatically generate and evaluate a deployment for a consumer device has been developed. This system and the motivation behind its development are described below, including directions for its future development.

1. Introduction

In a component-based system the deployment of components dictates its characteristics. A poor deployment can increase inter-component communications costs, memory requirements, degrade performance and affect a range of other properties. Conversely, a good deployment can optimize resource usage and performance. For consumer device manufacturers resource usage is important, as efficient usage can mean reduced hardware costs.

To streamline software generation the use of a product-line approach is necessary to support efficient component reuse across a product family. However, this can lead to problems in managing the optimization of the end system, when large numbers of components are used. Another problem can stem from the focus on logical architecture of current product-line designs.

To support designers of products, tools are needed to guide deployment. These should allow them to check architectures for property-based criteria in order to choose the most efficient.

The paper below describes a sample component-based product-line for a consumer device, which allows the flexible deployment of system components. This provides us with a good test-bed to study the potential of automated deployment generation and evaluation. The latter sections contain an outline of an approach for this and a prototype tool developed at Nokia Research Center. It also outlines several areas of potential development of this tool.

2. Motivation

The advantage of monolithic software construction is that each system can be optimized for resource usage, performance etc. As the number of systems increases this approach begins to become inefficient, especially as the complexity of the system and the number of developers increase. This is further exacerbated by globalization of software development. The result is that developers cannot know all the software to a sufficient level of detail to allow a high level of optimization. Additionally, when more systems are being produced, building each system from scratch becomes untenable, given the combined design, development and testing overheads.

To meet these concerns software reuse must be explicitly supported across products. Therefore Nokia has adopted a product-line approach to phone software development, along the lines of [1]. This involves identifying domain specific architectures and components. These are combined into a reference architecture that is specialized for individual products - see Figure 1.

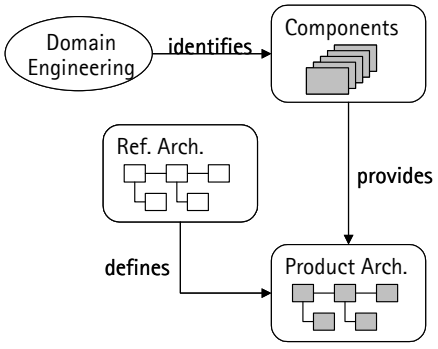


Figure 1: Product generation inside a product-line

There are several problems with implementing this approach in resource-constrained devices. Some stem from the approach itself and others from the needs of the domain.

The core problem with this approach is that it concerns itself with the logical composition of the system. The core functionality, applications and components are defined, but not how they are deployed. This results in systems that fulfill their functional requirements and certain quality attributes, but are not optimized for individual products. This creates product specific overheads, such as memory requirements and performance costs.

3. System Architecture

To utilize the product-line approach, a basic reference architecture is needed that covers all the systems developed. The fundamental software architecture inside the system being developed is a layer style, as shown in Figure 2. The uppermost layer contains applications that rely on middleware services. The lowest layer contains the necessary hardware abstraction and operating system. The middleware layer consists of software services that provide access to hardware and logical services.

From the point of view of deployment both applications and middleware can be considered as components. The difference is that applications do not support other services. This classification simplifies the deployment model of the system.

Some domain constraints affect the system architecture. One is that the number of threads executing in the system should be minimized. This will reduce the needed stack memory and simplify scheduling. Another is that certain services must run at the highest priority to meet real-time requirements. This forces a fixed number of threads and dictates the software that runs within them. The need to reduce the number of threads means that in certain circumstances components shared the same thread.

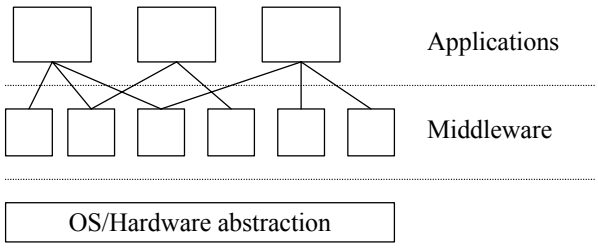


Figure 2: System architecture

Business constraints also affect this architecture. To enable large numbers of different products it was decided to increase the flexibility of the system to support reuse. To increase the reuse of components in different systems the coupling between them was minimized. As a result, components are location dependent, meaning that within the system a component-client cannot tell if the component is inside the same memory space or thread. This separates the logical structure of the system from the process structure. However, the reference and domain specific architectures place constraints on their deployment.

