Property Transformers For Compositional Reasoning

A Thesis

Presented in Partial Fulfillment of the Requirements for

the Degree Master of Science in the

Graduate School of The Ohio State University

By

Ayesha Mascarenhas, B.E.

* * * * *

The Ohio State University

2002

Master's Examination Committee:

Dr. Paolo A. G .Sivilotti, Adviser

Dr. Neelam Soundarajan

Approved by

Adviser

Department of Computer
and Information Science

ABSTRACT

The complexity of large systems makes them difficult to reason about. Techniques based on compositional reasoning provide a way to manage this complexity. Compositional reasoning is the ability to reason with the help of simple rules about a large system based on knowledge of smaller components. In order to achieve compositional reasoning the properties used to describe components must lend themselves easily to composition.

Recently, a new taxonomy of properties with particularly simple rules for composition has been proposed. Existential properties and universal properties are classes of such properties with simple rules for composition. In addition, property transformers for converting properties that are not existential or universal, to properties that are have also been proposed.

In this thesis, we present two new formulae for two of these property transformers. These formulae provide a novel interpretation of the property transformers they represent. In addition, we describe techniques to incorporate these properties with simple rules for composition into the CORBA and web services component-based development frameworks.

To my parents

ACKNOWLEDGMENTS

This work was made possible through the help and encouragement that I received from so many people. I am thankful to all of them. My academic advisor Dr. Paul Sivilotti who introduced me to the area of research I worked on and sparked my interest in the subject. His ability to uncover the subtle meaning behind the not-so-obvious has been inspiring. I am fortunate to have had the opportunity to work with him.

Thanks to Jason, Nigamanth and Scott for many good times and many great discussions. Thanks also to the members of the RSRG research group. The feedback and encouragement I received from this group was invaluable. Special thanks to Dr. Bruce Weide for always being tremendously supportive and helpful. Thanks to the members of the Distributed Systems Research Group for useful feedback related to my work.

I would also like to acknowledge all my friends in Columbus who made my time here so much more enjoyable. Finally, thanks to my family for always being encouraging and supportive.

VITA

November 17, 1977	Born - Mumbai, India
June 1999	B.E. Computers, Fr. Conceicao Rodrigues College of Engineering, University of Mumbai, India
Sept 2000- present	Graduate Reasearch and Teaching Associate, Department of Computer and Information Science, The Ohio State University.

PUBLICATIONS

Research Publications

Ayesha Mascarenhas and Paolo A.G.Sivilotti "A Paradigm for Component-Based Software Development in a Distributed Environment" *International Conference on Parallel and Distributed Processing Techniques and Applications* 2002.

FIELDS OF STUDY

Major Field: Computer and Information Science

TABLE OF CONTENTS

LIST OF FIGURES

Figure **Page**

CHAPTER 1

INTRODUCTION

1.1 Goal

Software applications are becoming increasingly distributed. These distributed systems are particularly well-suited for compositional development, due to them being made up of distributed sub-systems each of which can be treated as a component. By compositional development, we mean the development of systems from smaller and simpler components. As systems get larger, reasoning about these systems becomes more complex. One way to deal with this complexity is to reason about the behavior of systems based on the behavior of smaller and simpler components. Many techniques for specifying the behavior of distributed components already exist [1, 2, 3, 4]. However, a simple compositional development mechanism requires that the properties used to describe component behavior lend themselves easily to composition.

Our overall research objective is to develop a theory for compositional reasoning about software systems. By compositional reasoning, we mean the ability to reason about large software systems based on the knowledge of properties that hold on smaller and simpler components using simple rules. Compositional properties are properties that allow us, using simple rules, to determine system properties from the

individual component properties. Recently, a start has been made toward this goal by means of classifying certain compositional properties [5, 6]. Toward this goal, we aim to determine and prove characteristics of certain compositional properties. We also look at how these properties could be incorporated into current component-based development frameworks for distributed systems.

1.2 Motivation

A distributed system is a collection of software components located at geographically separated sites and collaborating to achieve a task. Communication between distributed components is usually through message-passing techniques. The nature of these systems makes them difficult to reason about and develop. For example, in concurrent systems, it may be difficult to reproduce a trace that precipitated an error. Traditional techniques for specifying sequential systems using pre/post conditions [7, 8] are not suitable to distributed systems [9, 10]. The potential concurrency in distributed systems may make it impossible for the caller of a function to guarantee that the pre-condition is satisfied when a function begins executing. Secondly, pre/post condition specifications do not express progress properties that are inherent in the specification of distributed systems.

Instead, one method of specifying behavior in distributed systems is by using safety and liveness properties [1, 2]. Informally, safety properties specify that "nothing bad happens" while liveness properties specify "something good eventually happens". An example of a safety property is: A component in a broadcast system must hold an authorizing token before it broadcasts. An example of a liveness property is: Once a component secures the token it must eventually relenquish it. One manner in which

these safety and liveness properties are expressed is by using temporal operators. Temporal operators *stable* and [11] denote safety properties while operators *transient* [12] and *leads-to* [11] denote liveness properties. We will take a closer look at these operators in Section 2.3.

Although component behavior can be formally specified using these safety and liveness properties, these properties do not necessarily reduce the effort spent by client developers when reasoning about the behavior of the component in the client application. A component description in terms of safety and liveness properties may provide the client developer with complete and precise specifications of the component behavior; however, the client developer now has to spend time reasoning (compositionally) about how the component will interact with the rest of the client application — time and effort that could already have been spent just once if spent by the developers of the component. Developers of components apply some form of reasoning, formal or otherwise, when developing, testing, performing walkthroughs, etc. During these activities, the developer has an idea of the interaction of the component in possibly different situations, which means the developer can write possibly open system specifications (descriptions of the system that can hold in any environment). Compositional properties allow us to capture precisely this behavior in components. While specifications in terms of safety and liveness properties allow reasoning reuse, they might not be sufficient for clients to reason compositionally. Instead, specifications need to also lend themselves easily to composition.

Recently, a new taxonomy of compositional properties has been introduced [5, 6]. This taxonomy identifies some fundamental classes of properties: *universal*, *existential*, and others, that enjoy particularly simple compositional theories. An existential

property is a property such that if a component satisfies the property, then any system that contains that component satisfies that property. A universal property is a property such that if all the components in a system satisfy the property, then the system satisfies that property.

As an example of an existential property, consider a distributed component that broadcasts streaming video, generating at least 56Kbps of continuous network traffic. Regardless of how this component is used in a larger system, that larger system will also generate at least 56Kbps of continuous network traffic. As an example of a universal property, consider a system that uses token-passing to manage access to a shared critical resource. If every component guarantees it neither creates nor destroys tokens, the larger system neither creates nor destroys tokens.

An interesting result of this classification technique is that, in the domain of distributed systems, the basic safety property *stable* is universal, while the basic liveness property *transient* is existential. Hence, we are now able to reason compositionally by describing distributed systems components in terms of these properties. Not all component properties, however, can be expressed in terms of compositional properties. The liveness property *leads-to*, for example, is neither existential nor universal. We would like to reason about the behavior of systems even in cases where the properties that hold on the constituent components are non-compositional.

To address this concern, *property transformers* based on weakest/strongest existential/universal properties can be used to transform properties that do not exhibit this nice compositional flavor into properties that do [6]. For example, a strongest existential transformer for some property P gives the strongest property that is weaker than P and that is also existential. If we know the existential properties that a

component exhibits, we also know something about any system that contains this component, and we can reason about the system as a whole, based on the properties of that one component. Similarly, a weakest existential transformer for some property P gives the weakest property that is stronger than P and that is also existential.

While the effort required to determine compositional properties on components might seem significant, the benefits in terms of simple compositional reasoning far outweigh these initial costs. The main benefits of component-based software development are in the reuse of developed components based on provided specifications. A modular compositional mechanism depends upon the availability of component specifications that are easy to compose.

Finally, while the research community focuses on formal methods in order to deal with the complexity in development of distributed systems, the industrial community tries to tackle the same problem through the development of tools, standards and middleware. These tools ease the process of development by hiding low-level communication and development details. However, in order to reason about the correctness of components, behavioral specifications are still necessary. In this thesis, we look at how compositional properties can be used to enhance the descriptions of components.

1.3 Contribution of Thesis

We define a new property transformer in Section 3.3 and show that this transformer is the same as the strongest existential property transformer. In Section 4.2 we define a new property transformer and show that it is the same as the strongest uiversal property transformer. These two new formulae provide a novel interpretation of the strongest existential and strongest universal property transformers respectively.

In addition, we describe techniques to incorporate the use of compositional properties using current technologies. In particular, we focus on two popular component-based development frameworks — CORBA and web services.

1.4 Thesis Outline

Chapter 2 provides the necessary background material and an introduction to the existing results on composition using some existential and universal properties, along with the notation used through the rest of this thesis. Chapter 3 provides our theorem for the strongest existential property transformer, a proof of correctness, and a practical application. Chapter 4 provides our theorem for the strongest universal property transformer along with a proof of correctness. Chapter 5 discusses a software development paradigm incorporating compositional properties and its application to distributed frameworks including CORBA and web services. We present related work and our conclusions in Chapter 6.

CHAPTER 2

BACKGROUND

2.1 An Introduction to Compositional Properties

Informally, compositional properties are properties that allow us to determine system properties from component properties using simple rules. An example is the mass of the composition of two physical objects: This property can be obtained, using a simple rule of addition, from the masses of the individual objects. However, the heat radiated by an object cannot be obtained simply from the heat radiated by the individual components. This property depends on more complex combinations of the properties of the constituent objects and is therefore not considered to be a compositional property.

Similarly, distributed systems are the result of composing distributed components. We would like the properties that hold on these components to have simple rules for composition. This will allow client developers to reason compositionally about the behavior of components in client applications.

The rest of this chapter describes some background material and relevant known results from related work.

2.2 Notation

Predicates are boolean-valued functions. The properties we refer to, are predicates. Function application is denoted using a dot. Function application has higher precedence than boolean operators, and associates to the left. The letters P, Q, R represent properties, while F, G, H, K, L represent components. For example, $P.F$ represents the boolean: property P holds on F.

Square brackets indicate that a predicate is true *everywhere*. For example, $[P]$ means P is true for all components.

Not all components can be composed with each other. We denote components that are composable by the $\surd$ binary operator. This operator is asymetric. For example, $F \surd G$ indicates that F can be composed with G. It is not necessary however, that G can be composed with F.

