A Framework for Software Component Interface Specification and Analysis

by

Matthew Hoyt

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2001

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners. I understand that my thesis may be made electronically available to the public.

Abstract

Although markets are emerging for commercial off-the-shelf components (such as Sun JavaBeans), there are many barriers to widespread component adoption. This is due to the inherent 'black-box' nature of software components: developers have no knowledge or control of the component's internal characteristics. Without source or design details, developers only have the component's interface, documentation and test results to answer important questions about reliability, proper use, behavior and performance. The current best practice of specifying a component's capabilities by providing only the syntax and informal documentation is insufficient to assemble mission or safety-critical systems successfully.

To address these problems we have developed a framework for creating and analyzing the concise specifications of components and their related interfaces. The framework extends a formal model for software architecture descriptions to support the specification of a range of terms. With formal component specifications developers can use the framework to analyze the properties of individual components or of entire systems. Unlike other approaches, the formal basis and implementation of our framework enhance understanding and automates much of the component analysis process.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

The goal of component-based software engineering (CBSE) is to enable the assembly of software systems from existing, independent, components. CBSE is a natural progression in the evolution of software engineering as the use of prefabricated components is fundamental to any mature engineering discipline [Szy97]. For example, automobiles are complex machines, but many models share common components such as engines, suspension and brakes. It is expected that CBSE will finally make widespread software reuse practical and, as observed in other engineering fields, result in reduced costs, and a faster time-to-market for software systems.

Despite the current successes of software components, most notably Visual Basic [Mau00], there are many barriers to the widespread adoption of software components. In the market, most components are shipped in a binary form that leads to a 'black-box' effect – a system developer has no knowledge of the component's internals. Without access to the sources, component users have to rely on documen-

tation to answer questions about important information including the component's behavior and resource requirements. Unfortunately, component documentation often does not contain enough detail to overcome the 'black-box' effect. Many cite this as the main reason CBSE and software component markets have yet to be widely embraced, especially in the area of mission-critical systems.

To address these problems, we have developed a framework for providing detailed, concise specifications of components. Based upon formal specifications, the framework provides both structural and behavioral reasoning capabilities.

1.2 Component-based Software Engineering

In [Szy97], Szyperksi proposes that CBSE provides the middle ground between custom software development and commercial off-the-shelf (COTS) software. Custom built software is often very expensive, as development and maintenance require significant resources. However, such systems can be modified and extended to meet the requirements exactly. On the other hand, COTS software is substantially less expensive, but also may not match user requirements, creating problems. By allowing system developers to compose systems from COTS components, CBSE provides the advantages of both types of software. COTS components can be obtained for less cost than equivalent custom development but can be custom assembled to match the requirements.

1.2.1 Benefits

There are many potential benefits of CBSE. These benefits draw from those that have been realized by component use in other fields of engineering.

- **Higher quality and reliability.** Because it is expected that COTS components will have already been sufficiently tested , there will be less potential for error in the assembled system leading to increased quality. As well, reusing the same component across applications leads to increased consistency, leading to benefits for both users and developers [Wil01].

- **Increased adaptability.** Component-based systems are easily adapted because of the modular nature of components. Components can not only be reused, but components can be replaced with minimal impact on the system. This allows for easier upgrades and enhancements. As well, many component standards allow components developed in different languages and environments to interact together, increasing the number of possible implementations.

- **Access to expertise.** Component vendors often have specialized domain knowledge that is encapsulated within their products. This can significantly reduce analysis and design time while providing a technically superior solution. As well, vendors may ensure conformance to industry and domain standards [Wil01].

- **Reduced risk.** The resources and risk required to develop and maintain a component is delegated to the component vendor [Wil01].

- **Reduced effort and increased focus.** If most system requirements can be met through existing components, less effort is required. This frees up skills and allows developers to focus on the specific domain issues [O'R99].

1.3 Software Components

The field of CBSE is emerging quickly, but it is not yet well understood. There are varying opinions and disagreement about what constitutes a software component [KB98], although most research and application focuses on COTS components defined in [Szy97] as:

> '...binary units of independent production, acquisition and deployment that interact to form a functioning system.'

Characterizing a component as a "binary unit" implies that it is in an executable form, and that the implementation is hidden from the client. Binary components reduce complexity for clients by hiding implementation details and allow vendors to provide alternate implementations and maintain trade secrets. Independence is essential if multiple vendors and clients are to be allowed, which is key to the success of CBSE [Han98].

1.3.1 Standards

To assist in the development of components and increase interoperability, most COTS components are constructed to conform to a component standard. Component standards, in varying degrees, provide the 'wiring' that allows components to interact with one another. Although there are many standards, the field is dominated by three commercial entries.

- **Microsoft's Component Object Model (COM).** The COM is a binary standard for components. As a binary standard, no restrictions are placed on the implementation language. The interface is the only required entity of

a COM component. Distributed COM (DCOM) and COM+ are variants of COM that provide support for distributed systems and transactions [Mic01].

- **Sun's JavaBeans.** JavaBeans are an extension to the Java platform. A bean is a set of classes combined with resources. Beans are designed with tool support in mind and can expose design and runtime interfaces. The bean model provides support for methods, events and properties. Enterprise JavaBeans (EJB) are variant of JavaBeans that are server-based and support persistent components and transactions [Sun01].

- **OMG's Common Object Request Broker Architecture (CORBA).** The Object Management Group (OMG) is a consortium of companies attempting to standardize interoperable enterprise applications. CORBA is an infrastructure that allows applications to work over networks without restrictions on language, platform or protocol. To create a CORBA object, a specification is written in an interface definition language (IDL) which is compiled creating code that allows clients to access the interface. During execution this stub code invokes the object request broker (ORB) which handles communication to the CORBA object. OMG has also defined CORBAServices, a specification for providing directories, transaction support and other services similar to those offered by JavaBeans and COM [OMG01].

1.3.2 Barriers to Component Adoption

As mentioned above in Section 1.1, CBSE has yet to mature to the level of other areas of engineering and software component specifications are rarely detailed or complete. The current best practice for specifying a component's capabilities is with an interface definition language (IDL) provided by component standards along

```
module stopwatch {
  interface Stopwatch {
    void start(); // Start the stopwatch
    void stop(); // Stop the stopwatch
    readonly attribute double elapsedTime;
      // Attribute to store the elapsed time between
      // the last start-stop pair. This statement
      // generates no variable,
      // only a pair of set/get
      // statements for this variable.
     };
};
```

Figure 1.1: CORBA IDL description for a stop watch component [Ram98]

with informal documentation. Figure 1.1 shows a simple CORBA IDL description. Current IDLs are restricted to expressing only the syntactic aspects of a component's interface: types, operation names, parameters, return values and exceptions. This is a major obstacle that is preventing the widespread adoption of software components. Developers need more information to assemble component-based applications successfully. Without improving the current best practice, component usage will be restricted by the following issues:

- **Extra Effort.** Without a clear and concise description of the component's interface, effort is wasted in understanding and applying the component each time it is used. If too much effort is required, it may outweigh the benefits of component use.

- **Trust.** A major barrier to component use is a lack of trust. Poorly specified components do not provide developers with sufficient information to judge if the component can be used in a given context. Because unexpected component behavior and interactions are unacceptable for mission-critical systems,

developers may not be convinced to use a component, for example if it does not specify how it deals with certain types of invalid data or what external calls it makes or what resources it consumes [BJPW99].

- **Undocumented Features.** Some components are difficult to integrate. Integration problems are most often because of incomplete specifications. In order to use a component, system developers will test the component extensively to reveal unspecified properties. Troubles arise if the discovered properties no longer hold for a different or future version of the component. In practice, it is difficult to avoid dependencies [Szy97] on unspecified properties.

- **Misuse.** System developers complain that a lack of specification details cause them to make wrong assumptions about components. This is one of the major causes of trouble during integration and maintenance. [DAC00]. Component vendors often fail to specify their assumptions when developing the components.

  A commonly cited example of the dangers of inadequate specification and wrong assumptions is the costly Ariane-5 disaster [JM97]. The launcher crashed after forty seconds due to a floating-point error in a module reused from the Ariane-4 rocket. Although the module never failed during Ariane-4 missions, it crashed because Ariane-5 had a different trajectory. This was never tested, as the head contractor assumed it would operate based on the previous missions and the vendor's qualifications [Gar98].

- **Performance.** Currently, important component properties including storage and memory requirements, as well as response times are rarely specified in the documentation. This can lead to potential disaster as completed systems may be oversized and perform poorly [GAO95].

- **Location and Selection.** Component catalogs that allow developers to browse through various vendors' offerings are growing in popularity. Many firms are creating on-line *marketplaces* for components. Most of the available components provide services for user interfaces, security, communications and data manipulation [Wil01]. Unfortunately, these emerging catalogs have yet to add value to component specifications providing only a summary of the available documentation. As well, there are very few components of substantial complexity [Szy97, Wil00b]. Although this is a first step, there are still many challenges facing system developers who are attempting to acquire a component.

  Often, it is difficult to find components that meet the requirements or components that could be easily adapted. [DAC00]. In [Wil00a], Wilkes concluded that a method for comparing components is also needed. These issues remain pressing problems for many reasons.

  Current component specification approaches contain only the information required to use the component. As well, this information is typically stored in the proprietary format of a specific development environment or repository [Wil00b]. Without any further specification, there is very little indication of how a component behaves [GGM98]. Lamela in [Lam00] suggests that a lack of standards for describing component requirements and components themselves is preventing the growth of components, but Szyperksi [Szy97] states that there is currently no clear method for specifying components.

  The emerging component catalogs only offer simple search mechanisms, either based on a keyword search of components' available documentation or by domain category selection. Incomplete and inconsistent specifications make it difficult to find appropriate components because searches may exclude rele-

vant components that use differing terminology in their specification. As well, components that don't meet the requirements may be included in the results. Differing terminology also complicates comparison and adaptation processes. Without detailed information about the component, it is extremely difficult to evaluate the component short of integrating it with the system [Voa98].

- **Organization and Management.** In addition to the technical barriers to component use outlined previously, there are many other important issues that need to be addressed. These issues are not directly addressed within this thesis, but are outlined to provide a full picture of the challenges of CBSE.

  Most important are the risks associated with using software developed by a third party and, in most cases, without access to the source. Although [Tal98] focuses on the defense industry and its problems with COTS components, many of the same arguments apply to less critical systems in business [Wil01]. Issues such as reliability, assessment methods and vendor stability need to be addressed. Because reliability is so important, the cost of assessing COTS components outweighs any benefits, and with the longevity of many military applications, (e.g., 15 to 30 years for an airplane) vendor stability is important.

  From a development perspective, there are psychological barriers. System developers, like everyone else, often fear the new and things they cannot control [Lam00]. As well, many developers succumb to the the *not-invented-here syndrome* but fail to understand the cost of in-house development compared to COTS components.

1.4 Proposed Solution

To address some of problems and to enhance the appeal of component-based development, we propose to extend component interfaces to specify additional properties. A component's extended specification can be viewed as a *contract* between the client using the component, and the implementation of the component providing the services. The terms of the contract provide parameters against which the client's use and component's service can be validated. This ability addresses many of the problems described above, giving system developers benchmarks, assurances, and compositional analysis abilities before implementation and integration.

We have developed a framework that provides a basis for specifying and analyzing component contracts. Our approach attempts to unite the salient specification terms and features of existing work in the area of extended component interfaces and contracts. Because this is an emerging area, most approaches do not adequately address the needs of component users. Our framework attempts to provide a foundation to address these shortcomings. Specifically our approach:

- is generic, and can be adapted to various component standards and paradigms.
- attempts to address and include the widest range of specification terms that would be of interest to component users.
- is formally defined; removing ambiguity and increasing the potential for tool-support.
- provides a base implementation that can be used to construct and analyze contracts.

1.5 Thesis Outline

This chapter has introduced CBSE, its potential benefits and current limitations. We propose a framework that supports the specification and analysis of component contracts.

Chapter 2 introduces the origins of contracts within object-oriented programming and surveys a number of existing proposals for extending the interface of software components. The chapter concludes with an analysis and comparison of the surveyed approaches.

Chapters 3,4, and 5 describe the requirements, design and implementation of the framework respectively. The requirements are drawn from the recommendations, experience and techniques surveyed in Chapter 2. The framework is based upon another, formally defined, framework for the validation of software architectures. The elements and extensions to the original framework are covered in Chapter 4. Details of the implementation in XSB Prolog/XMC model checking environment as well as example uses are provided in Chapter 5.

Finally, our conclusions and recommendations for further work are presented in Chapter 6.

Chapter 2

Component Contracts

2.1 Introduction

This chapter introduces the origins of extended interface specifications and contracts within the area of software engineering with a focus on Meyer's popular "Design-by-Contract" [Mey92]. To determine the requirements for our framework, an extensive survey of approaches is presented in this chapter. From this survey, we have identified the common and important specification terms and features to be included in the framework.

2.1.1 Extended Specifications

From Section 1.3.2, it is clear that strengthening a component's interface specification could potentially alleviate many of the existing obstacles to component usage. We believe a complete and concise specification of the component's interface is essential to provide developers with the knowledge they need to construct component-based systems successfully.

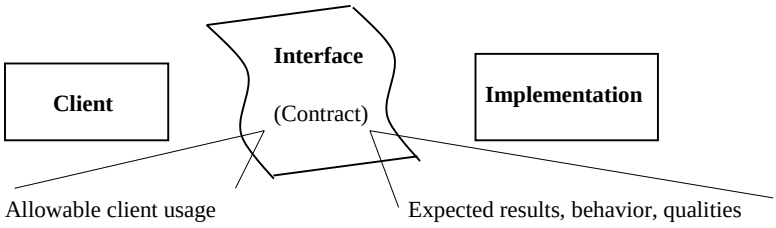

Figure 2.1: A Component Interface as a Contract

Because CBSE has so many expected advantages and applications, research in the area is ongoing in different fields. Within software engineering, software component technology is being developed in the fields such as software architecture, formal methods, object-oriented and agent-oriented systems, reuse and requirements specification. Industry attempts are much less ambitious and focus on standardizing small sets of non-functional requirements. progress is hampered by the competing standards offered by various organizations.

A detailed interface specification can be used as a basis for a contract. As mentioned in Section 1.4, we can use the terms of the interface, or contract as parameters against which the client and implementation can be verified and validated. In Figure 2.1, the contract terms describe the allowable usages of the component by the client, and the expected results and behavior of the implementation. These contracts can address many of the issues discussed previously by providing component users with assurances, guarantees and analysis abilities.

The following sections introduce the background and roots of contracts in object-oriented systems, which face many of the same issues as component-based systems at a finer level. Following the introduction, a survey of proposals and approaches for component contract specification and implementation is provided.

When considering an extended interface specification as a contract, it is impor-

tant to avoid over-specification, including more detail than is required. This may restrict both the applicability of a component and its possible implementations [BS97].

2.2 Object-Oriented Contracts

Many of the issues that affect software components have arisen earlier in object-oriented programming (OOP). This section introduces the two most significant contributions in the area of OOP and contracts. The notion of contracts in software development is attributed to Meyer [Mey92]. His original motivation in developing contracts was to reduce the complexity and increase the reliability of OO systems. Another contribution, the OO-contracts of Helm et al [HHG90] focus on specifying the behavior and interactions between objects in a system.

2.2.1 Design by Contract

Meyer introduced the idea of "Design by Contract" (DbC) in [Mey92] to increase the reliability and correctness of object-oriented (OO) software by introducing a set of principles to deal with software errors systematically.

Central to DbC is the notion of a contract. When developing a system, one may choose to program a solution to a problem, or "contract" it out to a subroutine or class.

In life, a contract between two parties, a client and a supplier includes obligations each party must fulfill to receive the benefits of the agreement. For example, the contract between a person, a client ordering fast food in a restaurant, the supplier: the client is obliged to wait in line, prepay for his meal, seat and clean up

Party:	Client	Supplier
Obligations:	Must prepay, Self-serve	Prepare meal quickly
Benefits:	Low-cost, Fast service	No wait staff, No unpaid bills

Figure 2.2: Contract details for fast food restaurant

after themselves to receive the benefit of a low-cost meal and little waiting time. On the other hand, the restaurant is obliged to have the meal ready quickly to receive the benefits of payment without dealing with wait staff and unpaid bills. For clarity, this example contract is summarized in Figure 2.2. A contract clearly specifies what a client must do in order to receive the benefit, and protects the supplier by specifying the conditions for which it is responsible.

Applying contracts to a software context, clients (callers) are protected by specifying what they must provide to receive a result from the supplier (callee). Suppliers (callee) are protected because the contract specifies what input is acceptable and is not liable for invalid input.

Because software is correct only relative to its specification, Meyer proposed implementing parts of a system's specification as correctness formulae (also known as Hoare triples) which are expressions of the form:

$$\{P\}A\{Q\}$$

where, for any execution of A, P holds initially and terminates in a state where Q holds. P and Q are referred to as the precondition and postconditions respectively. A caller must meet the obligations of the pre-condition to receive the benefits

provided by the callee. The callee, in-turn, upon receiving input meeting the pre-condition must meet the requirements of the postcondition. From the restaurant example:

{Customer prepays}{Meal prepared}{Wait time less than 5 minutes}

A simple example, consider the a function that maps a positive value between 1 and 100 to a range between 1 and 10:

$$\{0 < x \leq 100\} y := map(x) \{0 < y \leq 10\}$$

The pre-condition requires that the value of x be between 1 and 100 to ensure the postcondition of the *map* function returning a value between 1 and 10. DbC underlies the design of Meyer's Eiffel programming system [Mey91] and pre- and postconditions are implemented as assertions. Violated assertions generate exceptions during run-time. Invariants are another type of assertion that are applied to classes to restrict the set of allowable states that an object may have. Class invariants must hold upon object creation as well as preceding and following the invocation of a public method.