4. System Deployment

Deployment within this system, while being different from physical deployment, has many parallels. Instead of deploying to computing nodes, components are mapped to threads that support their execution. There are several problems associated with this deployment. These primarily result from the separation of the logical architecture and the deployment architecture, heightened by the need to support many different products.

The product-line approach, while having many advantages, also generates problems. These arise from the number of products supported and the number of associated components. For small-scale systems the properties of the component can be understood, and then best way to deploy them can be devised. For large numbers the possible deployment possibilities is huge, far beyond a designer's capability to optimize.

The products developed by Nokia vary in many respects; this is reflected in their different software and hardware architectures. A given deployment that might work well for one system might not work for another, if the underlying hardware and devices changes.

The separation of the logical and deployment concerns is considered necessary for flexible product development. It allows designers to specify the logical content irrespective of the underlying process model. However, as designers can use a non-optimal number of threads, clearly a method to guide the designer is needed.

5. Automated Deployment

The goal of the work presented here is to describe an approach to analyzing and producing an optimum component deployment for this system. The approach needs only to be comparatively correct; if one deployment to analyzed to be better than another this should also be reflected in the real system. This will allow designers to compare different deployments while allowing initial research work into model characteristics and requirements.

In devising a placement strategy for mapping component onto threads, we have identified that there are similarities in the optimization techniques for configuring parallel applications to run on multiprocessor systems.

The goal of these techniques is to find the optimum mapping of routines/subsystems to processors; this parallels our desired mapping of functional elements to executing entities. We have therefore examined processor-mapping strategies [2][3][4][5][6] in multiprocessor systems as a first step.

To maximize the throughput of a multiprocessor system, the use of multitasking and multithreading have been investigated extensively with some very positive results. In particular, schemes having a balanced combination of multitasking and multithreading have been most encouraging. Both these are established concepts in operating systems: multitasking hides the latency of slow I/O devices; multithreading to hides the latency of slow memory operations.

The table below shows some of the conceptual parallels:

Processor Allocation	Component Deployment
Processor	Thread
Task	Component
Inter-processor communication time	Inter-thread communication time
Inter-task communication time	Internal thread communication time
Computational characteristics	Computational characteristics

Figure 3: Comparison of multiprocessor mappings and component deployment

Other parallels exist in the goals of deployment; these are to improve performance, by reducing the cost of multi-processor deployment. For example, in distributed systems the communication between nodes is much more expensive than inter-process interaction. This is similar to our system, where inter-thread communication is much faster than inter-thread communication.

Another important parallel arise from architectural constraints on both our system and multiprocessor systems. An example is the need to handle resource locking between components, and the need to separate or co-locate components inside a deployment entity. These constraints have a major impact on the end performance and early analysis can avoid architectural problems later in development.

The initial approach taken has been to apply appropriate processor allocation strategies to deploying components, by substituting the relevant parameters in the algorithms. Currently, there are two optimization techniques used for multi-processor mapping; model-based and program transformation.

Model-based techniques are normally deployed at the system design stage before implementation. In general the model of a system design is improved upon, based on the resource constraints in the target system, and an optimal system structure, or configuration is generated for implementation. This approach is static in nature, in the sense that all mapping decisions are made at design time.

Program transformation is used after a system has been implemented. The idea is to optimize the program code, based upon the resource constraints and/or application specific requirements. This approach can be either static or dynamic. Static transformation re-arranges the initial code such that during execution, the system will have optimal performance. Dynamic transformation occurs at run time; it supports task migration and can be used for load balancing and fault tolerance.

As the current goal of our work is to develop a tool to produce an optimal architecture for a given fixed product the model-based approach was chosen.

6. Deployment Generation/Analysis Tool

To test this approach an experimental tool was developed at Nokia Research Center. This was used to test the potential for a deployment generation tool based on multiprocessor mapping strategies. An overview of the tool's structure is given in Figure 4. This structure aims to separate components from system properties and domain constraints.

