UiO : Department of Informatics
University of Oslo

Algebraic Component Composition in the UML

Martin Harbu Bielecki
Master's Thesis Spring 2014

Algebraic Component Composition in the UML

Martin Harbu Bielecki

2nd May 2014

Abstract

A software component is a reusable, high-level software entity with provided and required interfaces. Software components can be composed together in order to build software systems. When modeling component-based software, the traditional way is to manually attach their provided and required interfaces. This way of modeling is supported by the Unified Modeling Language, which is an OMG standard for object-oriented modeling.

As an alternative to this traditional approach, *algebraic component composition* is an approach that emphasizes the definition of *atomic* components and their compositions in an algebraic manner. The compositions are done via algebraic expressions. The result of an algebraic expression is a new *composite component* which is automatically derived from the components used in the expression. Component-based modeling in the algebraic manner is not supported by the standard UML.

The result of this master's thesis is a prototype which creates UML models of components and algebraic component compositions. The prototype accepts a textual specification of the model which it type-checks and translates to the corresponding UML model. A UML profile is used to extend the UML metamodel to support algebraic component composition.

Contents

List of Figures

List of Tables

Acknowledgements

First of all, I would like to thank my supervisor Volker Stolz for excellent guidance, helpful discussions during the work on this thesis and for giving me a push in the right direction when I needed it. I would also like to thank my family for supporting me, and the guys at the eight floor for good company. Thanks to Simen and Audun for proofreading. At last but not least, a special thanks to June for being understanding and supportive in the process of writing this thesis.

Martin Harbu Bielecki
University of Oslo
May 2014

Chapter 1

Introduction

This master's thesis is done at the Precise Modeling and Analysis (PMA) group at the Institute of Informatics at the University of Oslo. The thesis aims at discovering how the Unified Modeling Language (UML) can be used to support *algebraic component composition* when modeling component-based software, and how the algebraic composition approach compares to the more traditional component composition approach offered in component models such as rCOS[1]. This involves developing a prototype tool for creating such models from a textual input specification, together with the ability to validate these models in order to ensure model correctness. This chapter provides a short background to the topic of software modeling and software components, before moving on to the motivation behind the thesis, the problem statement and the methodology. The last section describes the outline of the thesis.

1.1 Background

Creating models of a software system serves different purposes during software development and creates a better understanding of the system for a diverse set of stakeholders. Creating system models as a way to document the system specification is a widely used technique [Som06, chapter 8]. This technique utilizes a set of graphical models that can describe business processes, the problem domain, the behaviour and structure of the system that is going to be programmed. The behaviour models show how the system should behave, for example the data flow or event reaction. The structural models describe the static structure of the system and the relationship with the different entities, such as classes and components. These models unify the understanding of the domain for each project participant and is helpful in analysis, design and development. Models help capturing the system requirements, and it is cheaper to alter the model to accommodate for requirement changes early in the development process than later, when it might require more

[1] http://rcos.iist.unu.edu/

work to get the change implemented. A software system model is an abstraction of the problem at hand, and leaving out the details is the most important aspect of such a model [Som06]. In an abstraction, the parts and details that are not important are left out, so the result is a model where only the interesting parts are present.

A *software component* is, in its simplest form, a reusable, high level software entity which encapsulates its functionality and contains two sets of interfaces, the *required set* and *provided set*, which serves as "connectors" to the outside environment. A required interface of a component specifies what the component needs in order to function properly, and a provided interface exposes functionality which the component offers. These components can be joined together by attaching their interfaces, if they are compatible, to create software systems. The joining of components through their interfaces is known as *component composition*. Software components gave rise to a branch in software engineering called *component-based software engineering*, or *CBSE* for short [Som06].

A model of a component-based system is a structured model with a high level of abstraction and describes the different component entities and their relationships. The traditional way of viewing such models is by defining the components, their interfaces and the links between these interfaces and viewing them in graphical component diagrams. This can be helpful to understand how the sub-components of a system are put together.

1.2 Motivation

Creating graphical models in the UML is a widely used technique in software development and a de facto standard for object-oriented modeling [Som06]. A graphical UML diagram is a visualization of a *semantic* UML model, which is a tree hierarchy of the elements that a model consists of. The UML supports the modeling of software components in the traditional way by specifying the components, the interfaces and the links between these interfaces as model elements. The traditional component composition approach emphasizes the connection of component interfaces, without necessarily creating a new component in the process. This is the approach utilized by the rCOS component model.

However, as an alternative to this traditional way of modeling components, the rCOS component model proposes in a recent article by Dong et al. [Don+13] a way to write *textual* specifications of components which contains a set of atomic components (an atomic component is not created from other components through composition) and *algebraic composition expressions*, which denotes the compositions of these components. The composition expressions are built up from a set of predefined *component operators*, which are applied to the components. The result of applying an operator to one or two (depending on the operator type) components is a new component, which again

can be used to create other composite components. In other words, the algebraic approach is a way of saying “take one or two components and apply an operator to it for deriving a new component”. An algebraic component composition expression can be arbitrarily long, and contain sub-expressions. The new component is computed automatically. The new composite component will contain a set of features, e.g. interfaces and methods, which is based on the features of the operands and the component operator used. This approach offers a new way of modeling components which encourages the work flow of defining atomic components and using the operators to create other components to create the final model.

The problem is that there is no support in the UML to represent such algebraic component compositions. There is no way to know *which* components where used to create a composite components and what the resulting component is going to look like. A "plain" UML component does not contain this information by itself, so we need a way to add it to the UML model without the need to do any manual modeling.

A common approach to extend the UML is with the use of UML profiles. A profile is a UML package which contains stereotypes used to create custom model elements by extending existing ones. The profile is then applied to a UML model. When the model is created, a natural step in the model development process is for the developer, designer or analyst to be able to validate the UML model. This is necessary to ensure correct compositions and component specifications.

1.3 Problem statement

The motivation above can be used to formulate the two following main problem statements:

1. How can the Unified Modeling Language, extended with a UML profile, be used to support and represent models of textual algebraic component specifications?

2. What are the benefits of the algebraic component composition approach in component-based modeling?

1.4 Methods

1.4.1 Literature studies

Literature studies form the base of the research done in this master thesis. A lot of literature can be found which covers the fields of components, component models, component-based engineering, component modeling and the UML. One of the more well-known books about component software is written by Szyperski [Szy02], and Fowler [Fow03] provides a nice introduction to the UML. The literature about the rCOS method, tool and component model, especially the ones covering

component specification and composition operators, e.g. [Don+13] were highly relevant as well. For the full list of literature, please refer to the reference section.

1.4.2 Prototype implementation

A part of this thesis is an implementation of a prototype tool which creates UML models, applied with a profile, of the textual algebraic component specifications mentioned in the motivation. The prototype developed is an extension to the existing rCOS tool. The rCOS components act as a layer on top of the UML (see figure 1.1). The rCOS tool offers functionality for creating UML models from a textual component specification (the language is called rCOSP) and contains a UML profile for creating models of rCOS components. However, the tool does not yet support algebraic component composition and the creation of UML models from algebraic specifications. Section 3.3 provides an example of how the rCOS tool, with the use of rCOSP, creates specifications of components.

Being able to work on an extension to this tool was an advantage because it was possible to reuse functionality already implemented in the tool. The final prototype parses a component specification written in a domain specific language (called rCOSPN) that supports algebraic component specifications, type-checks it and translates this specification to a semantic UML model. The model can be validated at all times after its creation to ensure correctness if it is altered. The prototype runs on the Java Virtual Machine and was implemented using Test-driven development. The prototype can be found at http://www.mn.uio.no/ifi/english/research/groups/pma/completedmasters/2014/bielecki/.

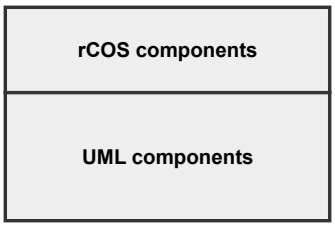


Figure 1.1: rCOS components are built on top of UML components

1.4.3 Tool evaluation

Several design decisions arose during development. We discuss the impact of them in chapter 7. We evaluated both the rCOSP and the rCOSPN languages based on the software model of a small airport. The component specification language and algebraic component composition approach was compared with the traditional way of component composition.

1.5 Thesis outline

The thesis is outlined as follows:

Chapter 1: Introduction gives an overview of the thesis topic, describes the motivation, problem statement and methodology.

Chapter 2: Components, component models and the UML gives a general background on component-based software engineering, defines components and component models and gives a brief introduction to the Unified Modeling Language (UML) and explains what we mean by the term "UML model".

Chapter 3: A look at different component models will see how components and component models are used in the "real world", and gives an introduction to rCOS, an academic component model and software engineering tool for component specification and composition. This chapter will also introduce the algebraic component composition as a new way to create models of component systems.

Chapter 4: The rCOSPN language introduces a new component specification language which focuses on easy component specification with support for algebraic component composition expressions.

Chapter 5: From code to model is a technical chapter which describes the implementation of the rCOSPN language and how the UML model is constructed from an rCOSPN specification. This includes a study of the relevant part of the rCOS UML profile.

Chapter 6: Type-checking and model validation will show how the rCOSPN specification, and the composition expressions in particular, is type-checked during parsing and how the UML model can be validated to ensure the correctness of the model after its creation.

Chapter 7: Discussion looks at prototype design decisions, tool limitations, alternatives to using UML profiles and compares the algebraic approach with the traditional approach of component composition.

Chapter 8: Conclusion and future work provides a summary of the work and describes possibilities for future work.

Chapter 2

Components, component models and the UML

2.1 Introduction

The development process of software projects can be hard and complex to understand. To get a better understanding of the problem domain and structure of the software application being developed, the Unified Modeling Language[1] (UML) offers a wide variety of diagrams to help development teams in the development process. Diagrams such as class-, sequence- and use case diagrams are widely used techniques in describing and visualizing the static structure of software applications, object interaction and user interactions, respectively. The use of these diagrams are widely standardized in modeling object-oriented software.

The UML also offers functionality to describe and visualize software on a higher level of abstraction than using class diagrams, namely with the use of component diagrams. *A software component* groups together related functionality to form a bigger entity, and communicates with other components through required and provided interfaces. How the components are constructed and plugged together is defined in a *component model*. The UML component diagrams help visualize the components, how they are wired together (their relations), and the interfaces they provide and require. The component diagrams make up a high level visualization of an entire system or its subsystems. However, software components is not something that only exists within the UML world. During the late 1990s, a branch in software engineering called component-based software engineering (CBSE for short), appeared [Som06]. As the name implies, CBSE makes heavy use of components, and its foundation is the composition of reusable and standardized components.

This background chapter will begin with a brief overview of what component-based software engineering is, before explaining the concepts of software components and component models. We will also

[1] http://www.uml.org/

give a brief overview of the Unified Modeling Language and UML profiles.

2.2 Component-based software engineering

The foundation of component-based software engineering is reusable, standardized software components. The components that make up a system is loosely coupled independent units which sit on top of an underlying platform (see figure 2.1), and the essence of CBSE is how to define, implement and integrate these components into systems [Som06]. The use of existing components, which can be third party, can make the development of big software projects go faster, since much of the functionality required may already exist in these components. The components can then be mixed and matched together using some "glue code" to produce, at least some part of, the system. Sommerville [Som06] defines four key characteristics of CBSE.

1. The implementation of the components are hidden from their interfaces. This means that one component could be swapped with another without changing the behaviour of the application if they provide the same functionality specified by their interfaces.

2. To make component integration easier, component standards define how components should communicate and how their interfaces should be described. These standards are a part of a *software component model*.

3. Middleware is needed to glue the components together and support component integration. The CORBA middleware is an example of such middleware, and we will look at the CORBA Component Model in section 3.2.2.

4. A special development process that is focused toward CBSE is required. A rough overview over such a process is to first search for suitable components, before doing a selection of the available components and validate that they will work together.

Component vendors that base their business on development and sale of components is the long-term vision of CBSE. However, since the components can be black-box units where the code may not be available, there is a problem when it comes to trust. The components can have undocumented failures and non-functional properties which makes the component not behave as expected. Because of this, most components are developed and reused within a company [Som06].

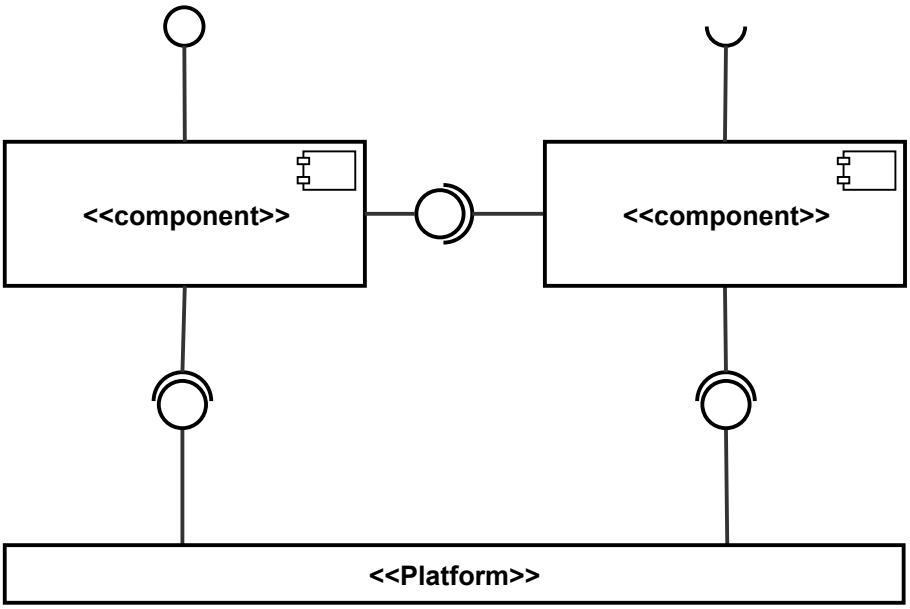


Figure 2.1: The traditional view of a component based-system. Figure derived from [Crn+11].

2.3 What are components and component models?

The word *component* is a broad and widely used term, and is not only used within the software and computer science areas. A general definition of a component is an entity that is *a part or element of a larger whole, especially a part of a machine or vehicle* [2]. This section will give definitions and further explanation of the terms "software component" and "component model", and explains the concept of component composition. The basic principles of these three elements are rather straight forward.

2.3.1 Software components

A computer has different hardware components such as CPU, memory modules, hard drives and external devices that together make up a complete system. The hardware components communicate through interfaces, and upgrading a component, for example the memory module, is easy if the new memory module complies to the interface offered by the motherboard. Similarly, a software application can be composed of many software components such as a database component, a user interface component and a component that handles the business logic. These components can again be composed of even smaller components. There is no set size of components, and they can be as

[2] http://www.oxforddictionaries.com/definition/english/component?q=component

small or as large as the component developer wants. It can be easy to believe that components and object orientation (OO) goes hand in hand, but it is not an absolute requirement that components contain classes and objects in the normal setting of the object oriented paradigm. They can, according to [Szy02] also be implemented using traditional procedures or consist of global static variables and methods. Purely functional entities using a functional programming language can also be an approach to create components. The implementation behind a component does not matter much, as long as its specification in terms of provided and required interfaces is clear. Each component is independent of other components, and can be composed together with other components to form an application. The composition will be explained and explored later, but it is important to note that composition is a major part of components. Without the ability to compose components together, the components would not be of much use beside what they implement on their own. The definition found in Szyperski's book [Szy02] defines components as follows:

Definition 2.1 (Software Component). A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.

This definition is divided in two parts; a technical part and a market-related part [Szy02]. The technical part covers independence, composition and the interfaces. The "explicit context dependencies" refers to the required interfaces, the composition of components defined in the component model and what the component needs in order to be deployed properly. The part about marketing is not of any particular interest for us in this thesis, but it covers the bit about deployment and third parties. There is another definition, found in [Crn+11], which is more specific and also mentions component models.

Definition 2.2 (Software Component). A software component is a software element that conforms to a component model and can be independently deployed without modification according to a composition standard.

As pointed out earlier, a software component specifies interfaces (or sometimes called *contracts*), either provided, required, or both, that it communicates through. The component encapsulates its implementation from the outside environment. The required interfaces specify the services a component need in order to function properly (the explicit context dependencies), while the provided interfaces defines the services it offers to other components (clients).

A component can contain any number of interfaces. Each method (or operation) in the interfaces can be supplied with pre- and postconditions (conditions that must be true before and after the method body has been executed, respectively) and invariants if the component model

supports it. In the UML, the *Object Constraint Language*[3] (OCL) is a declarative language used for specifying these conditions. For example, a simple postcondition to express that an object reference is not null before returning it can be written as **post: theObject != null**. When a client invokes a method and the input parameters and current state of the component satisfy the precondition, the component providing the method will make sure that the postcondition is met. If the precondition is not met, the behaviour of the invocation is undefined. Invariants are additions to pre- and postconditions which are conditions that must hold true at all times during execution. Extra-functional requirements such as resource and time usage can also be a part of the interfaces. For example, a component that provides functionality for drawing diagrams can be useful for a spreadsheet application, but useless for an animation application if the time spent drawing is slow. Components can be swapped with other components if the new component offers at least the same functionality and requires no more than the old one, i.e. they are interchangeable. A contract on how the component can be deployed, instantiated and how the instances behave through the interfaces is also provided as a part of the component specification (or rather, the component model).

2.3.2 Component models

The purpose of a component model is to define a standard for the implementation of components [Som06]. Some component models also specify the documentation and deployment instructions for the components. The definition found in [Crn+11] defines a component model to be the following:

Definition 2.3 (Software Component). A Component model defines standards for (i) properties that individual components must satisfy, and (ii) methods for composing components.

The standards make sure that the components that are developed within it can operate and be composed together. The most well-known models are the CORBA component model from the Object Management Group (OMG), the Enterprise Java Beans model from Sun (now Oracle), and the COM+ model from Microsoft [Som06]. However, many others exist, and different models target different domains [Crn+11]. We will take a look at a couple of different component models in the next chapter.

As described in the previous section, the interface specification defines a component. The component model describes the definition of the interfaces, that is, the name, parameters and exceptions of the operations. The usage information contains naming conventions, meta-data and a description of how the component can be customized to the application environment. The deployment information is an important

[3] http://www.omg.org/spec/OCL/

part of the model and specifies how the component should be packaged for deployment as an independent, executable unit [Szy02].

2.3.3 Composition of components

The process of composing components together is, naturally, an important aspect of component models. Sommerville [Som06] lists three ways of component composition:

- **Sequential composition:** the components are executed in sequence.
- **Hierarchical composition:** one component is directly using the (at least some) of the services provided by another component.
- **Additive composition:** at least two components are added together to form a new component. The interfaces of the new component are the composition of the constituent components, with possible duplicated operations removed.

Some glue code might be necessary to complete the compositions, but incompatibilities between the components can occur. For example, parameter incompatibilities between a method call and a declared method happens when the parameter types or number of parameters are different. Operation incompatibilities occurs when the names of the required and provided interfaces are different. Operation incompleteness is the case where the provided interface(s) of a component is a proper subset of the required interfaces of another component. To solve these incompatibilities, an adaptor component is created to glue the other two components together. The process of creating connections between components is also known as the *binding of components* [Don+13].

2.4 The Unified Modeling Language (UML)

The Unified modeling Language, or the UML for short, is a *de facto* standard for object-oriented modeling [Som06], and is managed by the Object Management Group[4] (OMG). The UML offers various diagramming abilities to aid software developers and designers in the software development process. Fowler [Fow03] lists three different modes that people use UML for:

- **Using UML as a sketch** to quickly communicate some aspect of the system. This is the activity most people use UML for.
- **Using UML as a blueprint** in cases where we want a complete and detailed design of the system.
- **Using UML as a programming language** for drawing diagrams that can be compiled/translated to code directly.

[4] http://www.omg.org/

The graphical diagrams in the UML can be divided into two types, *structure diagrams* and *behaviour diagrams*. Component diagrams are placed under the structure diagram classification, together with, among others, class diagrams. Among the most well-known behaviour diagrams are use case diagrams and sequence diagrams.