In Eiffel, assertions are specified as Boolean expressions. Functions can be used to emulate quantifiers and provide other validation services. Meyer acknowledges the weaknesses and drawbacks of this approach compared to a full formal specification language, but argues that it is an acceptable tradeoff in terms of reliability and simplicity for industrial software development [Mey92].

The main way Meyer's contracts contribute to software reliability is by providing a systematic method for implementing the terms of the software specifications leading to more potential errors being checked and detected. As well, DbC eliminates

the need for *defensive programming*, a technique that encourages ad-hoc handling of inputs to take care of all possible invalid terms. This leads to less complex and more readable code, further reducing the chance of error in the code [Mey97]. A detailed list of other advantages is presented in [MHKM95].

Contracts in object-oriented systems must hold in situations with inheritance and polymorphism and this leads to the notion of "sub-contracting." It is possible for a subclass to have a different effect than its superclass which can lead to potential troubles when a subclass is used in substitution. These situations can be addressed by forcing the subclass to adhere to the prime contractor, its superclass. A subclass is valid if the pre-condition is *weakened*, requiring less than its superclass does, and its postcondition is *strengthened*, returning at most what the super class does. From the previous example, *map*() could be replaced with:

$$\{x > 0\}y := map(x)\{4 > y > 9\}$$

This maps all positive values, which is weaker than the original limit that restricted values up to 100. The postcondition has been strengthened however, only mapping to values 5 through 8.

Regardless of the limitations to the expressiveness of this approach discussed below in Section 2.3.1, DbC has become very popular. It has been widely adapted to other programming languages, most notably Java. A survey of Java implementations can be found in [FLF01]. As well, DbC has been adapted for use in software specification notations, such as the Unified Modeling Language's (UML) Object Constraint Language (OCL) [WK99]. DbC has provided inspiration for much of the research in the area of component contracts.

ContractObserver
Subject **supports** [
value : Value
SetValue(val: Value) $\rightarrowtail \Delta value\{value = val\}$; Notify()
Notify() $\rightarrowtail \langle \| v : v \in Views : v \rightarrow$ Update() $\rangle$
Attach(o: Observer) $\rightarrowtail \{v \in Observers\}$
Detach(o: Observer) $\rightarrowtail \{v \notin Observers\}$
]
Observers: Set(Observer) **where each** Observer **supports** [
Update() $\rightarrowtail$ Action()
Action() $\rightarrowtail$ Subject $\rightarrow$ value {*Observer* **reflects** *Subject.value*}
]
invariant
$Subject.Setvalue(val) \rightarrowtail \langle \forall v : v \in Views : v$ **reflects** $Subject.value \rangle$
instantiation
$\langle v : v \in Views : \langle$ Subject $\rightarrow$ AttachView(v) $\rangle$
end contract

Figure 2.3: Contract for Observer Pattern

2.2.2 Contracts

Another notion of object-oriented contracts was proposed by Helm et al [HHG90] that provides more detailed terms than Meyer's DbC. Helm et al noticed that the behavior of an object cannot be inferred from its interface, leading to design and reuse problems. Contracts formalize the behavioral relationship between objects and definesa set of participants and their obligations.

Contractual obligations consist of *type obligations* requiring participants to support a certain interface and *causal obligations* which indicate an ordered sequence of actions that must be performed. As well, contracts define invariants and preconditions that participants must maintain.

Figure 2.3 is an example of a contract that specifies the requirements for a group of objects, a subject and its observers acting as the observer design pattern. Besides

the elements of DbC: pre-conditions, postconditions and invariants, contracts go further, indicating the protocol each object must adhere to.

2.3 Component Contracts

This section contains a survey of the area of component contract specification. Much of the work is based on the earlier work of object-oriented contracts and draws from many fields of software research. This includes a large body of work and we limit the survey to research directly concerned with the interface specification of COTS components – the user cannot access the underlying implementation of the component. The approaches surveyed next attempt to address the issues with software components described in Section 1.3.2.

2.3.1 Greybox Components

In [BW97] Büchi and Weck introduce *grey-box components*. Current interface descriptions can be considered *black box* specifications, only describing the behavior of the component in terms of the pre- and postconditions of its operations. Unfortunately, this approach cannot describe the complex interactions of components, for example, when callback functions are involved. On the other hand, revealing the full implementation of the component provides the user with too much detail. Commercial components should reduce the complexity of a system by hiding details from the user and too many details may lead to overspecification, restricting possible replacements or enhancements. This problem is also discussed in detail in [BS97].

Büchi and Weck's work is similar to the approach developed by Helm et al's

```
interface ITextModel {
  private seqof char text;
  private setof Observer observers;
  ⋮
  Delete (int pos);
    pre 0<=pos && pos<len(text)
    post (all int i: 0<=i && i<pos:
              text[i]==text@pre[i]) &&
         (all int i: pos<=i && i<len(text):
              text[i]==text@pre[i+1]) &&
         len(text)==len(text@pre)-1
         /* o.DeleteNotification(pos)
         called for all o ∈ observers */
  ⋮
}
```

(a)

```
⋮
Delete(pos: N) {
  pre: 0 ≤ pos < ‖text‖ {
  text' = text[0...pos-1] +
          text[pos + 1...‖text‖-1]
          do(o in observers) {
            o.deleteNotification(pos);
          }
}
⋮
```

(b)

Figure 2.4: Text model delete method. (a) shows pre-/postcondition specification (b) grey-box specification

OO contracts [HHG90]. Unlike the OO contracts, Greybox components are more flexible, differentiating between *mandatory* calls, which change state and *enquiry* calls that do not. Because enquiry calls have no side effects, only mandatory calls are required for contract compliance. Greybox components also support operational pre-conditions. As well, for increased usability, the grey-box specification language is designed as an extension of Java syntax thus appealing to many practitioners.

The following example demonstrates how the grey-box approach addresses the limitations of standard black box specification. Consider a component that manages text, and follows the observer design pattern by allowing interested observers to be notified when the text is modified. Figure 2.4 (a) shows a pre/postcondition of the *delete* method in Java, with common extensions. The pre-condition requires

that the character to be deleted is within range; the postcondition requires that the modified text sequence contain the same characters as the original excluding the deleted character. But, pre- and postconditions cannot indicate when the character is deleted. Is it deleted before the observers are notified? Or after? The grey-box specification shown in Figure 2.4 (b), using additional constructs addresses and indicates the character will be deleted before the observers are notified. The resemblance to the contracts of Helm et al is clear, except for the use of an imperative, Java-like notation.

In [BW99], Büchi and Weck provide a formal definition of correctness of implementations with respect to the grey-box specification language.

2.3.2 Framework for Component Interface Specification

In [Han98] and [Han00], Han presents a comprehensive framework for specifying component interfaces and their protocols. The framework addresses five aspects of interface specification: signatures (syntax), configuration (structure), behavior (semantics) interaction (protocols and constraints) and quality.

The *signature* of the interface defines the syntax of the attributes, operations and events accessible to users of the component. *Configuration* aspects describe the structure of the component in different contexts. Configurations are supported by specifying ports reflecting the roles the the component provides and requires within a composition. The *behavioral* aspect addresses the operational semantics of the operations through pre- and postconditions. *Interaction constraints* define the protocol of a component. The constraints guide the user of the component, and must be observed to prevent errors or unpredictable behavior. The final aspect characterizes the non-functional properties of the component. Quality characteris-

```
COMPONENT CM {
 ...
 CONFIGURATION ring {
 ...
  PORT ms_management {
   PROVIDE
    report_signature(IN MS m_id, IN Sig sig);
    report_alarm_attributes(IN MS m_id, IN Alarm alarm);
    report_perf_attributes(IN MS m_id, IN Perf perf);
   REQUIRE
    BOOLEAN enabled;
    request_signature;
    set_alarm_attributes(IN Alarm alarm);
    set_perf_attributes(IN Perf perf);
   SUBJECT_TO
    ^request_signature -> report_signature;
    (enabled) :>(report_alarm_attributes,
                 report_perf_attributes);
  }
 ...
}
```

Figure 2.5: Example of Interface Specification Framework

tics include the component's performance, reliability and security. Figure 2.5 shows part of the interface specification for the central manager (CM) of a telecommunications system. When connected in a ring setting, the CM has many roles. The figure provides the details of a simple managing services (MS) port. The CM provides three operations to the MS, and requires the MS to support one attribute and three operations. Constraints indicate that `request_signature` followed by `report_signature` are called upon connection. The second constraint requires `enabled` to be set before the report operations may be called.

In [Han00], Han argues that this framework provides a practical solution to the

interface specification problem because its notation is close to that of existing IDLs. The "light-weight" constraint-based approach to describe component behavior allows for incremental specification of the component.

2.3.3 CoCoNut

Reussner in [Reu01] presents a new model for describing component interfaces based on an extension of finite state machines to describe the component's protocol.

The model focuses on component composition issues including protocol adherence, and component adaptation in different contexts, interface matching and compatibility. The CoCoNut interface model consists of a *provides-interface* and an *requires-interface*.

The provides-interface models the services the component offers. This interface includes the classical interface such as CORBA IDL, as well as valid call sequences. Sequences are modeled with the *provides-automaton*. The automaton defines a subset of all allowable sequences to the offered services. Figure 2.6(a) shows the allowable call sequences to a VideoMail component.

The requires-interface models the possible call sequences the component makes to external services. This is modeled in the *function-requires automaton*. Figure 2.6(b) shows the function-requires automaton for the `play` service of the VideoMail component.

Inserting the function-requires automaton into the provides-automaton forms the *component-requires automaton*. The complete VideoMail *component-requires* automaton is shown in Figure 2.6(c). Assuming a similar automaton for the `stop` service, the combined automatons form the component behavior. Based on the languages recognized by the automatons, Reussner outlines an algorithm that can

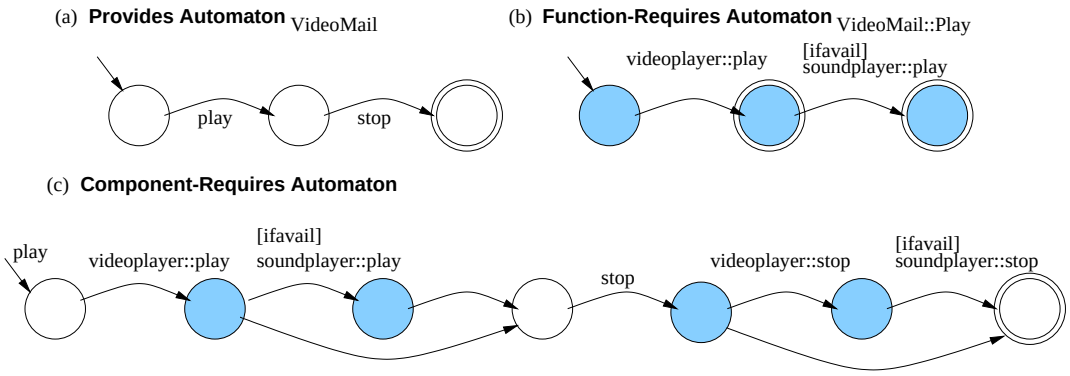

Figure 2.6: Automatons representing: (a) the services provided by a VideoMail component, (b) the required services for the VideoMail::play component and (c) the combined automatons representing the behavior of the component.

be used to adapt the component provides interface, depending on availability of services. Another algorithm can ascertain whether two components are equivalent, or if one component can substitute for another. For composition purposes, the minimum required functionality can be described by an automaton. As well, unwanted functionality, such as access to the file system which could cause a security problem is also supported.

For the prototype system, Enterprise JavaBeans (EJBs) [Ber00] are the target component standard. Developers do not explicitly create the automaton, but place annotations in the source code which is then pre-compiled to create the automatons.

2.3.4 Requirement/Assurances Contracts

Rausch in [Rau00] presents a system for specifying components and their composition. The work focuses on validating the behavior of a system as its constituent components evolve. The semantics of a component are usually described using a

```
COMPONENT TextBox
 REQUIRES INTERFACE Observer
  WITH METHODS
   update() : void
 ASSURES INTERFACE TextModel
  WITH LOCALS
   observers : Set(Observer)
   text : String
  WITH METHODS
   getText() : String -> return text
   addText(t : String) : void ->
    new text {text = text + t};
    < obs: obs in Observers :
           obs -> update() >
```

```
RA-CONTRACT HelpListContract
  INSTANTIATION
    l: ListBox
    t: TextBox
    t.TextModel.observers.add(hl)
    l.ListModel.observable = ht
  PREDICATE MAPPING:
    REQUIRED l.Observable
    ASSURED BY t.TextModel

RA-CONTRACT HelpTextContract
  ...
  PREDICATE MAPPING:
    REQUIRED t.Observer
    ASSURED BY l.ListModel
```

Figure 2.7: Example Requirements/Assurances Contract

number of documents: design documents such as class diagrams, IDL specification, interaction documents such as UML sequence diagrams and implementation documents. This separation of component information can lead to difficulties updating the information and impact assessment when a component changes because the dependencies are not explicit.

To address this problem Rausch introduces requirements/assurances contracts. Each component is described individually indicating what it requires from its environment, and what the component assures it will provide, given its requirements are met. The left side of Figure 2.7 shows the specification of a `TextBox` component that requires observers with an `update` method. Given observers, the `TextBox` assures it will provide two methods, and that when text is added, observers will be notified. Contracts are formed between components and include component instances, their connections and an explicit mapping between required and assured interfaces. The right side Figure 2.7 shows two contracts formed between a `TextBox`

and an observing `ListBox` that gets updated when the text changes.

To ensure contract conformance, the system designer has to "prove" the correctness of the syntax and behavior of each member of the contract. Rausch proposes to start with the conjunction of all predicates assured in the contract and ensure it ends with all required predicates.

If a component changes, the contract can be re-checked to ensure all the required predicates can be reached. Requirements/assurances contracts provide a way to ensure designers are aware of the consequences of component changes.

2.3.5 Contract Aware Components

In [BJPW99], Beugnard et al propose a four-level contract for components to increase trust. In mission-critical systems, it is important to judge whether a component can be used correctly in a given context. A component contract provides the parameters against which the component can be validated.

The first level, basic contracts, encompass current IDLs and provide the syntactic elements necessary to operate with a component standard. To define the requirements and effects of the operation precisely, behavioral contracts are defined using Meyer's design by contract [Mey92]. The next level, synchronization contracts, define the component in distributed and concurrent contexts. Synchronization contracts are defined by instances of strategies, such as mutual exclusion. Quality-of-service contracts form the fourth level and address features of the component beyond their behavior such as response time and result quality.

The contracts of Beugnard et al are more than detailed interface specifications, but are entities themselves and are based on the framework for contracts in distributed systems [LPJ98]. In a system where components are bound dynamically,

contracts are revealed through a query to the component. Depending on the context, or the required services, a component may present many different contracts to the component. As well, certain terms of the contract may be negotiable. For example, a client may request an increase in the precision of a result. A client may only use the services of a component after a contract has been selected.

The exact form of a contract is not discussed but the authors suggest UML formalisms encoded in XML. To enforce the contracts, a run-time monitoring system, is implemented as an extension to the underlying component middleware.

2.3.6 Contract-based Design

In [Gie00], Geise introduces contract-based design to address the the problem with synchronization in component-based systems. Callback routines break the strict layering where outgoing calls can only be made to a lower-level entity. In the simple case, self-recursion can occur where a component invokes a call that, in-turn invokes its calling component leading to deadlock. In a concurrent system, a component that is currently blocked, may prevent other components from delivering their callbacks, once again leading to deadlock.

To address this problem, Geise proposes to specify components using Object Coordination Nets (OCoNs) [GGW99], a variant of Petri Nets [BRR87], for specifying the behavior and synchronization aspects of object-oriented systems. According to Geise, contracts should not only include a description of protocols and coordinating sequences but a functional specification that details the pre- and postconditions, and non-functional properties. Contract-based design is structured as an extension to UML. A contract is the combination of a standard interface that describes the signatures and a protocol net to describe the interactions. Figure 2.8 shows con-

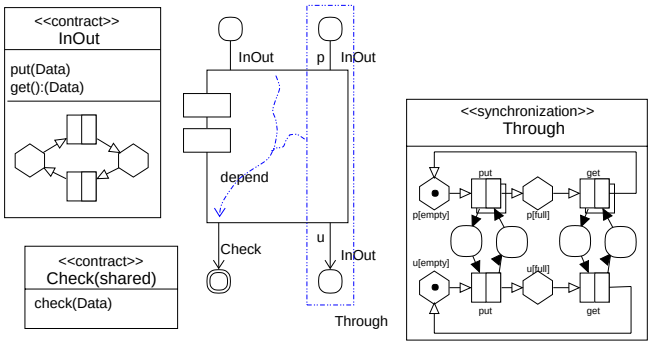

Figure 2.8: Example contract from [Gie00]

tracts defined as a UML <<contract>> stereotype. Components have both *provide* interfaces (denoted p), which are the services the component provides and *uses* interfaces (denoted u), indicating the services the component requires. An exclusive contract (denoted by a single circle) means each contract is served by a distinct instance of the component. A shared contract (denoted by a a double circle) means the contract is shared among many components.