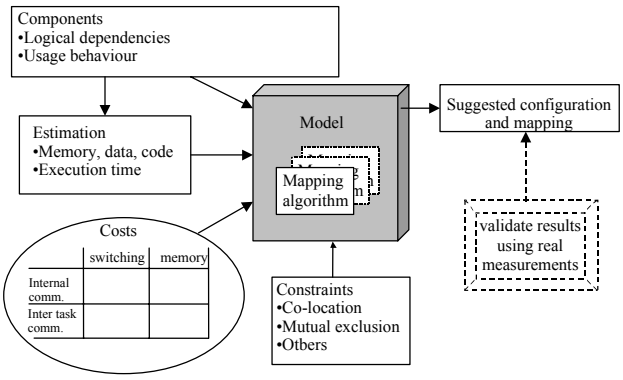


Figure 4: Overview of the deployment analysis tool

The components deployed are defined by the logical architecture of a product. This can be viewed as a dependency graph, starting with applications and terminating at logical components or device drivers. To each edge in the graph a weight is assigned based on the level of coupling between the components. If one component only rarely uses another then this is reflected by a low value; similarly, if a component relies heavily on another a high value is assigned.

The system information consisted of the costs of inter-thread communication versus in-thread communication.

As these values are used to comparatively evaluate the deployment only their correct ratio is important.

Architectural constraints such as co-location mutual exclusion can be added as input to the model. This allows some domain concerns to be reused for different product configurations.

Initial studies with sample logical architectures and the mapping algorithm from [6] have shown that this model, although simple, works. This algorithm was used to generate deployments for a deeply nested, a fully connected and a hybrid component graph. The results all showed a dramatic difference in estimated performance. In one typical case, where there is a deep nesting of components, the generated mapping of the components to threads resulted in an estimated execution time one tenth of a naïve mapping.

Considering the use of statistical data to define the level of component interaction the results are approximate. It is the use of statistical data that is this models weak point, however this is unavoidable due to the difficulties of a complete analysis. Therefore, we see this model as a good way to identify potential deployment architectures, prior to exhaustive testing.

7. Future work

The current modeling approach provides a good basis for further research in the areas of model development, integration with the product-line process and more in depth handling of components.

The model used above can be improved by providing a richer set of choices for the developer. It only used one slightly altered algorithm from the multi-processor domain; this could be expanded with more algorithms. To support a wide variety of products, the architectural constraints must be developed to provide more domain specific modeling. This might have to be reflected in the component descriptions used in the model. Finally, the values used to evaluate the resources used should be validated against a real system.

As the movement from logical definition to deployment is a crucial step in the development of any system, this tool could be altered to assist the product-line process. A designer could select the necessary components and this tool would generate a deployment and an evaluation. In this case the simplicity of the tool's underlying model and its facilitation for fast analysis would provide immediate and effective feedback to the designer.

The algorithms identified so far are a direct result of their application domain. From this it can be concluded that there is possibility to tune, adapt and develop algorithms specifically for use in generating optimum component deployments. As mentioned above, the static nature of the current software means that the model-based approach to optimization is feasible. In the future, there might be a need for dynamic program transformations that respond to a user's needs.

Not all components are as fine-grained as the ones described above. In many cases components can be composed of other components and objects. The deployment of these entities across a set of tasks also needs to be tackled. In this case the current component model is not sufficient and will need to be expanded to address the internal deployment of a larger-grained component.

8. Acknowledgments

The authors would like to thank Nokia Mobile Phones/SW-RTA for funding the development of the prototype.

9. References

[1] P. Clemens and L. Northop, Software Product Lines, Addison Wesley, 2002

[2] A. Mitschele-Thiel, "Hierarchical optimization of parallel applications", Proceedings of the Second International Workshop on Software Engineering for Parallel and Distributed Systems, 1997 pp 222-233.

[3] H.L. Muller, P.W.A.Stallard and D.H.D. Warren, "Multitasking and multithreading on a multiprocessor with virtual shared memory", Proceedings of the Second International Symposium on High-Performance Computer Architecture, 1996 pp 212-221.

[4] E. Smirni, C.A. Childers, E. Rosti and L.W. Dowdy, "Thread placement on the Intel Paragon: modeling and experimentation", Proceedings of the Third International Workshop on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems, 1995 pp 226-231.

[5] K. Taura and A. Chien, "A heuristic algorithm for mapping communicating tasks on heterogeneous resources", Proceedings of the 9th Heterogeneous Computing Workshop, 2000 pp 102-115.