The modeling elements are backed by a *metamodel*. The metamodel is a model which defines the concepts of the language [Fow03]. Unfortunately, the UML metamodel does not cover model facilities for every application domain. Sometimes there are elements in the domain for which there are no equivalent model elements in the UML metamodel. As a remedy for this limitation, the UML is easy to extend, and in fact is designed to be extended [CD00]. For these cases, a solution is to create a *UML profile* that is tailored towards the particular domain. A UML profile is a package that extends a reference metamodel [Gro11], for example the UML metamodel. The goal of a profile is to adapt the metamodel to the specific domain [Gro11]. A profile can contain *stereotypes*. Stereotypes allow the developer or designer to create new modeling elements by extending existing ones. Users of the UML can then apply the customized UML profile to their model in order to use these stereotypes.

While a UML profile extends a (UML) metamodel, there is another way to define custom UML model elements. This is to create a custom metamodel which suits the domain at hand. This is discussed in chapter 7 where we look at alternatives to using profiles.

2.4.1 A note on nested classifiers in the UML

The UML Superstructure specification [Gro11] defines a UML classifier as the following: ”A classifier is a classification of instances, it describes a set of instances that have features in common”. For example, classes, interfaces and components are considered classifiers in the UML. These classifiers can be nested, i.e. be defined within other classifiers. In this thesis, we say that the classifier defined within another classifier is local to the enclosing classifier. For example, component `A` can contain component `B`, where the `B` is local to `A`. The effect of this is to reduce the visibility of classifiers which are not “relevant” for other than the enclosing classifier.

2.5 The term “model” defined

The word “model” is ambiguous, and its meaning depends on the context the word is used in. Generally when talking about UML models, it is implicitly given that it is the graphical UML models that are talked about. In this thesis we define a UML model as a collection of *model elements*, which again contains a set of features. This definition is a derivative of the definition found in [WK03]. This definition is also in line with the Object Management Groups description of an UML

model in the UML Superstructure specification [Gro11] which says that ”It contains a (hierarchical) set of elements that together describe the system being modeled”. We call this the *semantic* UML model. An example can be seen in figure 2.2.

The semantic UML model is not tied to any particular UML diagrams. However, the semantic UML model in the sense we just defined can be drawn as one or more graphical diagrams. We say that the diagram *shows* the model [WK03]. Each type of diagram shows the model from a different viewpoint. Even though an element is present in the model, it does not necessarily have to be drawn in the diagram. A component diagram of the semantic UML model in figure 2.2 can be seen in figure 2.3.

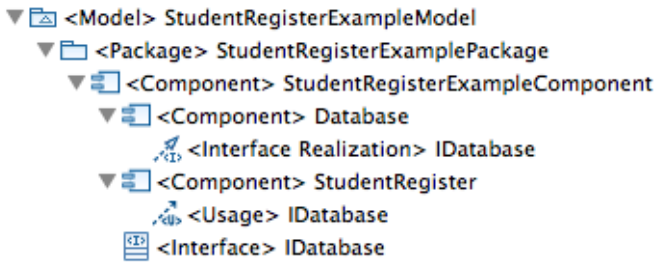


Figure 2.2: A semantic UML model

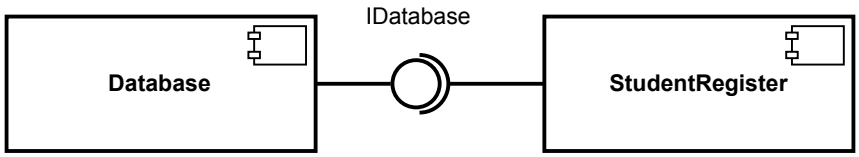


Figure 2.3: A graphical component diagram

2.6 Summary

Component-based software engineering (CBSE) is a software development approach where one uses and assembles components to create a system. A component is a software entity which exposes interfaces, both required and provided. The interfaces are used to communicate with other components, and the required interface of one component can be plugged into the provided interface of another component if they are compatible. A component model defines a standard for the implementation of components and component composition.

The Unified Modeling Language (UML) is a standard for object-oriented modeling and supports a wide array of graphical model diagrams. The UML can be extended with UML profiles to support new types of model elements. In this thesis, the semantic UML model, which we simply call the UML model, is a hierarchical representation of the model elements. We say that the graphical diagrams show the UML model from different viewpoints.

Chapter 3

A look at different component models

3.1 Introduction

The previous chapter covered the theoretical background of components, component models, component-based software engineering and the UML. In this chapter we will take a look at a couple of component models to see how the theory is put into practice.

The chapter is organized as follows: the first section (section 3.2) will present two component models used in "real-world" software. The first one is the OSGi framework (section 3.2.1), which is a light-weight component model on the Java platform. The second component model we will look at is the CORBA Component Model (CCM) in section 3.2.2. CCM is, together with Microsoft COM and Enterprise JavaBeans (EJB) one of the most well-known and mature component models. Section 3.3 will explore the rCOS[1] component model (Refinement of Component and Object Systems). We will explore how we can use *rCOSP*, a component specification language bundled with rCOS, to create a textual specification of a small airport system which consists of different components and their compositions. This example will demonstrate how the current way of component composition in rCOSP works. Section 3.5 introduces the idea of *algebraic component operators* as a tool for component composition. Using these composition operators we can treat composition as an expression, where the new component is computed as a result of the components used as operands in the expression.

3.2 "Real-world" components

This section will present an overview of two component models used in software, the OSGi framework and the CORBA Component Model, to see how components can be utilized in practice.

[1] http://rcos.iist.unu.edu/

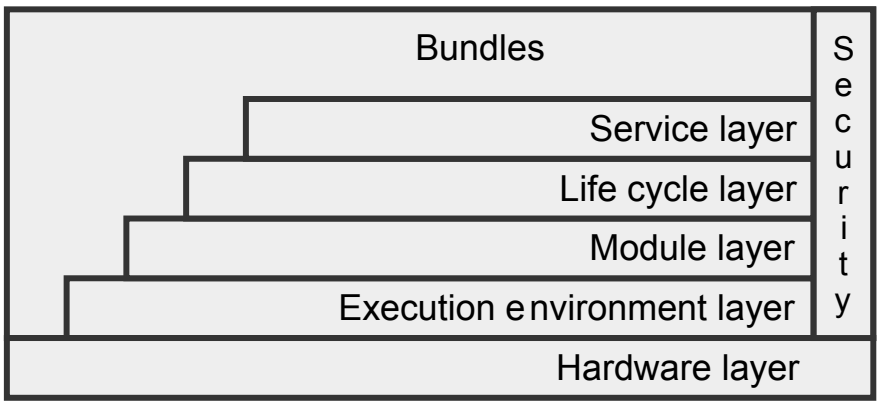


Figure 3.1: Layers of the OSGi framework

3.2.1 The OSGi framework

OSGi[2] is a framework for creating modular systems. It is the only mature solution for Java today [BE13, p. 9]. The OSGi specification is maintained by the OSGi alliance, and several implementations of the specification exist. Two of the most popular implementations are the Apache Felix[3] and Eclipse Equinox[4], the latter which the Eclipse IDE is built on top of [BE13].

The OSGi functionality is layered, and the layer outline can be seen in figure 3.1. The OSGi term for component is *module*, and in OSGi they come in the form as *bundles* [The12] and are defined in the *module layer*. A bundle is a collection of Java classes which are packaged in a JAR-file. JAR stands for **J**ava **Ar**chive, and is a way to create a package of Java class files and other resources to make it available for distribution. In OSGi, additional resources such as a manifest file is a part of the JAR-file, and it describes what the JAR-file contains. The bundles specify *exported* and *imported* packages and this information is specified as headers in the manifest file. The exported packages contain functionality that is provided by the bundle, and the packages that are imported are the dependencies it requires. It is also possible to require entire bundles. The OSGi 5 specification [The12] states that if a bundle requires another bundle, the framework must include every exported package from this bundle, and attach it to the requiring bundle. This mechanism wires bundles directly together.

The *life cycle layer* handles the life cycle of the framework itself and the bundles installed in the framework [BE13]. The specification says that each bundle can be in a number of different states:

[2] http://www.osgi.org/Main/HomePage
[3] https://felix.apache.org/
[4] http://www.eclipse.org/equinox/

Installed: the bundle is installed in the framework. However, it can not be started because of missing dependencies [BE13].
Resolved: every dependency of the bundle is available, and the bundle is ready to start or has been stopped.
Starting: the bundle is starting.
Active: the bundle is running.
Stopping: the bundle has been stopped.
Uninstalled: the bundle is removed.

The bundle mechanism makes it possible for bundles to hide their actual implementation from each other and expose their functionality through interfaces. These interfaces can be implemented by Java objects, which are known as OSGi services, and be published in a *service registry*. These OSGi services live on the *service layer*, and the Java object is owned by a bundle, which it also runs in [The12]. Bundles can register and search for services in the registry. The framework handles the connection between the bundles that offer the service and the bundle that requires it, so if a bundle goes to the **stopping** state, the services associated with it are removed from the service registry. The composition of bundles (or components) is therefore *dynamic*, since the coupling of bundles is done at run-time.

3.2.2 CORBA

The Common Object Request Broker Architecture, commonly known as CORBA, is a standard developed and maintained by the Object Management Group (OMG). CORBA aims to create interoperability between different programming languages, platforms and implementations [Szy02], also in distributed systems. CORBA based programs are able to communicate with each other. This interoperability is solved by hiding the implementation of objects from the outside environment via common interfaces definitions. In the CORBA Object Model, a client written in Java can invoke a method in an object written in another language, because the object interface is defined in the OMG *Interface Definition Language*, or IDL for short. The IDL is a common language that can be bound to a number of different programming languages, for example Java, C++ and C [Szy02]. A client stub and an object skeleton is compiled from the IDL. The stub and skeleton will redirect traffic to the real implementations, although, for the other end, they appear to be the real object. When doing a method call, the client specifies the method it wants to call on a specific object reference and marshals the method arguments using the IDL. The IDL at the receivers end unmarshals the arguments and invokes the method [Gro13]. The *Object Request Broker* (ORB) handles the call and keeps information about where to send the invocation request. Figure 3.2 is a high-level abstraction which shows how a method call travels through these layers. If the system is distributed and the client and receiver is remote, the ORB on the client side

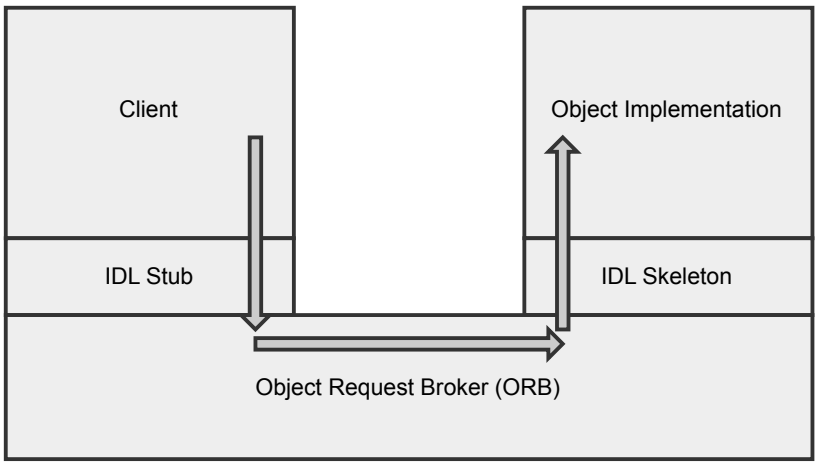


Figure 3.2: Method invocation in CORBA

routes the invocation to the ORB on the receiver side.

The *CORBA Component Model* (CCM) is an extension of the CORBA Object Model [WSO00]. A CCM application contains a number of CCM components, which can come from different vendors. The OMG CCM specification [Gro06] describes a CCM component type as a structure which encapsulates its internal representation and implementation and that contains set of features, which can be described by an IDL component definition. Concrete component instances are instantiated by a component type. These instances are managed by one or more *component homes*, which handles component life cycles and component access. A *basic* CCM component is a "component wrapper" for a CORBA object. An *extended* component has a greater set of features, such as ports. A port is a component feature that allows clients to interact with the component, and [Gro06] lists four different kinds:

- Facets, which are provided interfaces.
- Receptacles, which are required interfaces. The facets and receptacles of components are connected to each other [Szy02].
- Event sources, which sends out events on an event channel.
- Event sinks, which consumes events of a specific type. The event source and sink provides a publish/subscribe system for events.

The connections between a facet and receptacle can be set in component configuration or be made and destroyed dynamically during runtime [Szy02].

3.3 The rCOS Modeler

According to [Don+13], the complexity of software development can be divided into four fundamental attributes:

1. the complexity of the domain application,
2. the difficulty of managing the development process,
3. the flexibility possible through software,
4. the problem of characterizing the behaviour of software systems.

The first attribute says that it is hard to interpret and grasp the domain for the application, for example when it comes to software design. The second attribute states that it can be hard to manage the development process, for example the handling of large teams and a changing requirement specification. The third attribute refers to the wide range of, among others, software architectures, tools, algorithms, protocols that is offered, and that there are many possibilities to consider when developing an application. The fourth attribute is about the complexity of capturing the semantic behaviour of a system, which can be used for analysis, validation and verification for correctness [Don+13].

The concept *separation of concerns* can help handle this software complexity [Che+07]. As the name implies, this concept emphasizes divide and conquer of the software application, where different *views* are made for the different development stages. The views can for example be static structural views, interaction views and dynamic views. When integrated together, these views can model the complete system [Che+07]. An approach to separation of concerns is the OMG's *Model-Driven Architecture (MDA)*[5]. In MDA, different system models are built to handle the complexity. MDA focuses on creating platform independent models (PIMs) which are then translated to platform specific models (PSMs) via some kind of model mapping. However, this approach does not address the semantic behaviour of the application. In order to validate and verify correctness of the models, the developer has to do the work manually [Don+13].

Refinement of Component and Object Systems[6] (rCOS) is an attempt to seal the gap between the models in MDA and semantic validation and verification. rCOS is an approach to component-based MDA, where the key concepts are component architectures with hierarchical component composition, refinement between models and model transformations [Che+07]. The models built using rCOS can be of different levels of abstractions, and be reasoned about through refinement and transformations. The rCOS tool also supports code generation from these design models. The weak parts about rCOS is that is does not consider deployment, does not support dynamic (run-time) composition and does not consider extra-functional requirements.

[5] http://www.omg.org/mda/
[6] http://rcos.iist.unu.edu/

The rCOS component model

The rCOS component model is hierarchical and specifies two types of components, *service components* and *process components*. The former is often referred to as just *component* and the latter as *process* [Che+07]. The service component is passive and will wait for client calls, while the process component is active and can call other components without waiting for clients. Components can be *closed*, which means that they do not depend on functionality (no required interfaces) or *open*, which means that they have required interfaces.

The rCOSP specification language

rCOSP is a component definition language provided by rCOS. It allows the developer to textually describe and specify an architectural view of the components and their compositions. It provides some of the same language constructs as an object-oriented programming language, such as interfaces and classes, and its syntax is similar to the Java programming language. In addition to interfaces and classes, rCOSP introduces two new constructs, namely *contracts* and *components*. Below is a short description of the most important parts of rCOSP and how the component composition works. The next section shows how these parts can be used to model a small airport which consists of different components composed together and highlights the different features of rCOSP.

Interface: An rCOSP interface contains method declarations and field declarations. Each interface defines a set of operations, and can extend other interfaces. Interfaces cannot share names within the same scope, and cannot be partially implemented. By partial implementation we mean that the classes that implement the interface must give definitions to every method.

Class: A class in rCOSP is similar to a class in Java. It is a blueprint for creating data objects, and defines custom data types used in the components. A class can implement functionality to realize interfaces and contracts.

Contracts: A contract is associated with an interface and contains additional information which is needed when using the component, without needing to know how the component internally works. Each method declared in the interface can be extended in the contract with implemented functionality and protocols, but this is optional. The contract can be viewed as a layer between a component and an interface. Each defined interface can be associated with more than one contract. This allows interface reuse, because the developer can create a contract that is tailored towards a specific component without defining a new interface.

Protocols: A protocol puts restrictions on the method invocation order, e.g. that the method “loadPassengers” has to be invoked before the “unloadPassengers” method. The protocol is defined in the contract, outside the method bodies.

Pre- and postconditions: The pre- and postconditions are "abstract" specifications, in contrast to a "concrete" implementation we know from traditional programming with e.g. declarations, variables and assignments. The conditions are written as [pre : expression1, post : expression2], or using syntactic sugaring as simply [expression1 ⊢ expression2]. These conditions are written inside the method bodies, e.g. in the contracts and classes.

Components: Components are the most interesting building block. A component contains an optional component composition expression, declarations of provided and required contracts and can define internal classes, interfaces and contracts. Components can also contain internal/nested component definitions, which means that components can be nested. This opens up for local component compositions.

Provided contracts: The provided contract(s) of a component can either be by a class implementation or a delegation. A delegation happens when the contract is provided by a sub-component.

Required contracts: The required contract(s) of a component simply states what functionality the component needs to have.

Component composition: Component composition expressions tell us how the required and provided contracts of two components are plugged together. The plugging of interfaces is then given implicitly by the expressions. As an example, assume that we have a component `C1` with provided contract `A`, and a component `C2` which requires `A`. In the specification of component `C2` we can express the composition of the contract `A` between `C` and `C2` as composition : C1 ! C2 [(C2.A <– C1.A)].

3.3.1 Airport example in rCOSP

We will now explore rCOSP using an example model of a small airport. The example will demonstrate how the airport components can be specified using the language, and how the components are composed to form a model of the system. Figure 3.3 shows a plain UML component diagram of the airport system using the standard UML *ball-and-socket* notation for component wiring. This diagram shows composition in the traditional perspective. In this notation, the ball represents a provided interface, and a socket represents a required interface. Later in this example, we will see the rCOS component diagram of this model.

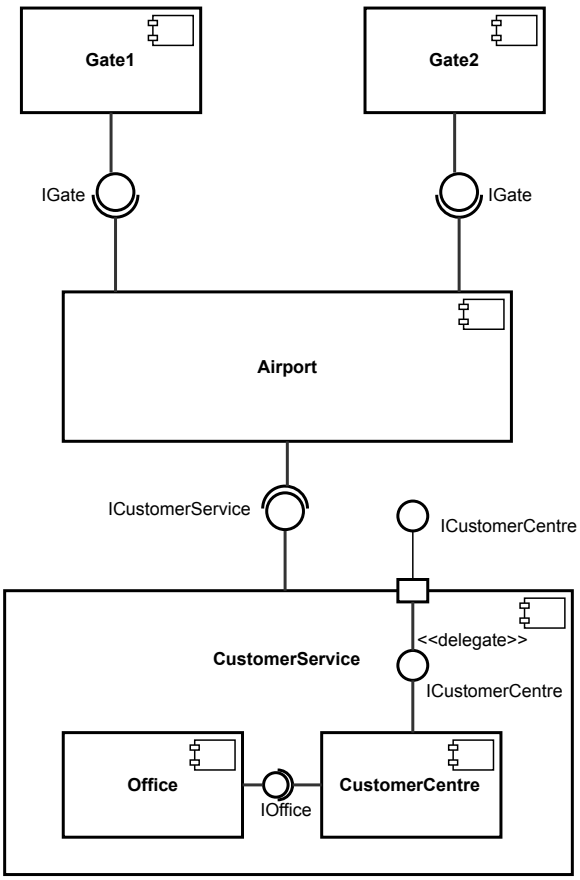


Figure 3.3: The UML component diagram of the airport

The gate component

An airport gate is a moveable bridge that connects the airport terminal and the aircraft. Our airport will contain two such gates, but there is no limit to how many gates can be included. Two is chosen to keep the model relatively small.

The gates are, naturally, modelled as components. Each component provides a contract of type `GateCtr` which is associated with the interface `IGate`. This interface declares two methods, `loadPassengers` and `unloadPassengers`, and the contract lay down a restriction on the `loadPassengers` method by specifying a pre- and postcondition. The precondition says that the number of passengers has to be greater than zero if this method invocation is to have a valid response. In this case the postcondition is defined as simply `true`, which means every state holds. The contract also specifies a protocol, which says that the `loadPassengers` method has to be called before the `unloadPassengers` method. The interface and contract can be viewed in listing 3.1.