Composition is denoted by $\circ$. For example, $F \circ G$ represents the system consisting of F composed with G where it is assumed that this composition *is* possible, that is, where $F \surd G$. Composition is closed which means that the composition of two components is also a component. Composition is asymetric and associative. For any F, G and H that can be composed: $((F \circ G) \circ H) \equiv (F \circ (G \circ H))$

We assume the existence of a $UNIT$ component that can always be composed with any component such that: $UNIT \circ F = F \circ UNIT = F$

The general notation we use for quantification is $(O\, i : r.i : t.i)$ where O is the operator and must be binary, symmetric, associative and must have an identity element. i is the free variable. $r.i$ is the range and must be a predicate on the free variable i. $t.i$ is the term and must be an expression that may contain i. The type of this expression must be the same as the type of the operands of the operator.

Operators $\wedge$ and $\vee$ are represented by special symbols $\forall$ and $\exists$. For example, the quantified assertions: $(\forall\, a\, :\, a \text{ in } F\, :\, \{P\}a\{Q\}\,)$ indicates that $\{P\}a\{Q\}$ holds for all actions in F. The tuple $\{P\}a\{Q\}$ denotes the fact that the execution of a in any state satisfying predicate P results in a state satisfying predicate Q, if the execution of a terminates. We assume that the execution of every action terminates.

Similarly, $(\exists\, a\, :\, a \text{ in } F\, :\, \{P\}a\{Q\}\,)$ indicates that $\{P\}a\{Q\}$ holds for some statement in F. As a short-hand, when the range is the predicate **true**, it can be omitted all together. For example, we write: $(\exists\, a\, ::\, \{P\}a\{P\}\,)$

We follow the calculational style of presenting proofs [13]. Each step in a proof involves the application of one or more axioms or theorems, and is accompanied with a justification for that step.

2.3 An Introduction to Temporal Operators

We briefly describe the temporal operators **next**, **transient**, **ensures**, $\leadsto$, **stable**, **unless** and **invariant**. We describe the model of computation where these operators will be used later, in Section 2.6

The most basic safety operator is **next**. Informally, the property P **next** Q means that if a program ever enters a state satisfying P, its very next state will satisfy Q.

$$(P \textbf{ next } Q).F \;\triangleq\; (\forall\, a\, :\, a \text{ in } F\, :\, \{P\}a\{Q\}\,) \tag{2.1}$$

The most basic progress operator is **transient**. Informally, the property **transient**.P means that if a program ever enters a state satisfying P, there exists at least one action that always takes the program to a state where P is **false**.

$$\textbf{transient}.P.F \;\triangleq\; (\exists\, a\, :\, a \text{ in } F\, :\, \{P\}a\{\neg P\}\,) \tag{2.2}$$

Informally, property P **ensures** Q means that if P holds, it will continue to hold so long as Q does not hold, and eventually Q does hold.

$$(P \textbf{ ensures } Q).F \triangleq (P \wedge \neg Q \textbf{ next } P \vee Q) \wedge (\exists a : a \text{ in } F : \{P \wedge \neg Q\}a\{Q\}) \tag{2.3}$$

The *leads-to* property, denoted $\leadsto$, is defined by the next three equations. Informally, $P \leadsto Q$ means that if P is true at some point, Q will be true eventually.

$$P \textbf{ ensures } Q \Rightarrow P \leadsto Q \tag{2.4}$$

$$(P \leadsto Q) \wedge (Q \leadsto R) \Rightarrow P \leadsto R \tag{2.5}$$

$$(\forall P : P \in S : P \leadsto Q) \Rightarrow (\exists P : P \in S : P) \leadsto Q \tag{2.6}$$

Informally, **stable**.P means that once P becomes true, it remains true.

$$\textbf{stable}.P.F \triangleq (\forall a : a \text{ in } F : \{P\}a\{P\}) \tag{2.7}$$

Informally, P **unless** Q means that if P is true at some point, it remains true unless Q becomes true.

$$(P \textbf{ unless } Q).F \triangleq P \wedge \neg Q \textbf{ next } P \vee Q \tag{2.8}$$

A stable predicate that holds initially, is said to be **invariant**.

$$\textbf{invariant}.P \triangleq (\textit{initial condition} \Rightarrow P) \wedge \textbf{stable}.P \tag{2.9}$$

2.4 Overview of Existing Results

One simple composition rule is the rule that establishes that if a property holds on a component then that property holds on any system containing that component.

Properties that exhibit this behavior, are *existential* properties. Another simple composition rule, is one that establishes that if a property holds on all components in a system, then that property holds on the system. Properties that exhibit this behavior are *universal* properties [6].

exist and *univ* are predicates on properties. A property P is existential if and only if $exist.P$ is true, where:

$$exist.P \triangleq (\forall F, G : F \sqrt{} G : P.F \vee P.G \Rightarrow P.(F \circ G)) \tag{2.10}$$

Similarly, a property P is universal if and only if $univ.P$ is true, where:

$$univ.P \triangleq (\forall F, G : F \sqrt{} G : P.F \wedge P.G \Rightarrow P.(F \circ G)) \tag{2.11}$$

Not all properties of interest are existential or universal. Property transformers — that is, functions from properties to properties allow us to transform these non-compositional properties to compositional properties [6]. For example, on applying the strongest existential property transformer, denoted $\mathcal{SE}$, to a property P, we obtain $\mathcal{SE}.P$: the strongest property which is weaker than P and existential. Informally, it is the strongest property that can be deduced of any system that contains at least one component that satisfies P. $\mathcal{SE}.P$ has been defined as the conjunction of all the properties that are weaker than P and existential [6]:

$$\mathcal{SE}.P \triangleq (\forall Q : [P \Rightarrow Q] \wedge exist.Q : Q) \tag{2.12}$$

Some properties that have been proved about $\mathcal{SE}$ are:

$SE.P$ is always weaker than the property P.

$$[P \Rightarrow \mathcal{SE}.P] \tag{2.13}$$

When a property P is existential, its SE is the property P itself.

$$exist.P \;\equiv\; [SE.P \;\equiv\; P] \tag{2.14}$$

SE is disjunctive.

$$[SE.(P \vee Q) \;\equiv\; (SE.P) \vee (SE.Q)] \tag{2.15}$$

SE is monotonic.

$$[P \;\Rightarrow\; Q] \;\Rightarrow\; [SE.P \;\Rightarrow\; SE.Q] \tag{2.16}$$

Similarly, a strongest universal property transformer, denoted SU has been defined [6]. The strongest universal property $SU.P$ has been defined as the conjunction of all the properties that are weaker than P and universal:

$$SU.P \;\stackrel{\triangle}{=}\; (\forall Q \;:\; [P \;\Rightarrow\; Q] \wedge univ.Q \;:\; Q\;) \tag{2.17}$$

Some properties that have ben proved about SU are:

The $SU.P$ of a property P is always weaker than the property P.

$$[P \;\Rightarrow\; SU.P] \tag{2.18}$$

When a property P is universal, its SU is the property P itself.

$$univ.P \;\equiv\; [SU.P \;\equiv\; P] \tag{2.19}$$

SU is monotonic.

$$[P \;\Rightarrow\; Q] \;\Rightarrow\; [SU.P \;\Rightarrow\; SU.Q] \tag{2.20}$$

Informally, $SU.P$ is the strongest property that can be deduced of any system, all of whose components satisfy P. Finally, since all existential properties are universal,

$$[SU.P \;\Rightarrow\; SE.P] \tag{2.21}$$

Similarly, a weakest existential property $WE.P$ has been defined [6] as the disjunction of all the properties stronger than P and existential.

$$WE.P \triangleq (\exists Q : [Q \Rightarrow P] \wedge exist.Q : Q) \tag{2.22}$$

Informally, $WE.P$ is the weakest property of a component that ensures that *any* system that contains that component, will satisfy the property P.

It has been shown that $WE.(P \Rightarrow Q) \equiv P$ **guarantees** Q, where

$$(P \textbf{ guarantees } Q).F \triangleq (\forall H, K : H \sqrt{F} \sqrt{K} : P.(H \circ F \circ K) \Rightarrow Q.(H \circ F \circ K)) \tag{2.23}$$

This means that if F satisfies the property P **guarantees** Q ,for *any* system that F is a part of, if the system satisfies P then F on its own guarantees that the system will satisfy Q [5].

It has been shown that there exist some instances where properties do not have a unique weakest universal property [6]. Therefore, there is no definition for the weakest universal property transformer WU in the manner of the definitions for SE, SU and WE in Equations 2.12, 2.17 and 2.22 respectively.

2.5 Significance of Property Transformers

Property transformers are important because they allow us to reason about a system based on the known properties of constituent components even though the known properties are not compositional themselves. For example, the weakest existential property $WE.P$ for a property P, is the weakest property that is stronger than P and existential. The definition of WE as given in Equation 2.22 indicates that $WE.P$ is the disjunction of all the properties that are weaker that P and existential.

In addition to this definition, a fixed-point representation of WE has been given [5].

$$WE.P.F \triangleq (\forall H, K : H \sqrt{F} \sqrt{K} : P.(H \circ F \circ K)) \quad (2.24)$$

This characterization of WE indicates that any system containing a component satisfying $WE.P$ will satisfy the weaker property P. This is significant because a client developer looking for a component to satisfy some system property P — whether it is existential or not, can satisfy this system requirement, by including one component that satisfies $WE.P$. This is because $WE.P$ is stronger than P and existential.

Similarly, the strongest existential property $SE.P$ for property P, is the strongest property that can be deduced about *any* system that has at least one component that satisfies P. It is particularly useful to component developers who want to advertise components they develop. Using the SE property transformer component developers can determine strongest existential properties of non-existential properties and write compositional specifications for components.

The strongest universal property is similarly useful to component developers. $SU.P$ is the strongest property that can be deduced about any system all of whose components satisfy P.

Unlike the fixed-point representation of WE in Equation 2.24 however, there is no fixed-point representation of SE and SU. In sections 3.3 and 4.2, we provide fixed-point representations for property transformers SE and SU.