[6] S. Yalamanchili, L. Te Winkel, D. Perschbacher and B. Shenoy, "Genie: An environment for partitioning and mapping in embedded multiprocessors", Proceedings of the Fifth IEEE Symposium on Parallel and Distributed Processing, 1993 pp 522-529.

Reusing Verification Information of Incomplete Specifications

Rebeca P. Díaz Redondo, José J. Pazos Arias and Ana Fernández Vilas
Departamento de Enxeñería Telemática. University of Vigo. 36200 Vigo. Spain
{rebeca, jose, avilas}@det.uvigo.es

Abstract

The possibility of verifying systems during any phase of the software development process is one of the most significant advantages of using formal methods. Model checking is considered to be the broadest used formal verification technique, even though a great quantity of computing resources are needed to verify medium-large and large systems. As verification is present over the whole software process, these amount of resources is more critic in incremental and iterative life cycles. Our proposal focuses on reusing incomplete models and their verification results — which are obtained from a model checking algorithm— to reduce formal verification costs in this kind of life cycles.

1. Introduction

Reuse is a promising way to help improving software development and, even though it has been practiced in different ways over many years, it is still an emerging discipline. Although reusing material resources (basically code) has been made *ad-hoc* since programming was born, reusing more abstract level components like human resources (ideas, designs, etc.) is more attractive because of the possibility of increasing the reuse benefits. Our proposal [3] shares this last philosophy and it offers a methodology to reuse high abstract level components: incomplete specifications —obtained from transient phases of an iterative and incremental development process—; and their verification results —obtained from a model checking algorithm. These high abstract level reusable components are specified using a formal representation which is not only the pattern in a specification-based retrieval, but the content of the components, so we have a content-oriented retrieval.

The paper is organized as follows: following section focuses on describing the software development process where software reuse is going to be included; section 3 summarizes repository management; the process of reusing verification efforts —including mathematical and practical aspects— is detailed in sections 4 and 5; and, finally, summary and future work are exposed in section 6.

2. Context

Current software engineering practice addresses problems of building large and complex systems by the use of incremental development techniques. Formal methods are expected to be adapted to support this practice, outside their traditional role of verifying that a model meets certain fixed requirements. SCTL-MUS [5] is a formal methodology for software development of distributed systems which joins both tendencies: on the one hand, the totally formalization of the process, combining different FDTs; and, on the other hand, an incremental and iterative point of view. In figure 1 it is shown the first phase of this methodology, where a complete and consistent functional specification of the system is obtained from user's specification.

Using the many-valued logic SCTL [5] (S*imple* C*ausal* T*emporal* L*ogic*) allows the formal description of functional requirements without being too far from natural language semantic. A generic causal requirement in SCTL follows this pattern:

Premise$\Rightarrow\otimes$ **Consequence**,

which establishes a causing condition (premise); a temporal operator determining the applicability of the cause ($\Rightarrow\otimes$); and a condition which is the effect (consequence). Apart from causation, SCTL is a six-valued logic, even though it is only possible specifying three different values: possible or *true*, non possible or *false* and *unspecified*. This concept of unspecification is specially useful to deal with both incomplete and inconsistent information obtained by requirements capture, because although events will be possible or non possible at the final stage, in intermediate phases of the specification process it may be that users do not have enough information about them yet, so these events are *unspecified* at this phase.

In this methodology, SCTL requirements are translated into MUS (M*odel of* U*nspecified* S*tates*) graph by incremental synthesis. This state-transition formalism allows prototyping and feedback with users, and supports the consistency checking by using a model checking algorithm. MUS graphs are based on typical labeled-transitions graph, but including another facility: unspecification of its ele-