The gate component itself is pretty simple. It is a passive service component and does not contain any component compositions. Its only

Listing 3.1: The gate interface and contract

```
1 interface IGate {
2     public loadPassengers(int numPassengers);
3     public unloadPassengers();
4 }
5 contract GateCtr of IGate {
6     public loadPassengers(int numPassengers) {
7         [numPassengers > 0 |- true ]
8     }
9     protocol{ loadPassengers ; unloadPassengers }
10 }
```

task is to provide the `GateCtr` contract by a class called `GateClass`. The `GateClass` is a class which implements the two methods in the `IGate` interface. The class is not shown in the listing because of simplicity (the full example is in appendix A.1). Listing 3.2 shows the definition of two gate components, `Gate1` and `Gate2`.

Listing 3.2: Two gate components

```
1 component Gate1 {
2     provided GateCtr by GateClass;
3 }
4 component Gate2 {
5     provided GateCtr by GateClass;
6 }
```

The customer service component

The customer service component is slightly more complicated than the gate component, because it contains a composition of two local components. Recall that a component is local when they are defined inside another component. These local components are an office component and a customer centre component. The code is shown in listing 3.3. This listing only shows the components. Everything else related to these components, such as interfaces, contracts and classes are left out, but can be seen in the full specification in the appendix A.1. As we can see, the customer service component provides the interface `ICustomerService` by the `CustomerServiceCtr` contract, and *delegates* the provided contract `CustomerCentreCtr` by the local customer centre component.

The most interesting bit is the composition of the two local components. The composition expression says that the `Office` and `CustomerCentre` components are going to be composed. More specifically, the provided contract `OfficeCtr` in `Office` will be attached to the required `OfficeCtr` contract of `CustomerCentre`. This will allow the customer centre to use functionality provided by the office.

Listing 3.3: The customer service component

```
1 component CustomerService {
2     composition : Office | CustomerCentre
3             [(CustomerCentre.OfficeCtr <- Office.OfficeCtr)]
4
5     provided CustomerServiceCtr by CustomerServiceClass;
6     provided CustomerCentreCtr by
7         CustomerCentre.CustomerCentreCtr; /*Delegation*/
8     component Office {
9         provided OfficeCtr by OfficeClass;
10    }
11    component CustomerCentre {
12        provided CustomerCentreCtr by CustomerCentreClass;
13        required OfficeCtr;
14    }
15 }
```

The airport component

The airport component is a central piece in this example, and ties the previous components together. The airport requires two gates and a customer service. Listing 3.4 shows the code. The specification of the component is rather short, because it only requires functionality from other components and the composition of the components which provides this functionality.

An important thing to note here is that the airport will have more than one gate. This implies that we have to require multiple contracts of type `GateCtr`. We cannot however, require the same contract more than once. If we do this, we have no clue which of the two similar methods we call, and which of the gate components will be invoked, since both of them are denoted by the same contract name. For instance the invocation GateCtr.loadPassengers(42) is ambiguous. One way of solving this is to define multiple equal contracts, but with different names. In other words, by simple renaming of the contracts. The code is shown line 1 and 2 in listing 3.4. These lines define two new contracts, namely `Gate1Ctr` and `Gate2Ctr`. They extend `GateCtr`, but add nothing new, so in this way we can view them as equal to the "original" gate contract, although with different names. After renaming, we specify that the airport component requires two gate contracts, namely the one we just defined, together with one customer service. The composition expression shows how these are attached together with the provided contracts of the other components which we previously defined. We plug the provided `GateCtr` contracts of `Gate1` and `Gate2` into `Gate1Ctr` and `Gate2Ctr`, respectively. The provided `CustomerServiceCtr` by the customer service component is inserted into the airport's required contract with the same name.

Listing 3.4: The airport component

```
1 contract Gate1Ctr of IGate extends GateCtr {} /* Renaming */
2 contract Gate2Ctr of IGate extends GateCtr {} /* Renaming */
3 component Airport {
4     composition :  Gate1 | Gate2 | CustomerService
5                    [(CustomerServiceCtr <- CustomerService.
                         CustomerServiceCtr)
6                    (Gate1Ctr <- Gate1.GateCtr)
7                    (Gate2Ctr <- Gate2.GateCtr)]
8     required Gate1Ctr;
9     required Gate2Ctr;
10    required CustomerServiceCtr;
11 }
```

The final airport model

We can run the complete textual specification of the airport model through the rCOS tool. The result is an (semantic) UML model and an rCOS diagram. The rCOS diagramming capabilities extend the UML diagramming, and is tailored for rCOS models. The figures 3.4 and 3.5 show two rCOS component diagrams displayed in Eclipse after some manual adjustments have been applied (drag-and-drop diagramming). The first picture shows the gates and the airport, and the second shows the airport and the customer service. If we compare these diagrams to the diagram in figure 3.3, we find that they are similar in structure (hierarchical composition). However, the rCOS diagrams contain contracts instead of interfaces. The compositions are shown as dependencies (dotted arrows).

3.4 Issues with the current state of rCOSP

As we now have seen, the rCOSP component specification language allows us to textually specify components and their compositions. From here, the rCOS tool can take this specification as input and create the UML model. We will now highlight and discuss some aspects of the language that can be seen as limitations and propose improvements for them.

Multiple components of the same type

One of the problems that arises is when we want multiple components of the same type, as we did with the two gate components in the airport example. In the example, we created two gate components identical in structure, although with different names, to accommodate for the requirement of two gates demanded by the airport. This is a bit redundant, and ideally we would want to specify each type of component once, e.g. by one single specification of component of type `Gate`. This

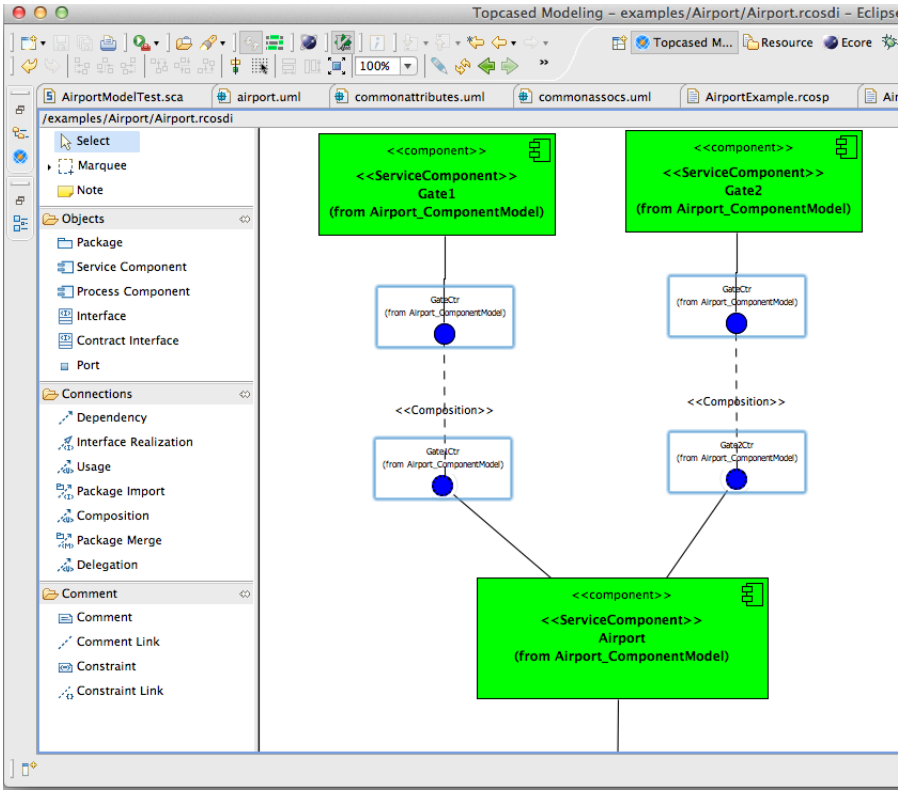


Figure 3.4: The rCOS component diagram of the Gate components

single component would be used in every composition that requires it.

Name clashes

Another problem we saw when modeling the airport was that we cannot require the same contract multiple times in a component, because this gives rise to an ambiguous specification when it comes to method invocations. In our specification, we solved this by defining multiple, equal contracts with different names (renaming). However, these are essentially duplicated specifications which we should avoid. A better solution would be an approach where we define the contract once. We will stick to the idea of renaming, but instead of doing the renaming by creating multiple contracts, we can introduce the idea of *enumerating* the required contracts. The enumeration can be expressed like required GateCtr[N], where the N in the square brackets denotes how many required contracts we want. A simple renaming where we append the number (2 to N) to each required contract would suffice in most cases. In the airport example, the result is two required contracts called `GateCtr` and `GateCtr2`, respectively, substituting N for 2. Upon encountering an expression of this type, the program would automatically create two (or N) required contracts of this type in the component. This would remove the need to create and rename contracts manually. This change would probably not be too hard to implement in rCOSP.

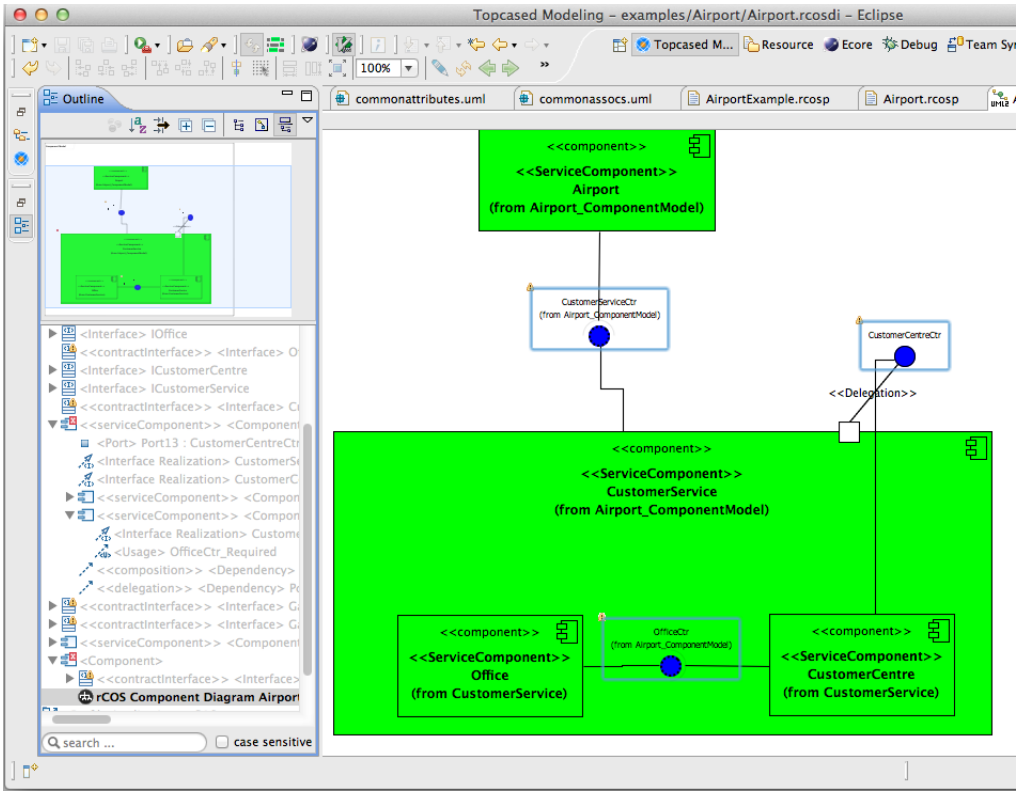


Figure 3.5: The rCOS component diagram of the CustomerService component

Composition

The last thing to point out is that the component composition can be improved. In the current state in rCOSP we need to explicitly specify the composition of components, by specifying what provided and required contracts are going to be plugged together. When modeling in the long run, this seems like a cumbersome and error-prone task. We need a simple way of modeling component compositions where the risk of errors is minimized.

3.5 Algebraic composition operators

We can take a new approach to the composition of components by introducing the idea of *algebraic component composition*. The algebraic composition is hierarchical in the sense that one component is directly using functionality provided by another. Composition is done via algebraic expressions, and the result of an expression is a new component. The algebraic composition expressions will have an infix notation. As an example, assume that we have two components, `C` and `B`, where `C` provides some functionality that `B` requires. To compose `C` and `B` using an algebraic operator, we can specify this as $K = C \parallel B$, where the $\parallel$ symbol means "parallel composition". The component `K` denotes

the new component created as a result of this expression.

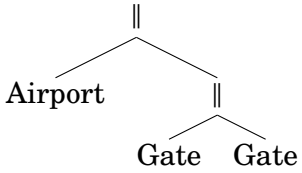
3.5.1 The term "algebraic" explained

In mathematics, something is *algebraic* when it is related to or involves *algebra* [7]. The oxford dictionary [8] further states that an algebra deals with letters and other symbols, which can represent numbers and quantities in formulae and equations. In mathematics, these letters and symbols can be applied to algebraic operations, for example addition and subtraction. This will result in *algebraic expressions*.

We can use similar definitions for these terms when switching from mathematics to component composition. The algebraic symbols represents components and the composition operators are the algebraic operations. When applying these two together we get the *algebraic component composition expressions*. The algebraic commutative and associative laws are valid for the *parallel* and *disjoint* composition operators, and this ensures that different expressions can yield components that are structurally equal although the expressions are manipulated differently.

3.5.2 Algebraic composition operator example

Take the following airport as an example to demonstrate. Assume we have a gate component which provides the contract `GateCtr` and an airport component which requires two gates, as before. Imagine that the two required contracts of the airport are now specified using enumeration and are called `GateCtr` and `GateCtr2`. The following algebraic composition expression creates a new airport: AirportSystem = Airport ‖ Gate ‖ Gate. The operator used here is called the parallel operator, and will be defined in the next chapter. However, note that the operator is binary. The expression can be represented as a tree:



The right subtree of the root node will compose two gates. The resulting "anonymous" component of this composition will then be composed with the `Airport` component which is the left subtree of the root. The result of this composition will be assigned to the `AirportSystem` variable.

[7] http://www.oxforddictionaries.com/definition/english/algebraic?q=algebraic
[8] http://www.oxforddictionaries.com/definition/english/algebra?q=algebra

3.6 Related work

As stated in the introduction (in section 1.4), the work done in this thesis is heavily influenced by, and based on, the rCOS component model and tool. The main contribution to this thesis is the work laid out in [Don+13]. This article defines a format and syntax for atomic component specification and component composition using component composition operators. We have adopted a simplified format of this atomic component specification language in our prototype tool, since it provides a readable syntax with focus on clear specification of components with provided and required interfaces. The language constructs that were left out were internal interfaces and actions. Please refer to chapter 4 for an introduction to this language.

[Don+13] defines five operators in total, three binary operators (*parallel*, *disjoint*, and *plugging*) and two unary operators (*renaming* and *restriction*). The prototype tool implements these operators, although in simplified versions in order to reflect the constructs that were left out from the language.

3.7 Summary

This chapter gave an overview of three component models. The OSGi framework and CORBA Component Model which are used in "real-world" software and rCOS which is an "academic" component model. We saw how components in the OSGi framework, called modules, export and import Java classes in order to create systems. The CORBA Component Model is an extension to the CORBA Object Model, which aims at creating interoperability between software, no matter what platform or programming language is used. The rCOS modeler is an approach to component-based MDA. The component model in rCOS is hierarchical with two types of components, services and processes. Components are composed by directly attaching required and provided contracts to each other, i.e. hierarchical composition. The rCOS tool comes with a component specification language, rCOSP, which opens up for text specifications of components. These specifications can be translated to rCOS UML models. However, there were some shortcomings with the language. The last section introduced algebraic component composition as an alternative to the traditional approach, and showed how we can use the algebraic approach to overcome the limitations by the rCOSP approach and at the same time view components and component composition from a different perspective.

Chapter 4

The rCOSPN language

4.1 Introduction

In this chapter we will introduce a new language for component specification and composition, called rCOSPN (for "rCOSP New"), and take a look at what it has to offer. The rCOSP specification language is used as a base for the new language, and much of its syntax and features are still used in rCOSPN, especially method bodies. However, there are also several changes and new features, and we will describe each of these in turn in the next section. We will also introduce the composition operators and their formal definitions. In the last section we go back to the airport example, and show how we can model it using the new language.

4.2 The goals of the language

Three main goals were in mind when creating the rCOSPN language. As stated in the previous chapter, rCOSPN is inspired by the component specification format found in [Don+13]. The first goal of rCOSPN is to provide a specification language which emphasizes readability and usability. By readability we mean that a component model specification should be easily understood just by glancing at the code. This requires a clear format for the specification of interfaces and atomic components. It should be clear what the behaviour of the composition operators are, and writing the composition expressions should be straight forward in most cases. By usability we mean that it should not require too much time to get acquainted with the language, and the user should quickly be able to write component specifications.

The second goal is the ability to extend the specification language in the future, and possibly integrate it further with the rCOS tool. Although creating structural models is the focus in rCOSPN, we chose to include behavioural modeling as well to some extent. Method bodies, protocols and pre- and postconditions are behavioural parts of the language, and are transferred untouched from rCOSP to rCOSPN. Using these elements have no impact on the structural UML

model created by the tool, but they are included nonetheless because behavioural models, e.g. sequence diagrams are an essential part of the rCOS tool which allows model transformations between different types of models. In the future, it might be relevant to implement functionality for integration between models made by rCOSPN and the rest of the rCOS tool.

The third goal is, naturally, to create UML models which reflects the specifications. This goal can be viewed in light of the first goal. The readable format of the textual specification should be relatively transferable to a clear and equally readable UML model. This implies that each language construct should have basic counterparts in the UML which can be utilized in model construction.

4.3 What is new?

The main changes from rCOSP, in addition to the introduction of the composition operators (section 4.4), are the removal of contracts (section 4.3.1), a new syntax for constructing atomic components (section 4.3.3), and allowing method bodies in interfaces (section 4.3.2).

4.3.1 Removal of contracts

The concepts of *contracts* and *interfaces* in rCOS are explained in section 3.3, but let us take a quick recap. In rCOS, an interface is just a set of *method signatures*. A method signature in rCOSP is the method name and the input and output parameters of the method. It does not have an explicit return type like in Java, but the type of the last output parameter is the return type for the method. The input and output parameters are delimited by a semicolon. Think of an interface as a label for a collection of method signatures. Components provide or require these interfaces through contracts. We call this *interface realizations* and *interface usages*, respectively. A contract therefore plays the role as a layer between an interface and the component, and the user of the component is not required to have any knowledge about how the component implements the operations. The contract also augments the interface with one or more protocols and adds pre- and postconditions to operations. A contract can contain code for implementing an operation, but it is not required. Contracts are sometimes called *contract interfaces*, while the regular interfaces are called *syntactic interfaces* [Gro14].

In the new language, we went for a stripped down solution where contracts are no longer a part of the language. The reason was that we wanted a simple specification language, as stated in section 4.2. The functionality that contracts employ is, as stated above, operation implementations (method bodies), protocols and pre- and postconditions. All of these are optional in contracts. The effect is that it is possible to have an empty contract, as we saw in the airport example,

which really adds nothing to the model besides an empty model element. The method bodies, protocols and pre- and postconditions are not restricted to contracts only, and can be put elsewhere, for example in the component itself.

A positive thing about contracts is that they allow interface reuse, but the specification might require more maintenance. For example, one simple change in an interface might require that every accompanying contract is changed accordingly.

4.3.2 Interfaces

Since contracts now are out of the picture, there are two alternatives when it comes to handling the optional method bodies, protocols and pre- and postconditions that were in the contracts. The first solution is to keep everything in the component specifications, and not allow these constructs to be anywhere else. This solution seems a little rigid, because having the possibility to have e.g. optional method bodies is handy when it comes to code reuse. The second solution is to allow these constructs to be optional in the interfaces, and this is the approach realized in the prototype.

The reason why we went for this approach is that we were inspired by the concept of *traits* in the Scala programming language [1]. In Scala, a trait is a unit of code reuse [OSV11], similar to an interface in Java. One of the biggest differences between a Scala trait and a Java interface is that a trait can be partially implemented. Partial implementation in this setting means that some (or all) of the methods declared in the trait can be implemented in the trait. We adopt this approach in the rCOSPN language, and the effect is that the interface can provide base functionality which is common for different components. If a component specification designer wants to create a more specific functionality of a method, it can be overridden by the method implementation in the components. The biggest advantage of this is that it allows for code reuse since an interface is just a collection of method signatures (and now possibly method implementations). It does not matter for the specification if the method implementation lives in the interface or the component.