2.6 Examples of Composition

We consider a simple model of composition [5] to illustrate the compositional properties we have discussed. Components and systems are represented by bags of colored blocks. Composition is the union of the contents of the components. An

empty bag is the $UNIT$ component. In this model, components (bags) can always be composed. Let F represent a bag with 1 red and 1 green block. One property of F, is that it has exactly 2 blocks. The property *has exactly 2 blocks* is neither existential (compose F with any non-$UNIT$ component), nor universal (compose F with any other components having the same property and the composition will have more than 2 blocks). However, the property *has at least 1 red block* is existential. Similarly the property *has no blue blocks* is universal. Finally, the component F also satisfies the existential property *at least 1 blue block* **guarantees** *at least 3 colors*. For any system that F is a part of, if the system has *at least 1 blue block* then F on its own, guarantees that the system will have *at least 3 colors* by virtue of its own properties — two different colored non-blue blocks.

A more interesting and relevant model is one where components are programs. These programs consist of a set of variables, a specification of their initial values, and a set of actions. A program execution starts from any state satisfying the initial condition and continues forever. At each step of execution, an assignment statement is selected non-deterministically and executed. The execution of an action can be conditional on a predicate or guard being true. For example, $even.x \longrightarrow x := x + 1$. This action is enabled when the guard is true i.e. x is even. Composition of programs is the set-union of the sets of variables and the sets of actions and the conjunction of the initial conditions. A program with a single action : the *skip* action, is the $UNIT$ component.

We describe the properties that hold on components in this model using the temporal operators we described in Section 2.3. A simple component F is shown in Figure 2.1. The component description consists of a safety property, $(\forall k : k \text{ in } int : x =$

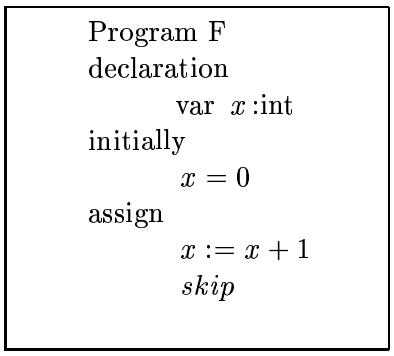

Figure 2.1: component F

k **next** $k \leq x \leq k+1$) which states that x is non-decreasing, and a progress property, **transient**.$(x = k)$ which states that x must change sometime i.e. we cannot keep picking the *skip* action. The safety property is a universal property (if no component ever decreases x, it will be monotonically non-decreasing). The progress property is an existential property (no matter what component we compose F with, there will always exist an action that changes x).

The component F also satisfies the property $\{x = 0\} \leadsto \{x > 10\}$. $\leadsto$ properties however, are not compositional. For example, we compose F with component G in Figure 2.2. Each component individually satisfies the property $\{x = 0\} \leadsto \{x > 10\}$. However, since the second action in G can now be enabled, it is not necessary that x is non-decreasing, and the $\leadsto$ property does not hold.

From the above two examples, it is clear that these compositional properties are not restricted to one particular domain of composition. At the same time, these properties allow us to reason easily about system properties based on the constituent component properties.

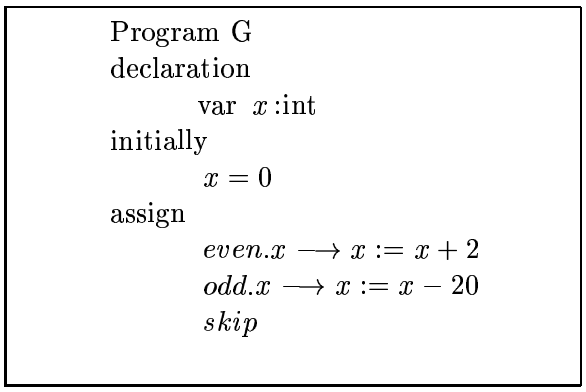

Figure 2.2: component G

2.7 Current Industry Frameworks for Compositional development

One popular framework for building distributed systems is CORBA [14]. A more recent technology is web services [15]. This section provides a brief introduction to both technologies.

2.7.1 CORBA

CORBA (Common Object Request Broker Architecture) [14] is the Object Management Group (OMG) standard for an architecture that allows applications to communicate over networks. Several commercial as well as free implementations of the standard are available, such as VisiBroker [16] and Orbacus [17].

The CORBA standard is platform, operating system, programming language and network independent. CORBA applications consist of running software units — objects. For any request, the message-initiating object is often called the client, while the target object is often called the server. The standard defines an Object Request

Broker (ORB) which resides on every workstation involved in the communication, as responsible for routing requests from clients to servers and responses the other way. The standard also defines a standard communication protocol, the Internet Inter-ORB protocol (IIOP), for the ORBs to communicate.

Objects are defined through an interface using the OMG Interface Definition Language (IDL). This IDL is independent of any programming language, but mappings to most popular languages such as Java, C++, C, Perl, etc. are available. The IDL parsers that do this mapping generate stubs and skeletons. Stubs contain the necessary code to allow clients to communicate with remote server objects as if they were local objects, and skeletons are the server-side complement to client stubs. The stubs and skeletons perform tasks like marshalling and demarshalling. The use of this IDL enables the separation of the interface from the implementation, and clients access objects only through the advertised interface. The ORB maintains an Interface Repository (IR) that contains the IDL interface definitions of objects.

CORBA is a mature and proven technology. Its main advantages over other middleware are its object-oriented nature, excellent language support, and a large number of well-defined standard services such as naming and trading services. Although CORBA can be used over the web, it has not been widely accepted as the technology to perform inter-organization communication.

2.7.2 Web Services

Web services [15] are the latest middleware technology. Web services use simple message- based protocols to exchange data across the Internet. A web service is a programming application that exposes an interface accessible over the Internet via

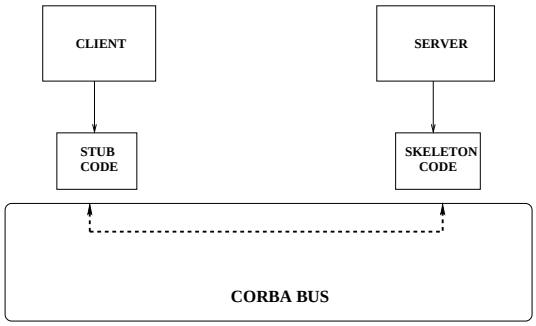

Figure 2.3: CORBA Architecture

standard Internet protocols and data exchange formats like HTTP and XML [18]. They are currently being hailed as the building blocks for constructing hte next generation of distributed systems.

The web service architecture depends on four standards to provide the necessary features to build distributed applications. The Extensible Markup Language (XML) is the standard used to represent data. While XML allows data to be represented in a standard way, it does not define the data types. The World Wide Web Consortium (W3C) XML Schema [19] defines some of these types, and is often used by web services platforms as the type system. The Simple Object Access Protocol (SOAP) provides the mechanism to access the services [20]. The SOAP standard provides specifications of SOAP messages and how they are to be sent over HTTP. Finally, the Web Services Description Language (WSDL) is an XML-based language for describing the web services, parameters to be sent, and return values [21].

A web service provider develops an application and exposes it over the Internet using SOAP. This is done by providing a description in WSDL and registering the service at the Universal Description, Discovery and Integration (UDDI) registry [22],

which is itself a web service accessible via a SOAP interface. A client application will locate a web service through the UDDI registry. The client can download the WSDL description of the service, and tools like Visual Studio.NET [23] are available to generate the necessary client stubs.

Unlike CORBA, web services were developed specifically with the web in mind. As a result, web services have been generally accepted by the web community, as the way to perform inter-enterprise communication. They have also received the backing of most of the major software vendors. The main disadvantage web services currently face, is that the web services architecture standard is still being developed. The standards they rely on, like SOAP, are still ambiguous, resulting in incompatible implementations.

CHAPTER 3

STRONGEST EXISTENTIAL PROPERTY TRANSFORMER

3.1 Introduction to Property Transformers

Not all properties of interest that hold on components are necessarily compositional. There is a need to be able to transform non-compositional properties to compositional properties. Property transformers, that is, functions from properties to properties, allow us to do this.

For example, the strongest existential property transformer $\mathcal{SE}$ is a function from a property to the strongest property that is existential and weaker than the original property. Since $\mathcal{SE}.P$ is weaker than P, any component that satisfies property P, also satisfies the weaker property $\mathcal{SE}.P$. Moreover, since $\mathcal{SE}.P$ is existential, any system containing a component that satisfies this property $\mathcal{SE}.P$ will also satisfy $\mathcal{SE}.P$. For some component satisfying a property P, $\mathcal{SE}.P$ is the strongest property that can be deduced of any system that component is a part of. [6].

Since $\mathcal{SE}$ allows us to express what holds on *any* system based on what we know about components, it is useful to component developers. Component developers can use the $\mathcal{SE}$ property transformer to determine strongest existential properties that

hold on component they develop. This allows them to say the maximum they can about the components they develop without restrictions on the system in which the component can be placed. In this chapter we introduce a new property transformer — γ, and prove some interesting properties of γ.

The **ensures** temporal operator is used to specify liveness properties on components. **ensures** is not existential. In this chapter, we define the $\mathcal{SE}$ for the property P **ensures** Q, and provide a proof of correctness. We also provide an illustrative example of how property transformers can be used for providing compositional specifications.

3.2 The Property Transformer — $\mathcal{SE}$

In the following equation Q is a property that is weaker than property P and existential.

$$Q : [P \Rightarrow Q] \wedge exist.Q \tag{3.1}$$

A property is a solution to Equation 3.1 if and only if it is existential and weaker than P. We know that if the conjunction of all the solutions to an equation is itself a solution to that equation, this conjunction is the *unique strongest solution* to that equation [13].

The strongest existential property weaker than P, $\mathcal{SE}.P$ has been defined in Equation 2.12, as the conjunction of all the solutions to Equation 3.1. It has been proved that $\mathcal{SE}.P$ is itself a solution to Equation 3.1 [6]. Since $\mathcal{SE}.P$ is the conjunction of all the solutions to Equation 3.1 and itself a solution to Equation 3.1, $\mathcal{SE}.P$ is the *unique strongest solution* to Equation 3.1.

3.3 The Property Transformer — γ

We define a new property transformer — γ below.

$$\gamma.P.F \triangleq (\exists H, K, L : F = H \circ K \circ L : P.H \vee P.K \vee P.L) \qquad (3.2)$$

Equation 3.2 says that if $\gamma.P$ holds on some system F, then there exists some way to decompose F, so that the stronger property P holds on at least one component of that decomposition.

The following sub-section lists some properties we have proved about γ.

