JRIALTO

AN IMPLEMENTATION OF THE HETEROGENEOUS RIALTO MODELLING LANGUAGE

Andreas Dahlin

Master's Thesis on Embedded Systems
Supervisor: Johan Lilius
Embedded Systems Laboratory
Faculty of Technology
Åbo Akademi University
March 2007

ABSTRACT

The importance of software in embedded products is continuously increasing. Programmable processors are nowadays integrated into every System-on-Chip (SoC), which provides developers with new possibilities and also challenges. New and improved design theories, methods and tools for designing and verifying these embedded systems are necessary in order to benefit from the rapidly improving fabrication processes for the hardware used in the systems. This Master's thesis presents an implementation of the second version of the Rialto language and describes the translation of models specified in the Unified Modeling Language to Rialto. Rialto is a small heterogeneous kernel language, with a sound mathematical definition, that can be used for expressing computations in several models of computation. Our implementation, called JRialto, is being developed in the Embedded Systems Laboratory at Åbo Akademi University. JRialto can be used for interpretation and simulation of Rialto 2.0 programs. A case study of a JPEG encoder modelled in UML 2.0 and simulated using JRialto will be presented.

Keywords: Models of Computation, Rialto Language, Heterogeneous Embedded Systems, Interpreter, UML, Simulation

Sammandrag

Introduktion

Under de senaste 20 åren har inbyggda datorsystem blivit allt mer utbredda i samhället i takt med att deras popularitet har ökat. Tidigare förknippades inbyggda system mest med bilarnas låsningsfria bromsar, flygplanselektronik och kontrollsystem för produktionsanläggningar medan man numera kan påträffa inbyggda system inom många olika områden. De nya användningsområdena har inneburit att vanliga konsumenter har blivit en ny och viktig målgrupp. Konsumentmarknaden är kostnadskänslig och ställer även annorlunda krav vad beträffar utveckling och underhåll.

Utveckling av inbyggda system innebär alltid en avvägning mellan vilka delar av systemet som skall utföras med hjälp av effektiva men dyra hårdvarukretsar och vilken funktionalitet som med fördel kan skötas av mjukvara. Under de senaste årtiondena har hårdvarukretsarna utvecklats avsevärt, medan de metoder som används för att planera och integrera mjukvaran i systemen är i stort sett oförändrade. Nya metoder för att beskriva inbyggda system är nödvändiga för att effektivt dra nytta av förbättrade hårdvarukretsar. I synnerhet finns ett behov av språk som kan beskriva så kallade heterogena system, dvs. system som kännetecknas av att de beskrivs av flera olika beräkningsmodeller. En mobiltelefon är ett bra exempel på ett heterogent system. Rialto är ett av de språk som klarar av att beskriva heterogena system. Språket kan även användas som ett mellanspråk vid kodgenerering från modeller skapade i diverse grafiska modelleringsspråk.

Syftet med det här diplomarbetet är att ge en introduktion till Rialto-språket, samt att visa hur UML 2.0-modeller kan översättas till Rialto och även att skapa en implementering av språket. Implementeringen, som heter JRialto, utgör den praktiska delen av diplomarbetet. För att påvisa JRialtos funktionalitet modellerade vi en JPEG-omkodare i Rialto och genomförde en simulering av omkodaren i vårt program.

Rialto

Rialto är ett heterogent språk som utvecklas vid laboratoriet för inbyggda system vid Åbo Akademi. Språket har en matematiskt solid semantik för att representera olika beräkningsmodeller, vilket samtidigt gör det möjligt att använda formella metoder för att verifiera att programmen är korrekta. Rialto använder sig av ändliga tillståndsmaskiner för att tolka och kompilera program. Effektiva algoritmer används för att optimera de ändliga tillståndsmaskinerna.

Dag Björklund presenterade Rialto i sin doktorsavhandling i januari 2005. Doktorsavhandlingen omfattar endast en liten del av den långsiktiga målsättningen för språket, t.ex. har endast sådana modeller som saknar tidsnotation studerats. Hans forskning visar att Rialto kan användas som ett mellanspråk för kodgenerering från såväl UML-modeller som SDF-modeller. Den långsiktiga målsättningen med Rialto är att effektivt kunna beskriva heterogena system som omfattas av flera hierarkiska beräkningsmodeller.

Rialto 2.0

Man har påpekat att vissa brister existerar i den första versionen av Rialto. Därför är en ny version av språket, Rialto 2.0, under utveckling. De grundläggande koncepten är fortfarande de samma, men flera förbättringar och förändringar har gjorts i den nya versionen. Den viktigaste förändringen gäller hanteringen av schemaläggningspolicyn i Rialto. En schemaläggningspolicy är Rialtos sätt att beskriva och kapsla in en beräkningsmodell. I Rialto 2.0 är det möjligt, till skillnad från i Rialto 1.0, att lägga till nya policyn utan att modifiera språkets implementering. An annan väsentlig skillnad är också att den underliggande beräkningsmodellen för Rialto har ändrats från att vara en entrådig maskin som driver programkörningen till att nu utgöras av en procedur i två steg.

Schemaläggningspolicyn

Som ovan nämnt, är schemaläggningspolicyn en mekanism för att kapsla in olika beräkningsmodeller och lösa de schemaläggningsbeslut som beräkningsmodellerna ger upphov till. I en situation då t.ex. två samtidiga påståenden kan utföras, är det inte påståendena själva som hanterar parallellismen mellan dem; den uppgiften tar en schemaläggningspolicy hand om. I Rialto 2.0 kan en schemaläggningspolicy skrivas direkt i språket, vilket medför att alla syntaktiska element som finns i Rialto kan användas även i en policy. Nackdelen är att programhögen blir relativt komplex att hantera. För närvarande har ett fåtal policyn för sekventiell, interfolierad samt stegbaserad körning implementerats.

Syntax och semantik

Rialto har en syntax och semantik som har påverkats av diverse specifikationsspråk. En del gemensamma koncept, såsom tillstånd och avbrott, förekommer i de flesta specifikationsspråk. Tillstånd spelar en mycket central roll i Rialto eftersom de kan användas för att kapsla in delsystem som använder sig av olika beräkningsmodeller. Rialto har en formell semantik som definierar ett programs exakta betydelse och därmed gör det möjligt att bevisa de egenskaper som ett program har. Den formella semantiken ger samtidigt ett regelverk för kompilering av program till olika lågnivåspråk.

JRialto – en implementering av Rialto

JRialto är en javabaserad implementering av Rialto som har utvecklats som en del av diplomarbetet. JRialto kan användas för att tolka, simulera och avlusa program skrivna i Rialto 2.0. Observera att nuvarande version av JRialto enbart är avsedd för att användas för forskning och utveckling av Rialto-språket. Av den orsaken har tolkningshastigheten inte prioriterats, inte heller har kodgenererings- och optimeringsmodulerna inkluderats i programmet.

Den metod för mjukvarudesign som har använts under utvecklingen av JRialto kan bäst beskrivas som en kombination av Agile-metoden och den vattenfallsbaserade metoden. Stor vikt har fästs vid att göra implementeringen så modularised som möjligt, eftersom det är viktigt att enkelt kunna byta ut och återanvända moduler under programmets hela livslängd. En fördel med programvaruutveckling i Java är att modulariseringen underlättas genom att man kan gruppera källkoden i paket.

I varje programmerings- och modelleringsspråk är det viktigt att ha möjligheten att representera data på flera olika sätt. Rialto är dock ett sådant språk som enbart behöver inkludera och definiera de mest elementära datatyperna, medan mera avancerade datatyper endast deklareras i Rialto. Den deklarerade datatypen måste då finnas implementerad i det valda målspråket. Elementära datatyper som booleska variabler, flyttal, heltal, etiketter och strängar finns implementerade i JRialto, liksom behållartyper för köer och mängder.

Tolken

Den del av JRialto som tar hand om själva programkörningen kallas tolk. Tolken har som uppgift att tolka de påståenden som finns i programmet i enlighet med språkets semantik. För att underlätta tolkens arbete byggs ett abstrakt syntaxträd först upp från programmets källkod med hjälp av lexikalanalys och språkparsning. Tolken i JRialto har, förutom tolkningen, även som uppgift att hantera sådana externa händelser som kan påverka tolkningen av programmet.

Kärnan i tolken utgörs av den så kallade huvudslingan. Huvudslingan är en oändlig slinga som ansvarar för uppdateringen av programräknaren och utför de påståenden som finns i programmet. Programräknaren innehåller en referens till det följande påståendet som skall utföras. Att schemaläggningspolicyna skall påverka programkörningen medför att det inte är helt trivialt att uppdatera programräknarens värde. I det fall att programräknaren har antagit ett specialvärde schemaläggs en policy för körning; policyn avgör utifrån programhögens innehåll vilket påstående som skall utföras. Rialto har med andra ord en komplicerad programhög som hanterar både körningen av schemaläggningspolicyn och vanliga påståenden.

JRialto kan inte enbart användas till att tolka program i syfte att undersöka programmets körning. Man kan även utföra mera verklighetstrogna simulationer genom att från programmet anropa specifika funktioner som har implementerats i något annat språk. På det här sättet kan man skapa realistisk utdata från det modellerade systemet. Den metod som används för att komma åt de externa funktionerna bygger på gränssnittet Java Native Interface (JNI). En annan form av kommunikation mellan ett Rialto-program och dess omgivning utgörs av externa händelser. Med extern händelse avses en sådan händelse som kan påverka det system som har modellerats i programmet men som har sitt ursprung i programmets omgivning. De externa händelserna måste definieras i en fil innan simuleringen startas. I filen specificeras ett namn och en tidpunkt för händelsen samt vilken del av det modellerade systemet den berör.

Tolkning av ett steg

Konceptet med steg är väsentligt i Rialto eftersom språket bygger på stegbaserade beräkningar. Vad är då ett steg? Ett steg är ett antal beräkningar som tar systemet från ett observerbart tillstånd till ett annat observerbart tillstånd. Det är viktigt att poängtera att systemet inte befinner sig i observerbara tillstånd under tiden steget utförs. Den schemaläggningspolicy som används för att schemalägga systemet i det aktuella tillståndet definierar även beräkningsstegets kornighet. Med kornighet avses här de enskilda arbetsuppgifterna storlek. Ett steg består av tre egentliga faser: beräkning av stegets innehåll, utförande av delstegen och manipulering av programhögen för att samla ihop systemets nya observerbara tillstånd.

Grafiska användargränssnittet

JRialto kan köras i både konsolläge och fönsterläge. Det senare erbjuder användaren ett lättanvänt grafiskt användargränssnitt. Det kan först te sig onödigt att inkludera ett grafiskt användargränssnitt för en tolk, eftersom programtolkning vanligen utförs från ett kommandoskal. Fördelen med ett grafiskt gränssnitt är att det ger en betydligt klarare överblick över den massiva information som presenteras i avlusningssyfte. Det grafiska användargränssnittet är utvecklat som en löstagbar modul, med andra ord kan modulen exkluderas från en specifik version utan att JRialto blir funktionsodugligt.

JRialtos användargränssnitt baserar sig på ett flertal fönster, vilka visar olika typ av information. I ett fönster presenteras det abstrakta syntaxträdet, medan trädet som innehåller programmets metamodell är tillgängligt i ett annat fönster. Det mest intressanta fönstret är dock det fönster som visar detaljerad information om tolkningen. Där finns information, i tabellform, om vilken policy som använts, värdet på programräknaren och innehållet i högen samt programmiljön. Varje påstående presenteras på två tabellrader; den första raden visar information om systemtillståndet innan tolkningen av påståendet, medan den andra raden berättar om systemtillståndet efter tolkningen av påståendet. För att se hur ett specifikt påstående påverkar systemet kan man alltså enkelt jämföra de två tabellraderna. Samtliga information som finns i tabellen kan exporteras till bl.a. HTML och LaTeX.

Användningen av UML-modeller i Rialto

Istället för att skriva Rialto-programmen manuellt i något textediteringsprogram kan man använda UML-modeller som utgångspunkt. Det faktum att UML-modeller har underliggande beräkningsmodeller utgör inget hinder eftersom beräkningsmodellerna kan beskrivas med en schemaläggningspolicy. UML-modellerna kan översättas automatiskt till Rialto-program, vilket innebär att Rialto kan användas som ett mellanspråk för målspecifik kodgenerering från UML-modeller.