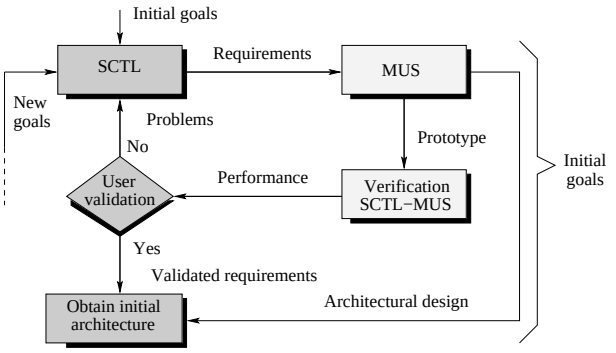


Figure 1. SCTL-MUS methodology

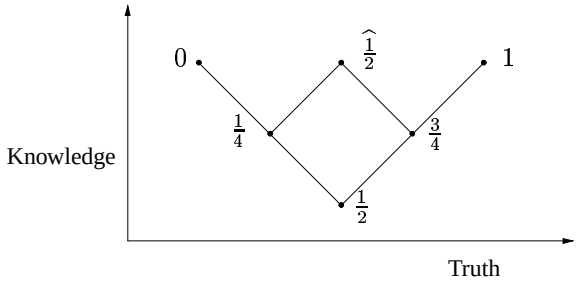


Figure 2. Knowledge and Truth partial orderings among degrees of satisfaction.

ments.

The degree of satisfaction of an SCTL requirement is based on causal propositions: "*an SCTL requirement is satisfied iff its premise is satisfied and its consequence is satisfied according to its temporal operator*". As SCTL-MUS methodology adds unspecification concept, this degree of satisfaction must not be *false* (nor *true*), just as the Boolean logic. In fact, it must have a degree of satisfaction related to its unspecification (totally or partially unspecified on the MUS model), because it can become *true* or *false* requirement, depending on how it is specified in future. Consequently, this methodology defines six different degrees of satisfaction, $\phi \in \Phi = \{0, \frac{1}{4}, \frac{1}{2}, \widehat{\frac{1}{2}}, \frac{3}{4}, 1\}$, which can be partially ordered according to a *knowledge level* ($\leq_c$) (figure 2) as follows:

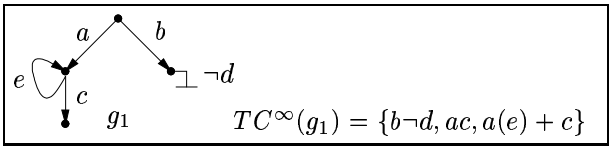
- $\{1, \widehat{\frac{1}{2}}, 0\}$ are the highest knowledge levels. We know at the current stage of the system the final degree of satisfaction of the property. The meaning of this verification results are the following ones: 1 or *true* means the requirement is satisfied; 0 or *false* implies the requirement is not satisfied; and $\widehat{\frac{1}{2}}$ or *contradictory* means the requirement cannot become *true* or *false*.
- $\{\frac{1}{4}, \frac{3}{4}\}$ are the middle knowledge levels. Although at the current stage of the system, the property is partially unspecified, we know its satisfaction tendency. That is, in a subsequent stage of specification, the degree of satisfaction will be $\frac{1}{4} \leq_c \phi'$ (respectively $\frac{3}{4} \leq_c \phi'$) for the current value $\frac{1}{4}$ (respectively $\frac{3}{4}$).
- $\{\frac{1}{2}\}$ is the lowest knowledge level. The property is totally unspecified at the current system's stage and we do not known any information about its future behaviour.

In short, the degree of satisfaction of an SCTL requirement varies according to its closeness to the *true* (or *false*) degree of satisfaction —partial order according to *truth level* (figure 2).

3. Lattice of reusable components

Establishing functional relationships among components enables defining component hierarchies or lattices to classify and retrieve them in an proper way. We define four partial order relationships among components and several metrics to quantify functional differences, which are needed to assess the necessity of making changes to existing components to satisfy query's specifications. As this paper focus on reusing verification efforts, we only describe here one of these identified functional relationships because the verification reuse process is based on it (the other ones are main pieces to reuse incomplete specifications).

MUS graphs can be organized on a lattice based on a function TC^∞ that associates with every MUS graph $g \in \mathbb{G}$ a set $TC^\infty(g)$, which is based on complete trace semantics [1]. Main differences with traditional ones are that TC^∞ takes into account both *true* and *false* events (for instance $\neg d$ in figure below), in order to differentiate *false* events from *unspecified* ones; and it includes infinite traces in $TC^\infty(g)$. An example of $TC^\infty(g)$ obtaining is shown in figure below.