3.3.1 Properties of γ

Lemma 1. *$\gamma.P$ is weaker than P.*

$$[P \Rightarrow \gamma.P]$$

Proof.

$\gamma.P.F$

$\equiv$ { definition of γ }

$(\exists H, K, L : F = H \circ K \circ L : P.H \vee P.K \vee P.L)$

$\Leftarrow$ { selecting H and $L = UNIT$ }

$(\exists K : F = UNIT \circ K \circ UNIT : P.UNIT \vee P.K)$

$\equiv$ { $UNIT \circ G \circ UNIT \equiv G$ }

$(\exists K : F = K : P.UNIT \vee P.K)$

$\equiv$ { one point rule }

$P.UNIT \vee P.F$

$\Leftarrow$ { $[Q \Rightarrow Q \vee R]$ }

$P.F$

□

Lemma 2. *$\gamma.P$ is existential.*

$$(\,\forall F, G \,:\, F\sqrt{G} \,:\, \gamma.P.F \,\vee\, \gamma.P.G \,\Rightarrow\, \gamma.P.(F \circ G)\,)$$

We first show that $\gamma.P.F \,\Rightarrow\, \gamma.P.(F \circ G)$ where $F\sqrt{G}$

Proof. $\gamma.P.F \,\wedge\, F\sqrt{G}$

$\equiv$ { definition of γ }

$(\,\exists H, K, L \,:\, F = H \circ K \circ L \,:\, P.H \,\vee\, P.K \,\vee\, P.L\,) \,\wedge\, F\sqrt{G}$

$\equiv$ { composing F with G }

$(\,\exists H, K, L \,:\, F \circ G = H \circ K \circ L \circ G \,:\, P.H \,\vee\, P.K \,\vee\, P.L\,)$

$\equiv$ { predicate calculus }

$(\,\exists H, K, L \,:\, F \circ G = H \circ K \circ L \circ G \,:\, P.H\,)$

$\vee\, (\,\exists H, K, L \,:\, F \circ G = H \circ K \circ L \circ G \,:\, P.K\,)$

$\vee\, (\,\exists H, K, L \,:\, F \circ G = H \circ K \circ L \circ G \,:\, P.L\,)$

$\Rightarrow$ { choosing $H' = H \circ K$ and $K' = L$, $L' = L \circ G$ and $L'' = G$ }

$(\,\exists H, K, L' \,:\, F \circ G = H \circ K \circ L' \,:\, P.H \,\vee\, P.K \,\vee\, P.L'\,)$

$\vee\, (\,\exists H', K', L'' \,:\, F \circ G = H' \circ K' \circ L'' \,:\, P.K' \,\vee\, P.H' \,\vee\, P.L''\,)$

$\equiv$ { predicate calculus }

$(\,\exists H, K, L \,:\, F \circ G = H \circ K \circ L \,:\, P.H \,\vee\, P.K \,\vee\, P.L\,)$

$\equiv$ { definition of γ }

$\gamma.P.(F \circ G)$

Similarly, it can be shown that $\gamma.P.G \,\Rightarrow\, \gamma.P.(F \circ G)$ where $F\sqrt{G}$. Thus, from Equation 2.10,

$$exist.(\gamma.P)$$

□

Theorem 1. *$\gamma.P$ is a solution to Equation 3.1.*

Proof. From Lemmas 1 and 2, we know that $[P \Rightarrow \gamma.P] \wedge exist.(\gamma.P)$

Thus, $\gamma.P$ is a solution to Equation 3.1. □

We show that γ is universally disjunctive, γ is monotonic, and for any P that is existential, P is equivalent to $\gamma.P$ in Lemmas 3, 4 and 5 respectively.

Lemma 3. *γ is universally disjunctive, i.e., for any set S,*

$$[\gamma.(\exists P : P \in S : P) \equiv (\exists P : P \in S : \gamma.P)]$$

Proof. $\gamma.(\exists P : P \in S : P).F$

$\equiv$ { definition of γ }

$(\exists H, K, L : F = H \circ K \circ L : (\exists P : P \in S : P).H$

$\vee (\exists P : P \in S : P).K \vee (\exists P : P \in S : P).L)$

$\equiv$ { interchanging existential operators }

$(\exists P : P \in S : (\exists H, K, L : F = H \circ K \circ L : P.H \vee P.K \vee P.L))$

$\equiv$ { definition of γ }

$(\exists P : P \in S : \gamma.P.F)$

$\equiv$ { $(\exists Q : Q \in S : Q.F) \equiv (\exists Q : Q \in S : Q).F$ }

$(\exists P : P \in S : \gamma.P).F$

□

Lemma 4. *γ is monotonic.*

$$[P \Rightarrow Q] \Rightarrow [\gamma.P \Rightarrow \gamma.Q]$$

Proof. $[P \Rightarrow Q]$

$\equiv$ { definition of implication }

$[Q \equiv P \vee Q]$

$\Rightarrow$ { Leibniz }

$[\gamma.Q \equiv \gamma.(P \vee Q)]$

$\equiv$ { γ is universally disjunctive, Lemma 3 }

$[\gamma.Q \equiv (\gamma.P \vee \gamma.Q)]$

$\equiv$ { definition of implication }

$[\gamma.P \Rightarrow \gamma.Q]$

□

Lemma 5. *If P is existential, then P is equivalent to $\gamma.P$.*

$$exist.P \equiv [P \equiv \gamma.P]$$

Proof.

$\boxed{\Leftarrow}$

$[P \equiv \gamma.P]$

$\equiv$ { γ is existential, Lemma 2 }

$[P \equiv \gamma.P] \wedge exist.(\gamma.P)$

$\Rightarrow$ { Leibniz }

$exist.P$

$\boxed{\Rightarrow}$

Assume $exist.P$. We already know from Lemma 2 that $[P \Rightarrow \gamma.P]$.

To show that $[P \equiv \gamma.P]$ all that remains to be shown is $[\gamma.P \Rightarrow P]$

$\gamma.P.F$

$$
\begin{array}{ll}
\equiv & \{ \quad \text{definition of } \gamma.P.F \quad \} \\
& (\exists H, K, L : F = H \circ K \circ L : P.H \vee P.K \vee P.L) \\
\Rightarrow & \{ \quad exist.P \quad \} \\
& (\exists H, K, L : F = H \circ K \circ L : P.(H \circ K \circ L)) \\
\equiv & \{ \quad F = H \circ K \circ L \quad \} \\
& (\exists H, K, L : F = H \circ K \circ L : P.F) \\
\equiv & \{ \quad F = UNIT \circ F \circ UNIT \quad \} \\
& P.F
\end{array}
$$

□

Lemma 6. *γ is the strongest solution of Equation 3.1.*

Proof. To show that $\gamma.P.F = (\exists H, K, L : F = H \circ K \circ L : P.H \vee P.K \vee P.L)$ is the strongest solution to Equation 3.1, we show that any Q which satisfies Equation 3.1 is weaker than $\gamma.P$

$$
\begin{array}{ll}
& [P \Rightarrow Q] \wedge exist.Q \\
\equiv & \{ \quad \text{Lemma } 5 \quad \} \\
& [P \Rightarrow Q] \wedge [Q \equiv \gamma.Q] \\
\Rightarrow & \{ \quad \gamma \text{ is monotonic, Lemma } 4 \quad \} \\
& [\gamma.P \Rightarrow \gamma.Q] \wedge [Q \equiv \gamma.Q] \\
\equiv & \{ \text{ replacing } \gamma.Q \text{ by } Q \quad \} \\
& [\gamma.P \Rightarrow Q]
\end{array}
$$

□

Theorem 2. *Proof.* From the uniqueness of the strongest existential property,

$$[\, \mathcal{SE} \;=\; \gamma \,]$$

□

3.3.2 Significance of γ

We showed that our new property transformer γ is equal to the strongest existential property transformer $\mathcal{SE}$ in Theorem 2. While the original definition of $\mathcal{SE}$ in Equation 2.12 defined $\mathcal{SE}$ as the conjunction of all the solutions to Equation 3.1, our definition of $\mathcal{SE}$ is a fixed-point definition.

This new definition, has a decompositional flavor to it. It states that for any system that satisfies the $\mathcal{SE}$ of some property P, there exists a way to decompose that system so that at least one component of that decomposition satisfies the stronger property P.

This interpretation of $\mathcal{SE}$, can provide us with a way to reverse engineer systems. For example, a developer looking for a component that satisfies the property P, can obtain this component by decomposing *any* system that satisfies the weaker property $\mathcal{SE}.P$. Since $\mathcal{SE}.P$ is weaker than P, it will be satisfied by a larger number of components as compared to P. This provides the developer a larger search-space to look for components satisfying property P.

3.4 Strongest Existential for Progress Property ensures

The progress property P **ensures** Q was defined earlier in Equation 2.3 as

$$(P \textbf{ ensures } Q).F \;\stackrel{\triangle}{=}\; (P \wedge \neg Q \textbf{ next } P \vee Q) \wedge (\exists\, a \,:\, a \text{ in } F \,:\, \{P \wedge \neg Q\} a \{Q\}\,) \tag{3.3}$$

As can be seen from the definition, P **ensures** Q is not existential : the property fails to hold on a system if component F is composed with any component, G that does not satisfy the property:

$$(P \wedge \neg Q \textbf{ next } P \vee Q)$$

ensures is a temporal operator that is used to describe liveness properties that hold on components. Even though the P **ensures** Q is not existential, once we know the SE for this property we know that $SE.(P$ **ensures** $Q)$ is a property of *any* system containing a component satisfying P **ensures** Q. In fact, from our new interpretation of SE, we know that *any* system that satisfies $SE.(P$ **ensures** $Q)$ can be decomposed to obtain a component that satisfies P **ensures** Q.

We will prove that $(\exists\, a :: \{P \wedge \neg Q\}a\{Q\})$ is the strongest property weaker than P **ensures** Q and existential.

Theorem 3. $SE.(P$ **ensures** $Q) \equiv (\exists\, a :: \{P \wedge \neg Q\}a\{Q\})$

From the definition in Equation 3.3, $(\exists\, a :: \{P \wedge \neg Q\}a\{Q\})$ is weaker than P **ensures** Q and existential.