Unified Modeling Language

Unified Modeling Language (UML) är som namnet antyder ett enhetligt modelleringsspråk, vilket kan användas för effektiv design, körning och underhåll av mjukvaruprocesser. UML har en grafisk representation för de flesta modellelement som kan användas för modellering i språket. UML utvecklas av det internationella och ideella konsortiet Object Management Group. Den nuvarande versionen, UML 2.0, innehåller 13 olika diagramtyper som används för att beskriva modellens beteende och växelverkan mellan olika delar av modellen samt modellens statiska struktur. Det är främst de diagram som beskriver beteende och växelverkan som är intressanta ur Rialtos synvinkel.

Översättning av enskilda diagramtyper

De element som de olika diagramtyperna innehåller kan oftast enkelt översättas till ett relativt kort kodblock i Rialto. Riktlinjer för översättningen av de enskilda elementen har utarbetats i det här diplomarbetet. De diagramtyper som beaktas är kommunikationsdiagram, klassdiagram, tillståndsmaskindiagram och aktivitetsdiagram.

Kommunikationsdiagram beskriver kommunikationsflödet mellan systemets olika delar, eller mer specifikt kommunikationsflödet mellan olika objekt i systemet. Kommunikationsdiagram används i Rialto för närvarande till att ge en bild av sambandet mellan olika objekt och samtidigt skapa de kommunikationskanaler som är nödvändiga för att objekten sinsemellan skall kunna utbyta information. I kommunikationsdiagram representeras objekt av så kallade livlinor, vilka enbart har en etikett och inget innehåll. Livlinornas innehåll och beteende kan specificeras med hjälp av andra diagramtyper, t.ex. klassdiagram eller tillståndsmaskindiagram. Klassdiagram beskriver objektens statiska struktur och används därmed i Rialto främst för att deklarera variabler.

Tillståndsmaskindiagram beskriver alltså en del av det modellerade systemet i detalj. Ett tillståndsmaskindiagram består av en eller flera tillståndsmaskiner som beskriver delsystemets olika tillstånd samt övergången från ett tillstånd till ett annat. Tillståndsmaskiner är ytterst lämpliga för att beskriva inbyggda system, då dessa oftast reagerar på stimuli som leder till en förändring av systemets tillstånd. Översättningen av tillståndsmaskindiagram från UML till Rialto är intuitiv, eftersom tillstånd är grundläggande element i båda språken. När systemet befinner sig i ett tillstånd är det möjligt att något arbete eller någon process utförs så länge systemet befinner sig i det tillståndet; det arbetet kan i sin tur beskrivas av en annan tillståndsmaskin eller som en aktivitet.

Aktiviteter beskrivs av aktivitetsdiagram. Aktivitetsdiagram är besläktade med tillståndsmaskindiagram och båda kan beskriva det detaljerade beteendet hos delsystem och effekter. Skillnaden mellan aktiviteter och tillståndsmaskiner ligger i typen av flöde de beskriver. Aktiviteter beskriver arbetsflöden medan tillståndsmaskiner beskriver flöden mellan olika tillstånd. För närvarande är det endast de kontrollrelaterade elementen i aktivitetsdiagram som är relevanta och beaktas vid översättningen till Rialto.

Simulering av externa händelser

Ibland är det användbart att se hur det modellerade systemet reagerar på händelser som inträffar under en simulering av systemet. Händelser syftar i det här fallet på händelser som har sitt ursprung i systemets omgivning men som systemet förväntas reagera på. Sådana externa händelser kan som bekant specificeras i en fil, som sedan beaktas under simuleringen av programmet. Filen behöver inte skapas manuellt, utan kan istället genereras från användningsfalls- och sekvensdiagram. JRialto utgår från användningsfallsdiagram i vilka olika aktörer är associerade med användningsfall. Ett användningsfall specificeras med hjälp av ett sekvensdiagram som innehåller en grafisk representation över de externa händelser som skall beaktas.

Automatisk översättning från UML till Rialto

En komplett systemmodell kan fås genom att kombinera de ovan beskrivna diagramtyperna. Den kompletta modellen kan då översättas som en helhet till Rialto. Översättningen kan göras automatiskt, även om det är en betydligt mer besvärlig process än att enbart översätta diagram enskilt. En modell som omfattar flera olika diagramtyper består oftast av ett kommunikationsdiagram som beskriver systemet på en hög nivå, medan kommunikationsdiagrammets livlinor beskrivs av ett klassdiagram. Klassens beteende beskrivs sedan i detalj av ett aktivitetsdiagram eller ett tillståndsmaskindiagram. Resultatet av översättningen är ett korrekt och direkt körbart Rialto-program som inte kräver någon manuell modifikation.

Fallstudie av en JPEG-omkodare

Diplomarbetet omfattar även en fallstudie. Syftet med fallstudien är att visa hur en JPEG-omkodare kan modelleras i Rialto och simuleras i JRialto. Önskade utdata är en korrekt JPEG-omkodad bild som har skapats från den okomprimerade bitkarta som används som indata. JPEG (Joint Photographic Expert Group) är en standard för bildkompression som utvecklades år 1992 av standardiseringsorganen ISO och ITU-T. JPEG lämpar sig särskilt bra för komprimering av färgfoton, eftersom de förluster i kvalitet som orsakas av JPEG-komprimeringen inte är speciellt märkbara för den typen av bilder. JPEG-komprimering kan utföras på flera olika sätt.

Den variant av JPEG som användes i fallstudien är standard-JPEG (baslinje JPEG). Standard-JPEG-algoritmen valdes eftersom den används på en stor mängd plattformar och samtidigt är lätt att förstå. Omkodaren modellerades som ett UML 2.0-aktivitetsdiagram. Aktivitetsdiagrammet har som uppgift att schemalägga omkodningen och anropa externa funktioner i ett JPEG-bibliotek för att utföra algoritmens delsteg rent konkret. Resultatet av simuleringen är en korrekt JPEG-omkodad bild, vilket både kan verifieras matematiskt och visuellt. Rialto kan därmed anses vara ett lämpligt språk för att modellera den här typen av schemaläggningsproblem, medan JRialto är kapabelt att tolka och ansluta modellen till externa funktionsbibliotek.

Slutsatser

Det här diplomarbetet har beskrivit hur UML-modeller kan översättas till Rialto samt presenterat en första implementering av Rialto 2.0. Rialto 2.0 är en vidareutvecklad version som bygger på samma grundläggande principer som Rialto 1.0. Rialto strävar på lång sikt till att kunna beskriva system som består av flera hierarkiska beräkningsmodeller genom att kapsla in delsystem med en specifik beräkningsmodell i tillstånd. De schemaläggningspolicyn som presenteras i arbetet visar på att Rialto för närvarande klarar av att korrekt beskriva olika beräkningsmodeller. Genom att jag har presenterat hur UML-modeller kan översättas till Rialto, har jag därmed också visat att Rialto kan användas som ett mellanspråk för kodgenerering från visuella modellspråk.

Ett omfattande arbete behövs ännu i framtiden för uppnå de långsiktiga mål har satts upp för Rialto. Ytterligare funktionalitet måste inkluderas i själva språket såväl som i dess implementering. Ett abstraktionslager för schemaläggningspolicyn borde definieras för att underlätta skapandet av nya policyn, vilket för tillfället är mycket arbetskrävande. Abstraktionslagret kunde dölja implementationsspecifika detaljer och förse utvecklaren med ett standardiserat och intuitivt gränssnitt för skapandet av policyn. Vad JRialto beträffar kunde översättningen från UML-modeller skötas av ett externt verktyg eller bibliotek. Slutligen borde även optimerings- och kodgenereringsmodulerna som finns i implementeringen av Rialto 1.0 implementeras.

TABLE OF CONTENTS

Abbreviations

API	Application Programming Interface
AST	Abstract Syntax Tree
DCT	Discrete Cosine Transform
FIFO	First In, First Out
FSM	Finite State Machine
GUI	Graphical User Interface
HTML	HyperText Markup Language
ISO	International Organization for Standardization
ITU	International Telecommunication Union
JFIF	JPEG File Interchange Format
JNI	Java Native Interface
JPEG	Joint Photographic Experts Group
LIFO	Last In, First Out
MCU	Minimum Coded Unit
MDA	Model-Driven Architecture
MOC	Models of Computation
OMG	Object Management Group
PC	Program Counter
RLE	Run-Length Encoding
RTC	Run to Completion
SDF	Synchronous Dataflow
SOS	Structured Operational Semantics
UML	Unified Modeling Language
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
XMI	XML Metadata Interchange
XML	Extensible Markup Language

List of Figures

1. Introduction

Embedded systems have become increasingly popular and wide spread during the last 20 years. From the situation in earlier years, when embedded systems mostly were associated with anti-braking systems for cars, aircraft electronics and plant control systems we have now moved towards a situation where embedded systems can found anywhere. Embedded systems are now used in all kinds of consumer electronics, as well as in smart buildings [13]. In smart buildings embedded systems can be used to increase comfort level, reduce energy consumption and improve safety and security. The new application areas for embedded systems have also implied new requirements on the systems. Modern embedded systems must not only be stable and reliable, they must also be able to interact with other embedded systems.

Because of the market demand for embedded systems to be used in new application areas, the customer base has undergone a significant change. Usual consumers have become an important customer group for embedded systems, which demands for instance new approaches for handling the maintenance of the end product during its entire lifetime. The consumer market is also very cost sensitive, requiring an efficient use of the hardware components available. As a consumer product often has a market window less than a year, during which time the product would have highest sales, the demand for a shorter development time is always present. A delay of only one month can in some cases result in a 20-30 percent loss in profits [3].

When designing an embedded system there is always a trade-off between implementing the desired functionality in hardware or in software. Functionality implemented in hardware is efficient, but unfortunately the functionality cannot be changed to match market demand after the component has been manufactured. This may lead to a situation where expensive hardware components in stock, become useless after the production of the product they were designed for has been stopped. Instead of using specially designed hardware to implement the desired functionality, it is possible to implement the same functionality in software running on more general-purpose hardware. Software has the advantage that it is usually easy to adapt to changing requirements concerning the functionality of the system. The drawback is that software calculations are less efficient. We can conclude, based on the above-mentioned facts that a proper mix between the amount of functionality implemented in software and hardware results in noticeable improvements to the entire embedded system. Unfortunately the design theories, methods and tools used to design, verify and integrate the software components in these complex systems have not developed as much as the hardware components. Hardware components have become faster and more reliable, but still their physical size has been reduced due to more advanced semiconductor fabrication processes. The situation is quite different in the field of design theory, where the lack of new and efficient design methods at the system-level is obvious.

System designers are today forced to design the entire system using one tech-

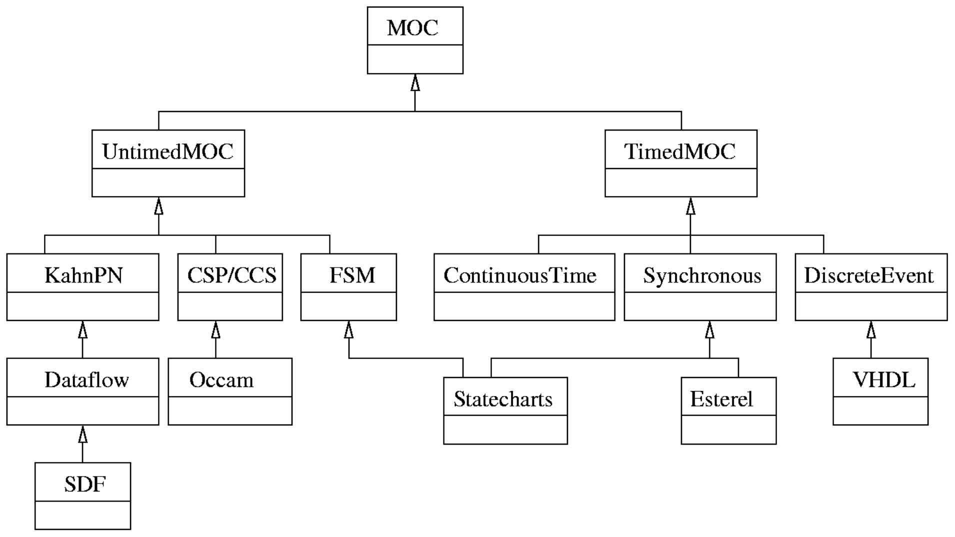


Figure 1.1: Classification of MOCs according to the abstraction of time (from [3])

nique instead of several. One often used technique is to write a functional specification of the system in C++ or another high-level language. Later in the design phase, some parts of the functional specification are decided to be realized in hardware, which means that these parts are described in a hardware description language (HDL) instead of C++. Because the system now is described in more than one language, the comprehensive system model is lost and thereby the ability to evaluate and simulate the entire system becomes very cumbersome. On the one hand, the situation described above is a clear problem to system designers today, especially to a designer without extensive experience and knowledge about certain pitfalls that must be avoided. On the other hand, the fact that modern embedded systems [3] require several different description techniques to describe the system in an efficient manner cannot be ignored. A mobile phone is an example of an embedded system that illustrates this requirement quite well. The mobile phone has a radio-frequency part, which consists of some analogue components; hence, it is best designed using analogue design techniques. The signal processing part is efficiently described by synchronous dataflow (SDF) techniques, while some other part of the mobile phone could be best described by a finite state machine.