Simple contracts are contracts that exist with no restrictions. Otherwise their operations are further restricted with the introduction of the <<synchronization>> stereotype by introducing synchronizations with other contracts of the same component. In Figure 2.8, the Through synchronization specifies how the operations of the provides contract are mapped to the uses contract of the same type. The dashed area indicates the scope of the synchronization.

Implicit dependencies relationships can be shown when an explicit synchronization contract is inappropriate. These are indicated with a dashed arrow. During composition, deadlock can be detected when a cycle is found among the contracts and their synchronizations.

```
interface CAR {
  attribute CAR-MODEL model;
  attribute set<CAR-OPTIONS> options;
  attribute OWNER ownedBy;
  attribute seq<TRANSFER> history;
  attribute MANUFACT madeBy;
  deliver(in CAR object, in MANUFACT src,
          in DEALER dest, in DATE time);
  sell(in CAR object, in OWNER src,
       in OWNER dest, in DATE time);
}
```

```
CAR ⊑ (and
      (the model CAR-MODEL) (at-least 1 model)
      (all options CAR-OPTIONS)
      (at-most 100 CAR-OPTIONS)
      (the serialNo INTEGER) (at-least 1 serialNo)
      ⋮
CAR ⊑ (the deliver DELIVER)
;; object co-located with the source
CAR ⊑ (same-as deliver.object.location
                deliver.src.location)
;; preconditions
;; car maker delivers it to the dealer
CAR ⊑ (same-as madeBy deliver.src)
;; owner can only sell car
CAR ⊑ (same-as madeBy deliver.src)
;; destinatation of sell is new owner
;; postconditions
CAR ⊑ (same-as ownedby sell.dst)
```

Figure 2.9: Example CORBA CAR Management service from [BD99]

2.3.7 Description Logic

In [BD99], Borgida and Devanbu propose description logics (DLs) as a formal basis for describing components. Borgida and Devanbu argue that current IDLs do not adequately fufill their role in representing the basic domain constructs and proper use of the component. To address the ambiguity and limited tool support of current interface descriptions, a formal approach is described that provides data invariants, pre- and postconditions and behavior descriptions that can be statically checked.

Although many formalisms such as the UML OCL exist for describing the elements of Borgidas and Devanbu's approach, their expressiveness prevents them from being decidable. DLs have their roots in the AI field of knowledge representation and reasoning. DLs are a decidable subset of first-order predicate logic and can be quite efficient.

The basic ontology of DLs are centered around objects, concepts and attributes, similar to the object paradigm of many component standards. On the left side

of Figure 2.9, a simple description of a car management service in the CORBA IDL is shown. On the right an equivalent encoding is provided in the EXPRESS description logic.

The reasoning capabilities of DLs can be used to determine if the client matches with the user's needs. Consistency checking ensures that pre- and postconditions are consistent with the invariants. Although DLs are limited, they provide a basis for checking operational and data specifications.

2.3.8 CORBA Interfaces with π-Calculus

In [CFTV00], Canal et al describe the application of the π-calulus [Mil99] to specify the expected order of calls to a CORBA Object.

The π-calculus is a process calculus, designed to describe dynamic concurrent systems and is used by Canal et al to model the interesting behavior of CORBA objects. The syntax of π-calculus is extended to bring it closer to an object-based notation and increase readability. The protocol information is stored separate from the current IDL definition to ensure compatibility with existing CORBA implementations and allow for different protocols to be associated with the same interface.

The formal basis of the π-calculus provides a number of advantages when statically checking the protocol conformance during design time. Properties such as deadlock-freedom within a system can be verified by modelling an application that uses the CORBA objects in π-calculus and then proving the property.

Based on the work presented in [CPT99], Canal et al demonstrate how the substitutability of an object can be verified. An object can be replaced only if the replacement maintains the same behavior with any existing clients. Object compatibility ensures that two connected objects interact properly.

To facilitate run-time checking and to ensure that implementations conform to their specifications, the use of interceptors, a CORBA facility is employed that allows programmers to insert code before or after an operation is executed. This can be used to create a run-time trace and ensure messages are compatible with the protocol behavior.

2.3.9 Component Contract Templates

In [DAC99a], Dong et al propose a template for software component specification. The information provided through a template provides assistance addressing many of the issues discussed in Section 1.3.2. The template goes further than many approaches which are limited to functional descriptions such as grey-box [BW97] to include many other properties.

The functional interface of a component should include the standard signature information as well as supporting documentation including class diagrams, state transition diagrams, data flow diagrams and collaboration diagrams. Component assumptions should not only include constraints such as the preconditions, but architectural design assumptions. Behavior should be specified using temporal properties.

Non-functional properties are also a key component of the template. These properties include performance, reliability and concurrency. Component templates should state the environmental requirements such as supported operating systems and languages and component standards.

Component vendors should include information about their products to assist users in making decisions. This information includes details about related products, collaborating components and examples.

Currently, the focus of the templates are the content, and their ability to address the needs of users. From a vendor's perspective, any new versions of the component should maintain the specification of the original component's template. If any changes are made, they should be highlighted. The information provides users clear guidelines for selecting and integrating the component.

2.3.10 Additional Work

In addition to the approaches already described , other proposals, findings are described next. Industrial approaches are not considered because they are still lacking any extensions to address the issues covered in Section 1.3.2 [Gil01].

Knowledge-Based Systems

In recent years, there has been work directed towards the development of methods for reusing the problem-solving methods (PSMs) of knowledge-based systems.

The Unified Problem-solving Method Description Language (UPML) was developed as a way to document PSMs to enable their reuse in the semi-automatic configuration of problem solving systems [GMF99].

A base ontology is defined to describe the PSMs. This ontology includes elements (sorts), a set of predicates based on the elements and a set of axioms to describe the relationship between the predicates.

A PSM, based upon the terminology defined in the base ontologies may include auxiliary definitions. A competence definition specifies the input and output parameters, the pre- and postconditions and any required subtasks. As well, an operational definition outlines the high-level operation of the PSM and where the

subtasks are used. Subtasks are defined in terms of their input and output parameters, a precondition and final goal.

Using the LOOM knowledge representation system [Bri93] which is based on the decidable subset description logic, a PSM can be automatically checked using a matching rule to see if it conforms to a task specification through concept subsumption.

Workshop on Object Interoperability

Recently, the Workshop on Object Interoperability (WOI) [VHT99, VHT00] covered many of the issues presented here. During the workshop, the participants identified four levels of information required from an object interface definition: signatures, protocol and behavior, functional and conceptual semantics and quality properties.

It was also recognized that effective specifications should be precise and based upon an underlying formal model. However, to increase practitioner acceptance, formality should be hidden as much as possible through the use of tools.

2.4 Comparison and Analysis

Amongst the various approaches for extending the specifications of software components a number of common elements emerge. In this section, we analyze the surveyed approaches for contract terms, specification structures and capabilities to address the issues outlined in Section 1.3.2. Based on the proposed approaches, we have identified a number of different aspects that should be considered for inclusion within a component contract.

2.4.1 Framework Requirements

One of the major problems with existing component documentation, discussed in Section 1.3.2 is ambiguous and incomplete natural language specification. Although not all of the surveyed approaches have well-defined semantics, all recommend more formal specifications to create concise specifications and reduce the possibility of misinterpretation or inconsistencies. Incomplete specifications are addressed by requiring that certain information be provided about the component.

To increase acceptance by developers, Büchi and Weck [BW97] recommend that formal specifications be 'hidden' from users. Their approach is to provide Java-like syntax and semantics for their specification language. Tool support can also increase acceptance among users by automating tasks and providing alternate abstractions.

2.4.2 Contract Terms

The most important aspect of a component contrac, is the information it provides to the user. Based on the survey of current proposals, the following areas have been identified for inclusion within a component contract:

Signature

As a basis, all approaches extend current IDLs and provide signature information. The contents of a component's signature vary depending on the standard to which it conforms to. Current IDLs only specify the services provided by a component, but components often rely on other components and services and many approaches (e.g. [Reu01]) include the signatures of a component's required services.

Operational semantics

Operational semantics include specification of the pre- and postconditions on operations and invariants that restrict the values of results and attributes. These are usually specified using the assertion language defined by Meyer for DbC [Mey92] or UML OCL [WK99]. Because these notations in general are undecidable and cannot be statically checked, most approaches do not include them or provide them only as structured comments for users' benefit or for run-time checking. The approach taken [BD99] is to restrict the language to a decidable subset, which can be statically checked at the cost of expressiveness.

Protocol and Behavior

A component's protocol and behavior are the two most commonly addressed areas in the surveyed approaches. The protocol of a component represents the allowable orderings of signature invocation to the component. The behavior of a component describes the results of a single invocation or sequence of invocations. A component's behavior may result in a change in value, an outgoing call or other action.

A variety of techniques are used to specify the protocol and behavior of a component. In [Han00], temporal logic statements are used, CoCoNut is based upon an extended finite state machine [Reu01], the grey-box approach uses statement specifications [BW99], Geise's approach [Gie00] is based on Petri Nets and the π-calculus process algebra is used in [CFTV00]. Preconditions can also be used to indicate the proper ordering of invocation on a component [VHT99], but have serious limitations [BW97].

Non-functional Properties

Despite the recommendation that non-functional properties should be included within a component contract [Han98, BJPW99, DAC99a, Reu01], there is currently very little research directed towards the non-functional requirements and software components. This is due to the difficulty recognizing and quantifying non-functional properties [VHT99].

In [BD00], Beus-Dukic outlines three areas of non-functional requirements for COTS components. *Architecture requirements* address a component's ability to be integrated into a system. Non-functional requirements associated with architecture include performance characteristics, reliability, security, reusability and portability. *Domain requirements* describe properties related to the component's environment. These include standards compliance, hardware requirements, related components, and design assumptions. *Organizational requirements* focus on the aspects of the vendor and customer. These include the vendor's credentials and market stability, standards conformance, upgrade policy, popularity, software development practices and support record.

Dong [DAC99a] lists many of the same properties described by Beus-Dukic. Beugnard [BJPW99] provides latency, throughput and precision as examples of non-functional properties of a component.

The difficulty specifying of non-functional properties is clear. The testing and procedures required to quantify and describe, for example, reliability can be expensive and something vendors may not even want to share. Voas [Voa01] proposes independent third-party laboratories to certify components. There are also difficulties in quantifying the performance of a component. Szyperski [Szy97] suggests that the big-O complexity of a component be specified to indicate its time and space

costs. Sitaraman [Sit01] outlines a number of shortcomings of this solution. In [WSHK01], given the latency of individual components, Wallnau et al were unable to predict the latency of a composition because of differing component dependencies. The ISO 9126 software quality standard is aimed at addressing these issues, but still requires investigation [SDF00].

Although there are many difficulties specifying the non-functional properties and requirements of components they are essential to users during the selection and evaluation of components. Often, the non-functional properties set components with similar functionality apart. When constructing a secure server system, all components that do not adhere to the selected security model can be eliminated.

Meta-information

One aspect of a component's specification that is overlooked by many approaches is the specification of meta-information to describe the elements of the component contract [VHT00].

Similar to the approach taken by the UPML [GMF99] and other agent-based approaches, ontologies are provided to describe the aspects of a specific domain. An ontology is a common language and is used to describe concepts within a domain [O'L00]. Ontologies have already been defined for a number of domains from healthcare to manufacturing [Ont01]. Describing the component in terms of defined domain ontologies removes potential ambiguities about their use. This eases the process of component selection, providing the user with a way to query a component description based on the ontologies it supports [TN99].

Roles and Dependencies

The focus of current IDLs is on specifying the provided services of a component. But, often components require access to other components or services to operate successfully [Szy97]. This has also been recognized and supported in a number of the approaches.

In both [Han98] and [CFTV00], the functionality of a component is divided into roles, each of which encapsulates a specific piece of functionality. This is analogous to the JavaBeans or COM standard, where a component may implement multiple interfaces.

EJBs also provide points of interaction through either the home or remote interface. The home interface of a component provides management functionality such as instance creation and deletion. The remote interface provides the component services.

2.4.3 Capabilities

In addition to contract terms, various approaches describe different ways that contracts can be used. At a minimum, every approach attempts to provide more concise, complete usage documentation for developers, but can also be used to determine how the component will behave when integrated with the rest of the system before integration, execution and testing.

Compatibility

One important capability is to ensure that two interacting components are interoperable. At the signature level this is already supported by existing IDLs through

type checking. Ensuring that the protocol and behavior of a component is compatible with other components is more challenging [VHT99]. Another important ability is to check whether one component can be substituted for another while maintaining the system requirements.

The Greybox [BW99] specification is based on the Refinement Calculus [BvW98] and provides a formal basis for determining if one component refines another. Similar to Meyer's DbC [Mey92], a component refines another if it provides 'more' to the client, while stratifying the substitution property. Geises' contract-based design [Gie00] also supports the refinement of components.

In [CFTV00] the components protocol is specified using the π-calculus. To check if one component can be substituted for another, bisimulation of the two component specifications produce identical behavior. Because a component may provide more, but have the same behavior as another component, a less restrictive form of protocol substitution was proposed in [CFTV00].

The CoCoNut model [Reu01], which is based on extended finite state machines, provides a number of component substitutability tests for components. The equality test determines if two components have the exact same protocol and behavior. Substitutability is a weaker test that checks if one component refines another. A similarity test is also proposed that provides a measure of how similar two components are.

The DL approach employed in [BD99] can be used to ensure compatibility between limited pre- and postconditions. By defining PSMs in terms of domain ontologies, Gaspari et al [GMF99] can construct reasoning tasks semi-automatically.

Compositional Reasoning

Component contracts are also used as a basis for reasoning about compositions of components. Often, developers are interested in determining before integration if the component instances interacting together will satisfy the system requirements.

Most approaches are interested in verifying the temporal properties of systems. Progress properties require that some event occurs at some time in the system whereas safety properties require that a specified event never occurs.

During contract-based design [Gie00], there is no explicit difference between components and their instances. Systems, or combinations of components can be connected with events or procedure calls. Since protocols are described using Petri Net, a well-defined formal method, it is possible to apply and check temporal properties such as deadlock avoidance. Canal et al take a similar approach using the π-calculus [CFTV00]. An application process can be created by instantiation component processes and a simulator can be used to verify safety properties.

The purpose of the contracts defined by Rausch [Rau00] is to ensure that as components evolve over time, system requirements still hold. Component specifications are composed together to form contracts. Contracts model a possible composition of components and include instances, connections, and predicates that reify the connection between the client and provider. This is followed by a proof, based on the protocol, indicating all client behavior is supported by the provider.

In addition to protocol conformance and temporal properties, non-functional properties of compositions can also be checked. [VHT99].

Contract Enforcement

Most research has been focused on the development of techniques that can statically verify the composition of component contracts. Unfortunately, since it is not feasible to verify component implementations formally against their specification, at run-time the specified behavior is not necessarily guaranteed.

Meyer's DbC addresses this problem by inserting assertions into the code. Any error conditions that arise cause the assertion to fail and generate an exception. Canal et al [CFTV00] for the case of CORBA recommend the use of interceptors to insert any checking or additional required information.

Although run-time checks have been proposed in a number of sources [Szy97, VHT99, BJPW99] it has not been the focus of current research. Log file analysis [AZ00] is a potentially useful technique that could be applied in this area. A log file of all events generated by a system, gathered from testing data or during run-time, can be matched against state-machines that define the behavior of a component or group of components. This could easily be adapted from any of the state based contract specifications such as [Gie00, CFTV00].

2.4.4 Limitations

There are currently no well developed systems for specifying and reasoning about software components. All of the approaches surveyed are proposals, theoretical frameworks or in early stages of development.

Many of the proposals lack formal semantics (e.g. [Rau00, Han98]), limiting the usefulness of specification outside of a documentation tool. Others, that have a formal basis, are not coherent and do not yet provide any notion of a framework

that could be useful to developers. As well, most of the approaches only address one aspect of component contracts.

2.5 Summary

This chapter provided an overview of the existing proposals for extending existing component interfaces. Comparing the different approaches reveals a number of common terms that belong within a component contract as well as functionality that should be provided with a framework to specify components. Despite the advances, current proposals still have a number of limitations.

Chapter 3

Framework Requirements

3.1 Introduction

This chapter describes the elements and the requirements of a framework for specifying analyzing component interface specifications or contracts. The framework scope is shown in Figure 3.1. There are two goals: provide a basis for specifying component contracts and provide methods for analyzing various aspects of the contracts.

Our notion of components is based on the model presented in [CD00]. This model is supported by well-known component standards including Microsoft's COM [Mic01], Sun's Javabeans [Sun01] and OMG's CORBA standard. Based on this model, we present the required structural elements of an interface specification.

Based on the recommendations and approaches of existing work surveyed in Chapter 2, the requirement and terms of a contract as well as the framework's analysis abilities are discussed in detail.