$TC^\infty(g)$ constitutes the observable behaviour of g according to TC^∞-criteria and it allows defining the equivalence relation $=^\infty_{\mathrm{TC}} \in \mathbb{G} \times \mathbb{G}$ given by $g =^\infty_{\mathrm{TC}} g' \Leftrightarrow TC^\infty(g) = TC^\infty(g')$, and the preorder $\sqsubseteq^\infty_{\mathrm{TC}} \in \mathbb{G} \times \mathbb{G}$ by $g \sqsubseteq^\infty_{\mathrm{TC}} g' \Leftrightarrow TC^\infty(g) \sqsubseteq TC^\infty(g')$. $\sqsubseteq^\infty_{\mathrm{TC}}$ provides a partial order between equivalence classes, that is, graph sets indistinguishable using TC^∞-observations, so $(\mathbb{G}, \sqsubseteq^\infty_{\mathrm{TC}})$ is a *partially ordered set*, or *poset*. A subset $G_1 \in \mathbb{G}$ is called a *chain* if every two graphs in G_1 are TC^∞-related.

Each reusable component (C) gathers both its functional specification, which is expressed by the set of SCTL re-

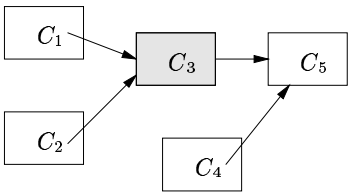


Figure 3. Chain of reusable components

quirements and modeled by the temporal evolution MUS graph (g), and an interface or *profile* information, which is automatically obtained from its functional characteristics to classify and retrieve it from the repository ($TC^{\infty}(g)$ is part of this interface). Besides this, a reusable component stores verification information, that is, the set of properties which had been verified on the MUS graph and their verification results (section 4).

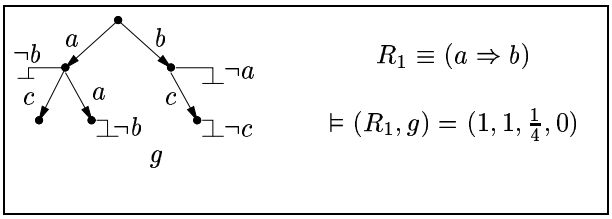
Each reusable component (C) is classified in the repository after finding its *correct place* in the lattice defined by TC^{∞} relation. That is, it is necessary looking for those components TC^{∞}-related to C[1] such as C is TC^{∞}-included on them, and those components TC^{∞}-related to C such as they are TC^{∞}-included on C (figure 3).

4. Reusable verification information

In order to store interesting verification information linked to each reusable component, we define four properties which summarize the degrees of satisfaction of an SCTL property R in the states of a MUS graph g:

- $\exists \Diamond R$ expresses that "*some trace of the system satisfies eventually* R" and its degree of satisfaction is denoted $\models (\exists \Diamond R, g)$.
- $\exists \Box R$ expresses that "*some trace of the system satisfies invariantly* R" and its degree of satisfaction is denoted $\models (\exists \Box R, g)$.
- $\forall \Diamond R$ expresses that "*every trace of the system satisfies eventually* R" and its degree of satisfaction is denoted $\models (\forall \Diamond R, g)$.
- $\forall \Box R$ expresses that "*every trace of the system satisfies invariantly* R" and its degree of satisfaction is denoted $\models (\forall \Box R, g)$.

To sum up, for each property verified in the MUS graph, we will have four derived properties whose degrees of satisfaction make up the degree of satisfaction of an SCTL property R in a MUS graph g, denoted $\models (R, g) = (\models (\exists \Diamond R, g), \models (\forall \Diamond R, g), \models (\exists \Box R, g), \models (\forall \Box R, g))$. This verification information is stored in the reusable component whose MUS graph is g, ready to be recovered whenever it is necessary.



An example of obtaining the degree of satisfaction of a property R_1 in a graph g is shown in figure above. After studying the degrees of satisfaction of R_1 in every state of g, $\models (R_1, g)$ is extracted. Its meaning is as follows: because of $\models (\forall \Diamond R_1, g) = 1$, every trace of g satisfies eventually R_1, that is, R_1 is a *liveness property* in g; since $\models (\exists \Box R_1, g) = \frac{1}{4}$, R_1 is partially specified in g, but regardless of future iterations, any trace of g does not satisfy invariantly R_1, that is, R_1 is *not a safety property* in g.