All systems are described using some kind of model of the system. A model is a simplification of the real system or even a simplification of another model [2]. The fact that a model can be defined as a simplification of another model is important, since it means that we can have a sequence of models, each simpler than the previous one, that all together describes a complex model. A model of computation (MoC) is a model, which describes the computational aspects of a model. Communication, synchronization and timing of concurrent processes belong to that category of aspects. The domain specific description of a model of computation implies that the MoC decides the abstraction level for concurrency, time and communication. A classification of different models of computation

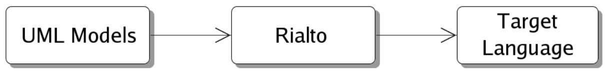


Figure 1.2: Rialto as an intermediate language

according to the abstraction of time is presented in figure 1.1. We can see that there are both models without and models with a notation of time. Esterel, VHDL and Occam are not actually models of computation, but they are categorized according to their underlying computational model.

When several models of computation are used to describe a system, the system is called heterogeneous. A heterogeneous system, where different components in the system are modelled using different models of computation is able to describe complex systems in a more efficient way since each model of computation do only need to solve the problem at its own abstraction level. Unfortunately, it is not easy to support heterogeneity. According to [3], there are essentially two approaches to dealing with heterogeneous systems. One approach is to create a framework, which defines interfaces that can handle the interaction between different existing models of computation and their respective description technique. The second approach, to create a language that can represent all models of computation, is to some extent the aim of the Rialto language.

Rialto is a small heterogeneous kernel language that is being developed in the Embedded Systems Laboratory at Åbo Akademi University. Rialto has sound mathematical semantics for representing different models of computation; hence, it provides a path to formal reasoning and verification. The language cannot only be used as a simulation engine by an implementation of the operational semantics, but it can also be compiled into a target language. Rialto programs can automatically be translated and generated from several visual front-end modelling languages such as the commonly used Unified Modeling Language. In figure 1.2, Rialto is used as an intermediate language between models described in the unified modeling language (UML) and a target language, for instance VHDL. It should be pointed out that Rialto still is under development and therefore it currently should only be used in a research environment. In a long-term perspective, Rialto will hopefully be able to contribute to the design and verification of embedded systems. The shorter development cycles of embedded systems makes it increasingly more important to be able to evaluate the effect of implementing functionality in software or hardware already in early stages of the design process.

1.1 Aim of Thesis

The aim of the thesis is to introduce the second version of the Rialto language, show how UML 2.0 models can be translated to Rialto and present an implementation of the language. The thesis also comprises a practical part. In the practical part an initial implementation of Rialto 2.0 was developed. The development of the implementation, named JRialto, was performed in a two person team consisting of undersigned and thesis worker Markus Dahlgård. My focus in the team has been on the development of the interpreter engine, the stack and the graphical user interface. The support for using scheduling policies in the interpreter and writing the policies in Rialto 2.0 were the most challenging parts. In order to show on the capabilities of JRialto, we modelled a JPEG encoder in UML and simulated the execution of the model in JRialto, producing a valid JPEG image as output.

1.2 Structure of Thesis

This thesis presents the implementation of the second version of the Rialto language and describes some of its uses for design, modelling and simulation.

An introduction to the Rialto language is given in Chapter 2, where also the background of Rialto is discussed. The structured-operational-semantics type semantics and syntax of Rialto is presented here together with the scheduling policies. The scheduling policies allow us to explore different models of computation.

Chapter3, JRialto - A Rialto Implementation, covers the implementation of the second version of the Rialto kernel language. JRialto is java-based implementation featuring interpretation of Rialto programs, code generation from UML diagrams and simulation of external events.

In Chapter 4, the translation from the Unified Modeling Language (UML) to Rialto is discussed in detail. The different diagrams of UML 2.0 represent different models of computation that requires translation to a corresponding Rialto model. An automated implementation of the translation process is presented.

In Chapter 5, a case study using JRialto for simulation is looked at. In order to test the functionality of our Rialto implementation we modelled a JPEG Encoder in UML and performed automated translation of this model into Rialto. The results from the simulation are presented.

Finally, some conclusions are given and future work suggested in Chapter 6.

2. The Rialto Language

In this chapter the Rialto kernel language is presented. The background of the language, including the fundamental ideas, is covered. The differences between the initial version of Rialto and the current version of the language, Rialto 2.0, are discussed. In the final section will give you a presentation of the syntax and semantics used in Rialto 2.0. A complete overview of the allowed syntax is here, as well as more general coding guidelines and short examples.

2.1 Introduction

Rialto is a small heterogeneous language developed in the Embedded Systems Laboratory at Åbo Akademi University. A heterogeneous language is a language that can describe heterogeneous systems; systems that is described using several models of computation [3]. Currently, the heterogeneity is a challenge to the design of embedded systems in general, because it is not possible to obtain comprehensive system models using the existing techniques. The long-term goal of Rialto is to be able to use it for efficiently describing systems that include several hierarchical levels of computational models.

Rialto has a formal semantics, which means that the language is formalized using structured-operational-semantics (SOS) rules. The formal semantics makes verification by formal methods possible and thereby the numbers of errors a designer can commit are reduced. The notion of a model of computation is central in Rialto. Models of computation are encapsulated into so-called scheduling policies, which makes it possible to separates the syntactic elements of the language from its semantic. Another feature of Rialto is the efficient implementation algorithms that make it possible to flatten hierarchical and complex Rialto programs.

Rialto is intended to be used as an intermediate language for code synthesis from visual modelling languages and their respective models of computation. Up to now, the unified modeling language (UML) has been used for visual modelling and the models created in UML have been translated into Rialto. Rialto can also be used for the execution of certain code blocks that have been written in different target languages. In that case, the execution is performed according to the computational model chosen.

The second version of the language, Rialto 2.0, is an evolution of Rialto 1.0. The most important improvement is the generic support for scheduling policies. It is only possible to use a number of predefined policies in Rialto 1.0, while it is possible to express new scheduling policies using Rialto 2.0 syntax.

2.2 Background

The Rialto language was presented in Dag Björklund's dissertation [3] in January 2005. The work he presented should be seen as initial work towards a greater goal. The long-term goal for Rialto is to create a small kernel language for the representation of many different models of computation. The lack of efficient modelling languages that can be used for modelling of heterogeneous systems is an large and well-identified problem in the area of system design for embedded systems. Rialto aims to contribute to the research in embedded systems design and provide a language, which supports efficient modelling and code synthesis from heterogeneous models. The ambitious long-term goals of Rialto cannot fit in a single thesis; therefore, the goals were narrowed down to cover only a slice of the existing categories of models of computation.

Rialto has a sound mathematical semantics; hence, it can be used as a mathematical operational framework. The framework can be extended to represent different models of computation and provide a path to formal reasoning, verification and code synthesis. Rialto can also be seen as an intermediate language between high-level graphical modelling languages and low-level platform specific target languages. Björklund states that his thesis is a study in code synthesis more than a study of models of computations and their combinations. The work on code generation, using Rialto as an intermediate language, shows that the Rialto approach provides a viable path to code generation. Models created in, for instance, the unified modeling language (UML) can be translated to Rialto, where an optimized finite state machine is created. Finally, source code in any selected target language can be synthesized from the state machine.

The optimization approach used for code synthesis in Rialto involves construction of finite state machines. The finite state machines are constructed from Rialto programs using the operation semantics. The finite state machines are reduced using S-GRAPHs. It should be pointed out that the current finite state machine approach assumes that two concurrent state blocks, which are compiled into separate finite state machines, are completely independent of each other. Synchronization between the states must be implemented trough communication mechanisms only. A software graph, S-GRAPH, is a directed acyclic graph used to describe a decision tree with assignments [3]. S-GRAPHs can be reduced by removing redundancies in the graph. Another important property of S-GRAPHs is that they are very suitable for code-size and performance estimation in embedded systems. Some additional optimizations are further applied to the reduced S-GRAPH, since a reduced S-GRAPH is not necessarily an optimal one. The optimized S-GRAPHs can now easily be compiled into different target languages.

The optimization steps makes the generated code less readable, but since the aim is to create a compiler that can generate complete and executable code directly from the model, the reduced readability can not be considered a clear disadvantage. Besides the UML models, synchronous dataflow (SDF) models were studied in Björklund's thesis. The conclusion from the study is that SDF models can be used for code synthesis from Rialto. An advantage is that the generated code does not contain any single appearance schedules, in which some elements only are scheduled for execution once, but on the other hand more control overhead cannot be avoided using the Rialto approach.

2.3 Rialto 2.0

Rialto 2.0 is the second version of the Rialto language. It features several improvements to address the issues pointed out in version 1.0. However, the basic conceptual ideas are still the same. An initial implementation of Rialto 2.0 has been implemented during the year. The implementation is still under continuous development and should therefore only be used for research of the Rialto language. The implementation is described in detail in chapter 3.

Several areas of work and differences between Rialto 1.0 and Rialto 2.0 are described in [10]. The perhaps most important improvement in Rialto 2.0 concerns the scheduling policies. As mentioned earlier in this thesis, scheduling policies are used to encapsulate different models of computation in the Rialto language. The idea of policies is not only used in Rialto, it exists also in the Ptolemy modelling environment and most operating systems. The novelty is the formalization of policies into Rialto, which makes it possible to reason formally about a program and apply formal methods to it. A number of scheduling policies is built into the Rialto 1.0 implementation, but there is no way to add new policies without changing the implementation. In Rialto 2.0, scheduling policies are expressed using the syntax and semantics of the Rialto language. This means that new policies can be created and existing policies can be modified without modifying the Rialto implementation itself. The relatively advanced execution stack defined in Rialto 2.0 is a direct consequence of the new way to express scheduling policies. The stack is still not completely defined in the language and therefore a detailed description of the stack can be found in the implementation chapter.

The underlying model of computation for Rialto has changed between version 1.0 and version 2.0. The computational model in Rialto 1.0 is a single threaded machine that stores the current state of the machine, while a program counter indicates which statement to execute. In the second version of the language, we first use a policy for the selection of a statement to execute. The execution of the selected statement is done according to the structured operational semantics of the statement. The formal semantics of Rialto makes it possible to verify programs. The approach used in [3] was to formalize the language semantics in the B language. The B language was originally developed in the 1980's and it is based on set theory and weakest precondition calculus. The problem with the B approach is that the language has its own model of computation, which makes it hard to define the scheduling policies. Rialto 2.0 will instead be formalized through the theorem-proving environment HOL.

Communication is another relevant aspect in modelling of heterogeneous systems. In order to keep things simple, communication models and timed models has been left out of design in Rialto 1.0. The communication aspects have not been considered yet in Rialto 2.0 either, but they will be considered later in the development process. Architectural mappings are sorely missing in Rialto currently, since the Rialto 1.0 compiler compiles everything into one finite state machine. As it is not desirable in the end to exclude support for architectural mappings, they have to be studied and supported in Rialto at some point of time.