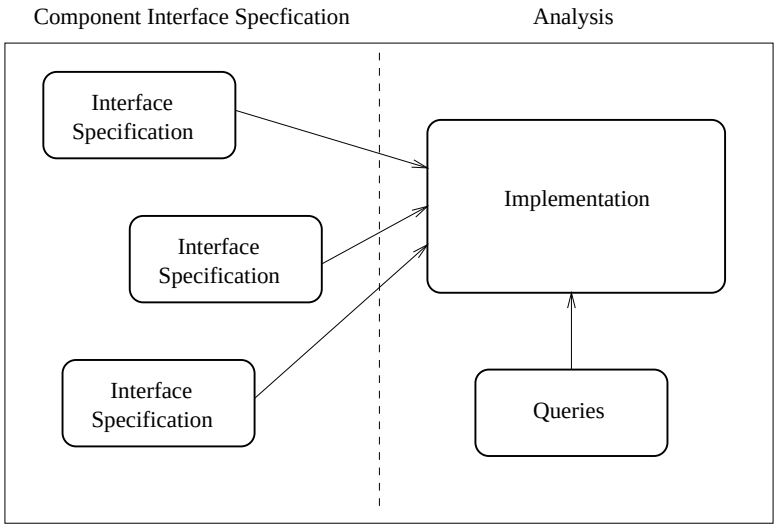

Figure 3.1: Framework scope: specification and analysis

3.1.1 Software Components

The term 'software component' is used in the same sense as defined in Chapter 2. In creating our framework, we follow the Cheeseman and Daniels [CD00] model. Assuming components are developed according to a component standard such as those mentioned in Section 1.3.1, their model breaks a software component into the following forms:

Interface An interface defines the interaction points and constraints to which a component to adhere to, and with which a client to interacts. Interfaces form *Usage contracts* that represent what a component should provide, and what a client can expect.

Component specification A component specification differs from an interface because it specifies the entire scope of a component. A component may support many interfaces and require the support of other interfaces. A component

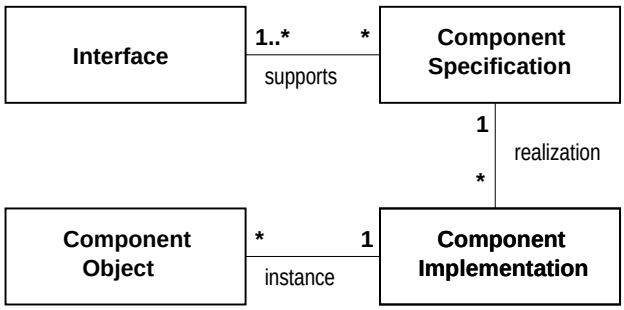

Figure 3.2: Component forms from [CD00]

specification defines the relationships between the various interfaces. Specifications form *Realization contracts* and must be adhered to by the developers of the component.

Component implementation A component's implementation consists of a compiled version of the source code that can be deployed. For a component to be deployable it must adhere to a component standard and be able to be instantiated independently.

Component object A component object is an instance of a component.

Figure 3.2 shows the relationship between the forms. An interface may be identified by many components and, as mentioned above, a single component can support many interfaces. A component specification may also have multiple differing implementations. These implementations can be instantiated as unique component objects.

It is important to remember the necessity of both interface and component specifications. Describing a component by the interfaces it supports often does not provide enough information to developers. Interface based descriptions are useful in ensuring that components can interact together correctly, and adhere to defined

constraints, but are insufficient for full compositional reasoning. When composing a group of components to build a system, a specification of how each component relates to each of its supported interfaces is required.

3.2 Contract Specification

Based on the component model introduced above, a contract should contain a description of the supported interfaces and a component specification describing the relationship between the supported interfaces.

Interfaces

Similar to other approaches [Han98, CFTV00], a component plays different roles, and provides different functionality depending on its context. The framework follows the JavaBeans and COM component standards by supporting these roles through multiple interfaces. An interface defines a related set of interaction points with the component such as methods, events, properties and exceptions.

Interfaces are not necessarily specific to a given component. For example, a JavaBean component may implement many Java interfaces to support multiple services. The OMG has formed domain task forces (DTFs) with the goal of standardizing the interfaces to services within specific domains. For example, the healthcare DTF [SIG01] has defined interfaces for clinical image access services and other industry services. Within the framework there are two types of interfaces. An *interface* defines the functionality for a single role and is denoted as a circle in Figure 3.3(a). *Composite interfaces* are the 'component specifications' and are formed as the set of interfaces and composite interfaces supported by a component. Figure

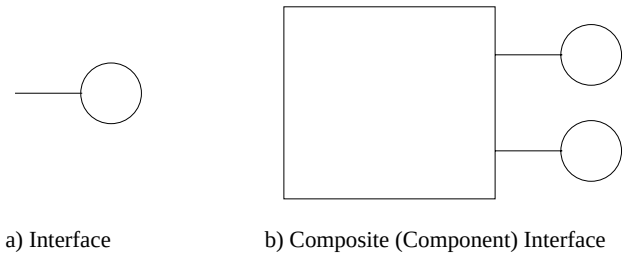

Figure 3.3: Interface Notation

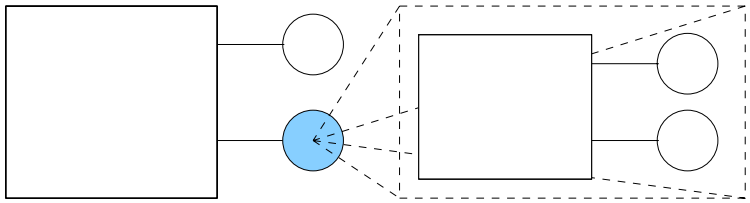

Figure 3.4: Composite Interface Notation

3.3(b) show composite interfaces denoted as a rectangle, with circles indicating the interface they support. Figure 3.4 shows the relationship between the interface types. Component support for the composite interfaces of other components is analogous to inheritance or aggregation. Composite interfaces are represented as a shaded interface circle.

Components not only provide services through interfaces, they may require access to other components that support certain interfaces. For example, in a JavaBeans implementation, required interfaces are those which are referenced within the bean.

3.2.1 Style

A very important aspect of contract specification is the style of the specifications. Based on existing work, the framework elements and description should be formally

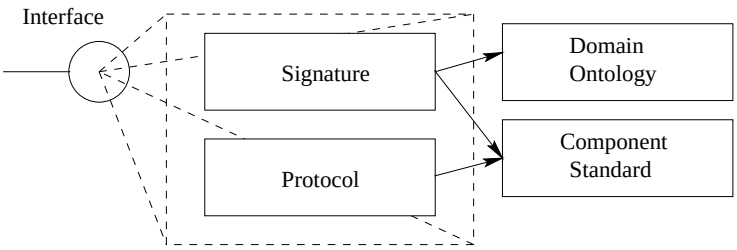

Figure 3.5: Interface Terms

defined to remove possible ambiguities and allow for tool support [BS97].

To encourage adoption and ease migration to component contracts, specification of specific terms should be optional. Often, component providers are not able to provide because of cost or other reasons complete specifications [GA99].

Another recommendation was that the description be readable to appeal to the average developer.

3.2.2 Terms

There are four types of terms that have been identified from the literature review for inclusion within a contract specification: signature, protocol and behavior, non-functional and meta. These types are based on terms that have been identified as useful by existing approaches. They are discussed next. Figure 3.5 shows the terms of an interface specification. The signature of the interface and a protocol describing the sequence of interactions is described in terms of a component and relevant domain ontologies.

Figure 3.6 shows the terms of a composite interface specification. The specification includes the supported interfaces of the component, as well as any interfaces it requires to operate. Specification terms include behavior and non-functional

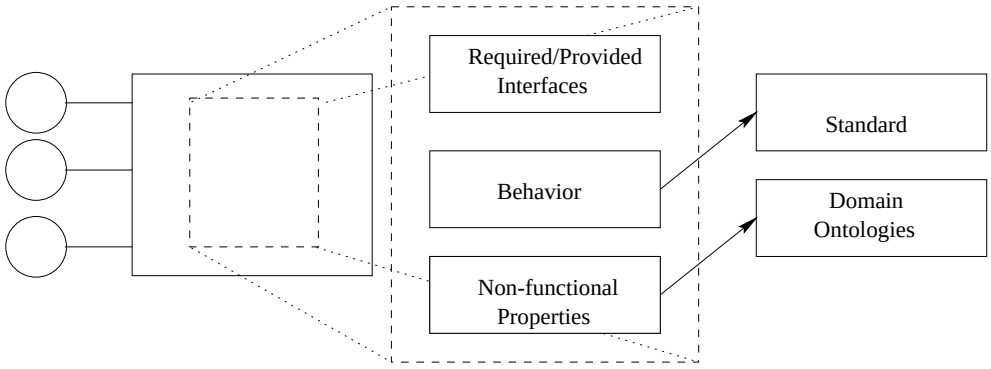

Figure 3.6: Composite Interface Terms

properties.

Signature

Each interface should contain, at a minimum, a signature describing access points. These are required to communicate with the component. This is the only type of term supported by existing IDLs. These terms include the names and types of methods, events, parameters, attributes and exceptions.

Protocol and Behavior

The protocol of a component represents the allowable sequence of interactions with the elements of an interface. Behavior refers to the response and actions a component performs when it is invoked by a client.

Within the framework, each interface may provide a description of, or constraints on the order that interface elements are invoked. A composite interface, representing the component's specification, describes the behavior of the component and represents the relationship between its supported interfaces.

Non-functional properties

The non-functional properties and qualities of a component are often left unspecified, but have been identified as an important requirement for widespread component adoption [Szy97]. Unfortunately, there is currently a lack of standards for most properties [BD00].

To support the specification of non-functional properties, the framework allows property types to be defined. Instances of the types are attached to a composite interface and have a value assigned to them. For each type, a function can be used to determine if the value is valid.

The value of potential properties can vary. A natural language comment may provide information about the vendor. The platform a component supports, may be one item from a standard enumerated set, it may be a number, indicating the base memory required in kilobytes or it may be a function, that produces a value based on its parameters.

Meta-information

To support interoperability and enhance the understanding of a component, meta-information can be supplied to describe the elements of the contract. Ontologies provide a standard for describing entities within a domain with constraints and relationships.

A component can be specified in terms of a defined ontology for the component standard that it supports. Domain specific ontologies can be used to describe the methods, parameters and non-functional properties of components.

Using a standardized conceptual model for a domain allows clients eases the

evaluation a component compared to others, or within the context of a system, because the concepts are similar.

3.3 Analysis

Although contracts are useful for understanding the operation of software components, machine-assisted analysis of these specifications would increase their appeal and usefulness to clients. To support analaysis, a formal specification is required. In the literature, potential uses for contracts beyond documentation have been proposed and are compared and summarized in Section 2.4.3. Based on existing proposals, the analysis capabilities of the framework are described below.

Property Verification

The basis of the framework's analysis capabilities is the satisfaction of a client's questions about component properties. A client can check whether an individual component exists that supports a property. A client could also check if a property holds for a composition of components.

Consistency

The framework should ensure that all contract specifications are consistent with respect to the framework's constraints, the supported component standard and any domain ontologies that are referenced.

The framework should also verify that the framework and component standard connection rules are valid for compositions.

Substitutability

The ability to substitute different component implementations with little or no system modification is one of the potential advantages of component-based systems. The ability to compare contracts to verify if one can replace another is a useful feature of the framework. The essential requirement of a substitution is that the external behavior of the component is transparent to its clients.

Substitutability is clearly defined with the following rules, based on the definitions used in previous work and discussed in Section 2.4.3. In general, a component can be replaced with another if the replacement:

1. provides at least the same services as the existing component.
2. requires no more services than the existing component.
3. has the same protocol as the current component.

For a substitution, signatures and protocols must match. The requirements for individual non-functional requirements are not well-defined. For example, a client may or may not require a replacement to operate at least as fast as the current component.

Compositions

Existing component standards and techniques lack mechanisms for specifying the architectures of systems and focus on connection standards, interfaces and services [CFTV00]. Using component contracts, a client can use the framework to construct and analyze the composition of a system from components. The composition can

be compared to a specification of the system to check if requirements have been satisfied.

The composition of systems can change over time, and the ability to specify the dynamic creations, connections and deletions of component instances within a system can help developers uncover errors in the system before implementation.

3.3.1 Enforcement

Because it is not practical to verify a component and its client implementation against the contract, run-time monitoring may be required in certain settings.

Monitoring support can be implemented within the component middleware [BJPW99] or dynamically generated and inserted in the 'glue code' between connections as in the JavaBeans standard [Sun01].

To ensure that the signature, protocol and behavior specifications of the contract are adhered to by both the implementation and the client, the monitor has the ability to check before and after each inter-component interaction. Full monitoring support of the non-functional properties of a system may require constant or periodic checking, regardless of interactions.

Support

To enhance usability and analysis as well as contract specification, the framework provides library of property templates that can be instantiated and extended by developers.

Property templates can be used to ensure that contracts and compositions are valid with respect to the component standards they support. Domain specific prop-

erty templates are provided by ontologies and can be used to analyze the signatures and non-functional characteristics of components. More general, property templates such as `deadlock` can be used to ensure that interactions with a component do not result in an error.

3.4 Summary

This chapter outlines the requirements of the framework. Contract terms and analysis capabilities have been drawn from the experience and recommendations of existing work to address the problems introduced in Section 1.3.2. A formal, structured specification will remove ambiguity and the possibility of misunderstandings. As well, it will reduce the amount of effort required to understand the component. Moreover, formal specifications are amenable to tool support, further assisting component users. For example, when attempting to locate components, contracts with formal specifications allow for more precise searching than standard text and keyword searches.

Chapter 4

Framework Design

This framework is based on another framework for software architecture validation developed by Lichtner [Lic00]. We chose this framework as a foundation for our work because CBSE and the field of software architecture are closely related. The high-level details of software architecture match well with the coherent functionality encapsulated by a component. All the abstractions introduced in Lichtner's software architecture framework (SAF) are applicable in the context of software component composition. This chapter describes the elements of the SAF and discusses the relationship between software architecture and components.

4.1 Software Architecture

The field of software architecture has emerged in response to the increasing complexity of software systems. Often when designing or describing complex software systems, informal "box-and-arrow" diagrams are used to describe the system at a high level. Formal and semi-formal notations classified as architectural descrip-

tion languages (ADLs) have been developed to assist system designers in reasoning about initial and high-level designs. Using a formalized notation, designers can use formal methods, tools or other techniques to perform analysis of high level designs to assess the potential costs of development or modification.

4.1.1 Lichtner's Software Architecture Framework

In [Lic00], Lichtner presents a formal framework for describing the architectural designs of software systems suitable for mechanical reasoning. His framework attempts to overcome many limitations of existing architectural notations including the inability to express certain elements or additional design information as well inadequate reasoning and validation capabilities.

Despite the growing emphasis on architectural designs, Lichtner noticed that there has been little effort directed at formalizing the underlying semantics of ADL models or architectural *meta-models* [LAC98]. Lichtner develops his framework by uniting the salient features of a range of existing architectural notations. To support specific notation features, or new advances, the notation allows for the definition of additional design information.

4.1.2 Model

In [Lic00], Lichtner describes the model for his SAF. The categories of Lichtner's SAF are defined by an entity and relationships that constrain the entity. The design-time model includes the categories that can be used within the system. The run-time model covers the instances of the design-time categories that have been configured to create a working system. The categories are summarized next. With

each category, relationships and predicates are provided as a basis for ensuring architectural descriptions are valid.

Design-time model

Elements are the basic entities that make up the architectural description. Elements may be *components* acting as computational or storage elements or *connectors* acting as 'glue' between interacting components. *Ports* are element connection points that define a service. *Architectural types* add meaning to the defined elements and ports by associating elements with the ports they support and by restricting connections between different port types.

Interfaces represent the interaction points through which an element communicates with the rest of the system. For a given element type, an interface is defined as a subset of ports supported by that type. To support behavioral modeling, events that each port can both observe and initiate are specified. All components and connectors are declared within a *library* and interact through an interface, and provide a sequence of operations that represent the behavior of the component.

Run-time model

Within a run-time model there are *instances* of components and connectors. As well, for each instance, their interface ports are also instantiated (*instantiated ports*).

Configurations are the central concept, containing the set of component and connector instances and connections. An *architecture* represents a complete description of the system and contains a collection of configurations. Configurations

within an architecture can also represent *composite* elements, and be used as elements within other configurations.

In addition to the categories, the architectural description contains construction operations to modify the state of a specific model. To ensure consistency, relationships define the state before and after an operation is applied.

4.1.3 Validation

Lichtner's Software Architecture Framework (SAF) also addresses the weak reasoning abilities of most notations. Most support parser-driven checks that ensure syntactic correctness and enforced constraints on typed entities. For verification of more advanced properties such as behavior, Lichtner provides room for various specification mechanisms.

For implementation purposes, Lichtner specifies the behavior of components and connectors using Hoare's CSP [Hoa85]. To distinguish between initiated and observed events, pass extra data, and identify sources, an event structure is provided.

During run-time, systems may change their topology by creating new instances or adding new connections. These dynamic systems are supported through a CSP process that describes the order of the construction events.

For verifying the structure and behavior of the system, Lichtner implements his SAF within the PVS theorem proving system. [ORS92]. Verification requires encoding the desired property into the same higher-order logic as the categories, and showing that the properties follow from the description [LAC99]. The flexibility of this approach allows a wide range of structural and behavioral properties to be verified.

Structural Properties

Structural properties are related to static elements of the system, and can cover both design-time and run-time categories. Examples of structural properties given in [LAC99, Lic00] include:

Completeness constraints: "All instance ports involved in at least one connection."

Style contraints: "All instances are of type Pipe or Filter."

Topology: "Pipes are only connected to filters."

Behavioral Properties

Behavioral properties describe the allowable states of a system are during execution, or in response to an event. Progress properties indicate that something should happen. Safety properties indicate that something should never happen. Examples of behavioral properties from [Lic00]:

- "Any event that was observed by A, was initiated by B."
- "An event A should not be initiated unless event B was observed."