The rCOS interfaces can contain field declarations. When a component provides an interface that contains one or more field declarations, these declarations are "moved" to the set of fields the component contains. The reason behind this is that, as mentioned, an interface is only a collection, or label, of methods and fields. The fields will belong to the component, not the provided interface.

The required interfaces in rCOSPN can be enumerated. As we will see, the algebraic composition works on method-level, not on interface-level. An enumeration of a required interface also renames the methods in the interface. This will make sure that we get no duplicated required

[1] http://www.scala-lang.org/

methods.

4.3.3 New component syntax

In this section we will take a look at how we can construct atomic components in rCOSPN. Recall that an atomic component is not created from other components. The new component syntax is formally defined in the Extended Backus-Naur Form (EBNF) grammar in listing 4.1. The first production rule in this grammar states that a component is either a service component or a process component, as before. Recall that a service component is passive and can only carry out operations when it receives stimulation from the outside (someone calls one of its methods). A process component is active and can initiate actions to other components. The second and third rule describe a service component and a process component, respectively. The "CNAME" rule is a production rule for identifiers that begins with an uppercase letter followed by zero or more letters or numbers (0-9). The fourth and fifth production rule shows what a component body and a process component body can consist of.

Other than being passive and active, there is another major difference between service components and process components. Processes act as *coordinators* between service components. We discovered two different ways to handle the process components. The first alternative is to treat them as normal service components with required and provided interfaces. They can then be implemented with a main method as an entry point which starts the active execution. This is the way it is done in rCOSP. The other alternative is to create a distinction between processes and service components. As can be seen in the fifth rule in the grammar, process components can only require interfaces, not provide them. This approach is described in [Don+13]. The effect is that the processes can focus on coordinating service components rather than be concerned of having to provide implementation themselves.

One of the effects of making the language simple is that it does not support nested component definitions and local composition expressions inside components.

An example component specification displaying the new syntax is shown in code listing 4.2. Here, we define two interfaces, `IAdminCRUD` and `IDatabase`. We then define a service component called `UserAdmin` which provides the `IAdminCRUD` interface and requires the `IDatabase`. A provided interface is denoted by the keyword **provided** followed by the interface name. The methods of the interface are implemented in the provided interface body. Some of the implementations of methods are left out in this example, but note that the `updateUser` method in the `IAdminCRUD` interface contains an implementation. Because of this, it is not necessary to include this method in the provided block in the component.

Listing 4.1: Component grammar

```
component ::= servicecomponent | processcomponent ;

servicecomponent ::= "component" CNAME
                        "{" componentbody* "}" ;

processcomponent ::= "process" CNAME
                        "{" processcomponentbody* "}" ;

componentbody ::= processcomponentbody | providedinterface ;

processcomponentbody ::= requiredinterface
                       | classdef
                       | field ;
```

Listing 4.2: User admin component

```
interface IAdminCRUD {
    public createUser(string name; User usr);
    public updateUser(string name, User usr) {
        user.setName(name)
    }
    public deleteUser(int id);
    public retrieveUser(string name; User usr);
}

interface IDatabase {
    public save(User usr);
    public load(int id; User usr);
}

component UserAdmin {
    provided IAdminCRUD {
        public createUser(string name; User usr) {
            usr := User.new(name);
            save(usr)
        }
        public deleteUser(int id) {
            /*Implementation*/
        }
        public retrieveUser(string name; User usr) {
            /*Implementation*/
        }
    }
    required IDatabase;
}
public class User {
    /*Implementation*/
}
```

4.3.4 Multiple identical interfaces

When modeling the airport in the previous chapter we saw that a problem arose when we wanted more than one `Gate` component. The solution was to create duplicate `Gate` component specifications, and create identical contracts via renaming so the airport component was able to require more than one `Gate`. To avoid code duplication and multiple identical specifications, we considered three alternatives.

In the first alternative, the solution is to split the component in a number of smaller components, each requiring exactly one of the multiple interfaces. In the airport example, this means that we create two airport components instead of one, where each will require one `Gate`. We can then "stitch" these components back together using composition with `Gate` components. The benefit is that there is no new syntax to implement. All we do is to split components into smaller ones. The disadvantages are that we do not avoid code duplication and we need to create more composition expressions to create the model.

The second alternative will use renaming of the duplicated required methods. In the component specification we state the requirement of two (or more) interfaces of the same type. Without renaming, this would cause ambiguous method calls since method calls are no longer prefixed by their contract name. Recall that if we specify the requirement of two contracts of type `GateCtr`, the method call GateCtr.loadPassengers(42) is ambiguous. However, using renaming, we can manually change the name of the methods that are duplicated, for example by adding a “renaming block” inside the component which explicitly states the renaming of methods. If a component requires two `IGate` interfaces, we will have two `loadPassengers` methods. Using renaming, we can rename one of them to, for example, `loadPassengers2`. The advantage is that code duplication is reduced. However, there is a risk for errors since manual work is required to rename the duplicated methods.

The third alternative, which is the alternative we went for, is a variant of the *enumeration* of required interfaces proposed in section 3.4. To require two `IGate` interfaces we simply write `require IGate[2];`. This will automatically rename the methods in one of the required `IGate` interfaces by simply appending the integer 2 to the method names. The other `IGate` interface will stay unchanged. Advantages of this approach are that renaming is automated and is less error prone and duplicated code is reduced. Ambiguous method calls are no longer a problem. The disadvantage is that the language syntax is extended, but we regarded the pay-off as greater than the cost of implementing the functionality in the prototype.

4.4 Component composition operators

In this section we introduce the composition operators and their definitions. The operators *parallel* (section 4.4.1), *disjoint* (section 4.4.2) and

plugging (section 4.4.5) are binary infix operators, while *renaming* (section 4.4.3) and *restriction*/*hiding* (section 4.4.4) are unary operators. The result of each operator application is a new component. We will go through each operator in turn, and we begin with the parallel operator. Every operator is a simpler version than the ones proposed in [Don+13]. Each will be demonstrated through examples. In order to do that, we need to define a couple of components (listing 4.3) to use as examples. The notation used in the definitions is explained in (table 4.1).

Table 4.1: Overview of notation used in operator definitions

Notation	Explanation
C.pIF	The union of the methods provided by the component C
C.rIF	The union of the methods required by the component C
C.vars	The set of variables defined in the component C

Listing 4.3: Two simple components

```
1  interface A {
2      public foo();
3      public bar();
4  }
5
6  interface B {
7      public fooz();
8      public baz();
9  }
10
11 component C {
12     provided A { /*Impl*/ }
13     provided B { /*Impl*/ }
14 }
15
16 component D {
17     required A;
18 }
```

4.4.1 The parallel operator

The *parallel* operator, denoted by $\|$, is, as mentioned, a binary, infix operator. This is an overloaded operator, meaning that the result depends on the type of the operands. The operands can be two process components (1), two passive service components (2), or one of each kind (3). The resulting component is a little bit different for each of these cases. The composition which involves processes is a special case of a composition involving two service components, and we will give all definitions in this section.

Definition 4.1 (Parallel composition of processes). Let P_1 and P_2 be two process components. Their parallel composition, $P_1 \parallel P_2$, results in a new component P_3, where

$$\begin{aligned} P_3.rIF &= P_1.rIF \cup P_2.rIF \\ P_3.vars &= P_1.vars \cup P_2.vars \end{aligned}$$

It is required that they do not have any variables in common.

The next definition applies to the cases where we have one process and one service component. The process will *coordinate* the behaviour of the service component.

Definition 4.2 (Coordination of components). Let P and C be a process component and a service component, respectively. Their parallel composition, $P \parallel C$, results in a component G, where

$$\begin{aligned} G.pIF &= C.pIF \\ G.rIF &= P.rIF \cup C.rIF \\ G.vars &= P.vars \cup C.vars \end{aligned}$$

It is required that the two components do not have any variables in common.

Definition 4.3 (Parallel composition of service components). Let C_1 and C_2 be two passive service components, either open or closed. Recall that a closed component do not have any required interfaces, while an open component does. Their parallel composition, $C_1 \parallel C_2$, results in a new component C_3, where

$$\begin{aligned} C_3.pIF &= C_1.pIF \cup C_2.pIF \\ C_3.rIF &= (C_1.rIF \cup C_2.rIF) \setminus (C_1.pIF \cup C_2.pIF) \\ C_3.vars &= C_1.vars \cup C_2.vars \end{aligned}$$

It is required that the two components do not have any variables or provided methods in common.

In every case, the parallel operator is commutative and associative since the composition never hides any methods of provided interfaces, unlike plugging (section 4.4.5). Let's look at an example which demonstrates definition 4.3.

Example 4.4. We can use the parallel operator on component `C` and `D`, shown in figure 4.3 to create a new component: $CD = C \parallel D$. The new component `CD` will provide the methods in interface `A` and `B` (the union), and since `C` provides `A`, the methods in this interface are not exposed as a required interface from `CD`.

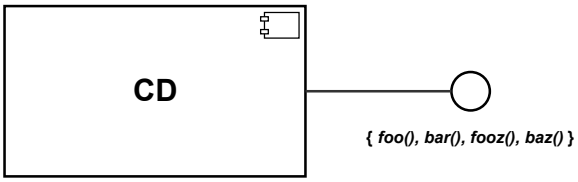


Figure 4.1: UML diagram of parallel component CD

4.4.2 The disjoint operator

The disjoint operator, denoted by $\otimes$, is a specialized version of the parallel operator. Two components are disjoint when their required interfaces do not overlap, and a component does not provide something that the other requires [Don+13].

Definition 4.5 (Disjoint composition of components). Let C_1 and C_2 be two disjoint components. Their disjoint composition, $C_1 \otimes C_2$ results in a new component where

$$\begin{aligned} C_3.pIF &= C_1.pIF \cup C_2.pIF \\ C_3.rIF &= C_1.rIF \cup C_2.rIF \\ C_3.vars &= C_1.vars \cup C_2.vars \end{aligned}$$

The disjoint operator is useful in expressions when it is a requirement that the two operands are disjoint. If they are not, we will get a type checking error and the whole expression will not pass the type checker. This operator is commutative and associative. The definition of disjoint composition involving one or more processes is the same as for the parallel composition operator.

4.4.3 Renaming

We can use the renaming functionality if we want to rename some of the methods in the interfaces of a component. Renaming applies to methods in the provided interface. Renaming methods can be handy when certain interfaces of two components do not match, but should be used carefully. There is no restriction to rename e.g. a method called `add` to `delete`.

Definition 4.6 (Method renaming). Let C_1 be a service component. The renaming of its methods is defined as

$$C_2 = C_1[old_1 \leftarrow new_1, \ldots, old_k \leftarrow new_k]$$

The result is a new component C_2 which is identical to C_1 except its methods $old_1, \ldots, old_k$ are renamed to $new_1, \ldots, new_k$.

Example 4.7. We can use the renaming operator on component C (listing 4.3) to create a new component where names of two of the provided methods have changed. $E = C[foo \leftarrow food, baz \leftarrow barz]$. The new component E will now provide the methods $\{food, baz, bar, fooz\}$.

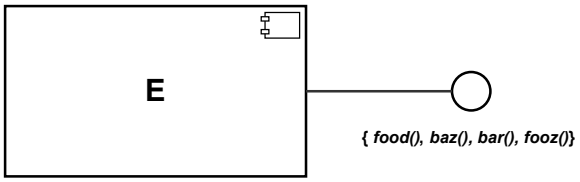


Figure 4.2: UML diagram of renamed component E

4.4.4 Restriction (hiding)

Restriction, also known as hiding, takes a list of method names that is going to be restricted from the environment outside the component.

Definition 4.8 (Method restriciton). Let C_1 be a service component. The restriction of at least one of its provided methods, denoted as $C_1 \setminus \{method_1, \ldots, method_k\}$ results in a new component C_2 where $C_2.pIF = C_1.pIF \setminus \{method_1, \ldots, method_k\}$.

Example 4.9. We can use the restriction operator on methods $\{foo, bar\}$ in component C to create a new component where these methods are hidden from the outside environment. This is expressed as $E = C \setminus \{foo, bar\}$.

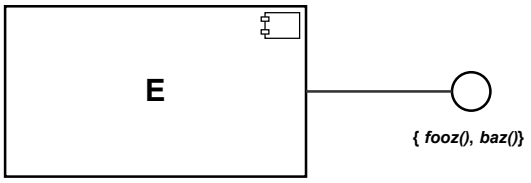


Figure 4.3: UML diagram of restricted component E

4.4.5 The plugging operator

The last composition operator to define is the plugging operator, denoted by $\ll$. This is a binary, infix operator, and is a combination of parallel composition and restriction.

Definition 4.10 (Plugging composition of components). Let C_1 and C_2 be two components. The plugging of C_1 with C_2 is defined as

$$C_3 = C_1 \ll C_2 = (C_1 \parallel C_2) \setminus C_2.rIF$$

It is required that $C_1 \parallel C_2$ is defined, and the restriction of $C_2.rIF$ only applies if $C_2.rIF \cap C_1.pIF \neq \emptyset$. The restriction operator, as shown in definition 4.8, applies to the provided interface of a component, so the effect of the plugging composition is that the provided methods of C_1 that C_2 requires are only available to C_2, and they are not exposed to the outside environment through the provided interface of C_3.

Example 4.11. In this example we will create a new component *CD* by using the plugging operator on component *C* and *D* in listing 4.3. The expression $CD = C \ll D$ creates a new component *CD* which provides only the methods found in interface *B*, since the methods in *A* are hidden and only accessible for component *D*.

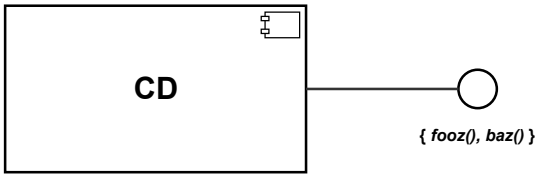


Figure 4.4: UML diagram of plugging component CD

4.4.6 Operator precedence and associativity

In this section we will take a look at the precedence and associativity of the operators. The EBNF grammar of the composition operators is given in listing 4.4. Specifying operator precedence is important to avoid ambiguous grammars. An ambiguous grammar can generate two different parse trees from the same input string [Lou97]. There are two basic methods to make a grammar unambiguous. The first one is to explicitly provide *disambiguating rules* which tells the parser which of the different parse trees is correct. This will not solve the unambiguity of the grammar, but we do not have to change the grammar to get the correct parse tree. The other method is to change the structure of the grammar so it forces the parser to construct the correct parse tree [Lou97]. As the grammar in 4.4 shows, the precedence of the operators are given by method two. Associativity deals with which side of the binary composition operator production rules in the grammar we put the recursive case. In our grammar, we have the recursion on the right side, making this right associative for the binary operators. This means that in cases of A = B ‖ C ‖ D, the composition of `C` and `D` is further down the parse tree, and the resulting component (of composing `C` and `D`) is then composed with `A`. This is equivalent with the parenthesized expression A = B ‖ (C ‖ D).

Of the binary operators, the plugging operator has the lowest precedence, followed by the parallel operator and with the disjoint operator with the highest precedence. This can be seen in listing 4.4. This means that the disjoint operator is "used" first, and is deeper in the parse tree than the parallel and plugging operators. The reason for this is that the primary use cases of the disjoint operator is to express a specific parallel composition when we require that the two operands are disjoint. Plugging has lowest precedence because we say that one wants to "plug" together the two component operands after they have been fully evaluated.

Listing 4.4: Component operator grammar

```
1  compositionexpr ::= CNAME "=" pluggingexpr ";" ;
2
3  pluggingexpr ::= parallelexpr ("≪" pluggingexpr)? ;
4
5  parallelexpr ::= disjointexpr ("‖" parallelexpr)? ;
6
7  disjointexpr ::= baseexpr ("⊗" disjointexpr)? ;
8
9  baseexpr ::=
10     "(" pluggingexpr ")"
11         (("\" "{" restrictionexprlist "}")?
12          | ("[" renameexprlist "]")? (pluggingexpr)?)
13     | CNAME "[" renameexprlist "]"
14     | CNAME "\" "{" restrictionexprlist "}"
15     | CNAME ;
```

When it comes to renaming and restriction, they work by taking the *closest component to the left*, and they take precedence over all the binary operators. In fact, the renaming and restriction can be seen as base cases of the binary operators, together with the case where we have a single name which identifies a component. As an example, imagine we have an expression like this: A = B ‖ C \ {method, method2}. With our grammar, this is equivalent to A = B ‖ (C \ {method, method2}). Here we use the restriction operator on component `C` before applying the parallel operator to `B` and the restricted component. If we want to first apply the parallel operator and then use restriction on the result, we have to parenthesize the expression like this: A = (B ‖ C) \ {method, method2};. Parenthesis can also be used in expressions where we want to override the natural precedence. For example, if we want parallel to take place over disjoint, we parenthesize the expression like this: A = (B ‖ C) ⊗ D;

4.5 Airport example revisited

In this section, we will go back to the airport example which we modelled in section 3.3.1, and see how we can model it using the new language. The full specification can be found in appendix A.2. We will utilize the composition operators to see how we can use them to create a real model. Note that there are some minor changes to the structure of the model in order to adjust it to the algebraic composition.

Recall that the airport consists of two `Gate` components and a `CustomerService` component. The airport component has dependency requirements on the gates and customer service. The gate interface, `IGate`, is shown in listing 4.5. In this example, there are no method bodies in the interfaces, it is just method signatures. The components will provide the implementation. The `Gate` component is also displayed in the same listing. The pre- and post conditions and protocols that

previously were in the contract have been moved to the provided interface of the component, and recall from section 3.3.1 that the behaviour of the methods was provided by the class *GateClass*. This is no longer the case, and the code is moved to the provided interfaces of the component instead. This will minimize the lines of code and create a more compact specification that is easier to understand.

Listing 4.5: The IGate interface and Gate component

```
1 interface IGate {
2     public loadPassengers(int numPassengers, Flight flight);
3     public unloadPassengers(Flight flight);
4 }
5 component Gate {
6     provided IGate {
7         public loadPassengers(int numPassengers, Flight flight)
              {
8             [numPassengers > 0 ⊢ true ];
9             flight.numPassengers := numPassengers
10        }
11
12        public unloadPassengers(Flight flight) {
13            flight.numPassengers := 0
14        }
15        protocol { loadPassengers ; unloadPassengers }
16    }
17 }
```

The customer service component is displayed in listing 4.6. Note that the specification of `Office` and `CustomerCentre` now are top-level specifications, instead of being internal components (see listing 3.3). The `CustomerCentre` component also provides `ICustomerCentre`, as before, and `ICustomerService`, which is different from the airport in chapter 3. The most interesting part of this example is the component composition which uses the plugging operator to compose the `Office` component and `CustomerCentre` component together. The result is a new component, `CustomerService`. This new component we just created will provide methods found in `ICustomerService` and `ICustomerCentre`. The methods in the provided interface `IOffice` is not provided to the outside environment by `CustomerService` because we use the plugging operator.

The airport component is shown in 4.7. We can see that the composition that previously occupied much of the specification in rCOSP of this otherwise trivial component is removed. This has reduced the complexity of the component to only require three interfaces. Two `IGate` interfaces are required, so we use the enumeration of required interfaces to denote that. This means that renaming of the methods of the "second" required `IGate` will take place, and the result is that the airport requires four methods; `loadPassengers, unloadPassengers, loadPassengers2, unloadPassengers2`. The composition expres-

Listing 4.6: The new CustomerCentre component

```
1 component Office {
2     provided IOffice {
3         public provideEmployee(;string employee) {/*Impl*/ }
4     }
5 }
6 component CustomerCentre {
7     provided ICustomerService {
8         public handleCustomer() {/*Impl*/}
9     }
10    provided ICustomerCentre {
11        public customerSupport(string inquiry; string support)
              {/*Impl*/}
12    }
13    required IOffice;
14 }
15
16 CustomerService = Office ≪ CustomerCentre;
```

sion can be seen below the component specification. First (line 6-7), we rename one `Gate` component in order to accommodate for the renaming of the required `IGate` interface in the airport component. The second expression (line 8) tells us to create a new component, `OneGateAirport`. This plugs the `GateRenamed` into the atomic airport. The third expression (line 9) plugs another `Gate` into this composite component we just created. Now we have two gates, but the customer service is missing. This is expressed in the last expression (line 10), where we plug the `CustomerService` component to the airport. Before doing so, note that we restrict, or hide, the provided `customerSupport` method. In the example, the expressions are split up to provide an easier overview. It would also be possible to create one big expression at the cost of readability. In the end the result is the same, namely one composite component that is created by other composite components, all the way down to the atomic components which we defined.