5. How to reuse verification efforts?

The defined classification scheme (section 3) implies that, for instance in figure 3, C_1 and C_2 are *functional parts* of C_3, being the last one a *functional part* of C_5. The main question in this situation is: how to know the degree of satisfaction of an SCTL property R in C_3, if we know the degrees of satisfaction of R in C_1, C_2 and C_5?. In this section we resolve this question after studying some mathematical aspects related to the ordering of degrees of satisfaction, and by applying these results to the proposed practical environment.

5.1. Mathematical aspects

Let $\sqsubseteq_e$ be a simulation relation between two states E_1, and E_2, denoted by $E_1 \sqsubseteq_e E_2$, satisfying: $\forall E_1' \mid E_1 \stackrel{\omega}{\rightarrow} E_1'$ then $\exists E_2' \mid E_2 \stackrel{\omega}{\rightarrow} E_2'$ and $E_1' \sqsubseteq_e E_2'$ and if $E_1 \stackrel{\omega}{\not\rightarrow}$ then $E_2 \stackrel{\omega}{\not\rightarrow}$. Let g and g' two MUS graphs, then g' simulates g, denoted $g \sqsubseteq_e g'$, iff $E_0 \sqsubseteq_e E_0'$, where E_0 is the initial state of g and E_0' the initial state of g'.

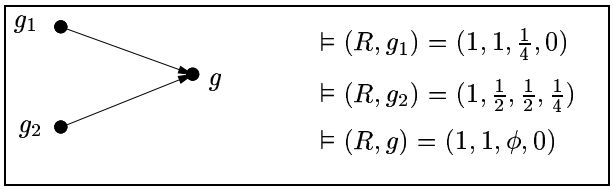
Property 1. *Let E and E' be two states satisfying $E \sqsubseteq_e E'$, then $\models (R, E) \leq_c \models (R, E')$. That is, the degree of satisfaction of a property R in E has a lower* knowledge level *than its degree of satisfaction in E'*[2].

As consequence of property 1, it is possible to extract verification information about the degree of satisfaction of one SCTL property R in a MUS graph g, that is, $\models (R, g)$, knowing $\models (R, g')$ and $\models (R, g'')$, where $g' \sqsubseteq_e g \sqsubseteq_e g''$, without running the verification algorithm. We have obtained different tables storing these reusable verification re-

[1] Two components C and C' are TC^{∞}-related ($C \sqsubseteq^{\infty}_{\mathrm{TC}} C'$ or $C' \sqsubseteq^{\infty}_{\mathrm{TC}} C$) iff their MUS graphs g and g' are TC^{∞}-related ($g \sqsubseteq^{\infty}_{\mathrm{TC}} g'$ or $g' \sqsubseteq^{\infty}_{\mathrm{TC}} g$).

[2] This property's demonstration is based on the structure of an SCTL requirement.

sults, but because of space reasons it is impossible to include them here.



A little example of what kind of verification information can be reused it is shown in figure above. In this example there are three components satisfying $g_1 \sqsubseteq_{\mathrm{e}} g$ and $g_2 \sqsubseteq_{\mathrm{e}} g$. After studying the degrees of satisfaction of a property $R \equiv (a \Rightarrow b)$ in both graphs, we can deduce that R is a liveness property in g and it is not a safety property, without running the model checking algorithm.

5.2. Practical aspects

The main problem of the solution which has been proposed in the previous section is comparing MUS graphs using the $\sqsubseteq_{\mathrm{e}}$ relationship in an efficient way. The following property offers a solution to this problem:

Property 2. *$\sqsubseteq_{\mathrm{e}}$ defines a partial order between MUS graphs, but, for deterministic graphs, it can be demonstrate that $\sqsubseteq_{\mathrm{e}}$ is totally equivalent to $\sqsubseteq^{\infty}_{\mathrm{TC}}$.*

because comparing components according to TC^{∞} relationship is much more efficient and equal effective (property 2) than comparing components according to $\sqsubseteq_{\mathrm{e}}$.

So, box labeled as *Verification SCTL-MUS* in figure 1 — where a set of properties $\{R_i\}$ are formally demonstrated on a MUS prototype g— may be replaced by the following steps:

1. Obtain the $TC^{\infty}(g)$ information to be able to locate in the repository the reusable components which are TC^{∞}-related to g.
2. Obtain the $TC^{\infty}(R_i)$ information in order to locate those functional requirements which are functionally equivalents to each R_i.
3. Retrieve those components whose classification distance to g is as little as possible and where verification information about functionally equivalent to $\{R_i\}$ properties are stored.
4. Extract verification information about $\models (R_i, g)\ \forall i$ from the recovered components.