4.2 Application to Component Contracts

The formal framework developed by Lichtner is well suited the description of component-based systems. Software components are often close representations to

the architectural components provided with many ADLs. By specifying the properties of the individual components and their composition in terms of the formal framework, we can reveal potential integration problems before the system is built. If applied to an existing system, integration problems may be revealed before changing a component in the system or adding new components. The ability to check for integration problems before implementation or deployment of a component-based system would address many of the issues raised in Chapter 2.

Although software architecture and CBSE have much in common, there are differences between the two areas. Where software architecture is focused on the high-level, conceptual design of a system, CBSE is based on designing systems from existing implementation pieces. ADLs have traditionally had weak support for important component notions such as independence, evolution and encapsulation [Han98, CFTV00]. To address these problems Lichtner's framework is extended to support the requirements of the contract framework.

4.2.1 Reference Model

To clarify the roles that software architecture plays, Wallnau et al [WSHK01] have developed a reference model for integrating software architecture and component technology. Figure 4.1 shows the elements of Lichtner's SAF in relation to the four levels of the reference model.

The bottom level, the assembly, is the actual implementation of the components and the 'glue code' to form the system. The next level describes the specification of the assembly and contains the description of the instances and connections of the components. Further up, types are defined. Both component and connector element types are included on this level. The top level defines the meta-types.

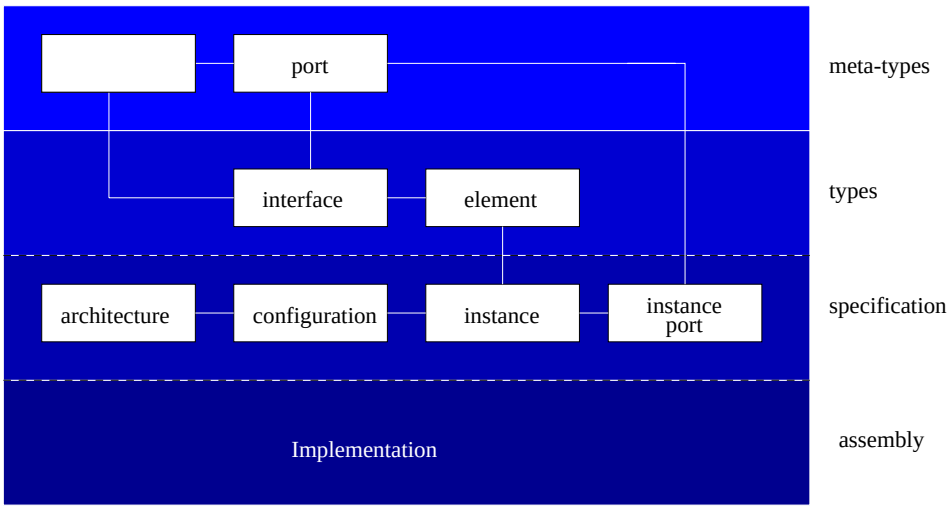

Figure 4.1: Elements of Lichtner's SAF in terms of the reference model

These definitions must be adhered to by all elements in the type level. Meta-types describe the details of the component standards. The framework provides support at all levels of the reference model.

4.3 Adapting the SAF

To meet the requirements for contracts and their analysis capabilities for the framework, the SAF of Lichtner is modified and extended. The changes reflect the differences between software architecture and CBSE. Software architecture and ADLs do not provide adequate support for encapsulation and incidence because they are focused on analyzing systems as a whole. As well, extensions to SAF are required to support the range of terms that may be included in a contract. To provide support for framework users, a collection of implementation base types and property templates should also be provided.

4.3.1 Model

Interfaces

A basic interface in our framework is analogous to a port in the SAF: combining related fuctionality. In [OH98] various strategies for assigning component methods and attributes are discussed. Because an interface may be specified independent of a specific component, additional information is provided with an interface. Besides the events the the ports can initiate and observe, a protocol description is also provided. As well, an interface may have non-functional properties attached.

Components and Composite Interfaces

In our framework, the SAF interface is considered a composite interface since it can contain multiple ports. A component is also another composite interface as it can contain multiple ports as well as multiple SAF interfaces. The notion of a component is extended to support roles, or multiple interfaces as described in Section 3.2. A component can now have more than one interface, forming a composite interface.

In addition to regular interfaces, composite interfaces may also reference other composite interfaces. This allows support for simple inheritance, polymorphism and aggregation.

4.3.2 Terms

Non-functional Properties

Non-functional properties can be attached to interfaces or components. Because the specification requirements for non-functional properties can vary greatly. To

support this, we introduce a property type. A property type is specified with at least one function that checks a value to see if it is a valid type of that property.

Meta-information

Any entity that is defined within the framework can be 'tagged' with a reference to a definition or concept from an ontology.

4.3.3 Analysis

In addition to the selection and analysis capabilities provided by the SAF, the framework has additional capabilities.

Consistency

When defining types, contracts and compositions, the framework should ensure that new information is consistent and valid with respect to the base definition provideds in [Lic00], additional rules defined for the extended framework, existing types and existing entities.

Substitutability

Additional operations are required to support checking of component equality and substitutability. To determine if two components are equivalent, they should possess the same signature, same behavior and same properties.

For substitution, the framework follows the rules defined in Section 3.3.

4.4 Summary

In this chapter, we have described a framework for analyzing architectural description that is extended to support the specification and analysis of component contracts.

Chapter 5

Framework Implementation

5.1 Introduction

In this chapter, we present a detailed description of a system that supports the verification of structural and behavioral composition of software components.

In contrast to Lichtner's implementation which employs theorem proving techniques, we use a combination of first-order predicate logic and model checking to verify properties. This combination allows for fast, automatic and simplified verification of system properties making our system accessible to most software developers.

This chapter introduces the theoretical background and pragmatics of the logic programming system XSB and model checker, XMC on which our implementation is based. In Section 5.4 the implementation and elements of our system are introduced. The final section presents examples of component specifications and their composition properties.

5.2 Theoretical Background

The implementation of SAF detailed in [Lic00] is based on the theorem-prover: Prototype Verification System (PVS) [ORS92]. Theorem proving techniques involve expressing the system and its desired properties as formulas in some mathematical logic. From the set of axioms and inference rules that define the logic, property proofs are constructed. Because PVS supports higher-order logic (HOL) it is very expressive and structural and, behavioral information can be easily specified. Behavioral specifications are encoded in Hoare's communicating sequential processes (CSP) [Hoa85]. As Lichtner acknowledges, the expansiveness of HOL comes at the cost of fully automated reasoning and user guidance is required to verify and construct proofs of most properties.

Because of the sophistication required to use a theorem-proving tool, Lichtner's approach may be too complex for use in normal software development. The framework is implemented using two complementary techniques: logic programming and model checking that allow for fast, automatic results.

5.2.1 Logic Programming

Logic programming is a programming style that is based on first-order logic. Prolog, the most popular logic programming language is based on Horn clauses.

To create a program a programmer provides a set of facts and rules. Facts can be considered specific instances of objects, whereas rules describe relationships between objects. To 'execute' the program, a goal is provided. Through a process called resolution, the Prolog runtime checks to see if the goal can be logically implied directly from existing facts, or by implication through rules.

There is a strong similarity between Prolog and relational database systems: facts, rules and goals are analogous to rows, relations and queries respectively. This makes Prolog well suited for maintaining the structural information of contracts.

5.2.2 Model Checking

Model checking has been used with success primarily in the areas of hardware and protocol verification. The current trend is to apply model checking techniques to the analysis of software system specifications [CW96].

The process of model checking is to determine whether a system specification possesses a certain property. A finite model of the system is constructed and properties are checked by performing an exhaustive state space search. It is guaranteed to terminate because model is finite. Properties are usually expressed as temporal logic formulas allowing verification about past, present and future states of the system. [CDD+98].

Unlike theorem proving techniques, model checking is generally faster and automatic. As well, model checking can be used in analyzing partial specifications providing useful information about a system before it is even fully designed. According to [CW96], the biggest strength of model checking is the ability to produce counter-examples which can be extremely helpful in debugging design errors.

The major weakness of model checking is the state explosion problem. The number of states in systems with many interacting components, or structures that can assume many values is potentially very large. Developing techniques to handle large state spaces is the main focus of research within the field and successes have led to widespread use within industry.

5.3 XSB Prolog/XMC

This section describes the XMC model checker developed at SUNY Stony Brook [CDD+98] for verifying temporal properties of concurrent systems. It is implemented with another SUNY Stony Brook project, the XSB tabled Prolog logic programming system [XSB01]. System and property specifications are encoded in the XL language.

5.3.1 XMC

Recent advances in logic programming technology allowed the development of XMC. These advances are due to the development of *tabled resolution* which has been implemented in the XSB logic programming system on which XMC is based. Tabled resolution resolves many of the inherent weaknesses of Prolog by providing termination on finite models and avoiding redundant computations. [CDD+98].

Despite variations in the system specification languages and property specification logics, the semantics are typically specified via structural recursion as fixed points (discussed below in Section 5.3.3) of certain types of functionals. These are similar to the semantics of logic programming systems. Hence model checking problems can be easily encoded into terms of a logic program. XMC was written in under two hundred lines of tabled XSB Prolog code consisting essentially of the declarative semantics of the specification language and property logic. Benchmarking results in [RRR+97] indicate XMC performs extremely well when compared to other prominent model checkers such as SPIN [SPI01] and the Concurrency Factory [CLSS96] which are coded in C/C++.

In addition to efficient performance, it is relatively easy to integrate other ap-

plications or extend aspects of the model checker because it is written in Prolog and compiled on the fly. Users are able to navigate the proof tree to view counter examples or evidence of success of property proof.

XMC has been successfully used in the verification of different protocols and algorithms [Lab00]. It has also been applied to software systems to verify the composition of design components [Don00].

The XMC specification language, XL consists of two parts. First, there is a process modelling language,a highly expressive extension of the value-passing Calculus of Communicating Systems (CCS) [Mil89]. As well there is a property specification language based on the modal μ-calculus [Koz83].

5.3.2 Process Modeling

CCS is a process calculus similar to CSP for describing systems consisting of concurrent, communicating components. XMC's CCS syntax is shown in Figure 5.1. CCS was designed to be flexible with a minimal set of operators [Fid94]. Because of experience using the model checker, the grammar and syntax XMC accepted by has changed since the earlier versions used in [CDD+98] and the most accurate version can be found in Figure 5.1. Prolog terms and predicates are used to represent values and computations respectively. XL takes from the value-passing CCS the notions of sequential composition (`o` or `;`), parallel composition (`|`), and choice (`#`). Although XL supports CCS-style restriction and relabeling to manage communication and derive instances from generic processes, XL's support for parameterized processes makes them unnecessary. Recursion is the only way to define an interactive process. Synchronized communication (`in`,`out`) supports value-passing which means processes can not only signal one another, but pass data values. Actions are

```
Pdef --> ( Pname ::= Pexp .)*

Pname --> Prolog Term

Pexp --> Pexp o Pexp    Prefix
  | Pexp ; Pexp         Prefix
  | Pexp # Pexp         Choice
  | Pexp '|' Pexp       Parallel Composition
  | Pexp @ PortMap      Relabelling
  | Pexp \ PortList     Restriction
  | Pname               Recursion
  | in(Port, Term)      Communication (input)
  | out(Port, Term)     Communication (output)
  | Action              Communication (non-sync)
  | Comp                Computation
                        (Prolog expression)
  | if(Comp,Pexp,Pexp)  Conditional
  | zero                Empty process
                        (0 in CCS)
  | nil                 Empty computation

PortMap -->
  [PortTerm/PortTerm (, PortTerm/PortTerm)*]

PortList -->
  {PortTerm (, PortTerm)* }

Port(Term) --> Prolog Term

Action --> Prolog Atom

Comp --> Prolog Predicate
```

Figure 5.1: XL Syntax Chart from [Don00]

globally observable and are used to indicate events that are relevant to properties. Computations are just Prolog computations, for example: `A is B + C`.

Prolog code can be in-lined with specifications allowing the powerful capability of arbitrarily detailed specifications, possibly to the level of implementation. User defined data types can be created from primitive types or from previous user defined types.

Similar to CCS, the basic object in XL is a process. Processes are defined using the `::=` operator and may contain parameters. A process consists of a sequence of the simple expressions defined above. A process can be invoked with parameters consistent with the defined types of the process parameters. A detailed description of the syntax and semantics of XL can be found in the XMC Users's Manual [Lab00]. An example XL specification of the Alternating Bit Protocol [Tan96] in which acknowledgements alternate between one and zero shown in Figure 5.2.

For system processes specified using a process algebra such as CCS, we can construct a representative labeled transition system (LTS). CCS specifications are encoded over a labeled transition system and are specified in Prolog using a `trans/3`

```
%%% The Alternating Bit Protocol

medium(Get, Put) ::=
    in(Get,Data);
    {  out(Put,Data)
       action(drop)
    };
    medium(Get, Put).

sender(AckIn, DataOut, Seq) ::=
    %% Seq is the sequence number of
    %% the next frame to be sent
    out(DataOut,Seq);
    {
        AckIn ? AckSeq;
        if AckSeq == Seq
        %% successful ack, next message
          then {
            NSeq is 1-Seq;
            sendnew(AckIn, DataOut, NSeq)
        }
        %% unexpected ack, resend message
           else sender(AckIn, DataOut, Seq)
    #
        %% upon timeout, resend message
        sender(AckIn, DataOut, Seq)
    }.

sendnew(AckIn, DataOut, Seq) ::=
    action(sendnew);
    sender(AckIn, DataOut, Seq).

receiver(DataIn, AckOut, Seq) ::=
    %% Seq is the expected next
    %% sequence number
    in(DataIn,RecSeq);
    if RecSeq == Seq
        then {
        NSeq is 1-Seq;
        action(recv);
        out(AckOut,RecSeq);
        receiver(DataIn, AckOut, NSeq)
    }
    else {
        %% unexpected seq, resend ack
        AckOut ! RecSeq;
        receiver(DataIn, AckOut, Seq)
    }.

abp ::=
      sendnew(R2S_out, S2R_in, 0)
      % sender -> receiver
    | medium(S2R_in, S2R_out)
      % receiver -> sender
    | medium(R2S_in, R2S_out)
    | receiver(S2R_out, R2S_in, 0).
```

Figure 5.2: Alternating Bit Protocol. Source from [Lab00]

clause, where `trans(S1(A),in(A,data),S2)` represents a transition from state `S1` with a port `A` to state `S2` on an action that inputs the value `data`. Details and transition rules can be found in [CDD+98].

5.3.3 Property Specification

The current release of XMC supports properties specified in the alternation-free fragment of the modal μ-calculus. The semantics of the modal μ-calculus can is described over sets of states of LTSs.

Properties are written using fixed point equations in the form of μ, the least fixed point operator and ν, the greatest fixed point operator. These operators reflect the computation involved: least fixed points start from the minimal element and then iteratively expand whereas greatest fixed points start from the maximal element and iteratively reduce it.

The modal μ-calculus also supports the logical connectives disjunction ($\vee$) and conjunction ($\wedge$) and the basic propositions $true$ and $false$. The diamond modality, $\langle a \rangle \phi$ is used to indicate that it is possible for an action a to occur and transition to a state where formula ϕ holds. The dual, $[a]\phi$ indicates that that formula ϕ holds in all reachable states (within one step) by an action a transition.

Using the fixed point operators, formulas are recursively defined allowing the definition of common temporal operators. For example:

$$\nu Z.\phi \wedge [-]Z$$

('$-$' represents any action) states that formula ϕ is true along every path. This is equivalent to the Computational Tree Logic (CTL) [CE81] operator $\forall \mathbf{G}\phi$, *always* ϕ.

```
D --> Z += F. (least fixed point)
    | Z -=  F. (greatest fixed point)
F --> Z | tt | ff | F \/ F | F /\ F | <A> F | [A] F | (F)

A --> action | {actions} | [-] A
```

Figure 5.3: Property specification syntax

$$\mu Y.\phi \vee \langle - \rangle Y$$

This formula is equivalent to the CTL property $\exists \mathbf{F}\phi$, *it is possible that* ϕ *will eventually hold.*

In general, μ refers to liveness properties, which require "something" to happen in contrast to ν which implies "always", or safety properties which require that certain conditions never occur [BS01].

Alternation depth refers to the level of non-trivial nesting of fixed points within a formula where adjacent fixed points are of a different type. XMC only supports the alternation-free fragment of the modal *mu*-calculus which means it only supports an alternation depth of 1 [CDD+98]. Although this limits the complexity of properties that can be expressed, formulas with alternation depth greater than 2 are difficult to understand [Bra99].

The syntax of XL's property specification is shown in Figure 5.3. In XL, the standard logical connectives (`\/` and `/\`) as well as the logical constants (`tt` and `ff`) are available for constructing formulas. `A` can represent an action, a set of actions (`{...}`), or the complement (`-`) of an action or set of actions (the complement of no actions is all actions). The two modalities, `<A> F` indicates that it is possible for formula `F` to hold after the action `A` whereas `[A] F` indicates that necessarily after the action `A`, the formula `F` holds.

Properties are specified using fixed point equations, where the operator `+=` denotes a *least fixed point* equation and the `-=` operator denotes a *greatest fixed point* equation.

Example

To demonstrate the use of XL's property specification language, here is an example related to the ABP source in Figure 5.2.

```
deadlock += [-]ff  <-> deadlock
```

This formula asks if deadlock is possible. Specifically it asks, "Is it true that, after an action (`[-]`), the system cannot progress (`ff`)? Or is it eventually possible on some execution path (`<-> deadlock`) for this to happen?"