Listing 4.7: The new airport component

```
1 component Airport {
2     required ICustomerService;
3     required IGate[2];
4 }
5
6 GateRenamed = Gate[unloadPassengers ← unloadPassengers2,
7                    loadPassengers ← loadPassengers2];
8 OneGateAirport = GateRenamed ≪ Airport;
9 TwoGatesAirport = Gate ≪ OneGateAirport;
10 MyAirport = CustomerService \ {customerSupport} ≪
       TwoGatesAirport;
```

4.6 Summary

In this chapter we presented a new language, rCOSPN, for component specification and composition. The language emphasizes an easy-to-use way of creating components, where the focus is on atomic components and their required and provided interfaces and the composition of these components. We looked at and defined the composition operators *parallel*, *disjoint*, *plugging*, *restriction*, and *renaming*. In the last section (section 4.5) of this chapter we went back to the airport example from section 3.3.1, and modeled it again using the new syntax and the composition operators. The result was a short and clear specification where the operators made compositions easy, without the need to be explicit about the attachment of required and provided interfaces.

Chapter 5

From code to model

5.1 Introduction

Having introduced the theoretical aspects of the component specification language and formally defined the composition operations *parallel, disjoint, plugging, renaming* and *restriction* in the previous chapter, this chapter is all about the practical aspects of the implementation of the language and the creation of the UML models. We will go through every stage that occurs when translating from a textual specification to the UML model, and will study the airport example again.

The chapter is outlined as follows: section 5.2 will go through the stack of technologies utilized in the implementation. Section 5.3 describes the rCOS UML profile, and is followed by a section which describes the structure of the UML models. We will also look at how we go from text to UML model (section 5.5) and see this in practice in section 5.6 where we create a UML model of our airport example.

5.2 Technology stack

Several technologies were used to implement the tool, and this section will give an overview of them. These are the Eclipse Modeling Framework (EMF), the parser generator ANTLR, Eclipse UML2, TOPCASED and the Scala programming language. The stack is depicted in figure 5.1.

5.2.1 The Eclipse Modeling Framework

Eclipse[1] is an open source project [Ste+09] and Integrated Development Environment (IDE) for software application development. It is primarily used for developing Java applications, but other languages are supported via plug-ins. *Eclipse Modeling Framework* (EMF) is a framework built on top of Eclipse. EMF tries to extinguish the gap between modeling and programming. Instead of creating a model of the

[1] https://www.eclipse.org/

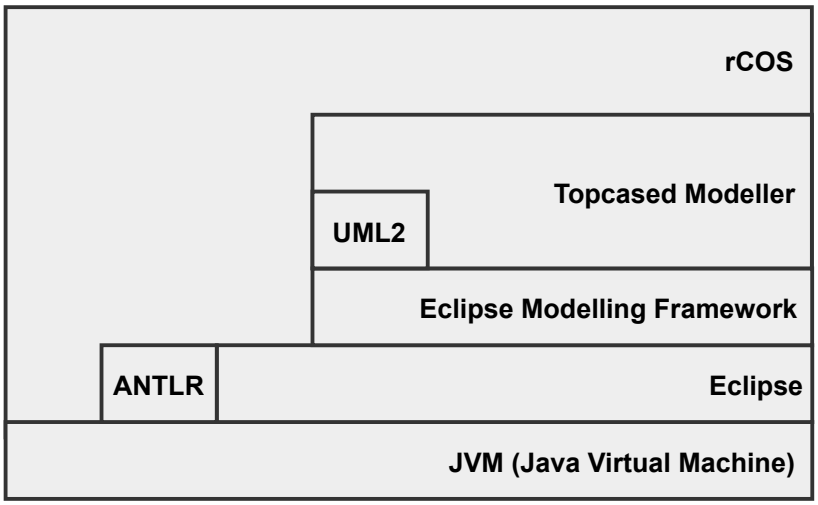


Figure 5.1: Technology stack

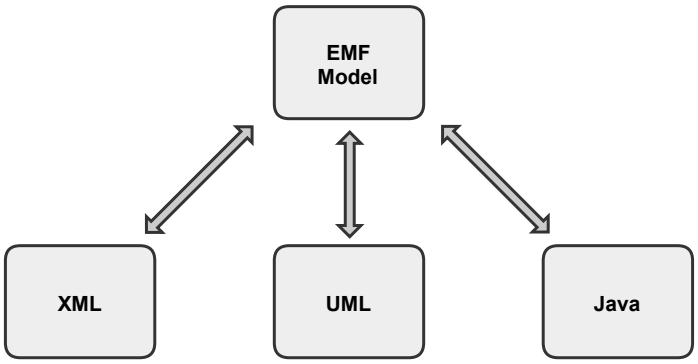


Figure 5.2: EMF relation between XML, UML and Java. Figure is derived from [Ste+09], page 14

software first and do the implementation after, the developer defines an EMF Model of the application domain in either XML Schema, UML or Java and then generates code from the model. Transformation between XML Schema, UML and Java is seamlessly supported, and EMF unifies these three technologies [Ste+09]. It is also possible to create the EMF Model directly in Eclipse and generate the other forms from it. The relationships are depicted in figure 5.2. The model which is used to build other models in EMF is the *Ecore* model [Ste+09]. The Ecore model itself is defined as an EMF model, which makes it a meta-model. The advantages of using EMF are code generation and model transformations. We will see later in this chapter how these provided facilities were used in the development of the tool.

5.2.2 ANTLR parser generator

Writing language recognition programs by hand can be a tedious task. A language recognizer consists of two parts, a *lexer* and a *parser* [Par07]. The lexer creates *tokens* from the stream of textual input. These tokens are consumed by the parser which analyses the syntactic structure of the input. If the input is not syntactically correct, as defined by the grammar, the parser will give an error. There are several tools which can generate a lexer and parser to a target language, e.g. Java, from a grammar specification written in a grammar language provided by the tool. Some of the most well-known for the Java programming language are JFlex[2], which generates a lexer, and CUP[3], which generates a parser. These two can be used in combination to create a full language recognizer. Another option is to use ANTLR[4], shorthand for **AN**other **T**ool for **L**anguage **R**ecognition, which generates both the lexer and the parser from a single grammar specification. In this project ANTLR version 3 is used, because it allows us to reuse parts from the rCOSP specification grammar, such as method bodies. Another reason for choosing ANTLR is the grammar development tool ANTLRWorks[5] which supports syntax highlighting, grammar debugging facilities and a syntax diagram visualizer, among others. This allows for quick grammar development. The last reason to choose ANTLR is that it supports tree building and tree grammars, which will be explained later in this section.

To generate a lexer and a parser with ANTLR, the grammar is given on Extended Backus-Naur Form (EBNF) in a grammar file. It is then possible to generate the lexer and parser to two separate Java files which can be compiled and run. Creating the recognizer is the first step to building an interpreter, translator or compiler. The next step is to do something beyond just recognizing the language, so we need to enhance the parser with *actions*. An action in ANTLR is a block of code, written in the target language [Par07] in the grammar, and inserted into the generated parser. Actions can be used to perform various tasks such as putting variables in a symbol table, retrieving information about production rule references and executing a method call. Actions outside production rules are used to declare global members (methods and variables). An action is triggered when the production rule it is associated with is invoked. Listing 5.1 shows two actions associated with the *component* production rule at line 7, one for each alternative in the rule. In this case, either an exception is thrown or the name of a component is set, depending on what alternative is matched. The grammar file can contain "import" statements which are directly inserted into the generated lexer and parser. This makes it possible to use external Java code in the generated code. In addition to defining

[2] http://www.jflex.de
[3] http://www2.cs.tum.edu/projects/cup/
[4] http://www.antlr3.org/
[5] http://www.antlr3.org/works/

Listing 5.1: ANTLR example

```
1 // in text parser grammar file
2 compositionExpr
3     : name=CNAME '=' p=pluggingCompositionExpr ';'
4     -> ^(COMPOSITION $name $p) ;
5
6 // in tree parser grammar file
7 composition
8     : ^(COMPOSITION name=CNAME name2=CNAME)
9       {throw new SimpleAssignmentException
10        ("Simple assignment at " + name + " = " + name);}
11    | ^(COMPOSITION name=CNAME comp=ASTcompositionExpr)
12      {m.setName($comp.r, $name.text);};
```

actions, production rules can be parameterized, have return values, and define scopes.

Sometimes a single pass through the input is not enough to solve a language problem [Par07]. We will later see how this applies to our parsing of rCOSPN. Using the lexer and parsing the token stream multiple times is one solution, but a more efficient way to do multiple passes is to construct one or more *abstract syntax trees*, or *AST* for short. An AST is a simplified parse tree. ANTLR version 3 supports construction of such trees during parsing with the use of rewrite rules. Line number 4 in listing 5.1 shows the construction of an AST when a composition is matched. The arrow "->" denotes the start of the rewrite rule. In this case, the root node **COMPOSITION** is created with the name and composition expression as child nodes. A *tree grammar* is created to parse the AST. A tree grammar generates an AST parser. Parsing the AST and triggering the appropriate action for each tree node makes it easy to translate the input into something else, like a UML model in our case. The actions trigger Java code which calls methods to build each model element.

5.2.3 Other technologies

Eclipse UML2

The Modeling Development Tools[6] (MDT) is an Eclipse project which offers implementations of industry standard metamodels and tools for creating models with these metamodel implementations. One of the tools offered is an implementation of the UML 2.x metamodel, called Eclipse UML2. This implementation makes it possible to define the UML models programmatically by using its API. Eclipse UML2 is used in the tool to build the UML models from the textual specification.

[6] http://www.eclipse.org/modeling/mdt/

The Scala programming language

The Scala programming language is a statically typed general purpose programming language that runs on the Java Virtual Machine (JVM). Scala code compiles to Java bytecode. This ensures interoperability with existing languages on the JVM such as Java. Calling code written in Java from Scala is a trivial task. Scala lives in both the object-oriented programming and functional programming paradigm. As an object-oriented language, every value in Scala is an object [OSV11]. Functional programming is about computation without side effects, immutable values and functions as first-class citizens. As a functional language, Scala supports all these features. The reason for choosing Scala as the programming language in this thesis is that it emphasizes concise and expressive code, supports interoperability with Java (the existing rCOS tool is built in Java), and due to personal preference. Eclipse supports Scala development through the use of the Scala IDE plug-in[7].

TOPCASED modeling Tools

TOPCASED[8] is an acronym for **T**oolkit in **OP**en-source for **C**ritical **A**pplications and **S**yst**E**ms **D**evelopment [Far13]. TOPCASED is a modeling tool which, among others, offers model editors, model transformations and model checking which is built on top of the Eclipse Modeling and the Graphical Editor frameworks.

5.3 The UML profile

As section 2.4 states, a profile is an extension mechanism to a referenced metamodel. This allows us to create model elements which are necessary for the modeling domain and are not available in the UML metamodel. The composition operators introduce a modeling problem that is not covered by the elements in the UML metamodel. The metamodel contains a component element, but this element does not record which operator and components, either atomic or composite, that were used in the composition to create the component. We need a way to express this declarative information in the UML. Figure 5.3 shows the rCOS UML profile part for modeling the compositions. The profile is a hierarchy of stereotypes. A UML stereotype is used to extend an existing UML model element. Stereotypes carry the annotation «**stereotype**» and are easily distinguishable from other model elements

The profile is built as an extension of the existing profile for rCOS, and only the part of the profile that covers the components and component composition is shown here. At the top of the hierarchy is the **rCOSComponent** stereotype. This stereotype has an extension association with the component metaclass. This kind of association

[7] http://scala-ide.org
[8] http://www.topcased.org/

is used to show that the properties of a metaclass are extended through the stereotype [Gro11]. The stereotype is attached to the extension via an *ExtensionEnd* property. An ExtensionEnd is navigable, which makes it possible to attach an instance of the stereotype to an instance of the extended classifier without modifying the classifier with a property [Gro11]. The rCOSComponent stereotype also contains a Component property which is attached to the *member end* of the extension association. This member end shows that a Component instance takes part in the association. The result of this is a seamless connection between an instance of an rCOSComponent and an instance of a UML component.

A stereotype can build on other stereotypes via a *generalization relationship*. As depicted in the profile diagram, the ServiceComponent, ProcessComponent and CompositionOperator stereotypes are more specified versions of the rCOSComponent. This concept is similar to the concept of relationship between a superclass and subclass, and the specific stereotype inherits the properties from the general stereotype. The ServiceComponent and ProcessComponent stereotypes are reserved for the atomic components and are applied to the atomic service components and process components respectively. The ComponentOperator stereotype defines the base for every composite component which is a result of an operator application. Since every operator applies to at least one component, the stereotype contains a property, `left`, which is typed by rCOSComponent. This property is used to refer to the left component operand in one of the three binary compositions or as the single component in the two unary operators.

Recall from the operator definitions in chapter four that the disjoint and plugging operators are specializations of the parallel operator. This is reflected in the profile which shows these two operators as UML stereotypes that extends the Parallel stereotype. The Parallel stereotype itself is an extension of ComponentOperator, and adds a property, **right**, which will hold the reference to the other argument. The two unary operators, renaming and hiding, are also defined as two stereotypes which extends the ComponentOperator stereotype. The Renaming stereotype contains a property, **ops**, which is a list of **RenamePair** objects. A RenamePair is a tuple which holds references to the old and new operation. The Hiding stereotype defines a property, **method**, which is the list of hidden operations.

With the profile defined in UML, generating Java code is a trivial task using EMF. Each stereotype gets translated to a Java interface and a class which implements this interface. As we will see, the Java code is used in the construction of the UML models.

5.4 The structure of the semantic UML models

We have stated earlier that the (semantic) UML model can be viewed as a hierarchical tree structure. An example model created by our tool can

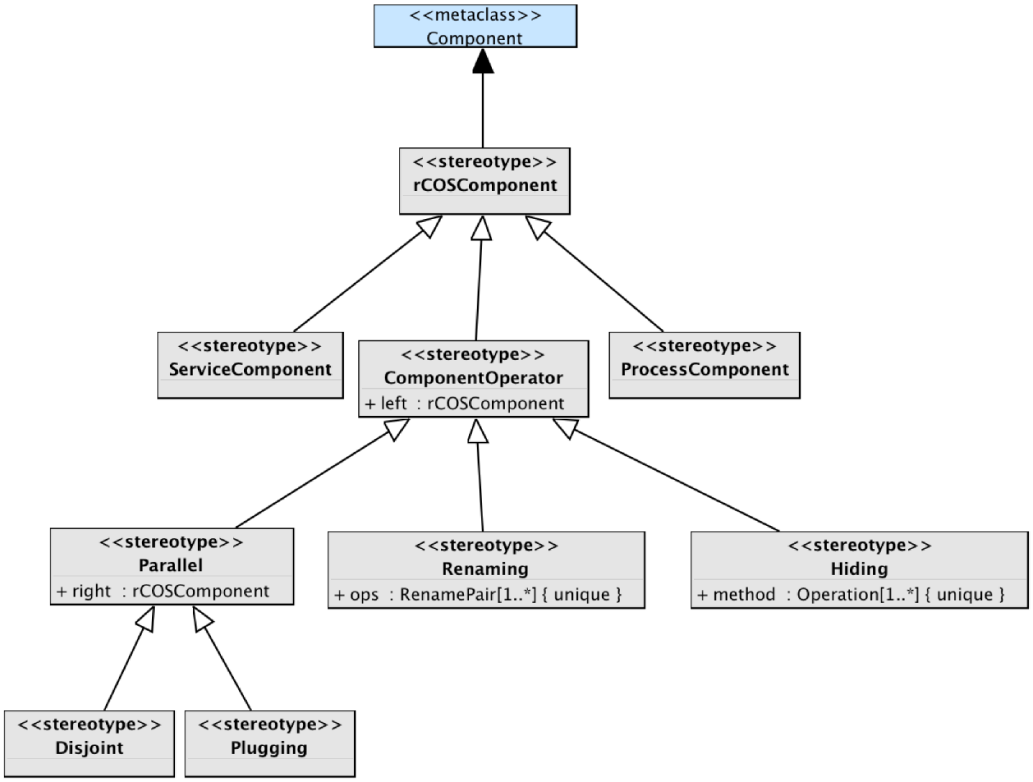


Figure 5.3: UML profile diagram

be seen in figure 5.4. This is the rCOS UML model created by feeding a slightly larger version of the specification given in listing 4.2 to the tool (the `Database` service component and a parallel composition between this component and the `UserAdmin` component is added).

We see that the rCOS UML profile is applied. The rCOS stereotype of a model element is shown with its name surrounded by guillemets. At the top of the hierarchy is the `<Model>` element. The model is defined as a UML package which captures a view of the system. Its child elements describes the complete system [Gro11]. The elements labeled with `<Package>` provide namespaces and are used to group elements. We can see that a class model package, a use case model package and a component model package are created as individual parts of the model in this example. In our case, the component model package is the most interesting package. We can see that the interfaces and components are added to this package.

5.5 How rCOSPN gets translated to a UML model

A model created with rCOSPN can be viewed from three different view points. We have already seen the first one, which is the textual representation defined by the rCOSPN language. The second

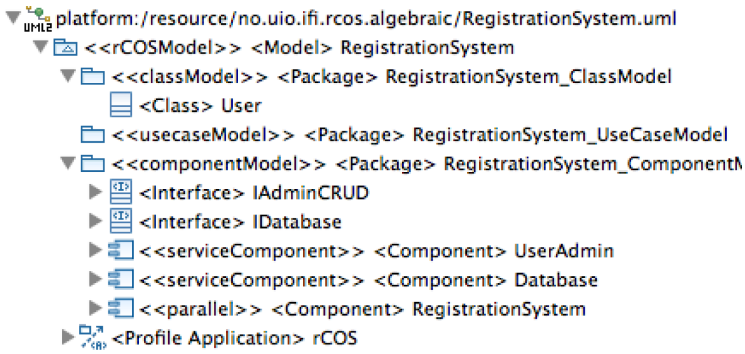


Figure 5.4: UML structure as seen in Eclipse

perspective is as a regular UML model. However, the UML model does not exhibit the declarative information obtained through the composition expressions, i.e. which operator was used to create the composite components and what the left and right operands were. The rCOS UML profile defined can be viewed as the third perspective, and shows the stereotype applications. This section shows how the UML model gets constructed and how the UML-profile is applied to the model elements.

5.5.1 Parsing

During model construction, the prototype builds an in-memory model in Java using the generated Java classes and the Eclipse UML2 API. When the parsing is done, this model is saved using a library function in EMF and can be opened with an UML model viewer. The model gets built during three phases of parsing and each phase is explained in turn below. The parsers are modifications of the parsers in the rCOS tool and adapted to support the new syntax and component composition expressions.

First phase: from text to AST

Parsing the textual input file is the first step in the model construction and is done during the first phase of the parsing. A grammar defined in ANTLR creates a lexer and parser which transforms the text to an AST using rewrite rules. This AST is used in the second and third phase by tree grammars.

Second phase: create basic UML elements

The parser in the second phase is an AST parser, which means that it is defined as a tree grammar in ANTLR (see section 5.2.2. This phase is the first phase in construction of the UML model. The grammar is augmented with actions which are executed when the corresponding production rule is matched. These actions will create the basic UML

elements in the model as Java objects. These elements are interfaces, atomic components and classes.

Third phase: compositions

The third parsing phase will parse the AST a second time using a different tree grammar. In this phase several important events happen. Attributes, methods and method bodies get created and added to their respective elements which were created during the second phase. This is functionality from the existing rCOSP parser. Provided and required interfaces are associated to their components. In cases of provided interfaces, the methods of the interface are added to the component in addition to being in the interface.