Proof. We need to show that $[(\exists\, a :: \{P \wedge \neg Q\}a\{Q\}) \Rightarrow SE.(P$ **ensures** $Q)]$. This can also be written as

$$(\forall F :: (\exists\, a :: \{P \wedge \neg Q\}a\{Q\}).F \Rightarrow SE.(P \textbf{ ensures } Q).F)$$

This proof consists of two cases. The first where $(\exists\, a :: \{P \wedge \neg Q\}a\{Q\}).F$ is false — in this case, $(\exists\, a :: \{P \wedge \neg Q\}a\{Q\}) \Rightarrow SE.(P$ **ensures** $Q)$ by virtue of the antecedent being **false**. We prove the second case where $(\exists\, a :: \{P \wedge \neg Q\}a\{Q\}).F$ is **true**, below

$(\exists a :: \{P \wedge \neg Q\}a\{Q\}).F$

$\equiv$ { Consider F to be the composition of two components H and K. Component H consists of only the action a and skip. Component K consists of the rest of the actions in F and skip }

$(\exists a :: \{P \wedge \neg Q\}a\{Q\}).(H \circ K)$

$\equiv$ { a in H }

$(\exists a :: \{P \wedge \neg Q\}a\{Q\}).H \wedge (H \sqrt{} K)$

$\equiv$ { H satisfies the definition of P **ensures** Q }

$(P$ **ensures** $Q).H \wedge (H \sqrt{} K)$

$\Rightarrow$ { $[P \Rightarrow SE.P]$, Equation 2.13 }

$SE.(P$ **ensures** $Q).H \wedge (H \sqrt{} K)$

$\Rightarrow$ { SE is existential and composing H and K }

$SE.(P$ **ensures** $Q).(H \circ K)$

$\equiv$ { initial assumption }

$SE.(P$ **ensures** $Q).(F)$

□

3.5 An Illustrative Example

The collection or detection of global state in a distributed system is a challenging task due to the absence of a global clock. One approach to gathering this information is the Chandy-Lamport algorithm for global state detection [24]. In this algorithm, a marker is used to initiate global state detection. This marker does not interfere with the underlying computation.

For this global state detection algorithm to work correctly, there must be one component that initiates the algorithm by performing a Marker Sending rule algorithm (MSR) given in Figure 3.2. In addition, every component in the system must follow the Marker Receiving rule (MRR) in Figure 3.1 upon the receipt of a marker.

We consider the compliance of components with these rules to be properties on the components. For example, any component that obeys MRR satisfies the property *MRR compliant*. Any component that obeys MSR satisfies the property *MSR compliant*. Any system that is capable of global state detection is said to satisfy the property *GSD/compliant*.

If all the components in a system satisfy the property *MRR compliant*, that system is also *MRR compliant* — it obeys the MRR on the receipt of a marker. Thus the property *MRR compliant* is universal.

The component that initiates the global state detection algorithm by performing the MSR, satisfies the property *MRR compliant* **guarantees** *GSD/compliant*. That is, if the entire system is *MRR compliant*, this component on its own guarantees that the entire system is *GSD/compliant*, by initiating the global state detection algorithm. This property is an existential property. Thus, a client developer needs to ensure that in any system that needs to be capable of collecting global state, all the components of the system are *MRR compliant* and the system has at least one component that satisfies the property *MRR compliant* **guarantees** *GSD/compliant*.

We can see from this example that a guarantees property allows us to describe the behavior of the component in any system. It does this by encorporating its requirements on the system as part of the property itself.

On receiving a marker along a channel C:
    If F has not already recorded its state then
        F records its state;
        F records the state of C as the empty sequence
    endif
    else F records the state of C as the sequence of messages received along C
        after F's state was recorded and before F received the marker along C.

Figure 3.1: Marker Receiving rule for component F

For each channel C incident upon and directed away from F:
    F sends one marker along C after F records its state and before F
    sends any further messages along C.

Figure 3.2: Marker Sending Rule for Component F

We can also express other properties of the components. Let F be the component that is capable of initiating the algorithm. This component can be in one of 3 states: computing (C), collecting state (CS) or idle (I).

Then,

$(\forall a : a \text{ in } F : \{C \wedge \neg SC\} a \{C \vee SC\}).F \wedge (\exists a : a \text{ in } F : \{C \wedge \neg SC\} a \{SC\}).F$

This can be written as $(C \textbf{ ensures } SC).F$. Tha is, if this component ever enter the state C, it will remain in a state satisfying C, or enter a state satisfying SC in the next step. From the result in Theorem 3, we know that $(\exists a : a \text{ in }$

$F : \{C \wedge \neg SC\}a\{SC\}$).F is the $\mathcal{SE}$ for the above property. Hence, this property holds on any system that contains component F.

In this example, even though the property C **ensures** SC that held on the component was not existential itself, we were able to use the $\mathcal{SE}$ property transformer, to obtain an existential property that did hold on the component.

Moreover, from the new interpretation of $\mathcal{SE}$ that we provided in Section 3.3 we know that *any* system that satisfies $(\exists a : a \text{ in } F : \{C \wedge \neg SC\}a\{SC\})$, can be decomposed such that C **ensures** SC holds on at least one component of that decomposition.

3.6 Summary

In this chapter, we provided an alternative fixed-point representation for strongest existential properties in Equation 3.2. This alternate characterization of $\mathcal{SE}$ denotes that any system that satisfies a strongest existential property $\mathcal{SE}.P$ can be decomposed so that one component of that decomposition satisfies the stronger property P. We provided a proof of correctness of this alternative representation, and then derived the strongest existential for a non-existential property **ensures** . We then provided an example of a situation where this property would be useful. The strongest existential property $\mathcal{SE}.P$ for any given property P is particularly useful to component developers who need to advertise the components they develop by providing strong descriptions of the components without restricting themselves to a particular deployment environment. Since the strongest existential property $\mathcal{SE}.P$ is the strongest

property weaker than P it represents the strongest property that holds on *any* system containing at least one component satisfying P, and can be used by developers to describe properties that hold on developed components.

CHAPTER 4

STRONGEST UNIVERSAL PROPERTY TRANSFORMER

4.1 The Property Transformer — *SU*

The strongest universal property transformer SU is a function from a property to the strongest property that is universal and weaker than the original property. Since $SU.P$ is weaker than P any component that satisfies property P, also satisfies the weaker property $SU.P$. Moreover, since $SU.P$ is universal, any system containing only components that satisfies this property $SU.P$ will also satisfy $SU.P$. Intuitively, $SU.P$ is the strongest property that can be deduced of any system that contains only components that satisfies the property P [6].

In the following equation Q is a property that is weaker than property P and universal.

$$Q : [P \Rightarrow Q] \wedge univ.Q \tag{4.1}$$

A property is a solution to Equation 4.1 if and only if it is universal and weaker than P. We know that if the conjunction of all the solutions to an equation is itself a solution to that equation, this conjunction is the unique strongest solution to that equation [13].

The strongest universal property weaker than P, $SU.P$ has been defined in Equation 2.17, as the conjunction of all the solutions to Equation 4.1. It has been proved that $SU.P$ is itself a solution to Equation 4.1 [6]. Since $SU.P$ is the conjunction of all the solutions to Equation 4.1 and itself a solution to Equation 4.1, $SU.P$ is the *unique strongest solution* to Equation 4.1.

4.2 The Property Transformer — δ

We define a new property transformer — δ below, where $n \geq 1$

$$\delta.P.F \triangleq (\exists F_1, F_2, ..F_n : F = F_1 \circ F_2 ... \circ F_n : P.F_1 \wedge P.F_2 ... \wedge P.F_n) \quad (4.2)$$

Equation 4.2 says that if $\delta.P$ holds on some system F, then there exists some finite decomposition of that system F, so that the stronger property P holds on all the constituent components of that decomposition.

The following sub-section lists some properties we have proved about δ. We show that $\delta.P$ is weaker than P, universal, and that every other solution to Equation 4.1 is weaker than $\delta.P$. Thus, by the uniqueness of a strongest solution to Equation 4.1 δ is equal to SU.

4.2.1 Properties of δ

Lemma 7. *$\delta.P$ is weaker than P.*

$$[P \Rightarrow \delta.P]$$

Proof. $P.F$

$\equiv$ { selecting $F_1 = F$ }

$(\exists F_1 : F = F_1 : P.F_1)$

$\Rightarrow$ { predicate calculus }

$(\exists F_1, F_2 ... F_n : F = F_1 \circ F_2 ... \circ F_n : P.F_1 \wedge P.F_2 ... \wedge P.F_n)$

$\equiv$ { definition of δ }

$\delta.P.F$

□

Lemma 8. *$\delta.P$ is universal.*

$$(\forall F, G : F \surd G : \delta.P.F \wedge \delta.P.G \Rightarrow \delta.P.(F \circ G))$$

We assume that $F \surd G$ and $\delta.P.F \wedge \delta.P.G$.

Proof. $\delta.P.F \wedge \delta.P.G \wedge F \surd G$

$\equiv$ { definition of δ }

$(\exists F_1, F_2 ... F_{n1} : F = F_1 \circ F_2 ... \circ F_{n1} : P.F_1 \wedge P.F_2 ... \wedge P.F_{n1})$

$\wedge (\exists G_1, G_2 ... G_{n2} : G = G_1 \circ G_2 ... \circ G_{n2} : P.G_1 \wedge P.G_2 ... \wedge P.G_{n2})$

$\wedge F \surd G$

$\equiv$ { composing F with G }

$(\exists F_1, F_2 ... F_{n1}, G_1, G_2, ... G_{n2} : F \circ G = F_1 \circ F_2 ... \circ F_{n1} \circ G_1$

$\circ G2 ... \circ G_{n2} : P.F_1 \wedge P.F_2 ... \wedge P.F_{n1} \wedge P.G_1 \wedge P.G_2 ... \wedge P.G_{n2})$

$\Rightarrow$ { predicate calculus }

$(\exists H_1, H_2 ... H_n : F \circ G = H_1 \circ H_2 ... \circ H_n : P.H_1 \wedge P.H_2 ... \wedge P.H_n)$

$\equiv$ { definition of $\delta.P$ }

$\delta.P.(F \circ G)$

Thus, from Equation 2.11,

$$univ.(\delta.P)$$

□

Theorem 4. *$\delta.P$ is a solution to Equation 4.1.*

Proof. From Lemmas 7 and 8, we know that $[P \Rightarrow \delta.P] \wedge univ.(\gamma.P)$

Thus, $\delta.P$ is a solution to Equation 4.1. □

We show that $[\delta.(P1 \wedge P2) \Rightarrow \delta.P1 \wedge \delta.P2]$, δ is monotonic, and for any P that is universal, P is equivalent to $\delta.P$ in Lemmas 9, 10 and 11 respectively.