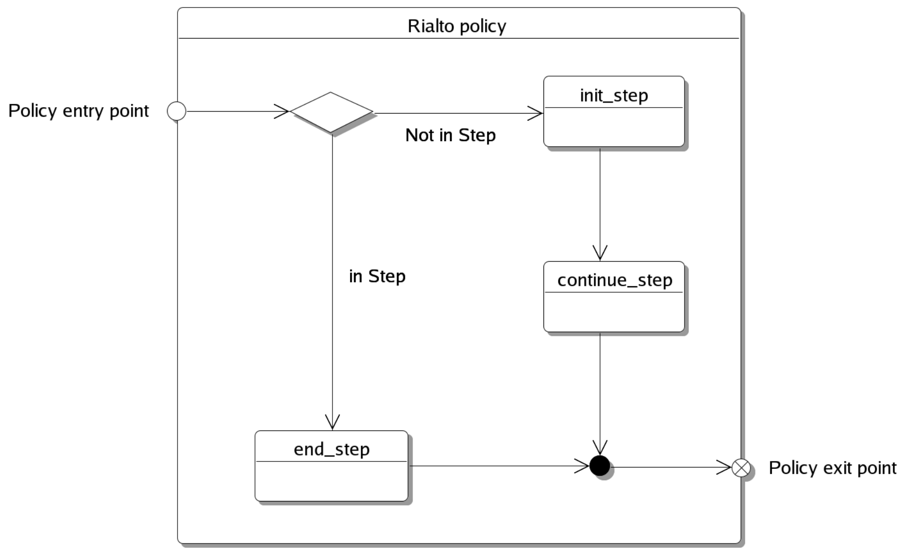


Figure 2.1: A model of a Rialto policy

2.3.1 Scheduling Policies

Scheduling policies are used as a mechanism to encapsulate different models of computation and solve the scheduling decisions they give rise to. For instance, in a situation where concurrent statements or code blocks exist in the program, they are not responsible of handling the parallelism between them. Instead we have scheduling policies, representing different models of computation, on top of the semantics of the statements that schedule the concurrent elements and thus resolves the nondeterminism left by the statements. Each state block can be assigned its own scheduling policy or more precisely an instance of a scheduling policy.

A major improvement in Rialto 2.0 is the support for writing policies in the Rialto language itself. This means that new policies can be added and existing policies can be modified without modifying the Rialto implementation used for interpretation of programs. All syntactic elements available in Rialto can also be used in a policy and since the operational semantics used for policies are the same as in normal programs, policies and normal programs are interpreted in the same manner. A policy can be seen as a subprogram of a Rialto program. Because policies are expressed in the mentioned manner, it is natural to use the same stack structure to represent the state of a policy as well as the state of the entire model. This approach is not completely straightforward to implement; several special functions are needed to distinguish between policy elements and elements in the model. When a policy is executing, labels of statements in the policy are on top of the stack. The policy must therefore modify stack elements below its own stack level to be able to schedule the execution of the Rialto model properly.

Even if a policy can be structured in many different ways, a particular structure is recommended to support the step based execution model. The policy protocol defines three possible states a policy can be in during execution. In figure 2.1, we can see an illustration of the Rialto policy protocol. We can see that there is an initialisation state (init_step), a running state (continue_step) and finally an exit state (end_step). The policy entry point symbolizes a situation where a policy is called upon to schedule the current program state, while the exit point symbolizes that the policy has completed its execution and decided which statement is next to be executed. In the initialisation state, the policy calculates the contents of the execution step and moves on to the running state, where one of the labels that should be executed during the step is selected. The policy returns the selected label, which implies that it is executed immediately after the policy execution is finished. After the selected label has been executed, the policy is called upon again. The difference is that the policy this time only will select the next label from the set of labels that should be executed during the step. Once again the policy returns the label to be executed and then the label is executed before the policy is called upon again. This procedure continues until there are no more labels to be executed in the step. When there are no more labels to execute the policy moves to the exit state, in which the policy manipulates the stack and collects the new state of the system. The policy has now moved the system from one observable state to another observable state.

A few policies have been implemented or taken into consideration in Rialto so far. Below is a short description of these policies. A code listing for the Rialto implementation of each policy (except rtc) can be found in Appendix B.

Default The default policy is used for simple and completely sequential executions. As the name suggests, the policy is used as the default choice of policy for state blocks. Scheduling decisions cannot be made by this policy, implying that it should only be used in situations where only one label exists in the active of the topmost stack element.

Interleaving The policy called interleaving represents a loose execution model, which means that its scheduling choice is undeterministic. The policy is suitable for scheduling concurrency in a loose manner. UML Interactions (found in communication diagrams) are scheduled by the interleaving policy. The lifelines in an interaction can be seen as objects that are running in separate threads and the interleaving policy is responsible to select, on a random basis, an object that should be scheduled for execution.

Step The scheduling policy named step is used when we want to allow the computation to proceed in steps. A statement is executed in each concurrent thread at each step. The step policy is suitable to use in situations where real parallelism should be allowed, regardless of the chosen model of computation. The policy can be seen, to some extent, as a replacement for the interleaving policy. UML Activities are scheduled by this policy in Rialto.

RTC The run-to-completion (RTC) policy is based on an assumption, which states that an event can only be dequeued and dispatched if the processing of the previous event is fully completed. In Rialto, the RTC policy runs until no more active labels exist in its scope. The RTC policy is used as the default policy for scheduling UML state machines. There is no Rialto 2.0 implementation of the RTC policy yet.

SDF The SDF policy implements a policy for handling static dataflow. Dataflow is a natural paradigm that can be used to describe digital signal processing (DSP) applications with concurrent implementations. Although SDF is an abbreviation for synchronous dataflow, its underlying model is not synchronous so it can rather be described as an untimed model of computation [3]. Synchronous dataflow is a special case of dataflow, which requires that the scheduling decisions for the system can be taken already at compile time. The SDF policy currently implemented in Rialto 2.0 can only handle predefined data sequences.

2.3.2 Syntax and Semantics

In this section, the syntax and semantics of Rialto 2.0 are covered. The underlying ideas for the Rialto syntax and semantics are first discussed in general. The syntax and semantics are then covered separately and more in detail. Some short code segments that exemplify the syntax of the language are presented in the end of this section. This section is based on theory from [3] and [10].

Rialto is a small language, originally designed for the description of UML state charts. The concept of states is essential in Rialto as well as in UML state charts. Labels are another important concept in Rialto. Each statement in a program must have a unique label, because the stack and execution of statements are dependent of labels. A label acts as an instruction address to a certain statement. The label can be specified in the Rialto program and if it is omitted a preprocessor generates a unique label for the statement. The syntax and semantics of Rialto has been influenced by other languages used for system specification. By looking at the different languages, a number of concepts that are common to most specification languages have been identified.

States, interrupts, concurrency, atomicity, communication policy and data are all concepts that exists in specification languages. As already mentioned, the notion of a state is central in Rialto. States are represented by a state block and a corresponding end state block. The state blocks can be hierarchical and concurrent. Interrupts (high priority events) that should be reacted upon immediately must be handled in some way in every programming language. In Rialto, the `trap` statement is used to monitor interrupts. Concurrency is a concept that can be interpreted in a number of ways. In our language, concurrent activity is denoted using the `par` statement, but the interpretation of the parallelism depends on the scheduling policy used.

It is often necessary to be able to define the level of atomicity in a programming language. Atomicity defines the smallest observable state change, in other words: several statements or actions that are executed as part of an atomic unit are observed as one single execution. In Rialto, atomicity is made explicit by a syntactic entity, namely the brackets `[ ]`. Communication is another important part of a model of computation. A communication policy states how different parts of a system should communicate with each other. Currently, communication is modelled with channels that for instance can represent the global event queue in a state machine. Handling of data has not been put in focus in Rialto; thus

n	$\in$	**Num**	numerals
a	$\in$	**Alph**	alpha
z	$\in$	**ANum**	alphanumericals
i	$\in$	**Ident**	identifier
x	$\in$	**Var**	variables
b	$\in$	**Bexp**	boolean expressions
e	$\in$	**Aexp**	arithmetic expressions
r	$\in$	**Rexp**	relational expressions
S	$\in$	**Stmt**	statements
T	$\in$	**Types**	data types
op_b	$\in$	**OpB**	boolean operators
op_a	$\in$	**OpA**	arithmetic operators
op_r	$\in$	**OpR**	relational operators
l	$\in$	**Labels**	labels

Stmt	=	{null, if, goto, trap, state, suspend, resume, par, print, return, assign}
Types	=	{Boolean, Label, Integer, String, Float, SetOfBoolean, SetOfLabel, SetOfInteger, SetOfString, FifoQueueOfBoolean, FifoQueueOfLabel, FifoQueueOfInteger, FifoQueueOfString, LifoQueueOfBoolean, LifoQueueOfLabel, LifoQueueOfInteger, LifoQueueOfString, StateConfig, SetOfFloat, FifoQueueOfFloat, LifoQueueOfFloat}
OpB	=	$\{\cap, \cup, \neg\}$
OpA	=	{+, -, $*$, /, %, ^}
OpR	=	$\{\equiv, \neq, <, >, \leq, \geq\}$
Ident	=	(a | _ | $) ($z$ | _ | $)*

Figure 2.2: Rialto Syntactic Categories

far, we have mostly concentrated on control. Only the most necessary primitive data types are currently defined in the language. All the mentioned syntactic elements are given by the grammar in figure 2.3. The defined syntactic categories are presented in figure 2.2.

It is often possible to get an intuitive understanding and meaning of a program by inspecting examples and syntactic elements. However, the exact meaning of a program is defined by its semantics. The assignment of a formal meaning to programs is called program semantics. An advantage of language that has a formal semantics is that it gives us the possibility to prove properties of programs in the language. The formal semantics of Rialto does not only validate the intuitive semantics, but more important, it gives precise rules for compiling Rialto programs into a low-level representation. The low-level representation can then be optimized and translated into a target language. The exact operational semantics of statements is defined in [10].

program	::=	**program** *name*
		decbody
		begin
		body
		end;
decbody	::=	(*externmetdec* | *externvardec* | *vardec* | *owndec* | *policdec*)*
externmetdec	::=	**externmethod** *name*();
externvardec	::=	(*l*:)? **externvariable** *name*: *T*;
vardec	::=	(*l*:)? **var** *name*: *T*;
owndec	::=	(*l*:)? **own** *name*: *T*;
policydec	::=	**policy** *name* *decbody* **begin** *body* **end;**
body	::=	((*l*:)? (*S* | *expr*);)*
nullstmt	::=	**null**
gotostmt	::=	**goto** *l*
ifstmt	::=	**if** *boolexpr* **then**
		body
		else
		body
		endif
parstmt	::=	**par**
		body
		||
		body
		endpar
statestmt	::=	**state**
		policy *name*;
		decbody
		begin
		body
		endstate
trapstmt	::=	**trap** *boolexpr* **do** *S* *body* **endtrap**
atomicstmt	::=	[*body*]
returnstmt	::=	**return** *l*
printstmt	::=	**print** (*n* | *a* | *z*) ((, (*n* | *a* | *z*))*)?
suspendstmt	::=	**suspend** *l*
resumestmt	::=	**resume** *l*
assignstmt	::=	*i* **:=** *expr*
expr	::=	*atomicexpr* | *boolexpr* | *arithexpr* | *relexpr* | *n*
atomicexpr	::=	*i* (((arglist)?))? (.*atomicexpr*)?
arglist	::=	(*i* | *n*) (((*arglist*)?))? ((,*arglist*) | (.*arglist*))?
boolexpr	::=	*expr OpB expr*
relexpr	::=	*expr OpR expr*
arithexpr	::=	*expr OpA expr*
name	::=	*i*

Figure 2.3: The Rialto 2.0 Grammar

The syntax and semantics of Rialto have now been described, but often a programming language is easier to understand from examples than from the theoretical rules. We will now present a few short Rialto program examples that might give a more intuitive understanding of the Rialto syntax. The program below is a very simple Rialto program; it does only write out a text to the console.

```
program myProgram
  begin
    print Hello;
end;
```

The program begins with the mandatory `program` statement, followed by a program name. The only statement used in this program is `print`, which writes the text `Hello` to the console. The following code is an example of a more realistic and useful program containing variable declarations, states and transitions between states.

```
program myProgram
  begin
    s1: state
      own counter    : Integer;
      var tmpCounter : Integer;
      begin
        counter := counter + 1;
        tmpCounter := tmpCounter + 1;
        goto s2;
    endstate;
    s2: state
      begin
        goto s1;
    endstate;
end;
```

The program defines two states, `s1` and `s2`. Two variables are declared in the first state, while we can see that the declaration section is empty in the second state. Both variables are integers, but their scopes are different as specified by the `own` and `var` keywords. The value of the variable `counter` is not affected by state transitions. For instance, if both variables contain the value 2 when a transition to `s2` is performed, and followed by an immediate transition back to state `s1`. The value of the variable `counter` is still 2, while the value of `tmpCounter` has been reset to `0`. Assignment of a value to a variable is also exemplified in the program above; `counter := counter + 1` results in an increment of the value stored in the variable `counter`. Variable types available in the JRialto implementation are described in section 3.5. The `goto` statements at the bottom of each state move the program from one state to the other; hence, this program never terminates. As mentioned several times in the thesis, a new feature in Rialto 2.0 is the possibility to define new policies using the syntax and semantics of the Rialto language.

```
program myProgram

  policy myPolicy
    var l : Label;
    begin
      ...
      return l;
  end;

  begin
    s: state
      policy myPolicy;
      begin
        goto s2;
    endstate;
end;
```

The program above defines and declares the policy `myPolicy` in the declaration section of the program. We can see that a variable is declared inside the policy to illustrate the fact that normal declarations and statements can be used in policies as well. `return` is a statement that only can be used in the body of a policy, because it returns the label to be executed next in the program. The `policy` statement is used to select the defined policy to be the scheduling policy of state `s`. The three programs presented can be used in combination with the complete grammar in figure 2.3 as a starting point for the creation of more advanced Rialto programs.