During this phase the ASTs of the composition expressions are parsed and the composite components get constructed. Upon encountering a composition expression, parser augmented actions will execute the correct code depending on what kind of composition it is. A composition expression can be viewed as a tree structure, and the composite component will be built bottom up starting at the leaves of the tree. We will go into more detail about composite component construction in section 5.5.3.

A limitation clearly manifests itself during this phase. The parser parses the input from top to bottom. In this case it means that a composition expression can not depend on a component which is a result of a composition expression defined later in the input. As an example, the following expression will create an error during this parse phase: A = B ‖ C; B = D ‖ E. As we can see, the first expression depends on `B`. However, `B` does not exist yet at this point of the parsing.

5.5.2 Model elements

As we now have seen, the UML model is created during parsing. The model elements get "translated" from text to UML via Java objects. We will now examine these translations further and see what each UML element consists of. Note that attributes, interfaces, classes and some parts of components are unchanged in this version of the tool from the previous version.

Attributes: Each attribute declared in a classifier will be added to the UML model as either a UML property (primitive types) or a UML association (non-primitive types).

Interface: An interface definition is translated to a plain UML interface with its declared methods and attributes.

Required interface: Each required interface declaration in a component is given as a *usage* dependency relationship in the UML model. The full requirement specification of a component is the union of the meth-

ods in the required interfaces. In cases where the required interface declaration is followed by a number denoting enumeration, the corresponding number of interfaces are created and their methods renamed. The component will have usage dependencies to all of these interfaces.

Provided interface: A provided interface declaration is translated to an *interface realization* relationship between the component and the interface. Each method defined in the provided interface block is added to the component as an *owned operation*. The type-checker will verify that a component provides all methods that are declared in the interface. For more about type-checking, please refer to chapter 6.

Class: A class in rCOSPN is simply constructed as a regular UML class. The classes are added to the rCOS class model package as top-level classifiers. The consequence is that there are no local classes even though they can be defined inside a component.

Component: The components are regular UML components with a certain rCOS stereotype applied. This will make them rCOS components. The stereotype applied depends on what kind of component it is: Service, Process, Parallel, Disjoint, Plugging, Renaming or Hiding. If the component is a composite component, a UML *dependency* relationship will be created. The component it depends on (one component for Renaming and Hiding, two components for Parallel, Disjoint and Plugging) is the supplier, and the composite component is the client in the dependency relationship. The purpose of having these UML dependencies is to be able to display in a UML diagram which components were used to create the composite component, they do not show which of the components were the left and right operands. The reason is that we do not have diagram capabilities to show the structure of the rCOS components and the UML dependencies work as a substitute for that.

5.5.3 Composition expressions

A composition expression can be arbitrarily large, i.e. contain subexpressions, and the abstract syntax tree of an expression is traversed in a post-order traversal. This means that the subtrees of the root are visited before the root. The root of a tree tells us what kind of composition it is. A composition using a binary operator has the left and right operand as its subtrees. Renaming and hiding have two subtrees as well; the component to apply the operation to and a list of names denoting the operations. The nature of post-order traversal implies that the composition is built bottom-up. The leaf nodes of the tree are either atomic or already defined composite components. The operators are implemented according to the definitions found in chapter 4.

The parallel operator will create a new component which provides the union of the methods provided by the left and right subtree. Each

provided method will be *copied* over to the new component. The required methods will be the union of the required methods of the left and right subtree, except those that are also provided. The union of attributes are copied over as well.

The disjoint operator is almost equal to the parallel, except that it is mandatory for the required methods of the left and right subtree to be disjoint and that one component does not provide something the other requires.

The plugging operator is equal to the parallel, except that the set difference between the provided and the required is calculated and exposed as the provided interface.

The hiding operator will create a new component based on its left subtree. The operations that are *not hidden* will be copied over to the new component. The operations that are hidden are added to the list of hidden operations. The effect of this is that we know *what* the component exposes (UML view) and *why* it exposes it (rCOS view).

The renaming operator behaves similar to the hiding. It copies every operation from the component in the left subtree, renamed if specified. It keeps a list of **RenamePair**s to keep track of which operations are renamed and to what.

The union of the provided interfaces is collected into one interface. On condition that the computed set of provided methods is not empty, a new UML interface is created. This interface is added to the context of the component. An UML interface is created for the computed set of required methods as well, given that the set is not empty.

The lack of the special operator symbols $\|$, $\otimes$ and $\ll$ on a normal keyboard is solved by allowing them to be replaced by || (two vertical bars), *o* (a single 'o' character) and << (two "less than" characters) in the composition expressions.

5.6 Example: creating the airport UML model

In this section we will run the tool on the airport specification given in section 4.5 and inspect the UML model both as a hierarchical tree and diagrammatically. As before we will begin with the `Gate` component, defined in listing 4.5. Figure 5.5 shows the tree view of the `IGate` interface and `Gate` component as displayed in the UML model view in Eclipse. The complete UML component model is appendix A.3. The `IGate` interface is a plain UML interface with its two owned operations. The `Gate` is an rCOS UML component with the service component stereotype applied. The component has an interface realization relationship with the interface.

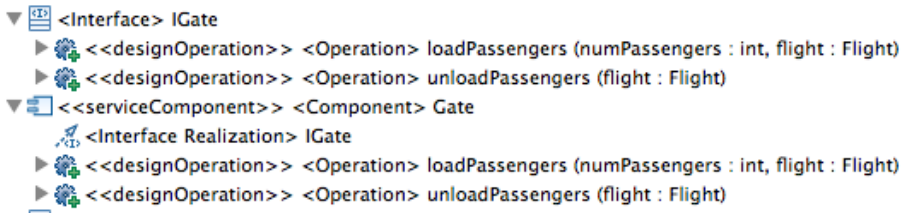


Figure 5.5: The IGate interface and Gate component in UML

Listing 4.6 defines the components `Office` and `CustomerCentre`. These are similar to the `Gate` component. This listing also contains the composition expression which creates the `CustomerService` component. The UML model of this component is depicted in figure 5.6 and its graphical counterpart, which is plain UML except showing the rCOS stereotypes, is in figure 5.7. This component is a result of applying the plugging operator, so this is the applied stereotype as well. The dotted arrows are the UML dependencies which refer to the two operands. Note that the `Office` and `CustomerCentre` components are shown outside of the `CustomerService` component in the diagram, and not inside. This is because they are considered top-level classifiers. We will discuss this in section 7.2.3. Recall the definition 4.4.5 which says that the provided methods of the left operand are going to be hidden if they are required by the right operand. This is reflected in the model, and the provided operation `provideEmployee` by the Office component (left operand) is not a part of the provided interface of the `CustomerService`, since `CustomerCentre` (right operand) requires it. The new provided (called `_provided` in the model) interface is calculated and added locally to the component. The component has an interface realization with this interface.

Figure 5.6: The CustomerService UML component

The final component, `MyAirport`, is shown in figure 5.8 and figure 5.9 as semantic model and graphical model, respectively. We see that it is a plugging composition. The expression which creates this model contains a sub-expression (see listing 4.7 line 10), that is the restriction on method `customerSupport` in component `CustomerService`. This is reflected in the model as the nested component `CustomerService_restricted`. The graphical diagram is shown in the Eclipse environment in figure 5.9.

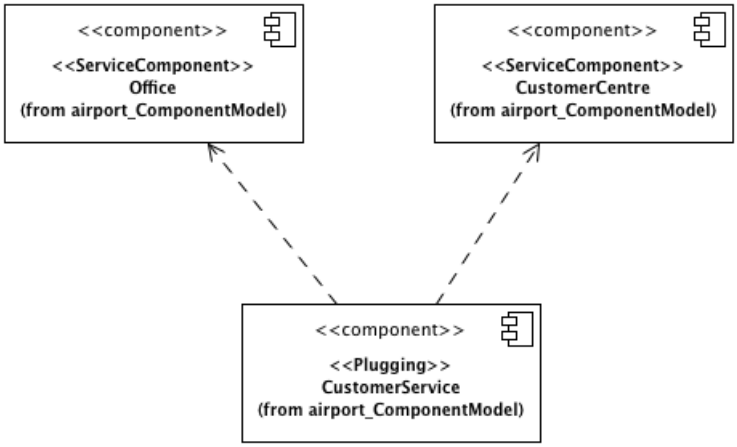


Figure 5.7: The CustomerService UML component diagram

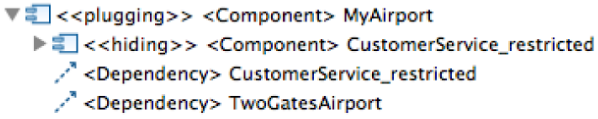


Figure 5.8: The MyAirport UML component

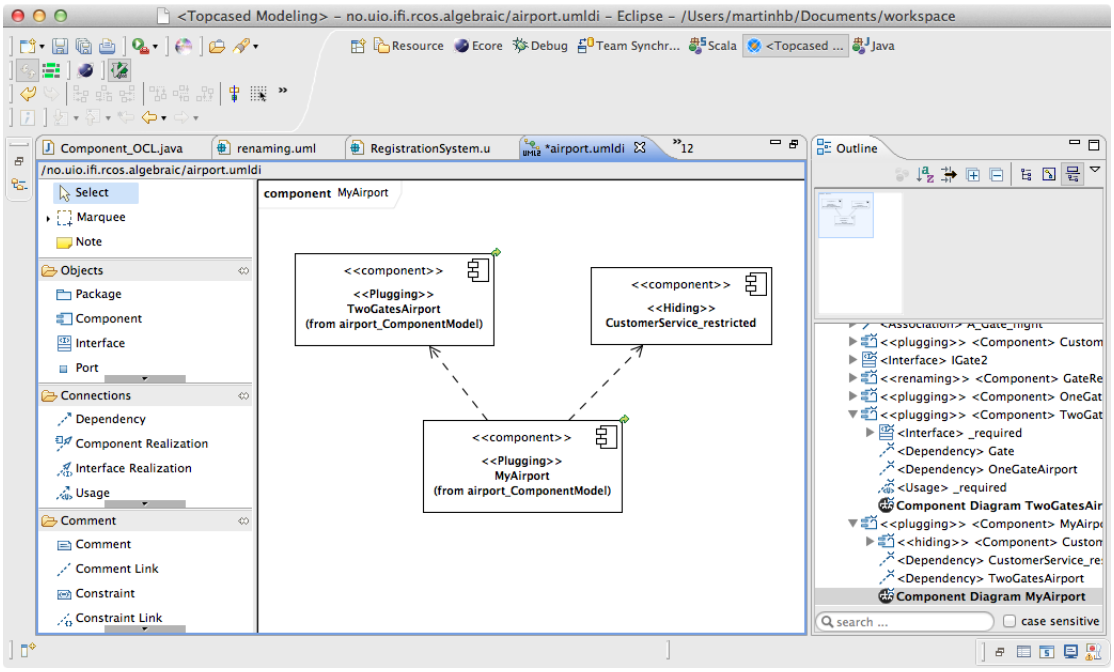


Figure 5.9: The final component, MyAirport, in Eclipse

5.7 Summary

We started this chapter by giving an overview of the technologies which were used to develop the tool. Among these were the Eclipse Modeling Framework which offers a convenient way to unify UML, XML and Java and easy conversion between these formats. We saw how the ANTLR parser generator was used to create parsers used for model building. The rCOS UML profile for was defined in section 5.3. This profile extends the UML metamodel with custom stereotypes for correct domain modeling. It exists one stereotype for each type of component, seven in total.

Several phases of parsing makes the transformation from textual specification to the UML model complete. Section 5.5 presented each stage: parsing text to AST, parsing the AST the first time to create basic UML model elements and parsing the AST a second time for "filling in" these basic elements. The composition expressions are handled during the last phase. This section also covered the details about constructing the composite components. During development a number of design decisions had to be made. These are presented in section 7.2.

The last section went back to the airport example and created a UML model of the airport which was given as a rCOSPN specification in chapter 4, and viewed the model both semantically and graphically.

Chapter 6

Type-checking and model validation

6.1 Introduction

To avoid creating erroneous UML models and violation of the component operator definitions it is vital to do various checks on the model entities, especially the components and composition expressions, in the different phases of the model construction. Type-checking the input specification during parsing is important to ensure that a correct model is created. Even if a model is valid immediately after construction (from an rCOSPN input specification), it is highly possible that it will be altered at a later point. Having the possibility to do model validation on the UML model whenever necessary is important to ensure the model stays correct at all times. If the model is correct, the validation passes. If not, we get one or more notifications that tells us which parts are wrong. The first part (section 6.2) of this chapter will look at the type-checking of components and component composition expressions, which is done during parsing. The second part (section 6.3) covers the validation of the UML models. The third part (section 6.4) lists all the type-checking and validation rules.

6.2 Type-checking

Mitchell [Mit02] defines a *type* as ”a collection of entities that share some common property” and *type-checking* as ”an algorithm that goes through the program to check that the types declared by the programmer agrees with the language requirements”. In our case the type is the collection of components. The scope of this thesis is the focus on the specification of atomic components and their compositions i.e. the structural model perspective, and we will only concentrate on the type-checking of components and composition expressions. However, the type-checking of rCOSPN specifications is not limited to components only. Arithmetic expressions, method invocations and method bodies are other examples where type-checking is important. The rCOS tool

which we build on offers type-checking for these categories which can be integrated in our tool.

The set of components which we just defined as a type can be divided into two groups, namely the atomic components and the composite components. Since the composite components depend on the atomic components, it is important to make sure that the atomics are correct before they are used in compositions. If they are not, even a small error may ripple through to the composite components, making the model wrong. The atomics are run through the type-checker immediately after they are created, which will do various checks such as seeing if the provided block(s) are complete (every method in the interface is provided) and that there is only one of each provided method. The full list can be seen in section 6.4. If the type-checker detects an error, the parsing is canceled, no model is built and the error is displayed to the user.

When an algebraic composition expression is encountered, the type-checking routine will be invoked before the composite component is created. Depending on the type of operator used, either binary or unary, this routine will inspect the expression and do various checks similar to what is done to the atomic components. If restriction or renaming is used, the component which this operator is applied to will be inspected. In renaming for example, it is verified that the methods that are going to be restricted exist in the component. For the binary operators both operands are checked, and tests such as making sure that they are compatible, for example no common provided methods, are executed. There are different checks for each type of composition. As an example, the expression `A = B ‖ C;` will run the parallel checking routine on the operands *A* and *B*. If they somehow do not fit together, e.g. overlapping provided operations, an error is generated and shown to the user.

The checks done during parsing could also be done during model validation, and we will see that this is in fact also the case for some of them (see the tables in section 6.4 for an overview of the rules). Given that the model can be validated after construction, type-checking as described above is not a critical requirement since the errors caught during this phase can also be caught during model validation. Therefore an alternative solution could be to do no type-checking during parsing at all. However, there is no reason to create a model if the specification is wrong, and if there is one error present in the specification we should get a notification about this at the moment it is discovered and not wait until the model is built so it can be validated.

6.3 Model validation

Once the rCOSPN input is parsed, type-checked and found correct, the output is the UML model with the rCOS profile applied. This model is a representation of the textual input specification, and can be edited further, for example by adding a new interface and a new component.

To be safe that these changes do not violate the rules, we can validate the model to ensure that it stays correct. We say that the model is valid when it conforms to the rules defined in section 6.4. In the tool, the model validation process can be invoked at all times. This means that errors are discovered quickly if the user alters the model after its construction, e.g. by adding an operation to an interface, but forgets to add it to the provided block of a component.

6.3.1 The Object Constraint Language

The Object Constraint Language (OCL), a declarative language, is defined as an add-on to the UML [WK03]. The OCL can be utilized to augment a UML model with expressions, constraints and information that cannot be expressed by the UML alone. For example, a UML class model can say that a `Car` has a set of seats and an association to the `Person` class, but not that the number of passengers must be less than or equal to the number of seats. In our implementation, we use the OCL to formulate the model validation rules as Boolean queries. The outcome of these queries (true or false) depends on whether the queried model element, the *query context* satisfy the rules. A query that checks that a component does not provide the same interface more than once is expressed as self.interfaceRealization->asBag()->isUnique(name). The keyword `self` refers to the context, i.e. the component currently tested. In our implementation, each OCL query is declared as a Java string and the context is set explicitly before each query.

6.4 Type-checking and model validation rules

How do we know when a model is correct? In order to type-check components and composition expressions, and validate the UML models we need to define a set of rules, or "checks", which the expressions and models must conform to. The rules of the atomic components can be found in table 6.1. These rules applies both to type-checking and model validation. The rules for the composite components which are created by one of the binary composition operators are found in table 6.2. These rules only apply to the model validation. The renaming and restricted (the unary composition) model validation rules are shown in table 6.3. Note that some rules are listed in multiple tables, since they apply to more than one category of components.

The type-checking and model validation rules for component composition expressions are described in table 6.4 and table 6.5. The former shows the rules for the binary operators, and the latter displays the rules for the unary operators.

Rule	Service	Process
There must be no duplicate variable definitions	X	X
The set of provided and required methods must be disjoint	X	-
Must provide an implementation of every method declared	X	-
Must not provide a method that is not declared in the interface	X	-
Must not provide the same interface twice	X	-
No duplicate provided methods	X	-
Must not provide the same method twice	X	-
Must not provide methods	-	X

Table 6.1: Type-checking and model validation rules for the atomic components

Rule	Parallel	Disjoint	Plugging
There must be no duplicate variable definitions	X	X	X
The set of provided and required methods must be disjoint	X	X	X
Must provide an implementation of every method declared	X	X	X
Must not provide a method that is not declared in the interface	X	X	X
No duplicate provided methods	X	X	X
Must not provide the same method twice	X	X	X
Should provide the union of the methods of the sub components	X	X	-
Should contain the union of the variables of the sub components	X	X	X
Should not require methods which are provided by one of the sub-components	X	X	X
Should require every method from both sub-components	-	X	-
Provided methods by the first operand which is required by the second is hidden	-	-	X

Table 6.2: Model validation rules for the binary composite components

Rule	Renaming	Restriction
There must be no duplicate variable definitions	X	X
The set of provided and required methods must be disjoint	X	X
Must provide an implementation of every method declared	X	X
Must not provide a method that is not declared in the interface	X	X
No duplicate provided methods	X	X
Must not provide the same method twice	X	X
Should still provide the methods that are not renamed	X	-
A renamed operation is not provided with its old name	X	-
Every renamed operation is provided with its new name	X	-
Should provide those methods that are not restricted	-	X
Should not provide those methods that are restricted	-	X

Table 6.3: Model validation rules for the unary composite components

Rule	Parallel	Disjoint	Plugging
No common variables	X	X	X
No common provided methods	X	X	X
No common required methods	-	X	-
One component do not provide a method the other requires	-	X	-

Table 6.4: Type-checking rules for the binary composition operators

Rule	Renaming	Hiding
The operations which are renamed or restricted must exist	X	X
Not rename an operation more than once	X	-

Table 6.5: Type-checking rules for the unary composition operators

6.5 Examples

The code listings 6.1 and 6.2 demonstrates the rule which says that a component can not provide a method that is not declared in one of the interface(s) it provides. The first listing shows the type-checking code of this rule written in Scala. This rule is applied to atomic components during parsing. Lines 2-5 collects every method from the interface(s) specified as provided. Line 7-8 fetches the set of the methods the component owns. The for loop in line 10-16 compares these sets, and throws an exception if an owned method is not found in the set of interface methods. The OCL equivalent of this rule is shown in the second listing. Lines 2-3 fetches the owned method set and the set of methods in the interface(s) it provides, respectively. Then we loop through the owned methods, and for each method check that an equivalent method exists in the other set. This evaluates to either `true` or `false`.