Lemma 9.

$$[\delta.(P1 \wedge P2) \Rightarrow \delta.P1 \wedge \delta.P2]$$

Proof. $\delta.(P1 \wedge P2).F$

$\equiv$ { definition of δ }

$(\exists F_1, F_2 ... F_n : F = F_1 \circ F_2 ... \circ F_n : (P1 \wedge P2).F_1$

$\wedge (P1 \wedge P2).F_2 ... \wedge (P1 \wedge P2).F_n)$

$\Rightarrow$ { $P \wedge Q \Rightarrow P$ and $F_1 = G_1 = H_1, ... F_n = G_n = H_n$ }

$(\exists G_1, G_2 ... G_n : F = G_1 \circ G_2 ... \circ G_n : P1.G_1 \wedge P1.G_2 ... \wedge P1.G_n)$

$\wedge (\exists H_1, H_2 ... H_n : F = H_1 \circ H_2 ... \circ H_n : P2.H_1 \wedge P2.H_2 ... \wedge P2.H_n)$

$\equiv$ { definition of δ }

$\delta.P1.F \wedge \delta.P2.F$

$\equiv$ { $P.F \wedge Q.F \equiv (P \wedge Q).F$ }

$\delta.P1 \wedge \delta.P2$

□

Lemma 10. *δ is monotonic*

$$[P \Rightarrow Q] \Rightarrow [\delta.P \Rightarrow \delta.Q]$$

Proof. $[P \Rightarrow Q]$

$\equiv$ { definition of implication }

$[P \equiv P \wedge Q]$

$\Rightarrow$ { Leibniz }

$[\delta.P \equiv \delta.(P \wedge Q)]$

$\Rightarrow$ { $\delta.(P1 \wedge P2) \Rightarrow \delta.P1 \wedge \delta.P2$, (Lemma 9) }

$[\delta.P \Rightarrow (\delta.P \wedge \delta.Q)]$

$\equiv$ { definition of implication }

$[\neg\delta.P \vee (\delta.P \wedge \delta.Q)]$

$\equiv$ { distributivity of $\vee$ over $\wedge$ }

$[\neg\delta.P \vee \delta.P \wedge \neg\delta.P \vee \delta.Q)]$

$\equiv$ { $\neg P \vee P \equiv$ **true** }

$[\neg\delta.P \vee \delta.Q)]$

$\equiv$ { definition of $\Rightarrow$ }

$[\delta.P \Rightarrow \delta.Q]$

□

Lemma 11. *If P is universal, then P is equivalent to $\delta.P$.*

$$univ.P \equiv [P \equiv \delta.P]$$

Proof. $\boxed{\Leftarrow}$

$[P \equiv \delta.P]$

$\equiv$ { δ is universal }

$[P \equiv \delta.P] \wedge univ.(\delta.P)$

$\Rightarrow$ { Leibniz }

$univ.P$

$\boxed{\Rightarrow}$

Assume $univ.P$. We already know from Lemma 7 that $[P \Rightarrow \delta.P]$. To show that $[P \equiv \delta.P]$ all that remains to be shown is $[\delta.P \Rightarrow P]$

$\delta.P.F$

$\equiv$ { definition of $\delta.P.F$ }

$(\exists G_1, G_2 ... G_n : F = G1 \circ G_2 ... \circ G_n : P.G_1 \wedge P.G_2 ... \wedge P.G_n)$

$\Rightarrow$ { $univ.P$ }

$(\exists G_1, G_2, ... G_n : F = G_1 \circ G_2 ... \circ G_n : P.(G_1 \circ G_2 ... \circ G_n))$

$\equiv$ { $F = G_1 \circ G_2 ... \circ G_n$ }

$(\exists G_1, G_2 ... G_n : F = G_1 \circ G_2 ... \circ G_n : P.F)$

$\equiv$ { $F = F$ }

$P.F$

□

Lemma 12. *δ is the strongest solution of Equation 4.1*

Proof. To show that $\delta.P.F = (\exists G_1, G_2 ... G_n : F = G_1 \circ G_2 \circ G_n : P.G_1 \wedge P.G_2 ... \wedge P.G_n)$ is the strongest solution to Equation 4.1, we must show that any Q which satisfies Equation 4.1 is weaker than $\delta.P$

$[P \Rightarrow Q] \wedge univ.Q$

$\equiv$ { Lemma 11 }

$[P \Rightarrow Q] \wedge [Q \equiv \delta.Q]$

$\Rightarrow$ { δ is monotonic, Lemma 10 }

$$
\begin{array}{ll}
 & [\delta.P \;\Rightarrow\; \delta.Q] \;\wedge\; [Q \;\equiv\; \delta.Q] \\
\equiv & \qquad \{ \text{ replacing } \delta.Q \text{ by } Q \quad \} \\
 & [\delta.P \;\Rightarrow\; Q]
\end{array}
$$

□

Theorem 5. *Proof.* From the uniqueness of the strongest universal property,

$$[\, \mathcal{SU} \;=\; \delta \,]$$

□

4.2.2 Significance of δ

We showed that our new property transformer δ is equal to the strongest universal property transformer $\mathcal{SU}$ in Theorem 5. While the original definition of $\mathcal{SU}$ in Equation 2.12 defined $\mathcal{SU}$ as the conjunction of all the solutions to Equation 3.1, our definition of $\mathcal{SE}$ is a fixed-point definition.

This new definition, has a decompositional flavor to it. It states that for any system that satisfies the $\mathcal{SU}$ of some property P, there exists a way to decompose that system so that all the components of that decomposition satisfy the stronger property P. This interpretation of $\mathcal{SU}$, can also provide us with a way to reverse engineer systems. For example, a developer looking for a component that satisfies the property P, can obtain this component by decomposing *any* system that satisfies the weaker property $\mathcal{SU}.P$.

4.3 Summary

In this chapter, we provided an alternative representation for strongest universal properties in Equation 4.2. This alternate characterization of $\mathcal{SU}$ denotes that any

system that satisfies a strongest universal property $\mathcal{SU}.P$ can be decomposed so that all the components of that decomposition satisfy the stronger property P. We provided a proof of correctness of this alternative representation.

CHAPTER 5

INTEGRATING COMPOSITIONAL SPECIFICATIONS USING CURRENT TECHNOLOGIES

5.1 Introduction

Distributed systems, although difficult to develop and reason about, offer the advantage of reuse of developed components due to their inherent loosely-coupled nature. Academic research dealing with the complexity in developing distributed systems often focuses on formal methods. The existential and universal properties we've seen earlier are a technique for enhancing formal methods of reasoning. The industry has been trying to deal with the same problem of complexity through the development of tools and middleware like CORBA [14] and web services [15]. These tools ease the process of development by hiding a significant amount of the underlying communication and development details. The need for behavioral specifications, however, still remains since these aid the client developer in terms of reasoning about components. We focus on two popular technologies, CORBA and web services, in order to see how we could incorporate advances in formal methods using current technologies.

5.2 An Approach to Encorporating Compositional Specifications

We consider two fundamental aspects to compositional development that these development frameworks should embody. The first is the need for component descriptions. In component-based enterprise frameworks such as CORBA, a component description consists simply of the component's syntactic interface; that is, a list of method signatures implemented by the component. The inadequacies of this approach are well- known and there have been several attempts to extend such interface descriptions with formal behavioral descriptions [4, 25]. Similarly, in the case of web services, a component description, is a list of messages that the web service accepts, and its replies. We would like to enhance these component descriptions to include behavioral specifications. In particular, we propose that component specifications consist of: (i) a traditional interface (method signatures), (ii) a declaration of abstract state (iii) a behavioral specification in terms of safety, liveness and compositional properties (iv) an informal description.

Secondly, in addition to good component descriptions, in order to support reuse, components and their associated descriptions of behavior must be easy to find and identify. Component providers must be able to advertise or publish these descriptions and component consumers must be able to browse or search these advertisements. All industrial middleware technologies already support some form of publication service or catalogue for syntactic interfaces. For example the CORBA standard defines several services such as the naming and trading services that are used to publish component descriptions and search for component descriptions. While the naming services require that the exact name of the required service be known, trading services allow queries

based on method signatures and simple properties. Similarly, web services make use of the UDDI registry to publish web service descriptions and allow searches for web services. We would like these services or catalogues to be naturally extended to include the augmented interfaces described earlier.

In Sections 5.3 and 5.4 we focus on how we can incorporate behavioral specifications and publising based on these specifications in CORBA and web services respectively.

The example we use in this chapter is that of a simple counter component. This component provides two methods:`Inc()` and `Clear()`. It maintains a local integer and increases this integer based on calls to its method `Inc`. A call to `Clear()`, resets the local integer to zero. The counter does not spontaneously change the value of the local integer. Informally, this component's safety specification states that either `Inc()` is invoked and the local integer increases, or `Clear()` is called and the local integer is reset, or no method is invoked and the count remains unchanged. There is no progress property for this particular component. In addition it is easy to see that if `Clear()` is never called, then, the value of the local variable will be non-decreasing. This property of the counter depends upon the behavior of its environment, and can be expressed as a **guarantees** property. That is, the counter component guarantees that its count is non-decreasing unless the environment calls its `Clear()` operation. The specification for this component is given in Figure 5.1.

5.3 CORBA

CORBA is used to build large heterogeneous systems that use the services of distributed components. These distributed objects can be viewed as components

Interface

Inc();

Clear();

Abstract State

integer: count

Safety

initially.$(count = 0)$

$(\forall j, k, l :: \#Inc = j \ \wedge \ \#Clear = k \ \wedge \ count = l$

next $\#Inc = j \ \wedge \ \#Clear = k \ \wedge \ count = l$

$\vee \ \#Inc = j + 1 \ \wedge \ \#Clear = k \ \wedge \ count = l + 1$

$\vee \ \#Inc = j \ \wedge \ \#Clear = k + 1 \ \wedge \ count = 0 \ )$

Existential

invariant.$(\#Clear = 0)$ **guarantees**

$(\forall l :: ((count = l)$ **next** $(l \leq count \leq l + 1)) \ )$

Informal Specification:

Inc() increases count by 1

Clear() sets count to 0

Figure 5.1: Description of Counter Component

and they provide services based on calls made to the methods they implement. In this model, composition is simply the acquisition of an object reference by another object. This object reference permits method invocations and hence interaction with a component's environment. (Here we ignore the Event Service, which provides event streams for interacting with the environment.)