3. JRialto - A Rialto Implementation

In this chapter, the Rialto implementation developed as a part of this thesis work will be presented. The goal with the implementation, JRialto, was to create an initial implementation of Rialto 2.0. The implementation can be used for executing and debugging both Rialto programs and the Rialto language itself. The overall software design of JRialto will be presented, as well as more detailed descriptions of the interpreter engine, the stack, the data types included, external function calls and the simulation of external events. A guide to JRialto's graphical user interface is available in the last section of the chapter.

3.1 Introduction to JRialto

JRialto is a java-based implementation of the Rialto kernel language. JRialto can be used for interpretation, simulation and debugging of Rialto 2.0 programs on all java-enabled platforms. The development of JRialto has been carried out in the Embedded Systems Laboratory at Åbo Akademi University since April 2006. The development team consisted of two master's thesis workers, undersigned and Markus Dahlgård, who where supervised by professor Johan Lilius.

It is possible to use and invoke JRialto in two different ways. By invoking JRialto in console mode, it is possible to let the Rialto program interpret without any user interaction. The only output shown in the console is the output that origins from certain print statements in the program and the interpretation is only terminated if a termination request is sent from within the model. The other, currently more useful, way to use JRialto is to enable the built-in graphical user interface. The graphical user interface enables the user to terminate the interpretation and inspect the interpretation results in detail, which includes traces from the execution of every single statement. The traces contain information on the value of the scheduling policy used, the program counter and the stack both before and after the execution of a statement. Additionally, the content of the environment at a point of time, including all declared variables and policies, can also be easily viewed and compared to the environment at another point of time.

It is important to point out that the current version of JRialto should only be used for research purposes and further development of the language. As a direct consequence of this, the efficiency in terms of execution speed has not been prioritized. The optimization and code writer modules, which would perform the flattening of the state machine and produce code for a selected target language have not been implemented yet. For the time being, the only way to produce target specific code is to convert the Rialto 2.0 program to a Rialto 1.0 program and execute the program using the Rialto 1.0 implementation.

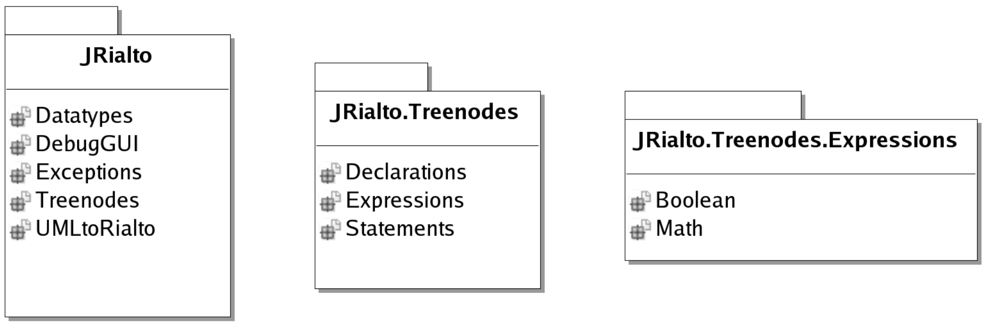


Figure 3.1: Structural overview of the packages

3.2 Software Design Overview

The software design process is often very crucial to the outcome of a software project. The design model chosen will most likely be reflected in the software produced. A great number of well-documented software design models exist and several of them are usually suitable for your specific project from a theoretical point of view. Nevertheless, the choice of an appropriate design model is not that simple in practice. The software method used in the JRialto project is best described as a combination of the Agile software development and the waterfall development model. The Agile software development method attempts to minimize risk by developing the software in short iterations, which last from one to four weeks. The waterfall model describes each of the development iterations in a sequential manner, including everything between the initial specification of requirements to testing and debugging the implemented functionality. It is very common in a long-term research project like Rialto that several different developers will improve the implementation during a number of years. This means that it is important to create a modular and easily extensible software solution, so that a new developer can utilize the existing code base instead of always implementing new features from scratch.

Despite the tight teamwork and instant information exchange within the team, we have tried to divide the development of JRialto into different modules and responsibility areas. The different modules are clearly reflected in the package structure. My focus has been on the implementation of the interpreter loop, including the scheduling policies, the UML translation framework and the graphical user interface. The debugging has been performed by executing small Rialto unit test-files and examining the interpretation results using the graphical user interface in JRialto.

The Java language encourages the developer to structure the software in several major building blocks, which are known as packages in Java. The classification of classes in different packages is not exactly defined; it is up to the developer to decide which classes that should be grouped together in a package. As a rule of thumb, the package structure should be based on the functionality and scope of the classes, i.e. classes that implement similar functionality or that implement functionality to the same part of the software should be grouped in the same package. From figure 3.1, we can see that the high-level packages in JRialto are

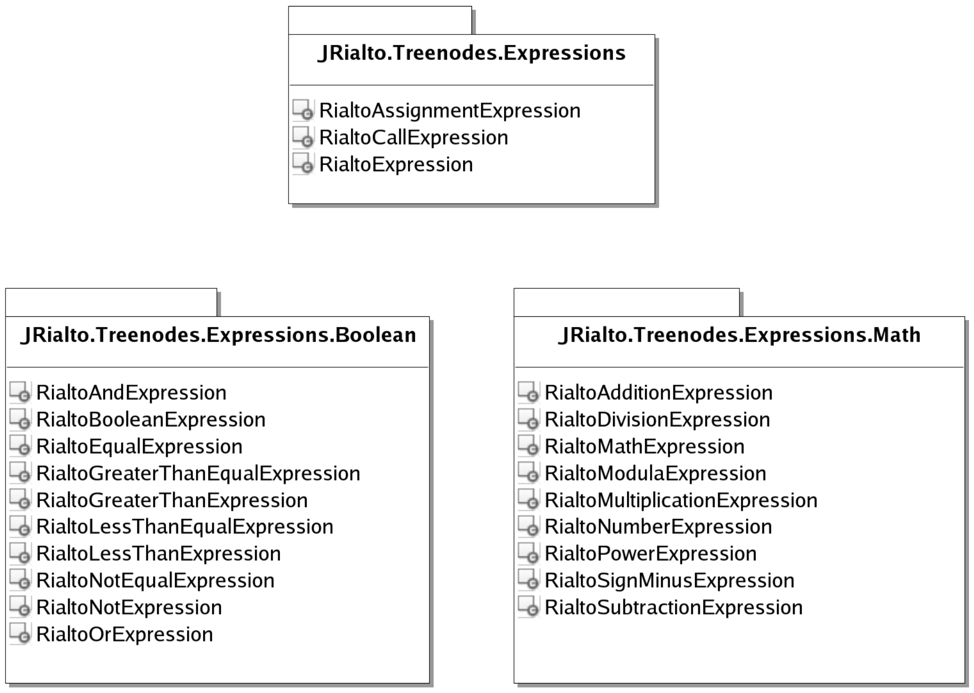


Figure 3.2: The Treenode packages implement the operational rules for executing a Rialto program

Datatypes, DebugGUI, Exceptions, Treenodes and UMLtoRialto. Each data type is implemented by its own class in the Datatypes package, while all exceptions are in the Exceptions package and so on and so forth.

The Treenode package is very essential to the whole implementation, as it contains the classes that implement the operational semantics of the Rialto language. The approach to use a class to represent each grammar rule is known as the *Interpreter pattern*. The Interpreter pattern should be used when the language to be interpreted is quite simple, can be represented by a syntax tree and efficiency is not a critical concern [8], both of these statements hold for the Rialto language. A tree node is a node in the syntax tree, in JRialto the super class *RialtoMetamodelObject* implements the general functionality of a tree node. The interpreter will call upon the functionality by visiting a tree node during interpretation (visitor pattern). In practice, this means that the interpreter executes the method `execute()`, available in all tree nodes, in the visited tree node.

From figure 3.1, we can see that tree nodes have been grouped in three different packages based on the type of grammar rule they implement. The Declarations package includes the classes that implement declarations of native Rialto variables, external variables, scheduling policies and external methods. The second tree node category consists of various expressions that are grouped together in the JRialto.Treenodes.Expressions package, which is further split into the packages JRialto.Treenodes.Expressions.Boolean and JRialto.Treenodes.Expressions.Math. As the package name indicates, the first package contains all expressions that give a boolean as their result while the Math package contains those expressions that return an integer or a float as their result. An overview of all expressions available

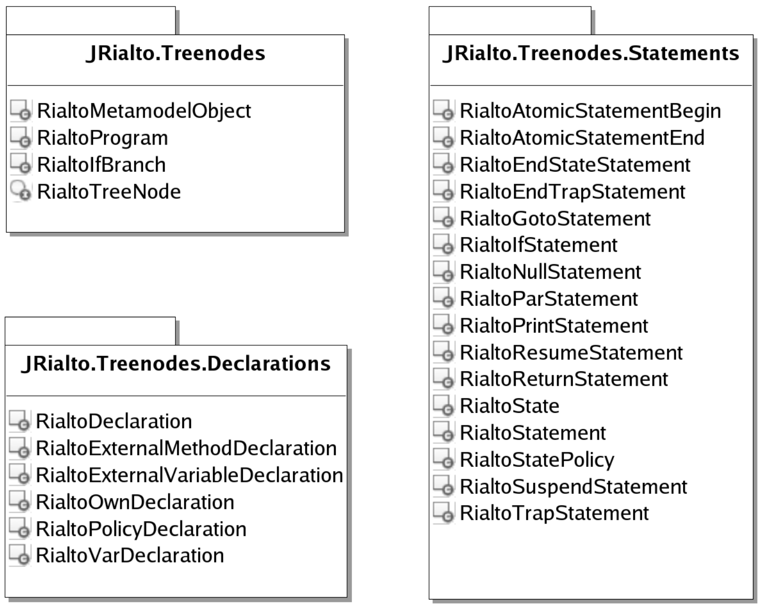


Figure 3.3: Expression packages

in Rialto is available in the package diagram in figure 3.2.

The final tree node type represents the normal statements in Rialto. The statements are listed in figure 3.3. An instance of the class that represents the appropriate statement is created when a statement is found during the creation of the Rialto metamodel tree. Most of the information necessary to include in the metamodel representation of the statement can be gathered already at this stage, however some information cannot be added until the statement is executed during the interpretation. Further implementation details on declarations, expressions and statements are available in [5].

Finally, a short description on the software design of the graphical user interface is provided in this paragraph. The DebugGUI package includes all classes that implements and contributes to the user interface. In fact, the modular design of JRialto would allow an exclusion of the DebugGUI package from a release without breaking the interpreter engine in JRialto. JRialto uses Swing for the implementation of the graphical user interface. Swing is a Java GUI toolkit, which provides sophisticated GUI components that still run the same on all platforms. Each window in JRialto is implemented as a separate class, while another class manages the invocation of the windows. The possibility to export the interpretation table to various file formats relies on an export plugin framework, which makes it easy to create new plugin modules for whatever file format needed.

3.3 Lexical Analyzer and Parser

A program cannot be interpreted directly from the file that contains the source code for a particular language, or at least it would be very inefficient and cumbersome to do so. It is also hard to guarantee that the content of the file conforms to the syntactic rules of the language if the file is interpreted directly. The solution is to initially check that the file contains correct syntax and then create an Abstract Syntax Tree (AST) that represents the file content in a hierarchical tree level. The modules used for this purpose are called lexical analyzer and parser [6].