Listing 6.1: Type-checking rule

```
1 def tcNotProvideUndeclaredMethod(component: Component) {
2     val provs = new OperationSet(Buffer.empty[Operation])
3     val providedInterfaces = component.getProvideds()
4     providedInterfaces.foreach(
5             inf => provs ++= inf.getOwnedOperations())
6
7     val owned = new OperationSet(Buffer.empty[Operation])
8     owned ++= component.getOwnedOperations()
9
10    for (o <- owned) {
11      if (!provs.contains(o)) {
12        throw new ProvidedUndeclaredOperationException(
13            "The component " + component.getName
14            + " provides method " + o.getName +
15            " which doesn't exist in any of the provided
                interface(s)")
16      }
17    }
18  }
```

Listing 6.2: OCL validation rule

```
1 let
2  set1 : Set(Operation) = self.ownedOperation->asSet(),
3  set2 : Set(Operation) = self.provided.getOperations()->asSet()
4 in
5  set1->forAll(
6   s1 | set2->exists(s2 | s1.name=s2.name and
7   (let
8    ppars:Sequence(Parameter)=s1.ownedParameter->asSequence(),
9    opars:Sequence(Parameter)=s2.ownedParameter->asSequence()
10  in
11   ppars->forAll(
12    ppar |
13    let
14     opar : Parameter = ppars->at(ppars->indexOf(ppar))
15    in
16     ppar.name=opar.name and ppar.type=opar.type))))
```

6.6 Summary

In this section we emphasized the importance of type-checking components and component expressions during parsing and being able to validate the UML models. It is vital to report any errors if the model designer writes an erroneous specification or introduces an error in the UML model. The model validation rules are written as OCL queries. At the end of this chapter was several tables which show every type-checking and model validation rule which is used to ensure correctness.

Chapter 7

Discussion

7.1 Introduction

This chapter will present an evaluation of the tool and discuss the prototype in light of the problem statement. Section 7.2 will look into the major design decisions made during the development of the prototype. After that follows a discussion on limitations of the tool. Alternatives to using UML profiles to support algebraic component composition are discussed in section 7.4, and section 7.5 will take a look back and compare the algebraic way of component composition with the "traditional" way of component composition. The last section describes how the algebraic component composition can be used in the "real world".

7.2 Prototype tool design decisions

7.2.1 Handling interface compatibility

Dealing with interface compatibility, i.e. when a required and provided interface of two components are compatible, was a topic we had to decide on how to handle when creating the prototype. In the Java programming language, entities can be considered "equal" if they implement the same interface. Consider an example where we have an interface called `Shape` and two classes `Circle` and `Square` which implements this interface. This allows us to pass instances of these two classes to a method where a `Shape` is required. However, if we create another interface `Form`, with the same method signatures as `Shape`, and change `Square` to implement this interface instead, we will get a compiler error if we try to pass an instance of `Square` to the method which requires a `Shape`. This is because these two interfaces are considered distinct types although they contain the same method signatures. This approach is not directly transferable to the algebraic component composition because we don't view an interface as a type. Deciding whether an interface fits with another based solely on the name would be too rigid and is not an optimal solution either.

Another way of describing interface compatibility is to view two interfaces as compatible if the one we would like to use (the provided interface) offers at least the same methods as the one required, but possibly more. The qualified names of the interfaces do not matter in this approach. So, if the required interface is a subset of the provided interface in terms of method signatures, they are compatible.

However, in the prototype the criteria for interface compatibility as described above was thought to be too strict. After all, in the tool, interfaces are just labeled collections of method signatures, and the algebraic compositions do not attach interfaces together in the traditional manner. As we know, they compute a new set of required and provided methods based on the component(s) used as argument(s) in the expression. If component `A` requires the methods `a, b` and `c` and component `B` provides methods `a` and `b` only, it should still be possible for `A` to use the two methods provided by `B` even though they are not compatible like described above. The approach where this is allowed is reflected in the definitions of the composition operators, and the solution implemented in the prototype. If there are any methods "left over" after a composition, they are simply required by the composite component. This approach emphasizes compatibility on "method-level" instead of "interface-level".

7.2.2 Static vs. dynamic component instances

A question that arose during development was how references to components should be handled in composition expressions. In object oriented programming it is common to define a class and instantiate it by creating objects (instances) of that class. A similar approach could be used in our case, where we create a new, independent "component instance" each time a component is used in a composition. The component instance is only used by the composition expression in which it was created. As an example, in the two expressions $A = B \parallel C$; $D = B \ll E$; the `B` component is used twice in two separate expressions, creating two independent instances. This would make the implementation more complex without adding any real value, because a component model in UML is a static model. Independent instances would effectively lead to duplications of components and a harder to maintain model. In our implementation, there exists only one instance of the component which is referred to by every composition expression using it.

7.2.3 The problem of setting component context

Recall from section 2.4.1 that components may be nested. Another way to express this is that components may be added to a *component context*. The context is a kind of scope, which is the environment of the component, and we say that if component `B` is nested inside `A`, `A` is the context of `B`. The context of a component depends on what kind of component it is. During parsing of the input, each atomic component,

whether it is a service or a process component, gets added as a top-level classifier. This means that it is attached to the global scope, and visible to every other classifier in the model. Components which are part of compositions, however, will be added to the context of the resulting component. For example, in the composition $K = A \parallel B \parallel C$, the "anonymous" composite component which is a result of $B \parallel C$ will be added to `K`'s context. This is fine because this composite component is only used by `K` and is not needed elsewhere. We do not take into consideration that $B \parallel C$ might be part of other compositions as well.

The problem arises when we consider the context of atomic components that are used in compositions. Following the idea described above, a component that is a part of a composition should be added to the composite component's context. However, a component can only be a part of one context at a time, i.e. it can only belong to one component. In the above example, if atomic component `A` was assigned to the context of `K`, it would *move* `A` from the top-level classifiers to the context of `K`. If `A` is used in another composition the user would be presented with an error saying that no component with that identifier exists.

Three solutions to this problem were considered. In the first solution, the notion of context would be dropped completely. This would make the model cluttered since every component would end up as a top-level classifier. This in turn would make the hierarchical view of the model flat. The second approach was to keep every "original" atomic component as a top-level classifier and create a new version of the atomic components each time they were used in a composition, effectively duplicating each component when it was needed (see section 7.2.2). This duplicate component would be added to the correct context. However, one of the core ideas is that we want only one of each atomic component in the model, which will be referred to by the sub-components in a composition. In the end, editing an atomic component, e.g. adding a method to its provided interface, should affect every composite component that depends on this component (see section 8.3 about future work). Duplication of components will make this harder to implement because of the extra bookkeeping details. The third option, which is the one we went for in the implementation, is to divide the components into two sets, one for root components and one for internal components. The root set contains every top-level component, both atomic and composite. The components in this set will not change context. The internal components are part of a composition and will be assigned to a context. In the example above where $K = A \parallel B \parallel C$, the component $B \parallel C$ will be added to `K`'s context, but `A` will not because it is in the root set. This is shown in the figure 7.1.

7.2.4 Handling interfaces

We have emphasized that an interface in rCOSPN is nothing more than a collection of methods signatures and optional method bodies. This means that two interfaces which contain exactly the same method

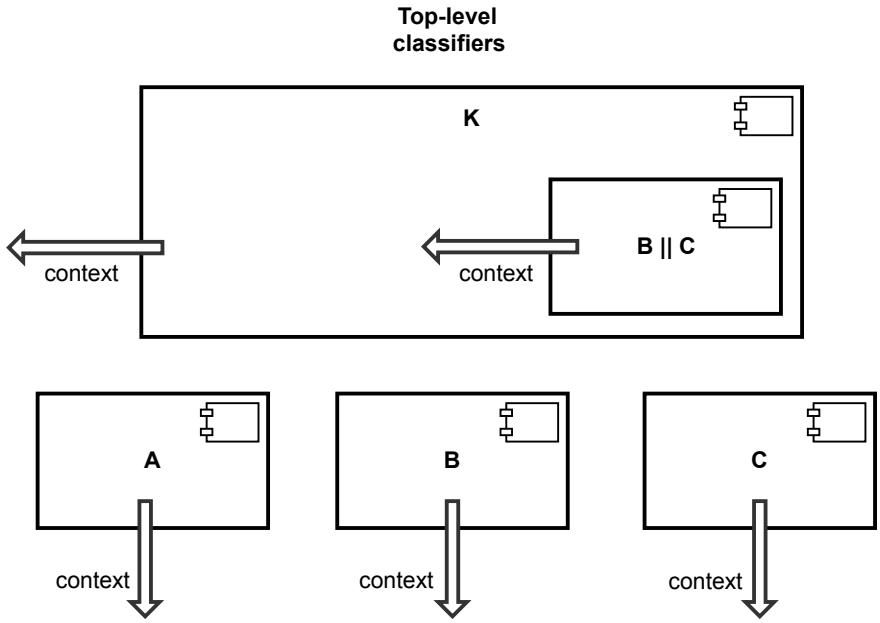


Figure 7.1: Component context

signatures can be viewed as equal, even if their names are different. When constructing the UML model, there are two ways to handle the interfaces; either discard them or translate them to UML interfaces. Discarding means that the only usage of an interface is in the textual specification as a convenient way of grouping related methods to reduce code duplication. During the UML model building, the interfaces will not persist and no UML interfaces will be created. The methods declared in an interface are copied over to each component. This could work with provided interfaces, if we assume that a component provides every method that it owns. In UML, we can inspect what methods a component provides by looking up its owned operations. Showing what a component requires is another matter and we discovered that it is harder to express without an UML interface. Copying the required operations to the component is not a solution, since it would make it impossible to differentiate between the provided and required methods and making us lose vital information about the model. We needed something in the implementation to keep the provided and required methods apart and a simple way of knowing what methods are provided and required. As we have seen, we went for creating UML interfaces, because it is a convenient way of bookkeeping method details and grouping methods. Another solution could be to keep simple lists of e.g. operation names in each component, one for provided and one for required.

7.2.5 No simple assignment

A simple assignment on the form `A = B;` is not allowed in rCOSPN. We felt that the use cases of such expressions would be rare and only apply in cases where the modeler wants to duplicate a component and give it a new name. This would have no direct effect on the model semantics. The expression can also be viewed as an empty renaming or hiding which would be an erroneous operation with no semantic meaning.

7.2.6 Renaming only applies to provided methods

Recall that the renaming of method names in a component only applies to the provided methods of the component. However, the definition of renaming proposed in [Don+13] says that renaming applies to both provided and required methods. In principle this would not cause any problems, but we felt that being able to rename required methods was a bit redundant. The purpose of renaming is to make components compatible for composition, and the renaming of provided methods only is enough to accomplish that. Another incentive to not allow renaming of required methods is that a potential error when renaming both provided and required could lead to the same name on a provided and required method, leading to a valid model becoming a non-valid model.

7.3 Tool limitations

7.3.1 Common variables

The component definitions for the binary operators state that it is required for the component operands to have disjoint sets of variables. This is important to ensure in order to avoid incorrect models. As an example, imagine the component specification given in listing 7.1. Both components contains the integer `i`, and provide methods which use this variable.

Listing 7.1: Common variable

```
1 component A {
2     int i;
3
4     /* provided methods which does something to 'i' */
5 }
6
7 component B {
8     int i;
9
10    /* provided methods which does something to 'i' */
11 }
12
13 //type-checking error: both contain 'i'
14 AB = A || B;
```

The type-checker will throw an error upon parsing the parallel composition expression saying that it is not possible to compose two components which contain the same variable. After all, these two components might do different things to `i`, for example by storing an important value. How should common variable(s) potentially be handled in the tool? One solution is to compose these two together as normal and create a component where only one of the `i`'s persists. This could potentially lead to a component where the methods from `A` will overwrite the value written by `B`, and vice versa, which is clearly not a good behaviour. Another solution is to place both `i`'s in the new component. However, this would cause a name clash in the component. So far this restriction seems good in order to avoid composite components like the one above.

The biggest impediment in this restriction is that it makes composing two components of the same kind a hard task, if they contain the same variable(s). Common provided methods can be remedied with a renaming expression, but renaming does not apply to variables. For example, imagine composing two memory cell components to create a two-place memory cell component. In the current version of the tool, there is no other way around this than simply defining these components *without* any variables.

7.3.2 Showing explicit method "attachment"

In a UML model created by the tool, there is no explicit way of telling which methods are "plugged" into each other in a binary composition. The model only reflects which components are used in a composition, and if one component provides something the other requires, these required methods are not carried over to the composite component. In many cases this is fine, because we want to automatically derive new components from existing ones without thinking of what methods went where. On the other side, in the cases where we do want to know, we have to manually inspect the UML model to discover the joining of required and provided methods.

7.4 Alternatives to UML profiles

In this thesis we looked into how the UML could support algebraic composition with the help of a UML profile. The main reasons for choosing a profile were that a profile is a lightweight, convenient and easy way to extend the UML metamodel with UML stereotypes which defines customized modeling elements by extending existing ones, and that we could utilize the rCOSP UML profile. In addition to using a profile, there are two other alternatives to consider. The first is to not extend the UML at all and stick to using plain components. The second is to create our own UML metamodel.

7.4.1 Using plain UML components

We have emphasized that the UML metamodel alone does not suffice to model the algebraic composition of components, since it does not contain the specialized modeling capabilities needed for this particular domain. Using plain UML could however work to some extent. After all, we still use the basic building blocks such as UML components and interfaces. As an example, imagine that we have a complete rCOS UML model with atomic components and algebraic compositions. If we take this model and *remove* the rCOS profile application, we would be left with a UML model with plain UML components and interfaces. This model could still be of some use and could be drawn in a diagram. However, it would lack the information associated with the rCOS and the algebraic compositions. For example, we can no longer distinguish the parallel compositions from the plugging compositions since the stereotypes are no longer present.

7.4.2 Creating a new metamodel

The field of metamodeling is beyond the scope of this thesis, but creating a new metamodel for the algebraic composition could be an alternative to using a profile. Although UML profiles are suitable for modeling the domain of algebraic composition, they are limited in the sense that they have to reference an existing metamodel. Simply put, a metamodel defines the concepts of the language [Fow03], for example it defines the concepts of classes, interfaces and components. The model elements we use in our ordinary models are instances of the concepts defined in the metamodel [WK03].

Building a new metamodel for our prototype would probably require more work than defining a profile. It would require us to create our own versions of modeling elements like interfaces, classes and components which are already available in the UML metamodel maintained by OMG. Using rCOS as a foundation would no longer be of use, since it uses the UML metamodel. The domain of algebraic composition is likely not big and specialized enough that creating an entirely new metamodel would be a huge improvement over using profiles. Besides, it would be harder to interchange models because it would require that the recipient had acquired the metamodel. This is in contrast to UML which is an industry standard for modeling.

7.5 Two worlds of component composition

We can view the algebraic composition and traditional composition as two different worlds of component compositions. Recall that by traditional composition we mean the normal hierarchical composition by the attachment of provided and required interfaces of components, without necessarily creating a new component in the process. Although it might be hard to directly compare the two composition paradigms

in several ways, we will in this section look at and discuss the major differences.

7.5.1 The nature of compositions

The algebraic paradigm is flexible and do not restrain composition when it comes to the structure of the component(s). The only requirement is that the compositions are valid according to the operator definitions. For example, composing two components which only provide functionality is a valid composition as long as the component arguments do not overlap in any way, e.g. by providing some of the same methods. The result is just a component which provides the union of the provided methods of arguments. In a more traditional component model, a composition like this would perhaps not result in anything being done since there is no attaching of a provided and a required interface involved.

One of the main differences between the two paradigms is the outcome of a composition. In algebraic composition, the result of a composition expression is always a new component regardless of the operator used. This is not the case for hierarchical composition, where we just attach interfaces to each other and the composition will not necessarily produce a new component in the process.

7.5.2 Component independence

In the prototype, the attributes and methods of the components used as arguments are *copied* over to the resulting component. The effect is that every component, no matter the type, is independent in the sense that it contains every attribute and method itself. The benefit is that we can remove the components used as arguments from the model, including the references to them, until only the composite components are left. The component is still fully functional since it is self-contained.

We can mimic the result of getting a new component as a result of composition in the traditional approach if we define a component with one or more local components. This component might *delegate* the provided functionality of the sub-components. In this case, the composite component serves as a wrapper for the sub-components, although it might appear for the outside environment that the composite component provides this functionality by itself. Removing the sub-components will affect the composite component, since it is not self-contained, and might cause the model to be wrong.

This difference is illustrated in figure 7.2, where the first diagram shows two atomic components, `A` and `B`, and their parallel composition, `C`. We see that the algebraic approach provides the methods `foo()` and `bar()` by itself, since they are copied to `C` from `A` and `B`. Recall that the UML dependencies are only relevant for the diagramming (the dotted arrows). The second diagram in the figure shows the approach where `C` is a wrapper for `A` and `B`. The interfaces of `A` and `B` are *delegated* through `C`, but the methods are still *a part of* `A` and `B`.

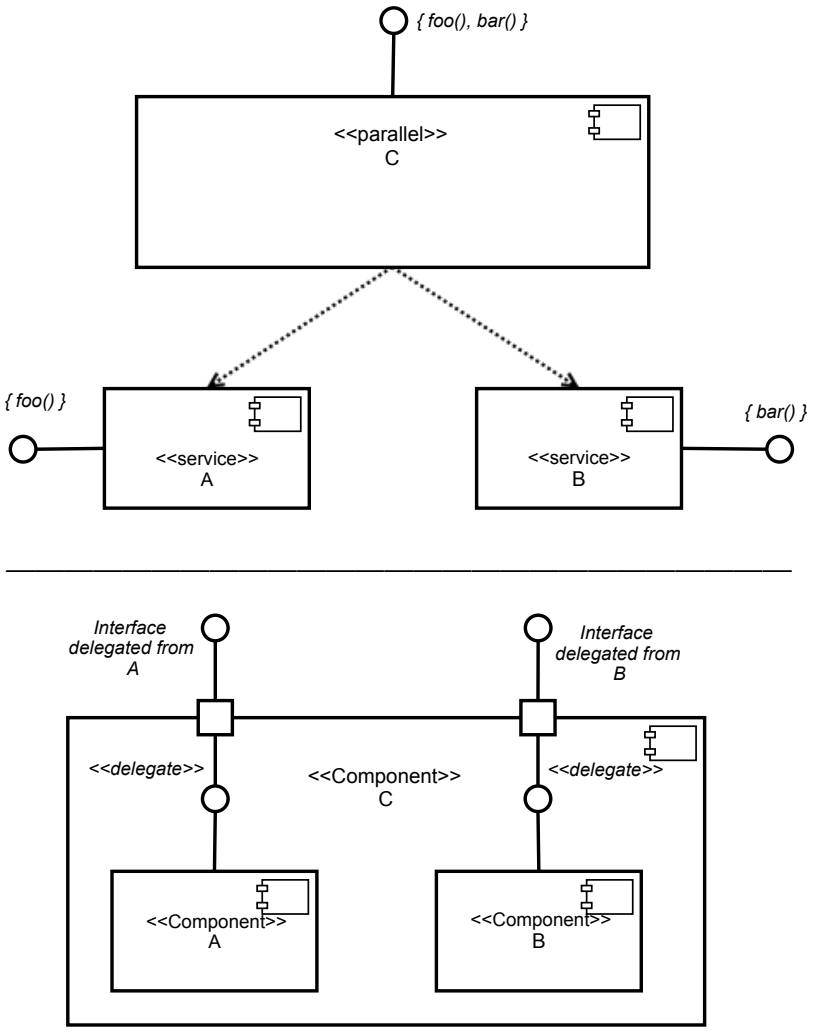


Figure 7.2: The algebraic (above) and traditional (below) approach

7.5.3 Explicit and implicit composition

Recall from section 3.3 that the composition in rCOSP is explicit, i.e. we have to explicitly specify the attaching of the provided and required contracts of components like in the following expression: composition A | B [(A.fooCtr <– B.fooCtr)]. The advantage is that we have complete control and knowledge of what goes where, at the cost of some manual work of writing out the compositions. After a quick glance at an rCOSP specification, it might be easier to understand the compositions and how the model is going to look like afterwards, in contrast to the algebraic composition in rCOSPN where the process of carrying out the compositions is automated and maybe not intuitive at first sight. This negatively impacts the goal of readability, which we earlier stated was one of the primary goals of rCOSPN.

However, the positive attribute is that the process of composition

is carried out automatically. The model developer or designer knows the provided and required set of methods of the atomic components, and simply specifies the compositions of these in an expression. The system will take care of figuring out the details of composition and the "attachment" of methods and building the new component based on the operator used in the expression.

7.6 Algebraic composition in "real-world" component-based modeling

The idea of doing component-based modeling in the algebraic approach is based on specifying atomic components and use these as building blocks to create composite components which again can used as bigger building blocks. As we have seen in this chapter, the approach offers several benefits when doing component-based modeling. For instance, the compositions are automatically deriving new information, there are less restrictions on component compositions and the need for interface compatibility is removed. As it stands now, however, most developers and designers probably do not want anything to do with the semantic UML model we create in the prototype. Graphical diagrams are more important and are more accessible to use than editing the semantic models. In section 8.3 we will elaborate on this topic.