5.3.1 Attaching Component Behavioral Specifications

In order to allow developers to specify their components a certificate-based extension to the CORBA IDL was developed *cidl* [4, 25]. Ordinary IDL captures interface names and method signatures. The CIDL extension allows the inclusion of abstract state and behavioral properties in a component's IDL. The extension makes use of pragmas to add information to the specification. According to the CORBA

```
interface counter {
  void Inc();
  void Clear();
  #pragma state short count;
  #pragma spec initially.(count == 0);
  #pragma spec (Forall j,k,l: (#Inc == j && #Clear == k && count == l)
      next (#Inc == j && #Clear == k && count == l ) or
           (#Inc == j+1 && #Clear == k && count == l+1 ) or
           (#Inc == j && #Clear == k+1 && count == 0 ));
  #pragma Existential (invariant.(#Clear==0) guarantees
           (Forall l: count==l next l <= count <= l+1));
}
```

Figure 5.2: cidl Description of Counter Component

standard, unrecognized pragmas are ignored by IDL parsers, thus backward compatibility of CIDL is guaranteed. We further extend CIDL to include existential and universal properties. For example, Figure 5.2 shows the CIDL description for the counter component. The #pragma state statement denotes the components abstract state. #pragma spec statements denote the safety and liveness properties that hold on components. #pragma Existential and #pragma Universal are used to denote compositional properties.

5.3.2 Publishing Specifications

The OMG Naming and Trading Services, are two of several services defined by the CORBA standard. A naming service permits clients to locate components by providing a symbolic name. This method of looking for components is not practical unless the client is aware of the exact symbolic name to look for. The CORBA Trading Service allows a rich set of queries based on method signatures and simple component properties. A trader stores advertisements known as *service offers* which

PropertyName	PropertyType	PropertyMode
AbstractState	CORBA::tcstring	Normal
Interface	CORBA::tcstring	Normal
Safety	CORBA::tcstring	Normal
Liveness	CORBA::tcstring	Normal
Exist	CORBA::tcstring	Normal
Univ	CORBA::tcstring	Normal
Informal	CORBA::tcstring	Normal

Figure 5.3: Properties for `ComponentSpec` Service

are descriptions of services along with an reference to the service provider. Services offers must have a specific *service type*. Service types correspond to categories and determine the kind of information that a client can expect from a service offer. The act of placing an advertisement, is called an *export* operation. The act of searching for a service is performed through an *service request* operation. Service requests contain constraint expressions which are used to determine the service offers that match the service request.

We leverage the discipline of publishing component interfaces with trading services to support the publication of compositional behavioral descriptions as well.

We define a new CORBA service type, the `ComponentSpec` service that includes the compositional properties . This service type has the fields for this service type listed in Figure 5.3 . All the fields are optional and modifiable by the exporter as indicated by the Normal mode. The field values are strings.

In order for a trader to be able to advertise service offerings from components meeting this `ComponentSpec` service, the trader needs to have added this new service type to its type repository. The trader must also provide the functionality to

allow component providers to create service offerings for this new type, allow client applications to search for service offers, perform imports, *etc.*

Now, any component that is developed to support these kind of compositional specifications needs to export a service offer of `ComponentSpec` service type to the trader with all the fields filled in appropriately. So for example, in the case of the counter component, the information present in Figure 5.2 will be used to export a service offer of `ComponentSpec` type.

Client developers can locate components by submitting service requests of service type `ComponentSpec`, and specifying constraints on the contents of the service offer.

This approach to attaching specifications by extending the interface definition language, allows a nice separation of the component interface and implementation. Secondly, the entire component description lies in one place — the interface, and can be retrieved from one place — the trader.

5.4 Web Services

A web service is a programming application that exposes an interface accessible over the Internet via standard Internet protocols and data exchange formats. In general, web service providers deploy services that can be used by distributed client applications. When a web-service is used by a client application it fits our model of a distributed component that is dynamically composed with the client application. Thus, web service providers should provide as much information as they can to client developers in order to help client developers reason about the behavior of this web service in their client application.

Web service interface descriptions are written in WSDL [21] and web-services will use the UDDI business registry to publish their interfaces and allow searches for services [22].

5.4.1 Attaching Component Behavioral Specifications

The component descriptions which are descriptions of messages, replies and bindings for the service, are described using WSDL and stored at the UDDI registry. WSDL is an XML based language, that allows us to describe this interface in a uniform manner.

We could follow the same approach as we did with CORBA — that of extending the Interface Definition Language in order to provide behavioral specifications with components. WSDL has a tag — the documentation tag that can contain any arbitary text or XML elements. This tag can be used to attach structured behavioral specifications. WSDL however, differs from IDL. Unlike IDL, WSDL is generally generated from the component source code using tools, as opposed to being used to generate service skeletons. Thus behavioral specifications for components need to be incorporated into the tool-generated WSDL for a service either manually, or by inserting them previously into the service code, so that tools can automatically generate the WSDL with attached behavioral specifications.

Another way to attach behavioral specifications would be to create a standard interface that includes methods like *get_Existential* and *get_Universal*. Every component that wants to indicate that it comes with attached behavioral specifications, must implement this standard interface by providing implementations of methods like *get_existential*. We prefer the first method of attaching specifications because the

behavioral descriptions of components reside in the component's WSDL description, that is its interface, instead of being a value returned by the service on invocation of the appropriate method.

Neither of our suggested techniques however allow for a clean separation of the service interface and service implementation. In both cases, the component behavioral description must be embedded with the component implementation or we must manually attach these behavioral specifications. This however, is a result of the web services architecture model where service descriptions are generated from implementations, unlike CORBA's model where the interface is used to obtain the service skeleton.

5.4.2 Publishing Specifications

The UDDI registry stores web services descriptions in the form of WSDL descriptions of the messages and replies for services and service bindings.

UDDI uses a data structure called the *tModel* to define industry standards. In order to standardize our component description format, we define a new tModel, `ComponentSpec Service`, that will be used to fingerprint web services that wish to indicate compliance with this compositional specification format. Figure 5.4 illustrates the `ComponentSpec Service` tModel. This tModel will be completely described in a WSDL document *cs.wsdl* that describes the service interface and protocol bindings for obtaining component specifications.

Services that want to indicate that they come with behavioral specifications do so by indicating compliance with this new tModel.

```
<tModel authorizedName="..." ...>
  <name> ComponentSpec Service </name>
  <description xml:lang="en">
   WSDL Description of a ComponentSpec
   service interface
  </description>
  <overviewDoc>
   ...
     <overviewURL>
      http://Component-definitions/cs.wsdl
     </overviewURL>
   ...
  <keyedReference tModelkey="uuid:C12345..." ...
   keyvalue="wsdlSpec"/>
</tModel>
```

Figure 5.4: tModel for the ComponentSpec Service

The UDDI *businessService* data structure is used to describe the services provided by a web service and the location where the web service can be accessed. Any web service provider (or component provider in our case) who wants to provide services that conform to our compositional specification format will use this `ComponentSpec` service tModel in the UDDI businessService data structure used to describe access to their service. In addition, the businessService will provide the accesspoint or location of the service. For example, the provider for the counter component will use our `ComponentSpec` service as shown in Figure 5.5.

Thus using the *tModel* structure provided by UDDI, we are able to classify components that do have accompanying compositional specifications.

```
<businessService businessKey="..." serviceKey="...">
  <name> CounterService </name>
    ...
    <bindingTemplates>
      <bindingTemplate>
        <accessPoint urlType="http">
          "http://www.sample.com/counter"
        </accessPoint>
      <tModelInstanceDetails>
        <tModelInstanceInfo tModelkey= "uuid:C12345">
        </tModelInstanceInfo>
      </tModelInstanceDetails>
    </bindingTemplate>
  </bindingTemplates>
</businessService>
```

Figure 5.5: businessService structure for counter component

The UDDI registry however, provides limited functionality for searching for components as opposed to CORBA's rich selection of services. The UDDI registry searching capabilities are based on the assumption that client developers know exactly what they are looking for. Thus, UDDI allows for services searches based on the service name, service providers name and implemented tModels.

Thus, even when components indicate through their businessService that they have behavioral specifications, the UDDI registry only helps client developers find a list of *all* components that do this. The client developer must now obtain each components WSDL individually in order to determine the actual behavioral specifications.

5.5 Summary

In this chapter, we described techniques to incorporate compositional behavioral specifications in two popular component-based development frameworks: CORBA and web services.

CORBA being a mature technology, provides a more flexible framework for incorporating behavioral specifications as opposed to web services. By simply extending the CORBA IDL and creating a new service type, we are able to describe and advertise components that have compositional properties.

Web services are still a very new technology. Although we were able to describe techniques to attach behavioral specifications to components, we were unable to separate the component interface and implementation cleanly. The querying services provided by web services were also not flexible enough to efficiently search for services based on their compositional properties. Instead client developers need to query each service in turn to find a suitable service. Most of the web services architecture is still not standardized and it is likely that the model will mature to allow us to incorporate these compositional behavioral specifications.

CHAPTER 6

CONCLUSIONS

6.1 Related Work

It is generally recognized that hierarchical decomposition of programs into smaller ones is necessary to handle the complexity in reasoning about large programs [26]. Moreover, it is necessary that we be able to reason about the larger system based on specifications of constituent components without knowledge of their internal details. To make this kind of reasoning possible, systems and components are specified using predicates that hold over their *observable behavior*. These predicates do not depend on any knowledge of the internal details of the constituent components. These kind of specifications are called *assertional* specifications.

6.1.1 Sequential Systems

Sequential systems have one thread of control. In the case of sequential programs, observable behavior can be fully-specified by pairs of initial and final states []. Two sequential programs with the same initial and final states are considered equivalent regardless of the differences in their intermediate steps. Hoare Logic is a formal axiom system for proving Hoare Triples of the form $\{pre\}S\{post\}$ [8]. S represents

a program construct and *pre* and *post* are assertions. Since the observable behavior of sequential program constructs is represented by pairs of initial and final states, this behavior can be specified in Hoare Logic. A large sequential system can be reasoned about on the basis of the specifications of its constituent program constructs without knowledge of the internal details of these components using Hoare Logic. The existential and universal properties we've discussed are general enough to be applicable to sequential systems as well. [9].