The lexical analyzer and parser in JRialto are implemented in the ANTLR language. ANTLR is a script language that combines the use of regular expressions with Java native code. The lexical analyzer defines the vocabulary of the language, based on the Rialto language specifications. If the lexical analyzer does not accept the content of the file an error message, including the line number it originated from, is reported to the user and the execution process is terminated. Once the lexical analyzer has accepted the syntax, it will call upon the parser module. The parser extracts the content accepted by the lexer and creates a tree structure according to some defined parser rules. Additionally some simple type checking is performed, duplicated variable names are detected and the destinations of all `goto` statements are validated. In our parser implementation, no information from the original file is excluded from the AST, which means that the original file could be recreated by simply iterating the AST.

The implementation of the lexical analyser and the parser are described more in detail in Dahlgård's thesis [5, Chapter 5]. In some cases, the AST contains all the information necessary for the interpretation, but to be able to interpret a Rialto program correctly more information must be added to the AST.

3.3.1 Metamodel Tree Generator

The metamodel tree generator is responsible for the creation of the Rialto metamodel tree from the AST delivered by the parser. If we recall the fact that each statement in Rialto is implemented as an class in JRialto, a nice way to create the metamodel is to let the metamodel tree generator walk through the AST and call upon a static method in the corresponding statement class when it finds an node (identified by an id) which matches a Rialto statement. This arrangement supports our implementation approach, according to which each Rialto statement should be responsible for the creation and execution of itself.

Additionally, labels are checked for their uniqueness when the metamodel is generated and statements without a user specified label are assigned a unique label. These automatically assigned labels reflect the position of the statement in the Rialto program file. Once the complete metamodel tree is generated, the interpreter can interpret the Rialto program based on the information in the tree.

3.4 Interpreter

In this section, the interpreter part of JRialto is covered. The interpreter is the most important part of JRialto, as it is responsible for the interpretation of Rialto programs. The interpreter does not interpret the rialto code directly; instead, it uses the Rialto metamodel tree as the starting point for the interpretation. The core in the interpreter is the main interpreter loop, which is described more in detail in the next section. Besides the main loop, the interpreter must take care of several other tasks. The helper methods specified in the Rialto 2.0 technical report [10] are implemented as methods in the interpreter. These helper methods implements metamodel tree related functionality needed by the main interpreter loop and some of the statements in Rialto. Details about the helper methods are available in [10]. There is also one helper method that is of great importance in Rialto policies, namely `calculateStep`. `calculateStep` is called upon when a policy wants to receive a partitioned set of active labels, based on the state of the system. The policy can use the partitioned set to execute some or all of the partitions returned from `calculateStep`; the purpose of the policy decides which of the partitions should be executed.

The support for external events is a special feature in JRialto (see section 3.4.4). The loading of the event files and insertion of the event into the modelled system at an appropriate time is also taken care of by the interpreter. During the interpretation, all information about the state of the model, the value of the program counter and environmental information are saved into a data structure. In order to avoid a reduced interpretation speed and increased memory consumption, the information is only stored if JRialto's graphical user interface is enabled.

3.4.1 Main Interpreter Loop

The main loop is the core of the interpreter. The loop is an infinite loop that is responsible for updating the program counter and executing Rialto statements. The program counter, `pc`, is a reference to the label of the rialto statement that should be executed next. In figure 3.4, the main loop is represented as an activity diagram. We can see from the diagram that the infinite main loop can only be interrupted and eventually terminated by the method `interruptInterpreter()`, which is called upon before each iteration of the main loop. The behaviour of `interruptInterpreter()` depends on if the graphical user interface (GUI) is enabled or not; if the GUI is enabled a dialog box will ask the user every thousand iteration if the interpretation should be terminated or continued, while the method never will force termination of the interpretation in console mode. The maximum number of interpretation steps has been restricted to 15000 in GUI mode in order to avoid excessive memory consumption due to the interpretation table statistics that are stored.

If the interpretation is decided to be continued, the next step is to update the program counter. It is not a trivial task to update the program counter in the Rialto language, because the execution order of the statements is affected by

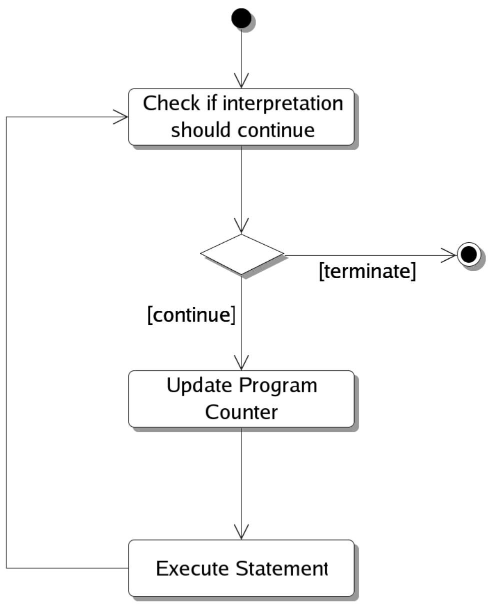


Figure 3.4: JRialto's main interpreter loop

the scheduling policies. The action *Update Program Counter* is illustrated more thorough in figure 3.5. The first decision regarding the program counter is based on the current value of the program counter. If the program counter is assigned the special value $\perp$, several steps must be performed in order to decide the new value of the program counter, otherwise the program counter has already been updated by the previously executed statement and no update is necessary.

The interpreter will now make eventual external events visible to the system. The next step is that the interpreter checks if the system currently is in an atomic unit. An atomic unit consists of several statements included in an Rialto atomic statement. An atomic unit is characterized by the fact that the statement within it should be executed sequentially without asking any policy for the execution order. This means that the program counter is updated so that it refers to the next statement in the metamodel tree. In the normal case when the interpreter is not in an atomic unit, the helper method `policy()` is called upon. This method returns the label to the scheduling policy that is responsible for scheduling the system in the current state. As the program counter now is set to refer to the label of the scheduling policy, the next executed statement will naturally be the first statement in the policy. This illustrates the fact that the interpreter is able to schedule normal execution of statements and execution of statements in a policy using the same routine.

The final remaining action of a main loop iteration is the execution of the statement associated with the label to which the program counter currently is referring. Each statement is responsible for its own execution implementation and when the execution returns the control to the interpreter, the interpreter will begin the next iteration by checking if the interpretation should continue.

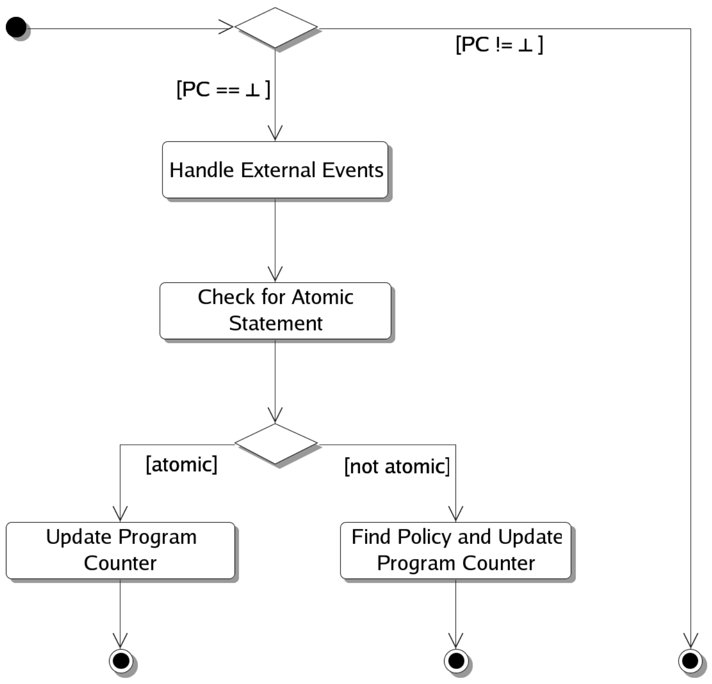


Figure 3.5: A detailed flow chart of Update Progam Counter

3.4.2 The Stack

The interpreter engine in JRialto is tightly related to an execution stack. The stack decides together with the program counter which statement is about to be executed next. The Rialto execution stack is presented in this implementation chapter, because the stack is not completely defined in the Rialto 2.0 language yet. The stack might be further developed in future implementations of Rialto, but the fundamental principles will probably remain unchanged.

The stack is more complex than most execution stacks since it must be able to describe the state of the system completely. The execution of statements in the program as well as in the policies uses the same stack. A stack element consists of an entire state configuration instead of only a single statement label as we can see in figure 3.6. A state configuration consist of an active set and a suspend set. Both sets can contain labels; labels in the active set are eligible for execution,

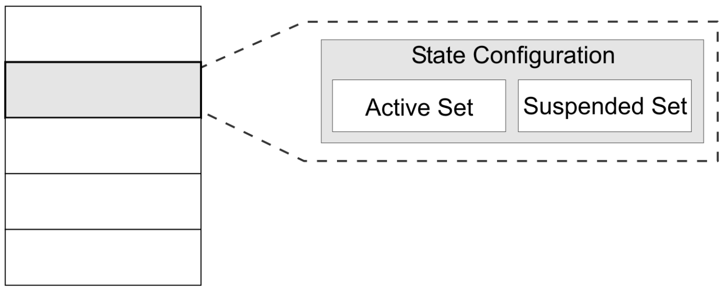


Figure 3.6: Stack implementation in JRialto

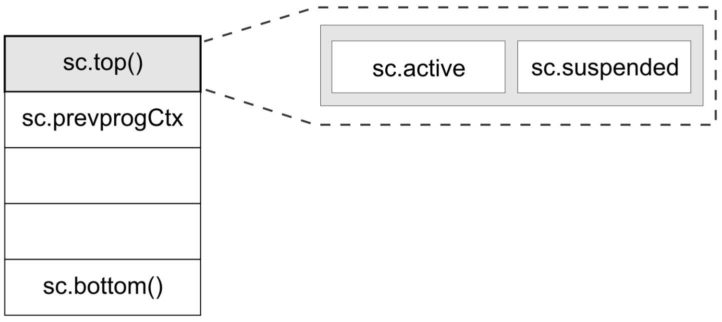


Figure 3.7: The different parts of the stack

while labels in the suspended set are not eligible for execution [10]. To be able to modify the stack in various ways a number of stack operations are available. These operations are used for the implementation of the operational semantics in every Rialto statement, but they are also accessible from Rialto code. The ability to access the stack operations from Rialto code is not important when we want to create a normal Rialto program, but it is crucial when we want to write a scheduling policy.

The execution stack in Rialto is named `sc`. We can see from figure 3.7 that the active and suspended set of the top-most state configuration can be accessed by shortcuts. The shortcut `sc.active` always points to the active set in that particular state configuration, while the `sc.suspend` points to the suspend set. In the case we want to access the entire state configuration instead of one of the sets the operation `sc.top()` can be used. The second element from the top of the stack is frequently accessed from most policies and therefore it can be accessed easily by using the `sc.prevprogCtx` shortcut. It is recommended to access the element by the shortcut instead of performing `sc.pop()`, accessing the element by `sc.active` or `sc.suspended` and finally performing a `sc.push()` to recover the stack.

The following stack operations are available in JRialto:

- bottom()
- pop()
- popFromBottom()
- popFromPrevProgCtx()
- push()
- pushAbovePrevProgCtx()
- pushToBottom()
- top()

3.4.3 External Function Calls

External function calls become necessary when the purpose of using JRialto is not only to examine the execution of the model without producing any real output from the system, but also to create concrete and correct output from the model. External function calls in this context, refers to the execution of functions implemented in some native language on the simulation platform. For instance, public functions in a library belong to that category. In JRialto, calls to external functions are supported through the Java Native Interface (JNI).

The Java Native Interface is a native programming interface specified by Sun Microsystems [22]. JNI can be used when an application cannot be written entirely in Java or when we want to call upon an already existing native procedure. Often a library that supplies you with the needed functionality already has been written in some native language; JNI is then a good method to make the library accessible from Java.

The loading of a native library should be performed in a java class file with a file name similar to the name of the Rialto file that contains the modelled system. If the modelled system is in the file `example.jr`, then the native library loading should be performed in a file named `example.class`. A detailed guide to writing a library loader can be found in [22]. JRialto will automatically search for a java class file named according to the above-mentioned convention. The external functions must also be declared in the Rialto program, before they can be used. The keyword `externmethod` is used for this purpose. An external function `my_function` is declared in the declaration section of the Rialto program by the code `externmethod my_function();`. The method can now be called upon anywhere in the rialto program by the statement `my_function();`. Currently the only functions supported are those that only return void and take no arguments. The case study presented in chapter 5 is an example of a situation where the need for supporting external function calls arises.