7.7 Summary

This chapter presented discussions of several aspects of the algebraic composition approach and the prototype. Many design decisions affected the implementation and result of the prototype, which unfortunately is not without limitations. Using a UML profile proved to be a decent solution in order to make the UML support component-based modeling in the algebraic manner, with the alternatives being to use plain UML or create a new metamodel for the algebraic domain. Section 7.5 provided a comparison between the algebraic approach and the "traditional" approach. As we could see, these two approaches are different in several ways, such as the automatic and explicit composition.

Chapter 8

Conclusion

8.1 Introduction

In this chapter we will give a summary of the thesis and conclude our work. We will also look at possible topics for future work.

8.2 Summary and Results

The first main goal of this master's thesis was to investigate how the Unified Modeling Language, extended with a UML profile, could be used to support and represent models of textual algebraic component specifications. The problem was that the UML metamodel does not have the modeling elements which satisfies the information we need to express when modeling in the algebraic approach, so we had to create them ourself. A UML profile is a package that is used to provide an extension to a referenced metamodel, which in our case is the UML metamodel. One of the benefits of using a UML profile is that it is a very convenient way of creating model elements which are tailored towards the modeling domain. The entities that extend the existing model elements are UML stereotypes, which are contained inside a profile. Another huge benefit was that we were able to build on the existing rCOS UML profile. This allowed us to reuse modeling elements from, and build on, the rCOS tool. Alternatives to using a UML profile were to stick to using "plain" UML or creating a specific metamodel. With the former we would still have modeling capabilities to some extent, but without the additional information associated with the algebraic compositions. The latter option could be a good solution. However, since we already use many of the modeling elements in the current UML Metamodel, creating a new metamodel would lead to building many of the model elements which are readily available in the UML Metamodel. We would also not be able to use the existing rCOS tool. The end result is that the UML profile provided a nice and simple way of making UML support algebraic component compositions.

A part of this thesis was to develop a prototype tool for algebraic component composition in the UML. This prototype takes a textual

specification written in the rCOSPN language as input, and creates the UML model from this specification. rCOSPN is a language with a clear and readable format which emphasizes algebraic component composition expressions. Chapter 4 presented the rCOSPN language, and chapter 5 covered the UML profile and the model construction. The tool also was able to type-check the algebraic composition expressions and validate the models. This was covered in chapter 6.

The second main goal was to discover what the benefits of the algebraic component composition approach in component-based modeling are. As discussed in chapter 7, the algebraic composition approach is quite different from the more traditional hierarchical composition in a number of different ways. The biggest strengths of the algebraic composition we implemented in the prototype is that it emphasizes the specification of small, reusable components which can be used as a base for creating several different kinds of composite components. The result of an algebraic composition expression is a new component where its properties are derived automatically based on the expression arguments. There is no need to explicitly state the details of composition, i.e. the attaching of a provided and a required interface. This is done automatically by the tool. The components are independent because they own their properties. Also, there is no need for two components to be compatible interface wise when doing a composition, since the joining of provided and required methods are done on method-level and not on interface-level.

8.3 Future work

In this thesis we have created a prototype and foundation for creating UML models of component-based systems in an algebraic manner. This foundation is a good starting point to further investigate how we can support the UML with these kinds of models in the realm of graphical diagrams (diagramming).

8.3.1 Extending the tool to UML diagramming

In this thesis we emphasize a workflow where we write a textual component specification in rCOSPN, run it through the tool and inspect the semantic UML model to do further editing and model validation. Tools such as TOPCASED contain functionality for drawing UML diagrams, and we can use these tools to draw a diagram of the model. Although a UML component diagram of an rCOS model drawn by TOPCASED shows the classifiers with the stereotypes applied, it does not contain diagramming capabilities that are specific to rCOS components, such as displaying which components are the left and right operands for a composite component. In a diagramming tool for rCOS components, we need to display this kind of information. A possible extension to the prototype is to create such diagramming capabilities in

the UML.

On the semantic UML model level, the editing of components and their compositions are still done manually before being validated. Besides creating a textual specification, editing the model graphically by "click-and-drag" components into the model is an approach which can speed up the process. However, in the graphical environment we do not have access to the composition operators. It can be cumbersome to have to edit the textual specification every time we want to create new atomic components and composition expressions, so having access to the operators graphically as well would be an advantage for efficient modeling.

Furthermore, the graphical part of the tool should make a distinction between the atomic components and the composite components, and put down a set of restrictions when it comes to the graphical editing of these classifiers. A suggestion is to make the composite components read-only since they are computed as a result of component composition and are defined in terms of other components. In this way, the editing of a composite component would have to go through an atomic component. The editing of an atomic component would imply that every composite component which is derived from this atomic would be changed as well. An additional functionality could be to let a composite component be tagged as writeable, and in the process make it independent from the component(s) it was derived from.

8.3.2 Behavioural modeling

A possible direction to investigate could be to look into behavioural modeling. The prototype supports method bodies in the textual specification, however this support is only transferred from rCOSP, which we know handles composition differently. For example, method calls are no longer prefixed by the contract name. The work could begin by adapting the syntax of method bodies to the algebraic approach, and implementing functionality for transforming models into behavioural models.

8.3.3 Code generation

It could also be possible to look into generating executable code, e.g. Java, from an rCOSPN specification. The rCOS tool can generate Java code from an rCOSP specification, but this has to be adapted to handle algebraic components and compositions.

Appendix A

Full airport examples and UML model

A.1 Complete airport specification in rCOSP

```
/* Gate */
interface IGate {
    public loadPassengers(int numPassengers);
    public unloadPassengers();
}

contract GateCtr of IGate {
    public loadPassengers(int numPassengers)
    {
       /* [pre : numPassengers > 0, post : true] */
        [numPassengers > 0 |- true ] /* Syntactic sugaring */
    }

    protocol{ loadPassengers ; unloadPassengers }
}

class Flight {
  int numPassengers;
  public Flight() {}
}

class GateClass {

  Flight flight;

  public GateClass() {
    flight := Flight.new()
  }

  public loadPassengers(int numPassengers;) {
    flight.numPassengers := numPassengers
  }

  public unloadPassengers() {
    flight.numPassengers := 0
  }
```

```
}

component Gate1 {
    provided GateCtr by GateClass;
}

component Gate2 {
    provided GateCtr by GateClass;
}

/* Customer Service */
interface IOffice {
    public provideEmployee(;string employee);
}

contract OfficeCtr of IOffice {}

interface ICustomerCentre {

    public customerSupport(string inquiry; string support);
}

contract CustomerCentreCtr of ICustomerCentre {
    public customerSupport(string inquiry; string support) {
        [inquiry != null |- support' != null]
    }
}

interface ICustomerService {
    public handleCustomer();
}

contract CustomerServiceCtr of ICustomerService {
}

component CustomerService {

    /* Composition of Office and CustomerCentre */
    composition : Office | CustomerCentre
                [ (CustomerCentre.OfficeCtr <- Office.OfficeCtr
                    ) ]

    provided CustomerServiceCtr by CustomerServiceClass;
    provided CustomerCentreCtr by CustomerCentre.
        CustomerCentreCtr;

    class CustomerServiceClass {
        public handleCustomer() {
        }
    }

    component Office {
        provided OfficeCtr by OfficeClass;

        class OfficeClass {
            public provideEmployee(;string employee) {
                /*Poor Bob gets assigned to all customer
                    inquiries. */
                employee := "Bob"
```

```
            }
        }
    }

    component CustomerCentre {
        provided CustomerCentreCtr by CustomerCentreClass;
        required OfficeCtr;

        class CustomerCentreClass {
            public customerSupport(string inquiry; string
                support) {
                 Var string emp;
                 emp := OfficeCtr.provideEmployee();
                 support := "Support provided by " + emp
                /*support := "Support provided"*/
            }
        }
    }
}

/* Renaming */
contract Gate1Ctr of IGate extends GateCtr {}
contract Gate2Ctr of IGate extends GateCtr {}

/* Airport */
component Airport {
    composition :  Gate1 | Gate2 | CustomerService
                  [
                   (Airport.CustomerServiceCtr <-
                       CustomerService.CustomerServiceCtr)
                   (Airport.Gate1Ctr <- Gate1.GateCtr)
                   (Airport.Gate2Ctr <- Gate2.GateCtr)
                  ]

    /* Gates. One required contract for each gate needed*/
    required Gate1Ctr;
    required Gate2Ctr;
    required CustomerServiceCtr;
}
```

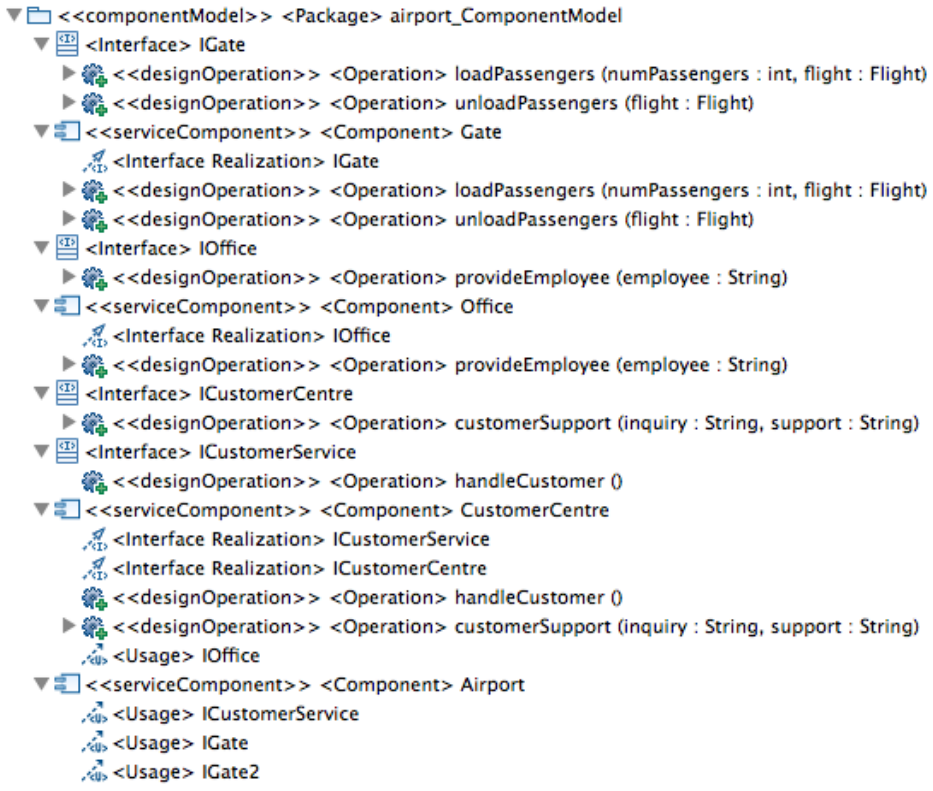
A.2 Complete airport specification in rCOSPN

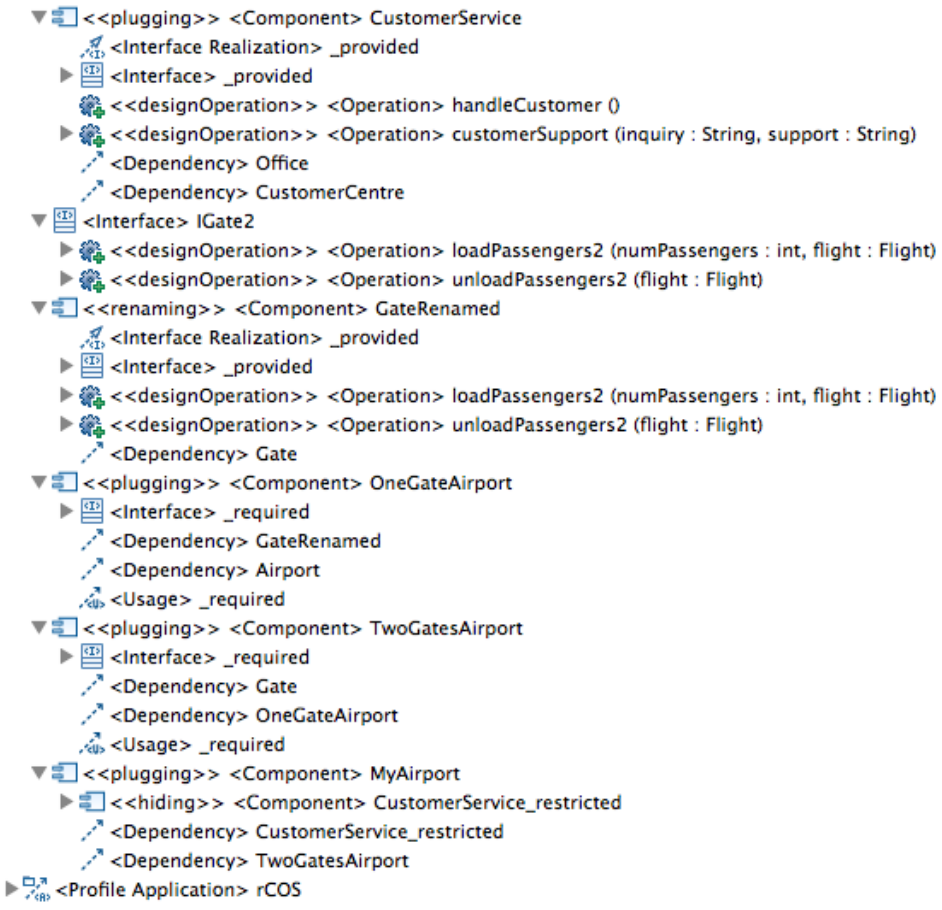
```
interface IGate {
    public loadPassengers(int numPassengers, Flight flight);
    public unloadPassengers(Flight flight);
}

component Gate {
    provided IGate {
        public loadPassengers(int numPassengers, Flight flight)
             {
            [numPassengers > 0 |- true ];
            flight.numPassengers := numPassengers
        }
```

```
        public unloadPassengers(Flight flight) {
            flight.numPassengers := 0
        }
        protocol { loadPassengers ; unloadPassengers }
    }
}

class Flight {
    int numPassengers;
    public Flight() {
    }
}

/* Office */
interface IOffice {
    public provideEmployee(;string employee);
}

component Office {
    provided IOffice {
        public provideEmployee(;string employee) {
            /*Poor Bob gets assigned to all customer inquiries.
                 */
            employee := "Bob"
        }
    }
}

/* Customer Service */

interface ICustomerCentre {
    public customerSupport(string inquiry; string support);
}

interface ICustomerService {
    public handleCustomer();
}

component CustomerCentre {

    provided ICustomerService {
        public handleCustomer() {
        }
    }

    provided ICustomerCentre {
        public customerSupport(string inquiry; string support)
            {
            [inquiry != null |- support != null];
            Var string emp;
            emp := provideEmployee(); //provided by IOffice
            support := "Support provided by " + emp
             /*support := "Support provided"*/
        }
    }
    required IOffice;
}

CustomerService = Office << CustomerCentre;
```

```
component Airport {
    required ICustomerService;
    required IGate[2];
}

GateRenamed = Gate[unloadPassengers <- unloadPassengers2,
    loadPassengers <- loadPassengers2];
OneGateAirport = GateRenamed << Airport;
TwoGatesAirport = Gate << OneGateAirport;
MyAirport = CustomerService \ {customerSupport} <<
    TwoGatesAirport ;
```

A.3 Complete airport UML model




<<plugging>> <Component> CustomerService
<Interface Realization> _provided
<Interface> _provided
<<designOperation>> <Operation> handleCustomer ()
<<designOperation>> <Operation> customerSupport (inquiry : String, support : String)
<Dependency> Office
<Dependency> CustomerCentre
<Interface> IGate2
<<designOperation>> <Operation> loadPassengers2 (numPassengers : int, flight : Flight)
<<designOperation>> <Operation> unloadPassengers2 (flight : Flight)
<<renaming>> <Component> GateRenamed
<Interface Realization> _provided
<Interface> _provided
<<designOperation>> <Operation> loadPassengers2 (numPassengers : int, flight : Flight)
<<designOperation>> <Operation> unloadPassengers2 (flight : Flight)
<Dependency> Gate
<<plugging>> <Component> OneGateAirport
<Interface> _required
<Dependency> GateRenamed
<Dependency> Airport
<Usage> _required
<<plugging>> <Component> TwoGatesAirport
<Interface> _required
<Dependency> Gate
<Dependency> OneGateAirport
<Usage> _required
<<plugging>> <Component> MyAirport
<<hiding>> <Component> CustomerService_restricted
<Dependency> CustomerService_restricted
<Dependency> TwoGatesAirport
<Profile Application> rCOS

Bibliography

[BE13] P. Bakker and B. Ertman. *Building Modular Cloud Apps with OSGi*. O'Reilly Media, 2013.

[CD00] John Cheesman and John Daniels. *UML Components: A Simple Process for Specifying Component-based Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000.

[Che+07] Zhenbang Chen et al. 'Modelling with Relational Calculus of Object and Component Systems - RCOS.' In: *CoCoME*. Ed. by Andreas Rausch et al. Vol. 5153. Lecture Notes in Computer Science. Springer, 2007, pp. 116–145. URL: http://dblp.uni-trier.de/db/conf/dagstuhl/cocome2007.html#ChenHHKLLLNORSYZ07.

[Crn+11] Ivica Crnkovic et al. 'A Classification Framework for Software Component Models'. In: *IEEE Transaction of Software Engineering* 37.5 (Oct. 2011), pp. 593–615. URL: http://www.mrtc.mdh.se/index.php?choice=publications&id=2139.

[Don+13] Ruzhen Dong et al. 'rCOS: Defining Meanings of Component-Based Software Architectures'. In: *Unifying Theories of Programming and Formal Engineering Methods*. Ed. by Zhiming Liu, Jim Woodcock and Huibiao Zhu. Vol. 8050. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 1–66. URL: http://dx.doi.org/10.1007/978-3-642-39721-9_1.

[Far13] Patrick Farail. *TOPCASED 2 hour presentation*. 2013. URL: http://www.topcased.org/ (visited on 19/03/2014).

[Fow03] Martin Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. 3rd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

[Gro06] Object Management Group. *CORBA Component Model Specification, OMG Available Specification, Version 4.0*. http://www.omg.org/spec/CCM/4.0/. 2006.

[Gro11] Object Management Group. *OMG Unified Modeling LanguageTM (OMG UML), Superstructure. Version 2.4.1*. http://www.omg.org/spec/UML/2.4.1/. 2011.

[Gro13] The Object Management Group. *CORBA basics*. 2013. URL: http://www.omg.org/gettingstarted/corbafaq.htm (visited on 16/02/2014).

[Gro14] The rCOS Group. *An rCOS walk-through*. Feb. 2014. URL: http://rcos.iist.unu.edu/index.php/rcos-modeler/tutorial.

[Lou97] Kenneth C. Louden. *Compiler Construction: Principles and Practice*. Boston, MA, USA: PWS Publishing Co., 1997.

[Mit02] J.C. Mitchell. *Concepts in Programming Languages*. Cambridge University Press, 2002.

[OSV11] Martin Odersky, Lex Spoon and Bill Venners. *Programming in Scala: A Comprehensive Step-by-Step Guide, 2Nd Edition*. 2nd. USA: Artima Incorporation, 2011.

[Par07] Terence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Programmers. Pragmatic Bookshelf, May 2007.

[Som06] Ian Sommerville. *Software Engineering: (Update) (8th Edition) (International Computer Science)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006.

[Ste+09] David Steinberg et al. *EMF: Eclipse Modeling Framework 2.0*. 2nd. Addison-Wesley Professional, 2009.

[Szy02] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. 2nd. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[The12] The OSGi Alliance. *OSGi Core Release 5 Specification*. http://www.osgi.org/Specifications. 2012.

[WK03] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. 2nd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

[WSO00] Nanbor Wang, Douglas C Schmidt and Carlos O'Ryan. *Overview of the CORBA Component Model*. 2000. URL: http://www.cs.wustl.edu/~schmidt/PDF/CBSE.pdf (visited on 16/02/2014).