6.1.2 Concurrent Systems

Distributed systems tend to have multiple threads of control that potentially execute in parallel. These concurrent threads communicate and synchronize with each other. Concurrent systems may communicate by means of shared memory or message passing. Distributed systems, due to the geographic distribution of constituent components, make use of message passing to carry out communication. The possibility of synchronization and communication between the concurrent components makes the intermediate states as important as the initial and final states. Thus, the observable behavior of these systems cannot be fully specified using only initial and final states. Hoare Logic is not compositional when concurrency is a part of the system [9], and is inadequate to specify distributed concurrent systems. Instead, *assume-guarantee*-type formalisms [27] are used to specify distributed concurrent systems. According to this formalism, in order to specify concurrent processes, their initial and final state behavior must be specified along with their interaction at intermediate points. A *rely-guarantee* formalism to specify concurrent systems using shared-variable communication was proposed by Cliff Jones [28]. Another formalism applicable to synchronous

message passing systems is the *assumption-commitment* formalism proposed by Misra and Chandy [29]. Both the rely-guarantee as well as assume-commitment formalisms are *assumption-guarantee* type formalisms.

6.1.3 Assume-Guarantee Formalisms

The rely-guarantee formalism specifies the amount of interference allowed from the environment of a component during its execution without endangering fulfilment of the purpose of that component [28]. This formalism is an extension of the Hoare triples format, with two additional predicates: a rely predicate that specifies the expected behavior of the environment, and a guarantee predicate that represents the task to be performed by that component. Both predicates characterize atomic actions. A rely-guarantee formula is represented as shown below

$$< rely, guar >: \{pre\}P\{post\}$$

pre and *post* are conditions on the initial and final states of a process P, and *rely* and *guarantee* are conditions on the environment of P.

The assumption-commitment formalism is similar to rely-guarantee, but applies to synchronous message passing systems. Here, the *assumption* and *commitment* predicates are predicates over the history of the component.

While the assume-guarantee formalisms for specifying concurrent systems are compositional, these techniques require the explicit description of possible environments in which the component might be placed, which is not the case with existential/universal type of specifications. The assume-guarantee type specifications are like component properties coupled with specific environments [5]. The universal/existential type specifications are just component properties.

6.2 Future Work

The work we have presented contributes a step toward developing a general theory of composition. A significant amount of work is needed though before our overall goal can be achieved. Some directions we believe this work can be extended include determining predicate transformers for other temporal operators like *leads-to*. The development of tools that can be used to support compositional development using compositional specifications will also be useful from a practical standpoint.

6.2.1 Property Transformers for Progress Property Leads-to

Almost all progress properties on components are expressed using the $\leadsto$ operator. The progress property $leads-to$ is neither existential nor universal.

We know that $SE.(P \leadsto Q)$ cannot be **true**, because if that were the case, then according to our characterization of $\mathcal{SE}$ in Equation 3.2, there should be a way to decompose *any* system (all systems satisfy **true**) so that at least one component of that decomposition satisfies $P \leadsto Q$. However, the $UNIT$ component satisfies **true**, and cannot be decomposed so that at least one component of that decomposition satisfies $P \leadsto Q$. Hence we conclude that there must be some property stronger than **true** that is the strongest existential property for $\leadsto$. One important direction for this work is to find out what this strongest existential property for $\leadsto$ is.

6.2.2 Tools to Aid Compositional Development

In a compositional development environment, ideally the major development work should lie in decomposing system requirements, and locating components that can provide these system properties. A development framework should facilitate the task

of reading accompanying component specifications. Moreover, an integrated development environment should seamlessly obtain a component's specification, display it to the developer, and calculate the result of the composition of different components (based on their accompanying compositional specifications). For example, on obtaining a component's specification, the tool would highlight that any system containing this component will have its existential properties.

Another useful task to automate is checking whether two given specifications are a match. Searches based on matching specifications are difficult, because a match must be made between a property as it is specified by the client and the specification as it is provided by the implementer. Model checkers can be used to check for matches in certain cases [30]. While it would be useful to be able to automate the decomposition of a given system specification into smaller specifications, or to calculate the result of combining specifications, or to determine whether a given property is existential or universal, these tasks are, in general, incomputable.

6.3 Summary

Our overall research goal is to develop a theory for the compositional construction of large software systems based on our knowledge of smaller simpler components. This requires that we are able to reason about the behavior of large systems and determine their specifications based on knowledge of the constituent components specifications only. To achieve this, component behavior needs to be specified using properties that lend themselves easily to composition.

Existential and universal properties are classes of properties with particularly simple rules for composition. An existential property is a property that holds on a system

whenever the system contains at least one component that has that property. A universal property is one that holds on a system whenever all the constituent components have that property. Properties that hold on components however, do not necessarily exhibit this compositional behavior. For example, the **ensures** property is not existential. Property transformers are functions that are used to transform non-compositional properties to their stronger/weaker compositional form. This allows us to specify the behavior of components in terms of stronger/weaker compositional properties despite the fact that the properties that hold on the component are themselves non-compositional.

Toward our research goal, we have provided a fixed-point representation of the strongest existential property transformer. We have also provided a fixed point representation for the strongest universal property transformer. Finally, we have described techniques by means of which, compositional specifications can be attached to components using current technologies like CORBA and web services.

BIBLIOGRAPHY

[1] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems*, vol. 1. Specification. 175 Fifth Avenue, New York, New York 10010: Springer-Verlag, 1992.

[2] B. Alpern and F. B. Schneider, "Defining liveness," *Information Processing Letters*, vol. 21, pp. 181–185, October 1985.

[3] P. A. G. Sivilotti, *A Method for the Specification, Composition, and Testing of Distributed Object Systems.* PhD thesis, California Institute of Technology, 256-80 Caltech, Pasadena, California 91125, December 1997. Available as CS-TR-97-31.

[4] P. A. G. Sivilotti and C. P. Giles, "The specification of distributed objects: Liveness and locality," in *Proceedings of CASCON '99* (S. A. MacKay and J. H. Johnson, eds.), (Toronto, Ontario, Canada), pp. 150–160, December 1999.

[5] M. Charpentier and K. M. Chandy, "Theorems about composition," in *Mathematics of Program Construction*, pp. 167–186, 2000.

[6] M. Charpentier and K. M. Chandy, "Reasoning about composition using property transformers and their conjugates," in *IFIP TCS*, pp. 580–595, 2000.

[7] R. W. Floyd, "Assigning meanings to programs," in *Proceedings Symposium on Applied Mathematics*, vol. 19, pp. 19–31, 1967.

[8] C. A. R. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, vol. 12, pp. 576–583, October 1969.

[9] W. P. deRoever et al., *Introduction to Compositional and Noncompositional Methods.* Cambridge, 2001.

[10] P. A. G. Sivilotti, "Specifying and testing the progress properties of distributed components," in *Workshop on Testing Distributed Component-Based Systems*, May 1999. part of the 21st International Conference on Software Engineering (ICSE).

[11] K. M. Chandy and J. Misra, *Parallel Program Design: A Foundation*. Reading, Massachusetts: Addison-Wesley Publishing Company, 1988.

[12] J. Misra, "A logic for concurrent programming: Safety," *Journal of Computer & Software Engineering*, vol. 3, no. 2, pp. 239–272, 1995.

[13] E. W. Dijkstra and C. S. Scholten, *Predicate Calculus and Program Semantics*. Texts and Monographs in Computer Science, 175 Fifth Avenue, New York, New York 10010: Springer-Verlag, 1990.

[14] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, February 2001. Revision 2.4.2.

[15] W3C Web Services Architecture Working Group, *W3C Web Services Architecture Requirements, Working Draft*, April 2002. available at www.w3c.org/2002/WD-wsa-reqs-20020429.

[16] Borland Corporation, *Visibroker For C++ 4.5, Product Documentation*, 2001. available at www.info.borland.com/techpubs/books/vbcpp.

[17] Object-Oriented Concepts, Inc., *ORBacus For C++ and Java*. Version 3.1.

[18] W3C XML Core Working Group, *XML 1.1, Working Draft*, April 2002. available at www.w3.org/TR/XML11.

[19] W3C XML Schema Working Group, *XML Schema:Formal Description, Working Draft*, March 2001. available at www.w3.org/TR/2001/WD-xmlschema-formal-20010320/.

[20] W3C Web Services Activities Group, *SOAP Version 1.2 Part1:Messaging Framework, Working Draft*, June 2002. available at www.w3.org/TR/2002/WD-soap12-part1-20020626.

[21] W3C Web Services Description Group, *Web Services Description Language (WSDL)1.1, Working Draft*, July 2002. available at www.w3.org/TR/wsdl12.

[22] UDDI Working Group, *UDDI Version 3.0*, July 2002. available at www.uddi.org/pubs/uddi-v3.00-published-20020719.htm.

[23] J. Richter, "Microsoft .NET framework delivers the platform for an integrated service oriented web," *MSDN Magazine*, vol. 15, no. 9, 2000. available at http://msdn.microsoft.com/msdnmag/issues/0900/Framework/Framework.asp.

[24] K. M. Chandy and L. Lamport, "Distributed snapshots: Determining global states of distributed systems," *ACM Transactions on Computer Systems*, vol. 3, pp. 63–75, February 1985.

[25] P. Krishnamurthy and P. A. G. Sivilotti, "The specification and testing of quantified progress properties in distributed systems," in *Proceedings of the 23rd International Conference on Software Engineering (ICSE)*, (Toronto, Canada), IEEE and ACM SIGSOFT, May 2001.

[26] E. W. Dijkstra, "Solution of a problem in concurrent programming control," *Communications of the ACM*, vol. 8, no. 9, 1965.

[27] M. Abadi and L. Lamport, "Composing specifications," *ACM Transactions on Programming Languages and Systems*, vol. 15, pp. 73–132, January 1993.

[28] C. B. Jones, "Tentative steps toward a development method for interfering programs," *ACM Transactions on Programming Languages and Systems*, vol. 5, no. 4, pp. 596–619, 1983.

[29] J. Misra and K. M. Chandy, "Proofs of networks of processes," *IEEE Transactions on Software Engineering*, vol. 7, no. 7, pp. 417–426, 1981.

[30] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. The MIT Press, 1999.