5.4 System Description

This section covers the implementation of Lichtner's adapted model in Prolog and the XMC model checker. Figure 5.4 shows an overview of the system. The system requires as input the following:

Framework and Library Defines the basic facts and operations required to specify and analyze a design in the framework. As well, a library provides predefined types and property templates.

User defined types and properties Defines the basic element types, port types and any additional properties for a specific architectural paradigm that the user requires.

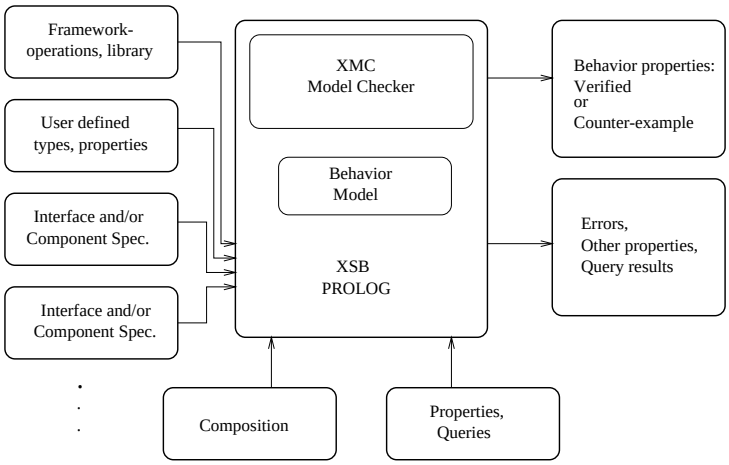

Figure 5.4: System Overview

Interfaces Individual interfaces can be specified. A component specification can be provided to form a contract.

Composition The design composition includes instances of elements, and how they interact to form a system.

Properties and Queries The last input consists of the properties to be verified as well as any other queries. Behavioral properties are specified in XL's version of the modal μ-calculus or by using and combining the properties provided with the system. Queries in Prolog can reveal details about structural properties of the design.

After processing and checking the design for structural consistency relative to the LSAF, the system generates a complete behavioral model based on the structural and operation information provided. This model is compiled by the XMC model checker. After compilation, temporal properties can be verified and the query results can be obtained. Additional properties and queries may also be resolved via Prolog.

5.4.1 Structural Model

The design composition consists of a number of elements. There are five areas that form the structural model, with types representing the lowest level building up to architectures which represent the composition of an entire system. Components, representing computations and data stores and connectors, representing interaction mediators are the fundamental building-blocks of the design. These two elements are analogous to classes in object-oriented programming languages and are similar within our architectural framework. The Prolog implementation represents the structural information internally as dynamically asserted facts. A specification is constructed using operations that ensure consistency of the design and the asserted facts. Although a specification is constructed using operations, the facts and other predefined rules, all documented in Appendix A, can be used to form queries about the structure of the design. They are summarized below and the implementation can be found in Appendix A. Specific details can be found in [Lic00].

To demonstrate the various aspects of the specification, examples of the encoding of the example system from [Lic00] are used. The example system is a simple remote procedure call (RPC) memory system. A client issues *read* and *write* to a remote memory system. This requires a clerk to package the request to the RPC component.

Basic Types

Type specifications for the design are analogous to types in programming languages. They define the basic architectural types include the elements and their ports. A mapping specifies what port types are supported by each element type as well as valid interactions between ports.

These types are specific to the design paradigm. For a low-level design, a *procedure* could be defined as an element type. A procedure would have two ports, a caller to invoke other procedures and a return port for return values to its caller.

The procedure type is the basis of Lichtner's example in [Lic00] and an equivalent encoding in our system.

```
% ADL Types
:- define_element_types([procedure]). :-
define_port_types([caller,definer]). :-
define_export_map([(procedure,caller),(procedure,definer)]). :-
define_port_map([(caller,definer),(definer,caller)]).
```

This example defines a procedure that exports two ports, and establishes the valid interactions between procedures. A more complex environment, such as Java, may establish the *class* as an element type. A class would export ports for method calls and returns and sending and receiving events.

Interfaces

Interfaces are element specifications and represent the interaction points of the element to the rest of the system. For an element type, an interface defines ports that are consistent with those exported by the type. For each port, a set of events are specified: those that the port can initiate, and those that the port can observe are specified.

The interface definition for a clerk element within the example system is shown next.

```
:- define_interface(clerk_interface, procedure, [
     (clerk_caller, caller, [remotecall],[normal, rpcfailure]),
     (clerk_definer, definer,[normal, memfailure],[read,write])
   ]).
```

In the RPC memory system, the clerk accepts the request from the client, and passes it to the RPC system. The clerk procedure expects to be called and observe a `read` or `write` event and will return `normal` or `memfailure`. When called it issues a `remotecall` and expects a return `normal` or `rpcfailure`.

Libraries

Libraries contain the sets of components and interfaces that have been declared. Components that have been added to the library are specified by an interface and a sequence of operations representing their behavior. The operation sequence is encoded in XL. Because XL allows Prolog code to be in-lined, the specifications may be quite abstract, representing only high-level actions or detailed to the level of full implementation. As with interfaces components are not required to have a behavioral specification at the cost of limiting verification to structural aspects of the design .

```
:- add_component(lib, clerk, clerk_interface).
:- add_behavior(lib, clerk,
     '% clerk_translate
      in(Clerk_definer, event(Request, Origin, Data)) o
      out(Clerk_caller, event(remotecall, Instance, [Request])) o
      {  {  in(Clerk_caller, event(rpcfailure, Instance, Data2)) o
            out(Clerk_definer, event(memfailure, Origin, Data2))
         }
        #
         {  in(Clerk_caller, event(normal, Instance, Data2)) o
            out(Clerk_definer, event(normal, Origin, Data2))
         }
      }.').
```

This example shows the definition of the clerk component specified by the clerk interface. The behavioral model is explained in detail below in Section 5.4.2 but

the operation of the clerk can be described: when the clerk is called with a request, it issues a `remotecall` and if the call returns `rpcfailure` it returns `memfailure`, otherwise it returns `normal`.

Configurations

Configurations represent a design, a set of instantiated components and connections between their ports. Connections are between instantiated ports of the component instances.

The topology of the RPC memory system is shown next:

```
% Create Configuration
:- initial_configuration(config,lib).
:- instantiate_component(config,rpc1,rpc).
:- instantiate_component(config,clerk1,clerk).
:- instantiate_component(config,client1,client).
:- instantiate_component(config,mem1,mem).

:- connect(config, client1,client_caller, clerk1,clerk_definer).
:- connect(config, clerk1,clerk_caller, rpc1,rpc_definer).
:- connect(config, rpc1,rpc_caller, mem1,mem_definer).
```

Four components, `rpc1, clerk1, client1, mem1` are instantiated from their respective components. There are three connections between the components. The client will call the clerk; the clerk will call the RPC procedure which in turn, issues a call to the memory.

Architecture

An architecture encapsulates a design and allows for more complex designs where an entire configuration is packaged as a single element.

5.4.2 Behavioral Model

The behavioral model is based on the configuration information provided in the design. Only components that are instantiated are considered when constructing the behavioral model. Each component is represented by a process which is defined with parameters for a specific instance name, and its instantiated port identifiers. This section describes how the behavioral model is constructed from the initial structural information. Using the clerk example:

```
clerk(Instance, Clerk_caller, Clerk_definer) ::=
```

The rest of definition consists of the operation sequence associated with the element specified in XL.

In XMC, each port in the model is assigned a unique identifier, also called a channel. In the system, instantiated port identifiers are constructed using a combination of its component instance name and the port name. So, the clerk component is instantiated as

```
clerk(clerk1, Clerk1_clerk_caller, Clerk1_clerk_definer)
```

Events

The event model for the system is quite similar to the event definition used in [Lic00]. Components expect to either receive events via XL's `in` operation, or send events using the `out` operation. An event has the form:

```
event(Action, Origin, Data)
```

An `Action` is the type of event; for a port, the set valid actions are those a port can initiate and observe. The `Origin` represents where the event came from, or is

directed to. Any parameters for the event can be placed in the `Data` parameter. For example, after being called the clerk procedure sends an event with action `remotecall`. The origin is the current instance of the clerk and parameter is the action with which the clerk was invoked.

Communication

Based on the operation sequence, a component does not specifically refer to an instance, or even a component with which it expects to communicate. These bindings are established through the connections between instantiated ports defined within a configuration. For each connection, an instance of the `connect` process, is created:

```
connect(I1,P1,Chan1,I2,P2,Chan2) ::=
  {  in(Chan1, event(A,O,D))
     o action(event(init,I1,P1,A,O,D))
     o action(event(obs,I2,P2,A,O,D))
     o out(Chan2, event(A,O,D))
     o connect(I1,P1,Chan1,I2,P2,Chan2)
  }
 #
  {  in(Chan2, event(A,O,D))
     o  action(event(init,I2,P9,A,O,D))
     o  action(event(obs,I1,P1,A,O,D))
     o  out(Chan1, event(A,O,D))
     o  connect(I1,P2,Chan1,I2,P2,Chan2)
  }.
```

Like events, the communication process is similar to the original defined in [Lic00]. It acts to mediate communication between two ports. To connect two ports, the process requires the names of both components' instances, ports and channels. The process does not add any functionality, but takes the provided information along with the values of each event to generate actions. These actions are used when

constructing system properties for verification and to allow properties to refer to the occurrence (or non-occurrence) of events. A connection instance, for the example RPC system between the client and the clerk:

```
connect(client1,client_caller,Client1_client_caller,}
        clerk1,clerk_definer,Clerk1_clerk_definer)}
```

Within the context of procedure element types, discussed above in Section 5.4.1, the client calls, or initially sends events to the clerk. The clerk in turn may send or return events to the client.

Composition

Similar to the structural model, each instantiated element and connection are part of a larger process. They are combined using parallel composition, which implies they act independently. In the system, the larger process is given the name of the configuration. If the configuration is a part of an architecture, and represents the composite behavior of an element then the configuration process includes the same parameters as an individual component.

The system takes advantage of XMC's support for user-defined data types to increase the chance of uncovering inconsistencies in the design. Instances and the unique event actions of all of the components involved in a design are used to construct two enumerated types. For events, actions and instances are restricted to those in their respective types; data is passed as a Prolog list. Any violation will generate a type error when the model is compiled by XMC. Using both data-types and automated generation ensures that the behavioral model is logically correct.

5.5 Design Verification and Queries

After the model has been successfully constructed in Prolog and compiled by the model checker, we can verify system properties or query the design.

5.5.1 Structural

The Prolog representation of structural information is well suited for specifying properties and queries. Below, a number of queries appear similar to those presented in [LAC99] as well as others demonstrating the ability to construct complex queries within Prolog easily. Currently as the rules defined are restricted to those relations specified in the formal model. If other relations are consistently used for queries, they can be defined and reused. As well, Prolog allows the definition of customized operations to enable a query form closer to 'natural language'.

- Stylistic Properties: all instances are either of type pipe or filter.

```
pipeFilterStyle(Config) :-
  library(Config,Lib),
  forall(instance(Config,_,Element),
    specifiedby(Lib,Element,Instance),
    (type_of(Lib,Instance,pipe);type_of(Lib,Instance,filter)) ).
```

- Topological Properties: Pipes are only connected to filters and vice-versa. Given a description of an architecture constructed using operations, implementation will ensure that architectural type constraints arc upheld. This requires manual guidance using PVS.

```
topology(Config) :-
  library(Config,Lib),
  forall(connections(Config,I1,P1,I2,P2),
```

```
    (inst2interface(Config,I1,Int1),
     inst2interface(Config,I2,Int2),
     ((type_of(Lib,Int1,filter_output),
       type_of(Lib,Int2,pipe_input));
      (type_of(Lib,Int1,pipe_output),
       type_of(Lib,Int2,filter_input)))) )
```

- Completeness Constraints: all ports of all instantiated elements are either connected or bound to ports of an interface (there are no unattached ports in the configuration).

```
portsAttached(Config) :-
  forall(port(Config,Instance,Port),
         (connection(Config,Instance,Port,_,_);
          connection(Config,_,_,Instance,Port);
          binding(_,Config,Instance,Port,_)) ).
```

- Total number of element templates in a configuration

```
totalElements(Config) :-
    library(Config,Lib),
    setof(Component, component(Lib,C), ComponentSet),
    size(ComponentSet).
```

- Set of connector templates unused in the configuration

```
connectorsUsed(Config,ConnectorSet) :-
  library(Config,Lib),
  setof(Connector,(connector(Lib,Connector), not
        instance(Config,_,Connector)),RonnectorSet).
```

- Set of outgoing and incoming connections of an instance within a configuration.

```
outgoing(Config, Inst, ConnSet) :-
  setof(Connection,
        (connection(Config,Inst,_,_,_);connection(Config,_,_,Inst_))
        ,ConnSet).
```

Behavioral

To verify behavioral properties, we need to express them in a modal μ-calculus formula. The formulas are evaluated in terms of actions. As mentioned previously, each time an event is sent across a pair of ports an action representing that event is generated. Design specific actions can also be generated within the process model of a component. The following properties are parameterized in terms of events but could be modified to support the verification of arbitrary actions within the system. Simple properties can be combined to produce more complex properties. Because the model checker is implemented in Prolog, queries about the behavior can be carried out by leaving some of the variables unbound. Instead of returning true or false, a set representing all possible values of the variable will be returned.

- An event will never occur in the system. This is useful for ensuring that a certain, potentially invalid action does not occur.

  ```
  never(S,I,P,A,O,D) -=
    [event(S,I,P,A,O,D)] ff /\ [-] never(S,I,P,A,O,D).
  ```

  The parameters (`S,I,P,A,O,D`) correspond to whether the event is being initiated or observed, the instance, port, action, origin and extra data respectively. For example,

  ```
  never(initiate,clerk1,clerk_caller,rpcfailure,clerk1,_)
  ```

  states that component `clerk1` never makes a call that initiates `rpcfailure`. Using the '`_`' indicates that any extra data parameters should be considered.

- It is possible for the event to occur in the system.

  ```
  poss(S,I,P,A,O,D)
    += <event(S,I,P,A,O,D)> tt \/ <-> poss(S,I,P,A,O,D).
  ```

- Eventually the event will occur in the system.

```
even(S,I,P,A,O,D)
  += <-> tt /\ [-event(S,I,P,A,O,D)] even(S,I,P,A,O,D).
```

- An event a does not occur unless event b occurs first.

```
notaunlessb(S,I,P,A,O,D,S1,I1,P1,A1,O1,D1) -=
    <event(S1,I1,P1,A1,O1,D1)> tt
 \/ (    [event(S,I,P,A,O,D)] ff
      /\ [-] notaunlessb(S,I,P,A,O,D,S1,I1,P1,A1,O1,D1)).
```

- Eventually an event a will occur, eventually followed by an event b.

```
athenb(S,I,P,A,O,D,S1,I1,P1,A1,O1,D1) -=
    [event(S,I,P,A,O,D)] even(S1,I1,P1,A1,O1,D1) /\
    [-event(S,I,P,A,O,D)] athenb(S,I,P,A,O,D,S1,I1,P1,A1,O1,D1).
```

It is important to notice that XMC does not appear to resolve unbound variables correctly in all cases. This can be fixed by creating a Prolog procedure which first ensures they are reachable, then evaluates the property. Templates for basic temporal properties can be recombined to form more complex properties. The finite-state verification patterns of Dwyer et al [DAC99b] provide a good basis for constructing these properties.

5.6 Extensions

This chapter has focused on the implementation of Lichtner's SAF. The following sections introduce the operations used to specify the extra terms and structure for the component contract framework.

5.6.1 Interfaces

A protocol can be attached to an interface to describe its proper usage to clients, and expected behavior to implementations. Protocol specifications can be either specified using the XL process specification language or as a set of temporal constraints. Temporal constraints are simply temporal properties of the component. Han in [Han00], recommends this approach over a process specification because it allows incremental specification, which is easier and reduces the chances of over-specification. Because clients may only connect to a single port, protocol descriptions are likewise restricted. The following code describes the protocol for the `clerk_definer` port of the `clerk_interface`:

```
:- add_protocol(clerk_interface, clerk_definer,
   'in(Clerk_definer), event(_, Origin, Data)) o
    { out(Clerk_definer, event(memfailure, Origin, Data2))
     #
      out(Clerk_definer, event(normal, Origin, Data2))
    }'
```

The alternate approach, using temporal constraints, uses another construction operation. A basic constraint that states: "all requests eventually return" is specified as:

```
:- add_constraint(clerk_interface,
     ifaevenb(event(obs,_,clerk_definer,_,Origin,_),
              event(init,Origin,clerk_definer,_,_,_)).
```

The operation is based on an instantiation of the `ifaevenb` property template that states that if the first event occurs, the other, in some finite time, must eventually follow.

5.6.2 Components

All components must be specified by at least one interface, but for our framework, a component may support multiple interfaces. Internally, for simplicity, there is a single 'global namespace', meaning the possibility for name collisions exists. This greatly simplifies the component specification because process specifications can reference ports and events directly.

For a component to support multiple regular interfaces, it only has to be specified by a single SAF interface that includes the desired ports. Components can refer to multiple interfaces with the following construction operation. For the following consider that the clerk also had access to a database:

```
:- add_interface_component(clerk,dbaccess_interface).
```

Component behavior, specified with the `add_behavior` operation could then reference ports within either the `clerk_interface` or the `dbaccess_interface`.