3.4.4 External Event Simulation

JRialto includes support for stimuli from external events during the simulations. In this context, an external event is defined as an event that is generated outside the modelled system, but to which the modelled system can react. No support is available for simulation of those events, which are intended to be consumed by an external system but still generated inside the modelled system. The external events must be defined in a file prior to the start of the simulation of the Rialto program. The file must have the same name as the Rialto program name. For instance, the external events that should be used during the simulation of the Rialto program `example.jr` should be defined in the event file `example.ev`. Once an event defined in the file is inserted into a part of the modelled system, it is treated according to the same rules as an internal event. On the implementation level, the insertion of an event to a part of the system is performed by adding the event to a queue defined in the Rialto program. Event files are actually plain text files with a simple notation convention for the representation of external events. Each event is defined in the following notation:

{ *time* , *event* , *queue* **}**;

As Rialto has no real notion of time, the time field here is basically the number of interpretation steps executed by the interpreter. An event can occur each time the interpreter asks a policy to decide which statement is about to be executed next, but not between the execution of statements in an atomic unit. An external event cannot occur between the statements included in an atomic statement. If one or several external events are specified to occur at some point of time when the interpreter is executing an atomic unit, the events will occur to the system after the execution of the atomic unit is completed. The *time* field must be an integer greater or equal to zero. Several events can be specified to use the same time value; in that case, all events will occur at the same interpretation time. The exact order in which the events are inserted into their respective queues is not defined. The *event* field defines the name of the event to be generated, while the last field *queue* specifies into which queue the event should be added. The name of the queue must be equal to the name of a queue defined in the Rialto program. A complete event file can for instance have the following content:

```
{0,    eventStartCounter,   externalEventQueue};
{80,   eventTogglePause,    externalEventQueue};
{120,  eventTogglePause,    externalEventQueue};
```

Instead of writing the event file manually in a text editor, the events can be described graphically in an UML editor by the use of use case diagrams and sequence diagrams. The event information included in the UML model is automatically extracted by JRialto and used for the generation of an event file. The modelling of external events in UML is described in detail in section 4.6.

3.5 Data Types

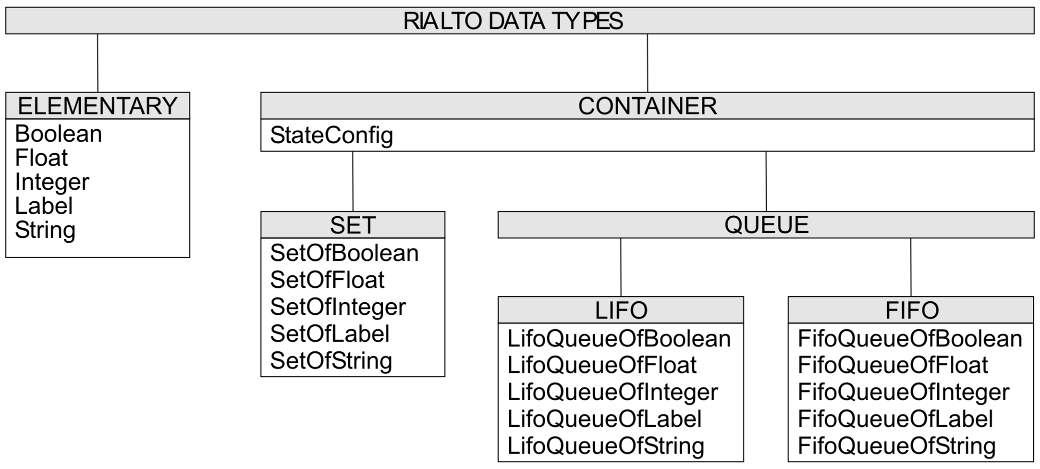


Figure 3.8: Overview of the data types available in JRialto

In every programming and modelling language it is important to be able to represent data in various ways. A large number of data types are necessary to represent different types of data in an efficient way in a concrete programming language, such as C++ or Java. To an intermediate language, as Rialto, the implementation of a complete set of data types is less important. Rialto should only include a few primitive data types like integers, floats and other data types necessary for controlling the flow and execution of the model. Types that are more complex are only declared in Rialto, while their implementation is deferred to a target language [3]. This is the same approach as in the ESTEREL language.

In figure 3.8, the data types implemented in JRialto is presented. We can see that they are structured into different categories depending on their characteristics. There are basic elementary types such as booleans, floats, integers, labels and strings but also container types. The `Label` data type is only used to represent the unique labels that can be associated to each Rialto statement. `String` is another data type that has a special intended use in Rialto. Obviously, it can be used to contain normal textual strings, but this data type is also used for representing events in Rialto 2.0. The container data types are data types that can contain several of the elementary data types. The container data types are divided into sets and queues, which both are necessary in order to be able to write a scheduling policy.

The structure among the data types in figure 3.8 does also reflect how the data types are implemented. Each data type is represented by a class in java. Each class implements a certain interface to ensure that at least a certain set of methods and operations are available to all data types. Some data types do also inherit features from the category they belong to, `SetOfBoolean` is for instance inherited from the super class `Set`. A more detailed presentation of the data type implementation in JRialto is available in Dahlgård's thesis [5]. Variable declarations, variable scope and the general usage of variables in Rialto are covered in the syntax and semantics section in chapter 2.

3.6 Execution of a Step

This section contains a small example that illustrates how a step is executed in JRialto. A step is a number of computations that takes the system from one observable state to another observable state. It is important to observe that the state of the system is not observable during the execution of a step. The granularity of the computation step is decided by the scheduling policy that represents the computational model of the current system state.

The state machine in figure 3.9 is used to illustrate the step based computational model in Rialto. The state machine consists of simple states and orthogonal composite states in different hierarchical levels. The parent state `s` is composed by two sub-states, `sp1` and `sp2` that exist in parallel. The state `s` is scheduled by an instance of the `step` policy and `sp1` is scheduled by another instance of the `step` policy, while an instance of the `interleaving` policy schedules `sp2`. All the simple states (s2, s3, s4, s5, s6 and s7) are scheduled by the completely sequential policy `default`.

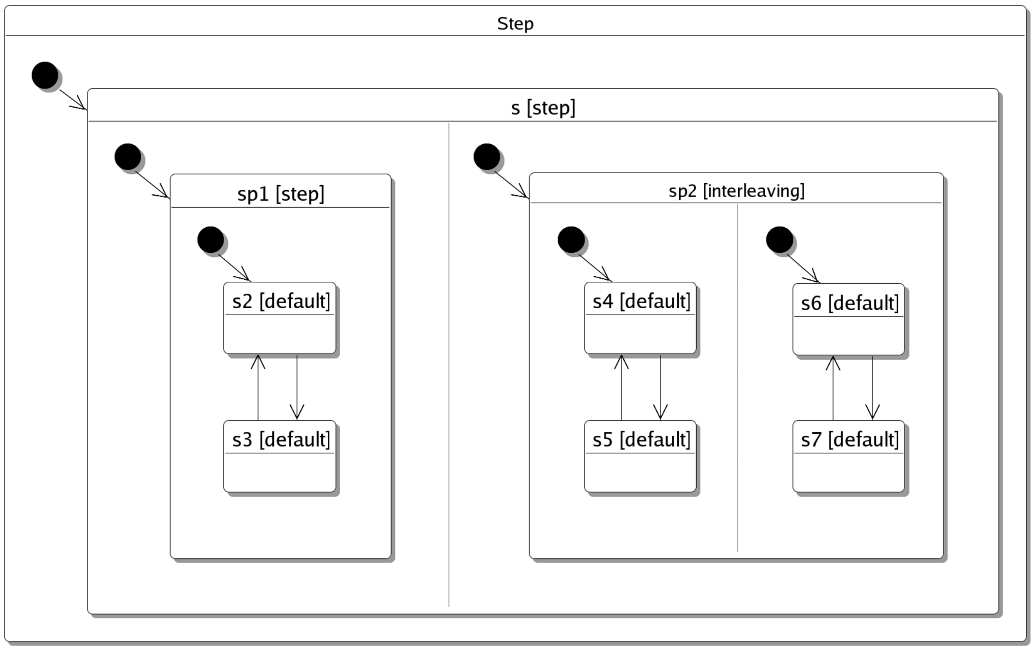


Figure 3.9: A state machine representing parallel execution scheduled by different policies

In the initial state, the state machine will be in the states `s2`, `s4` and `s6`. The policy that each of these states share is the policy associated with `s`, resulting in an execution of that policy. The `step` policy is defined to execute each orthogonal region; in this case both, the region containing `sp1` and the region containing `sp2` should be executed during the step. The policy must be able to arrange the current state `{s2, s4, s6}` in partitions that reflect to which region they belong, in this case the partitions will be `{s2}` and `{s4, s6}`. The execution of `s2` in the first part of the step is straightforward and it will result in a transition to the state `s3`. The next part of the step is to let the policy associated with `{s4, s6}` decide the execution of their region, namely the `interleaving` policy. The `interleaving` policy is defined to randomly execute one of the orthogonal regions of the state it is scheduling. This means that either `s4` or `s6` is executed, but never both during the same step. In the scenario where `s4` is executed the result of the interleaving would move the region into the state `{s5, s6}`, while the other scenario (execution of `s6`) would move the region into the state `{s4, s7}`. The step is completed by collecting the new state of the system. The instance of the `step` policy, which is scheduling `s` is responsible to collect the new observable state of the system. This state is either `{s3, s5, s6}` or `{s3, s4, s7}`. The next step will also be initiated by the policy of `s`, since the new state of the system also has `s` as its parent state.

The detailed stack trace table in figure 3.10 shows the stack content during a single execution step. The table should be read from left to right row by row. As we can see, the step moved the system during this particular execution from the initial state `{s2, s4, s6}` to the state `{s3, s6, s5}`. Code listings of all the three polices used in this example is available in Appendix B.