If the verification information obtained is not enough to know the required verification results, it is necessary to run the model checking algorithm, but this execution can be reduced depending on the available verification information.

6. Summary and future work

The work introduced in this paper focuses on reusing verification information linked to incomplete systems in a totally formalized, incremental and iterative software development process with the aim of minimizing its formal verification costs. That is, we propose reusing high abstract level verification information, as difference to other approaches like [4] where although reusing verification results is also proposed, they are less formalized proofs (simulation proofs) over code components (algorithms).

After studying different relationships among incomplete specifications, we have identified a criteria to compare functional specifications based on trace semantics and taking advance of unspecification inherent to incomplete models. Applying this criteria, we build a lattice of reusable components which allows avoiding formal verification tasks in the retrieval process. This entails a fast retrieval which is accurate enough to reuse verification information and it makes a difference between other proposals [2, 6, 7] where specification matching is based on theorem proving. We have also identified what verification information can be reused and, consequently, how to minimize formal verification tasks.

In order to continue this proposal, we are working on reusing verification results of *functional similar* properties with the given one; and with the possibility of dividing the given property into several properties. Both lines share the same goal: increasing the possibility of finding interesting verification information in the repository.

References

[1] *Handbook of Process Algebra*, chapter The Linear Time - Branching Time Spectrum I: The Semantics of Concrete, Sequential Processes. Elsevier Science.

[2] B. H. C. Cheng and J. J. Jeng. Reusing Analogous Components. *IEEE Trans. on Knowledge and Data Engineering*, 9(2), Mar. 1997.

[3] R. P. Díaz-Redondo and J. J. Pazos-Arias. Reuse of Verification Efforts and Incomplete Specifications in a Formalized, Iterative and Incremental Software Process. In *Proceedings of International Conference on Software Engineering (ICSE) Doctoral Symposium*, Toronto, Ontario (Canadá), May 2001.

[4] I. Keidar, R. Khazan, N. Lynch, and A. Shvartsman. An Inheritance-Based Technique for Building Simulation Proofs Incrementally. In *22nd International Conference on Software Engineering (ICSE)*, pages 478–487, June 2000.

[5] J. J. Pazos-Arias and J. García-Duque. SCTL-MUS: A Formal Methodology for Software Development of Distributed Systems. A Case Study. *Formal Aspects of Computing*, 13:50–91, 2001.

[6] J. Schumann and Fischer. NORA/HAMMR: Making Deduction-Based Software Component Retrieval Practical. In M. Lowry and Y. Ledru, editors, *Proceedings of the 12th International Conference Automated Software Engineering*, pages 246–254. IEEE Computer Society Press, Nov. 1997.

[7] A. M. Zaremski and J. M. Wing. Specification Matching of Software Components. *ACM Transactions on Software Engineering and Methodology*, 6(4):333–369, Oct. 1997.

Industrial experience of using a component-based approach to industrial robot control system development.

Peter Eriksson
ABB, Sweden
peter.j.eriksson@se.abb.com

Introduction

I will share some experience that we have gained during ten years of development of our today's robot controller software, supporting simulation systems and communication software. ABB produces and delivers industrial robot systems to a variety of application fields such as those for car manufacturing, foundry, painting and food packaging. Recently ABB has as the first robot manufacturer delivered more than 100.000 units to the market. The controller generation that this presentation will cover represents about half of the delivered systems. The controller software represents a huge and complex system with several million lines of code and several hundred man-years of development. Many different software engineering fields such as real-time, motion control, databases, application programming language, communication and human-machine interaction are combined in these products and increase the demands on the development process as well as the system architecture.

Experience and some highlights during ten years of using a component-based approach to system development

The subjects that will be covered can be divided in the following areas

- Organization
- Methods
- System architecture
- Test strategy
- Legal and commercial issues

Some examples and solutions that we have applied on the different subjects will be presented.

Present and future challenges, goals and obstacles for CBSE from my perspective

Many challenges and unsolved issues exist and even if we have been very successful during our development we are heavily dependent on the experience of individuals and on the maintaining of quality and system architecture. During the presentation I will highlight some of those issues that need to be addressed to establish a higher degree of stability and predictability in the type of component-based software architectures that we use.