5.6.3 Substitution

To answer the question, "Can this component replace the original with no change in behavior?" We have implemented and extended a bisimulation algorithm based on XMC. Bisimulation is the process of comparing two proccesses to see if they have the same transitions. To determine if two components are equivalent, bisimulation checks can be performed on the components' process specifications. Strict bisimulation requires that all transitions be identical. This is not very useful, as it is not likely to occur often across different versions of components. Weak bisimulation only requires that external transitions are similar, a more useful test of component equivalency.

The two operations can be executed within the XSB environment:

```
:- bisimilar(lib,component1, component2).

:- weakbisimilar(lib, component1, component2).
```

Bisimilarity checks are carried out on the processes specified with the `add_behavior` operation. Often though, component users are concerned with the behavior of the entire component. Another variant of weak bismulation is provided that allows only specific events to be included within the simulation. This is useful if one is only interested in the changes of a single port or event. This operation can also be used for verifying the protocol of an interface against a component specification that refers to it.

Substitutability as described in Section 3.3 requires that the component provides at least the same services as the original. In terms of the implementation, the replacement must have at least the same ports that observe events as the original. To check if no more services are required, the replacement must have no more than the original's ports to initiate events. As well, weak bisimulation against possible events in the original should hold in the replacement.

5.7 Summary

In this chapter, we have shown the details of the translation of Lichtner's framework to the XSB Prolog environment and XMC model checker. Descriptions consist of a set of Prolog facts that are asserted through framework construction operations.

There are many advantages to our approach. Using a model checker, as opposed to a theorem prover allows us to achieve fast automatic results. Any failures will

generate a counter-example. The Prolog basis also allows process specification to be arbitrarily detailed with the use of in-line program code. The property specification language, the μ-calculus, is computationally more expressive than other temporal logics such as LTL and CTL.

Finally, we have extended the framework by providing additional construction operations and analysis operations to meet the requirements of our framework.

Chapter 6

Conclusions and Future Work

6.1 Summary

Component-based software engineering has recently attracted significant attention. CBSE has many potential advantages that have been realized in other engineering disciplines that have adopted component-based construction techniques. Unfortunately, there are many barriers to widespread use of software components because of their black-box nature. As most commercial component vendors provide only a binary version, users must rely on the documentation and testing to understand the component. This can lead to problems as current documentation standards are inadequate and inferring unspecified properties from the implementation may lead to conflicts with future versions.

To address these problems we have designed and implemented a framework for specifying and analyzing component contracts. Component contracts are detailed specifications of a component's interfaces. The framework provides a basis for specifying terms of the contract as well as providing tools to assist in understanding.

Chapter 1 provided an introduction to CBSE and a discussion of the specific

limitations. In Chapter 2, an extensive survey of similar, existing approaches was presented. From the survey we drew the requirements of our framework. Chapter 3 covered the requirements of the framework based on the common and important contract terms and analysis capabilities. To implement our framework, we adapted and extended another framework for software architecture description. Chapter 4 discussed the software architecture, the relation to CBSE and the modifications that were made. Chapter 5 provided details of the framework implementation including our use of Prolog and model checking.

6.2 Conclusions

The thesis of this work is that a software architecture based framework provides a basis for specifying and analyzing the contracts that are required for adequate understanding of COTS components. During our work, we have contributed the following:

- A survey of existing approaches that highlights the importance and challenges of the problem. As well, we have attempted to address, and bring attention to the widest range of possible terms of interest to component users by drawing the common and important features from existing works.
- We have described how a framework for software architecture analysis can be applied to component interface specification. Recent work by Wallnau et al [WSHK01] supports our position.
- We have produced implementation of our framework using logic programming and model checking. In addition to providing another implementation

of Lichtner's framework we have implemented additional modifications and extensions to support our requirements.

- For the component implementor and user, the framework provides a formal basis for specifying contracts. Formal specifications, besides removing ambiguities, have allowed us to provide sophisticated analysis abilities to further aid understanding.

6.3 Recommendations and Future Work

The field of component interface specification and contracts has recently emerged, and there are many opportunities for further research. We have identified many ways in which our framework can be improved. In addition, a number of other general research issues have arisen from this work.

From a design view, more concise definitions are required. We have identified potential cases where invalid or unpredictable results may occur because the consistency rules are not strong enough. A notable example is the potential for name collision between interfaces within a composite interface. Currently, for simplicity more complex definitions have not been created.

For the framework to be useful, a large supporting library containing pre-defined component, port and interface types as well as potential non-functional properties should be provided to the users to reduce effort.

A final requirement that was not addressed in this thesis is run-time monitoring. Because it is not always feasible to verify an implementation against the contract, in critical situations both the component and client's operation need to be monitored to ensure there are no violations of the contract. A common recommendation for

monitoring behavior is to adapt the component middleware [CFTV00]. Ensuring that a component conforms to its non-functional properties such as memory usage and throughput is more difficult. The demand for monitoring capabilities appears to be low since expected overhead is high [VHT00].

An important step omitted from this thesis is the application of this framework to a reasonable case study. A case study would clearly demonstrate the usefulness of the framework in a practical application.

To increase acceptance by developers, the framework should be further developed into a full tool with an interface atop the current Prolog interpreter or integrated within an IDE. A user interface could provide more readable descriptions as in [BW99].

6.3.1 Summary

Our framework provides a formal basis for specifying and analyzing component contracts to overcome some of the existing problems with software components. To address these issues we have identified the salient specification terms and analysis capabilities required to support contracts. As well, we have developed an initial implementation to demonstrate the application of the framework and provide a foundation for further research.

Glossary

callback - A program may register a function (usually with a function pointer) to be invoked at a certain point in the program. Used to reduce the coupling in software: the behavior of program can be decided or changed at run-time.

CBSE - ComponentBased Software Engineering

COM - Component Object Model. A binary standard for defining object interfaces developed by Microsoft.

CORBA - Common Object Request Broker Architecture. An OMG standard for supporting objects across different systems, platforms and programming languages.

COTS - Commerical off the shelf. COTS software is developed for market sale. A user purchases the software as-is from the vendor. Usually available in a binary form without the implementation source.

EJB - Enterprise JavaBeans. A server based component architecture based on JavaBeans. EJB execute within a server that provides advanced services such as transactions, security and persistence.

JavaBeans - A component standard based on the Java programming language.

OCL - Object Constraint Language. A subset of UML for specifying constraints. There are four types of constraints: pre- and postconditions, invariants and guards.

OMG - Object Management Group. A consortium aimed a setting standards in object-oriented programming notably CORBA and UML

UML - Unified Modeling Language. UML is s a modeling language used to specify, visualize and document object-oriented systems.

Bibliography

[AZ00] J. H. Andrews and Y. Zhang. Broad-spectrum studies of log file analysis. In *ICSE 22*, pages 105–114, Limerick, Ireland, June 2000.

[BD99] A. Borgida and P. Devanbu. Adding more "dl" to idl: towards more knowledgeable component inter-operability. In *21st International Conference on Software Engineering (ICSE-21)*, Los Angeles, CA, May 1999.

[BD00] L. Beuse-Dukic. Non-functional requirements for cots software components. In *COTS Workshop: Continuing Collaborations for Successful COTS Development (ICSE 22)*, Limerick, Ireland, June 2000.

[Ber00] C. J. Berg. *Advanced Java 2, Development for Enterprise Applications*. PH-PTR, 2nd edition, 2000.

[BJPW99] A. Beugnard, J.-M. Jezequel, N. Plouzeau, and D. Watkins. Making components contract aware. *Computer*, 32(7):38–45, July 1999.

[Bra99] J. C. Bradfield. Fixpoint alternation: arithemetic transition systems, and the binary tree. *Theoretical Informatics and Applications*, 33(4/5):341–356, 1999.

[Bri93] D. Brill. The loom reference manual. Technical report, University of Southern California, 1993.

[BRR87] W. Brauer, W. Reisig, and G. Rozenberg. *Petri Nets: Central Models and Applications (LNCS 254)*. Springer Verlag, Berlin, 1987.

[BS97] M. Buchi and E. Sekerinski. Formal methods for component software: The refinement calculus perspective. In W. Weck, J. Bosch, and C. Szyperski, editors, *TUCS General Publication No. 5, Proceedings, Second International Workshop on Component-Oriented Programming (WCOP '97)*, pages 23–32, Turku, Finland, September 1997. Turku Centre for Computer Science.

[BS01] J. Bradfield and C. Stirling. Modal logics and mu-calculi: an introduction. In A. Ponse J. A. Bergstra and S. A. Smolka, editors, *Handbook of Process Algebra*, chapter 4, pages 293–330. Elsevier, 2001.

[BvW98] R. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Springer-Verlag, Berlin, 1998.

[BW97] M. Buchi and W. Weck. A plea for grey-box components. In G. T. Leavens and M. Sitaraman, editors, *Proceedings, Foundations of Component-Based Systems Workshop*, Zurich, Switzerland, September 1997.

[BW99] M. Buchi and W. Weck. The greybox approach: When blackbox specifications hide too much. Tucs technical report no 297a, Turku Centre for Computer Science, Turku, Finland, August 1999.

[CD00] J. Cheesman and J. Daniels. *UML Components: A Simple Process for Specifying Component-Based Software*. Addison-Wesley, 2000.

[CDD+98] B. Cui, Y. Dong, X. Du, K. N. Kumar, C. R. Ramakrishnan, I. V. Ramakrishnan, A. Roychoudhury, S. A. Smolka, and D. S. Warren. Logic programming and model checking. In *Proceedings of PLIP/ALP '98*, 1998.

[CE81] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. *LNCS 131*, pages 51–71, 1981.

[CFTV00] C. Canal, L. Fuentes, J. M. Troya, and A. Vallecillo. Extending corba interfaces with π-calculus for protocol compatibility. In *TOOLS Europe*, Mont Saint-Michel, France, June 2000.

[CLSS96] R. Cleaveland, P. M. Lewis, S. A. Smolka, and O. Sokolsky. The concurrency factory: A development environment for concurrent systems. In R. Alus and T. Henzinger, editors, *Computer-Aided Verfication (CAV '96) LNCS 1102*, New Brunswick, NJ, July 1996. Springer-Verlag.

[CPT99] C. Canal, E. Pimentel, and J. M. Troya. Conformance and refinement of behavior in π-calculus. In *Workshop on Component-based Development in Computational Logic*, Paris, France, September 1999.

[CW96] E. M. Clarke and J. M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4):626–642, December 1996.

[DAC99a] J. Dong, P. S. C. Alencar, and D. D. Cowan. A component specification template for cots-based software development. In *International Workshop on Ensuring Successful COTS Development (ICSE-21)*, Los Angeles, CA, May 1999.

[DAC99b] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *ICSE 21*, Los Angeles, CA, May 1999.

[DAC00] J. Dong, P. S. C. Alencar, and D. D. Cowan. Component contract templates: A rigorous approach for component specification and composition verification. In *Proceedings, OOPSLA'00 Workshop on Component Contracts*, October 2000.

[Don00] J. Dong. Model checking the composition of hypermedia design components. In *10th IBM Centre for Advanced Studies Conference (CASCON)*, pages 51–64, Toronto, ON, November 2000.

[Fid94] C. Fidge. A comparative introduction to csp, ccs and lotos. Technical Report 93-24, The University of Queensland, Queensland, Australia, April 1994.

[FLF01] R. B. Findler, M. Latendresse, and M. Felleisen. Object-oriented programming lanaguages need well-founded contracts. Technical Report TR01-372, Dept. of Computer Science, Rice University, Houston, TX, 2001.

[GA99] J. H. Gennari and M. Ackerman. Extra-technical information for method libraries. In *Proceedings, Twelfth Workshop on Knowledge Acquisition, Modeling and Management (KAW)*, Banff, AB, October 1999.

[GAO95] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch or why it's hard to build systems out of existing parts. In *Proceedings,*

International Conference on Software Engineering (ICSE 17), pages 179–185, April 1995.

[Gar98] K. Garlington. Critique of 'design by contract: The lessons of ariane'. http://www.flash.net/ kennieg/ariane.html, March 1998.

[GGM98] J. H. Gennari, W. Grosso, and M. Musen. A method-description language: An initial ontology with examples. In *Proceedings, Eleventh Workshop on Knowledge Acquisition, Modeling and Management (KAW)*, Banff, AB, April 1998.

[GGW99] H. Geise, J. Graf, and G. Wirtz. Closing the gap between object-oriented modeling of structure and behavior. In *2nd International Conference on the Unified Modeling Language*, Fort Collins, CO, October 1999.

[Gie00] H. Giese. Contract-based component system design. In J. Ralph and H. Sprague, editors, *33rd Hawaii International Conference on System Sciences*, Maui, HI, January 2000.

[Gil01] S. Gillmor. Generation xml. *XML Magazine*, 2(3), June/July 2001.

[GMF99] M. Gaspari, E. Motta, and D. Fensel. Automatic selection of problem solving libraries based on competence matching. In *Workshop on Object Interoperability (ECOOP '99)*, June 1999.

[Han98] J. Han. A comprehensive interface definition framework for software components. In *Proceedings, 1998 Asia-Pacific Software Engineering Conference (APSEC'98)*, pages 110–117. IEEE Computer Society, 1998.

[Han00] J. Han. Temporal logic based specifications of component interaction protocols. In *New Issues of Object Interoperability*, pages 43–52, Caceres, Spain, June 2000.

[HHG90] R. Helm, I. M. Holland, and D. Gangopadhyay. Contracts: Specifying behavioral compositions in object-oriented systems. In N. Meyrowitz, editor, *OOPSLA/ECOOP 90, ACM SIGPLAN Notices, 25(10)*, pages 169–180. ACM Press, October 1990.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[JM97] J.-M. Jezequel and B. Meyer. Design by contract: The lessons of ariane. *Computer*, 30(1):129–130, Jan 1997.

[KB98] W. Kozaczynski and G. Booch. Component-based software engineering. *Software*, 15(5):34–36, September/October 1998.

[Koz83] D. Kozen. Results on the propositional mu-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[Lab00] LMC Lab. Xmc: Users' guide and manual. http://www.cs.sunysb.edu/ lmc, June 2000.

[LAC98] K. Lichtner, P. S. C. Alencar, and D. Cowan. Using view-based models to formalize architecture description. In *Third International Software Architecture Workshop (ISAW-3)*, November 1998.

[LAC99] K. Lichtner, P. Alencar, and D. Cowan. A framework for software architecture verification. Technical Report CS-99-23, University of Waterloo, November 1999.

[Lam00] C. Lamela. *Breaking Down the Barrier to Software Component Technology*. IntellectMarket Inc., Santa Rosa CA, October 2000.

[Lic00] K. Lichtner. *A Framework for Machine-Assisted Software Architecture Validation*. PhD thesis, University of Waterloo, Waterloo, Ontario, 2000.

[LPJ98] S. Lorcy, N. Plouzeau, and J.-M. Jezequel. A framework managing quality of service contracts in distributed applications. In *26th Technology of Object-Oriented Languages and Systems (TOOLS-26)*. IEEE Computer Society, August 1998.

[Mau00] M. Maurer. Components: What if they gave a revolution and nobody came? *Computer*, 33(6):28–35, June 2000.

[Mey91] B. Meyer. *Eiffel: The Language*. Prentice Hall, 1991.

[Mey92] B. Meyer. Applying "design by contract". *Computer*, 25(10):41–51, October 1992.

[Mey97] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall PTR, 2nd edition, 1997.

[MHKM95] R. Mitchell, J. Howse, S. Kent, and I. Maung. Software contracts: a way forward. In *ICSE-17 Workshop on Research Issues in the Intersection of Software Engineering and Programming Languages*, Seattle, WA, April 1995.

[Mic01] Microsoft. Microsoft com technologies. http://www.microsoft.com/com/, January 2001.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice Hall International, 1989.

[Mil99] R. Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, May 1999.

[OH98] H. Ohlenbusch and G. T. Heineman. Complex ports and roles within software architecture. In *IBM Centre for Advanced Studies Conference (CASCON)*, December 1998.

[O'L00] D. E. O'Leary. Different firms, different ontologies, and no one best ontology. *IEEE Intelligent Systems*, 15(5):72–78, September/October 2000.

[OMG01] OMG. Corba website. http://www.omg.org/corba, January 2001.

[Ont01] Ontology.org. Ontology.org: Enabling virtual business. http://www.ontology.org, June 2001.

[O'R99] C. O'Reilly. Beanbag, an extensible framework for describing, storing and querying components. Master's thesis, University of Dublin, Dublin, Ireland, September 1999.

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. Pvs: A prototype verification system. In *Lecture Notes in Computer Science 607*, pages 748–752. Springer-Verlag, 1992.

[Ram98] N. Ramakrishnan. A simple corba idl program. *Pantheon Systems Journal (http://www.pantheon.com/journal)*, April 1998.

[Rau00] A. Rausch. Software evolution in componentware using requirements/assurances contracts. In *Proceedings, International Conference on Software Engineering (ICSE 22)*, pages 147–156. ACM, June 2000.

[Reu01] R. H. Reussner. Enhanced component interfaces to support dynamic adaption and extension. In *34th Hawaii International Conference on System Sciences*, Maui, HI, January 2001. IEEE.

[RRR+97] Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, S. A. Smolka, T. Swift, and D. S. Warren. Efficient model checking using tabled resolution. In *9th International Conference on Computer Aided Verification (CAV '97) LNCS 1243*, Haifa, Israel, July 1997. Springer-Verlag.

[SDF00] A. Schmietendorf, R. Dumke, and E. Foltin. Metrics-based asset assessment. *Software Engineering Notes*, 25(4):52–55, July 2000.

[SIG01] OMG Healthcare SIG. Healthcare domain task force. http://healthcare.omg.org, 2001.

[Sit01] M. Sitaraman. Compositional performance reasoning. In *4th ICSE Workshop on CBSE: Component Certification and Systems Prediction (ICSE 23)*, Toronto, ON, May 2001.

[SPI01] SPIN. Formal verification with spin. http://netlib.bell-labs.com/netlib/spin/whatispin.html, 2001.

[Sun01] Sun. Javabeans. http://java.sun.com/beans, January 2001.

[Szy97] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison Wesley, 1997.

[Tal98] N. Talbert. The cost of cots. *Computer*, 31(6):46–52, June 1998.

[Tan96] A. S. Tanenbaum. *Computer Networks*. Prenctice-Hall, 1996.

[TN99] S. Terzis and P. Nixon. Component trading. In J. Bosch, C. Szyperksi, and W. Weck, editors, *Research Report 17/99 Proceedings, Fourth International Workshop on Component-Oriented Programming (WCOP '99)*, Ronneby, Sweden, June 1999. University of Karlskrona/Ronneby.

[VHT99] A. Vallecillo, J. Hernandez, and J. M. Troya. Object interoperability. In *Object-Oriented Technology: ECOOP '99 Workshop Reader, LNCS No. 1743*, pages 1–21. Springer-Verlag, 1999.

[VHT00] A. Vallecillo, J. Hernandez, and J. M. Troya. Woi'00: New issues in object interoperability. In *ECOOP 2000 Workshop Reader, LNCS No. 1964*. Springer-Verlag, 2000.

[Voa98] J. M. Voas. Certifying off-the-shelf software components. *Computer*, 31(6):53–59, June 1998.

[Voa01] J. Voas. A road toward component predictability and thus quality certificates. In *4th ICSE Workshop on CBSE: Component Certification and Systems Prediction (ICSE 23)*, Toronto, ON, May 2001.

[Wil00a] L. Wilkes, editor. *Buying and Selling Components - CBDI Forum SIG Meeting*, Niagara, ON, September 2000. CBDi Forum.

[Wil00b] L. Wilkes. *How Do We Share Information on Components?* CBDi Forum, Surrey, UK, March 2000.

[Wil01] L. Wilkes. *Buying Software Components*. CBDi Forum, Surrey, UK, January 2001.

[WK99] J. B. Warmer and A. G. Kleppe. *The Object Constraint Language: Precise Modeling With UML*. Object Technology Series. Addison Wesley, 1999.

[WSHK01] K. Wallnau, J. Stafford, S. Hissam, and M. Klein. On the relationship of software architecture to software component technology. In *6th International Workshop on Component-Oriented Programming (ECOOP)*, Budapest, Hungary, June 2001.

[XSB01] XSB. The xsb logic programming system, v.2.3, 2001. http://xsb.sourceforge.net, 2001.

Appendix A

Prolog Implementation

This appendix describes the various Prolog clauses and procedures that define an architectural design in terms of Lichtner's software architecture framework. Section A.1 explains the facts and rules that are used to describe the design. Operations are procedures that add or change facts about the design and are explained in Section A.2. In general, facts are not declared or asserted directly in a specification, but through operations.

A.1 Facts and Rules

The following section describes the facts and rules used to describe the architectural design. As mentioned above, the following facts are not generally declared or asserted directly to create a design, but can be used as terms within a structural query. A fact represents a unique piece of information, whereas rules are derived from existing facts. Because the information provided by rules can always be derived from facts they are not necessary to define the structure of the design, but they are provided here to provide a close match with the original definitions of

[Lic00] and enhance readability and understanding.

A.1.1 Basic Types

Element Type (fact): `element_type(E).` *E is an element type.*

Port Type (fact): `port_type(P).` *P is a port type.*

Export Map (fact): `export_map(E,P).` *E is an element that exports port type P.*

Import Map (fact): `port_map(P1,P2).` *A connection can be formed between P1 and P2.*

A.1.2 Interfaces

Interface (rule): `interface(I)` *I is an interface.*

Type (fact): `type_of(I,E)` *I is an interface of element type E.*

Port type (fact): `port_type(I,P,PT)` *I is an interface that has a port P of port type PT.*

Interacts through (rule): `interacts_through(I,P)` *I is an interface with a port named P.*

Initiates (fact): `initiates(I,P,E)` *I is an interface with a port P that initiates event E.*

Observes (fact): `observes(I,P,E)` *I is an interface with a port P that observes event E.*

Event (rule): `event(I,E)` *E is an event observed or initiated by interface I.*

A.1.3 Libraries

Elements (rule): `element(L,E).` *L is a library that contains element E.*

Component (fact): `component(L,C,I).` *L is a library that contains: C a component specified by interface I.* As well, (rule): `component(L,C)`

Connector (fact): `connector(L,C,I).` *L is a library that contains: C a connector specified by interface I.* As well, (rule): `connector(L,C)`

Contains (fact): `contains(L,I).` *L is a library that contains interface I.*

Behaves Through (fact): `behaves_through(L,E,BS).` *L is a library that contains: BS is sequence of operations that represent the behavior of the element E.*

Specified by (rule): `specified_by(L,E,I).` *where L is a library that contains: E an element specified by interface I.*

A.1.4 Configurations

Instance (fact): `instance(C,I,E).` *C is a configuration that contains: I the instance name of the instantiated element, E.* Also, (rule): `instance(C,I).`, (rule): `inst2element(C,I,E)`.

Instantiated Port (fact): `port(C,I,P)` *C is a configuration that contains: P, the port of instance I.*

Connection (fact): `connection(C,I1,P1,I2,P2)` *C is a configuration that contains: I1:P1, the instantiated port connected to the instantiated port I2,P2.*

Instance to Interface (rule): `inst2interface(C,I,F)` *C is a configuration that contains I an instance specified by interface F.*

Instance to Component (rule): `instance to component(C,I,E)` *C is a configuration that contains an instance, I of component E.*

Instance to Connector (rule): `inst2connector(C,I,E)` *C is a configuration that contains an instance, I of connector E.*

A.1.5 Architecture

Composed of (fact): `composed_of(A,C)` *A is an architecture that includes configuration C.*

Implements (fact): `implements(A,C,E)` *A is an architecture that includes a configuration C, that implements element E.*

Binding (fact): `binding(A,C,I,P,CP)` *A is an architecture including configuration C. I,P form an instantiated port that is bound to CP, a port belonging to the element configuration C implements.*

A.2 Construction Operators

These operations are Prolog procedures that change the state of program by adding or changes established facts. These are used to construct the design. Operations also enforced the constraints given in [Lic00] to ensure structural consistency of design.

A.2.1 Basic Types

Define Element Types `define_element_types([E1,E2,...])`. *Defines elements E1, E2, ... of the list argument as element types.*

Define Port Types `define_port_types([P1,P2,....])`. *Defines elements P1, P2, ... of the list argument as port types.*

Define Export Map `define_export_map([(E1,P1),(E2,P2),...])`. *For each pair in the argument list, define element E exports port type P.*

Define Port Map `define_port_map([(P1,P2),(P3,P4),...])`. *For each set of port types in the argument list, define a connection between the types as valid.*

A.2.2 Interfaces

Define Interface `define_interface(I,E,[(P1,PT1),(P2,PT2),...])` *Define an interface I for element type E, and port(s) P with type PT.*

A.2.3 Library

Add Component `add_component(L,C,I)` *Add a component C specified by interface I to library L.*

Add Connector `add_connector(L,C,I)` *Add a connector C specified by interface I to library L.*

Add Behavior `add_behavior(L,E,BS)` *Add a sequence of operations, BS representing the behavior of element E to library L.*

A.2.4 Configurations

Initial Configuration `initial_configuration(C,L)` *Create a configuration C a configuration that references library L.*

Instantiate Component `instantiate_component(C,I,E)` *Create an instance I of a component E within configuration C.*

Instantiate Connector `instantiate_connector(C,I,E)` *Create an instance I of a connector E with configuration C.*

Connect `connect(C,I1,P1,I2,P2)` *Within configuration C connect port P1 of instance I1 to the instantiated port, I2:P2.*

A.2.5 Architecture

Add Configuration `add_configuration(A,C)` *Add a configuration C to architecture A.*

Implement Composite `implement_composite(A,C,E)` *A is an architecture containing C, a configuration that implements element E.*

Bind `bind(A,C,I,P,P2)` *A is an architecture and I,P an instantiated port that is bound to the element port that the configuration C implements.*

A.3 Behavior

For this implementation, behavior is specified in XL, the language of XMC [Lab00]. When constructing a behavioral model for an architectural design, the following operations do not assert new facts, but convert structural information and provided

operation sequences to XL and output to a terminal (i.e. the console or a file). Refer to Chapter 5 for a detailed explanation of how the system functions.

Event Instances `event_instance(C)` *Generate the set of instances involved in a connection for configuration C.*

Event Actions `event_action(C)` *Generate the set of actions that are initiated or observed by instances within configuration C.*

Instance Behavior `instance_behavior(C,I)` *Add instance I from configuration C.*

Connect Behavior `connect_behavior(C,I1,P2,I2,P2)` *From configuration C, connect instantiated port I1:P1 to instantiated port I2:P2.*

Config Behavior `config_behavior(C)` *Create a behavioral model for configuration C.*

Composite Behavior: Operation `composite_behavior(A,C,I)` *Add configuration C of architecture A that implements instance I.*

Bind Behavior `bind_behavior(A,C,CP,I,P)` *Adds binding between configuration C of architecture A that implements CP to instantiate port IP.*

Appendix B

RPC Memory

This appendix contains the initial encoding representing the design of the simple RPC memory system presented in [Lic00] and the intermediate output, the behavioral model in XL generated from the specification used by the model checker.

B.1 Design Specification

```
% RPC Memory ADL Description

% ADL Types
:- define_element_types([procedure]).
:- define_port_types([caller,definer]).
:- define_export_map([(procedure,caller),
                      (procedure,definer)]).
:- define_port_map([(caller,definer),
                    (definer,caller)]).

% Add Interfaces to Library
:- define_interface(client_interface,
     procedure, [
       (client_caller, caller,
          [read, write],
          [normal, memfailure])
   ]).

:- define_interface(clerk_interface,
     procedure, [
       (clerk_caller, caller,
          [remotecall],
          [normal, rpcfailure]),
       (clerk_definer, definer,
          [normal, memfailure],
          [read,write])
   ]).

:- define_interface(rpc_interface,
     procedure, [
       (rpc_caller,caller,
          [read, write],
          [normal]),
       (rpc_definer,definer,
          [normal, rpcfailure],
          [remotecall])
   ]).
```

```
:- define_interface(mem_interface,
     procedure, [
       (mem_definer,definer,
          [normal],
          [read,write])
   ]).

% Add Component Templates to Library
:- add_component(lib, client, client_interface).
:- add_behavior(lib, client,
  % client_read
  {  out(Client_caller,
       event(read, Instance, [null])) o
   { in(Client_caller,
       event(normal, Instance, Data))
    #
     in(Client_caller,
       event(memfailure, Origin, Data))
   }
  }
  #
  % client_write
  {  out(Client_caller,
       event(write, Instance, [null])) o
     { in(Client_caller,
         event(normal, Instance, Data))
      #
       in(Client_caller,
         event(memfailure, Instance, Data))
     }
   }.').
:- add_component(lib, clerk, clerk_interface).
:- add_behavior(lib, clerk,
  '% clerk_translate
  in(Clerk_definer,
    event(Request, Origin, Data)) o
  out(Clerk_caller,
    event(remotecall, Instance, [Request])) o
  {
   { in(Clerk_caller,
      event(rpcfailure, Instance, Data2)) o
     out(Clerk_definer,
      event(memfailure, Origin, Data2))
   }
   #
   { in(Clerk_caller,
       event(normal, Instance, Data2)) o
     out(Clerk_definer,
       event(normal, Origin, Data2))
   }.').
:- add_component(lib, rpc, rpc_interface).
:- add_behavior(lib, rpc,
  '% rpc_call
  in(Rpc_definer,
    event(remotecall, Origin, Data)) o
  {
   {  out(Rpc_caller,
         event(rpcfailure, Origin, [null]))
   }
   #
   {  pop(Data, Request, Rest);
      out(Rpc_caller,
        event(Request, Instance, Rest)) o
      in(Rpc_caller,
        event(Return, Instance, Data2)) o
      { out(Rpc_definer,
          event(Return, Origin, Data2))
       #
         out(Rpc_definer,
           event(rpcfailure, Origin, [null]))
      }
    }
   }.  {* pop([X | T],X,T). *} ').
:- add_component(lib, mem, mem_interface).
:- add_behavior(lib, mem,
  '% mem_read
   { in(Mem_definer,
       event(read, Origin, Data)) o
     out(Mem_definer,
       event(normal, Origin, [null]))
   }
   #
   % mem_write
   {  in(Mem_definer,
        event(write, Origin, Data)) o
      out(Mem_definer,
        event(normal, Origin, [null]))
   }.').

% Create Configuration
:- initial_configuration(config,lib).
:- instantiate_component(config,rpc1,rpc).
:- instantiate_component(config,clerk1,clerk).
:- instantiate_component(config,client1,client).
:- instantiate_component(config,mem1,mem).

:- connect(config, client1,client_caller,
                   clerk1,clerk_definer).
:- connect(config, clerk1,clerk_caller,
```

```
                  rpc1,rpc_definer).
:- connect(config, rpc1,rpc_caller,
                  mem1,mem_definer).
```

```
% Architecture
:- add_configuration(rpc_memory,config).
```

B.2 Behavioral Model

```
{* :- datatype(event_action).

:- datatype(event_instance).

:- datatype(event_type).
  event_type(event(Action,Origin,Data)) :-
    typeof(Action, event_action),
    typeof(Origin, event_instance),
    typeof(Data, list(_)).

  event_instance(rpc1).
  event_instance(clerk1).
  event_instance(client1).
  event_instance(mem1).


  event_action(memfailure).
  event_action(normal).
  event_action(read).
  event_action(remotecall).
  event_action(rpcfailure).
  event_action(write).
*}

connect(I1,P1,Chan1,I2,P2,Chan2) ::=
  {  in(Chan1, event(A,O,D))
     o action(event(init,I1,P1,A,O,D))
     o action(event(obs,I2,P2,A,O,D))
     o out(Chan2, event(A,O,D))
     o connect(I1,P1,Chan1,I2,P2,Chan2)
  }
 #
  {  in(Chan2, event(A,O,D))
     o  action(event(init,I2,P2,A,O,D))
     o  action(event(obs,I1,P1,A,O,D))
     o  out(Chan1, event(A,O,D))
     o  connect(I1,P1,Chan1,I2,P2,Chan2)
  }.

clerk(Instance, Clerk_caller, Clerk_definer) ::=
 % clerk_translate
  in(Clerk_definer,
    event(Request, Origin, Data)) o
  out(Clerk_caller,
    event(remotecall, Instance, [Request])) o
  {
   { in(Clerk_caller,
      event(rpcfailure, Instance, Data2)) o
     out(Clerk_definer,
      event(memfailure, Origin, Data2))
   }
   #
   { in(Clerk_caller,
       event(normal, Instance, Data2)) o
     out(Clerk_definer,
       event(normal, Origin, Data2))
   }
  }.
client(Instance, Client_caller) ::=
   % client_read
  {  out(Client_caller,
       event(read, Instance, [null])) o
   { in(Client_caller,
       event(normal, Instance, Data))
    #
     in(Client_caller,
       event(memfailure, Origin, Data))
   }
  }
  #
  % client_write
  {  out(Client_caller,
       event(write, Instance, [null])) o
     { in(Client_caller,
         event(normal, Instance, Data))
      #
       in(Client_caller,
         event(memfailure, Instance, Data))
     }
   }.
mem(Instance, Mem_definer) ::=
   '% mem_read
```

```
   { in(Mem_definer,
       event(read, Origin, Data)) o
     out(Mem_definer,
       event(normal, Origin, [null]))
   }
   #
   % mem_write
   {  in(Mem_definer,
        event(write, Origin, Data)) o
      out(Mem_definer,
        event(normal, Origin, [null]))
   }.').

rpc(Instance, Rpc_caller, Rpc_definer) ::=
  '% rpc_call
  in(Rpc_definer,
    event(remotecall, Origin, Data)) o
  {
   {  out(Rpc_caller,
          event(rpcfailure, Origin, [null]))
   }
   #
   {  pop(Data, Request, Rest);
      out(Rpc_caller,
        event(Request, Instance, Rest)) o
      in(Rpc_caller,
        event(Return, Instance, Data2)) o
      { out(Rpc_definer,
          event(Return, Origin, Data2))
        #
          out(Rpc_definer,
            event(rpcfailure, Origin, [null]))
      }
    }
   }.  {* pop([X | T],X,T). *} ').

config ::=
  rpc(rpc1, Rpc1_rpc_caller,
            Rpc1_rpc_definer) |
  clerk(clerk1, Clerk1_clerk_caller,
                Clerk1_clerk_definer) |
  client(client1, Client1_client_caller) |
  mem(mem1, Mem1_mem_definer) |
  connect(client1,client_caller,
          Client1_client_caller,
          clerk1,clerk_definer,
          Clerk1_clerk_definer) |
  connect(clerk1,clerk_caller,
          Clerk1_clerk_caller,
          rpc1,rpc_definer,
          Rpc1_rpc_definer) |
  connect(rpc1,rpc_caller,
          Rpc1_rpc_caller,
          mem1,mem_definer,
          Mem1_mem_definer) .
```