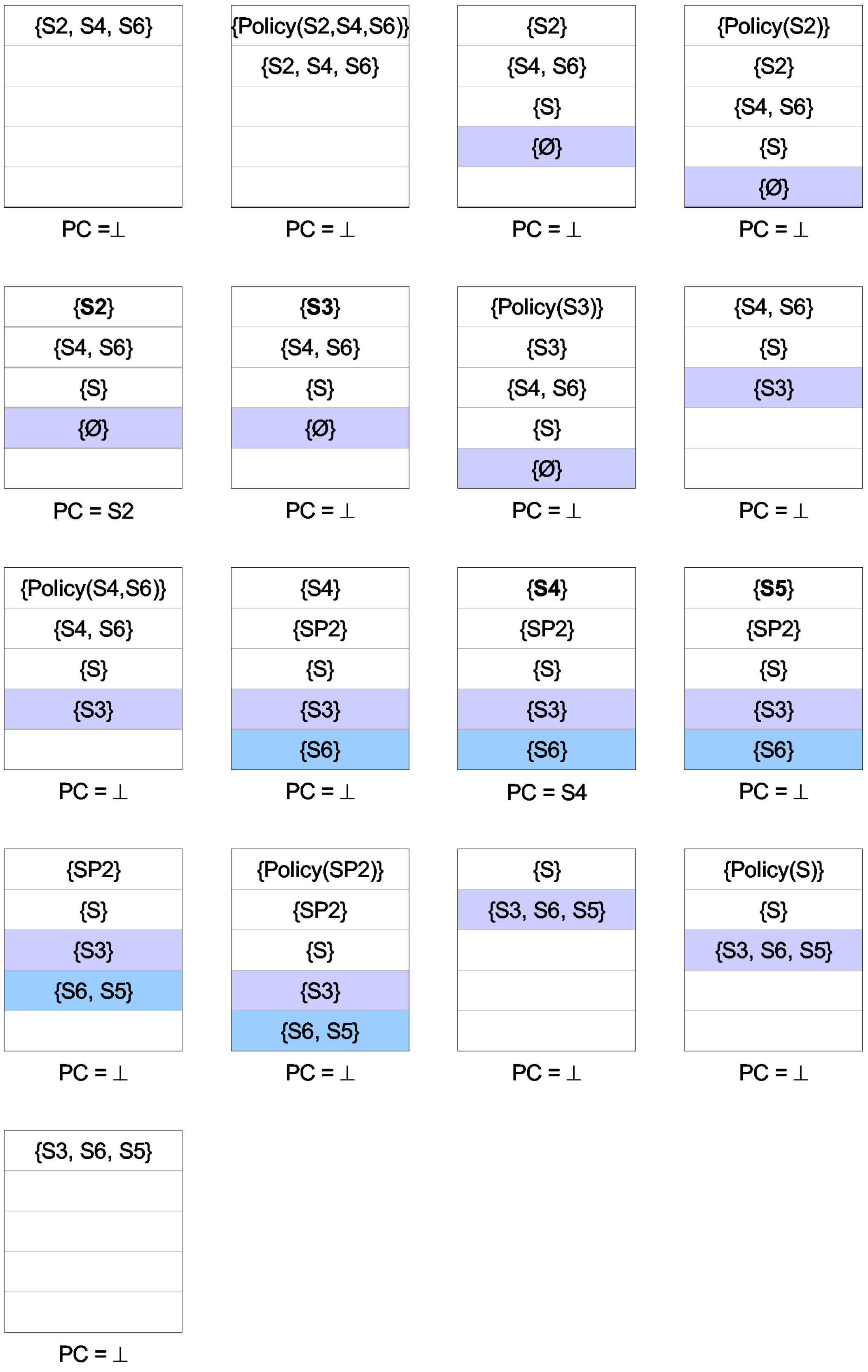


Figure 3.10: Stack trace from the execution of a step

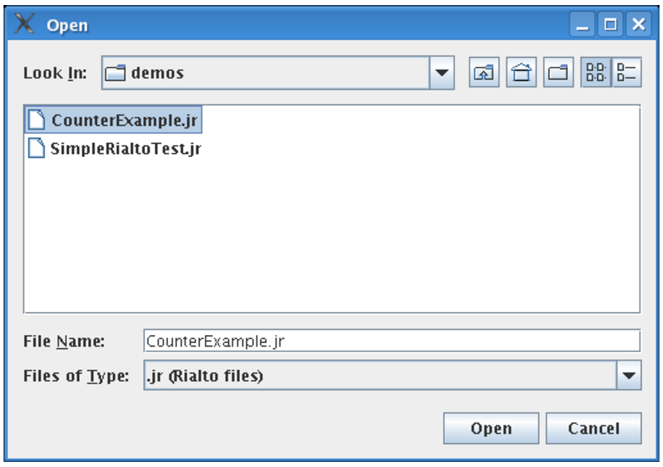


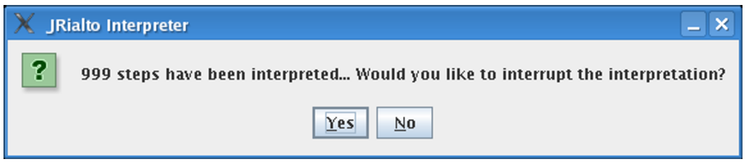
Figure 3.11: The open file dialog box in JRialto

3.7 Graphical User Interface

JRialto is equipped with a graphical user interface (GUI). The reason to include a graphical user interface in a software product like JRialto is perhaps not too obvious at first sight, since interpretation is a task that usually is performed from command line. Concerning JRialto, the graphical user interface has been necessary for the debugging of the interpreter and the Rialto language itself. Sometimes it is also useful to be able to track the interpretation of a program in order to see that the program actually executes as intended. Even if the extensive information available through the graphical user interface would be possible to present on the command line, it would be nearly impossible to present it in a user-friendly manner.

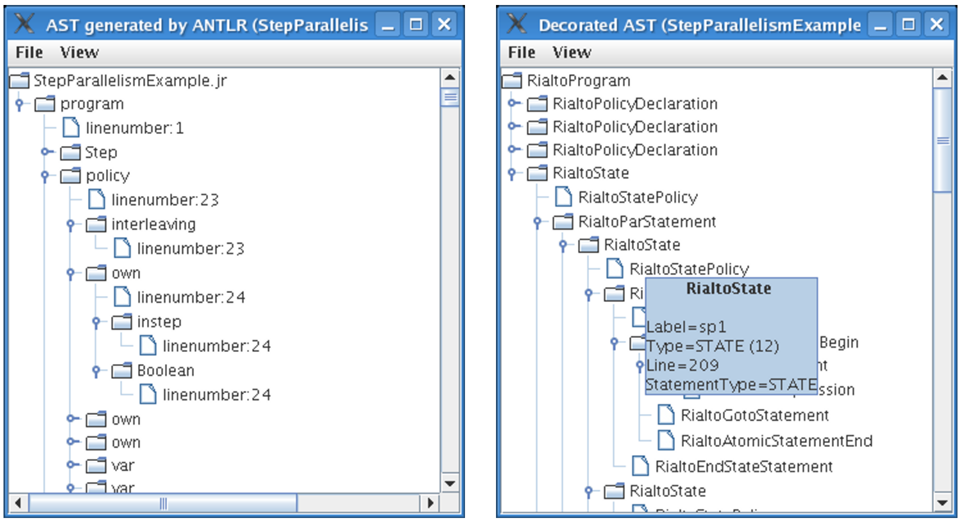
The user interface is developed as a detachable module of JRialto. This means that the GUI can be omitted from a certain JRialto release, without breaking any other feature. The GUI has not been developed with efficiency in focus, which might result in a less responsive interface on slow computers. However, it should also be pointed out that the overhead caused by the GUI during interpretation is only present if the GUI is enabled, otherwise no overhead and reduction in interpretation speed should be noticeable. The graphical user interface is built on the Java Swing GUI toolkit, which provides a powerful user interface that appears uniformly on all java-enabled platforms. So far, the GUI has been tested on machines running the Sun Java Runtime Environment on Linux, Mac and Windows operating systems.

In the rest of this section, a description of the user interface and its features will be given. The GUI consists of several windows, which can be moved and resized as needed. The *open file* dialog box (figure 3.11) is the first window that appears when JRialto is launched with the GUI enabled. It is a completely normal dialog box from which the user can choose a file to be opened, in this case a Rialto program file can be selected (.jr file name extension). If the *cancel* button is pushed, an informational message about the possible usage parameters in Rialto is shown and JRialto is terminated. Naturally, if a file is selected and



999 steps have been interpreted... Would you like to interrupt the interpretation?

Figure 3.12: Interrupt interpretation dialog box


AST generated by ANTLR (StepParallelis
File View
Decorated AST (StepParallelismExample
File View

```
StepParallelismExample.jr
  program
    linenumber:1
    Step
    policy
      linenumber:23
      interleaving
        linenumber:23
      own
        linenumber:24
        instep
          linenumber:24
        Boolean
          linenumber:24
      own
      own
      var
      var
```

```
RialtoProgram
  RialtoPolicyDeclaration
  RialtoPolicyDeclaration
  RialtoPolicyDeclaration
  RialtoState
    RialtoStatePolicy
    RialtoParStatement
      RialtoState
        RialtoStatePolicy
        Ri
                        Begin
                        nt
                        ssion
          RialtoGotoStatement
          RialtoAtomicStatementEnd
        RialtoEndStateStatement
      RialtoState
```

```
RialtoState

Label=sp1
Type=STATE (12)
Line=209
StatementType=STATE
```

a) b)

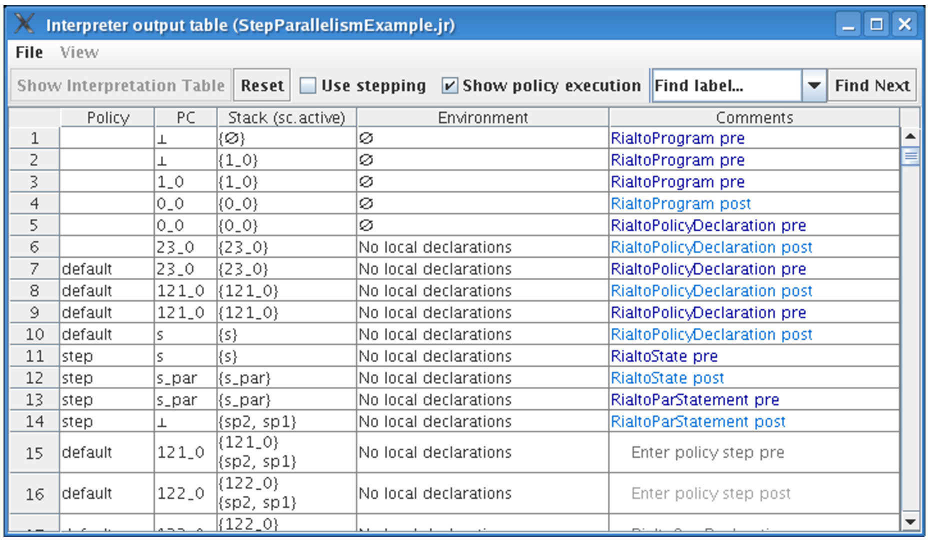
Figure 3.13: The windows for the a) Abstract Syntax Tree and b) Metamodel Tree Window

the *open* button is clicked, the selected file will be opened for interpretation. If the file chosen is a valid Rialto 2.0 program, it will be interpreted immediately without any interaction from the user. The dialog box in figure 3.12 will popup if the interpretation of the program is not completed after 1 000 steps of the program have been interpreted. The user can interrupt the interpretation or he can choose to let the interpretation continue for another 1 000 steps. However, the interpretation will finally be interrupted after 15 000 steps even if the user does not choose to stop the interpretation. The limit of 15 000 steps in GUI mode is motivated because the table would otherwise consume very much memory and become unusable.

Now when the interpretation has been either interrupted or completed, five different JRialto windows are visible on the screen. A window for the representation of the abstract syntax tree, another window in which the Rialto metamodel tree is shown, a third window that is intended to show the detailed information collected during the interpretation and finally a window that represents a redirected console. The AST window in figure 3.13a shows all tree nodes that exist in the abstract syntax tree. Nodes in both the AST window and the metamodel tree window can be expanded or collapsed by clicking on the nodes. The metamodel tree presented in the latter mentioned window shows the name of the objects, which are contained in each node. By moving the pointer over a node, information on that particular node becomes available in a tooltip box as we can see in figure 3.13b. The information available depends on the type of the object, but indepent of the object type a label and type is always shown. An option to expand or collapse all nodes in the abstract syntax tree or in the metamodel tree can be found in the *View* menu in respective window. The trees can be printed by selecting *File - Print.*

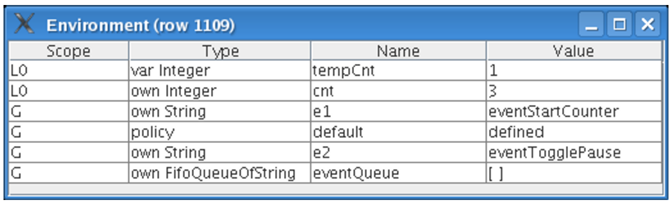
The *Interpreter output table* window found below the two tree windows is perhaps the most interesting, since it is possible to see detailed interpretation information in that window. It features the possibility to see the entire interpretation table at once or gradually. The window does not contain any information from the interpretation before the *Show Interpretation Table* button has been pressed. The name of the above-mentioned button does not indicate that the program should be interpreted, but rather only that the details from the interpretation should be shown. This reflects the fact that the Rialto program is interpreted already before the GUI is available. This does also explain why the dialog box in figure 3.12 asks if the user wants to interrupt the interpretation, already before the GUI is available. The conclusion here is that pressing the *Show Interpretation Table* button does not result in re-interpretation of the program; the information available from the initial information is only shown when the button is pressed. The entire table is shown immediately if the *Use stepping* checkbox is not checked, while only one interpretation step is shown if the box is checked. Repeated button presses causes more interpretation steps to be shown. The available interpretation data can be cleared from the table by pressing the *Reset* button. Another useful checkbox is placed to the right of the use stepping checkbox, namely the *Show policy execution* checkbox. By checking this box, information about the execution of the policies that is hidden by default will be shown in the table. All labels defined in the program can be searched for by using the *Find label* dropdown box. The search is performed only in the program counter column and the stack column.

JRialto's interpreter output table (figure 3.14) has five columns and two rows per executed statement, the first row shows the state of the system before the execution of the statement and the second row shows the state of the system after the execution of the statement. The first column, *Policy*, shows the name of the policy that is assigned to the state in which the executed statement resides. The value of the program counter is available in the *PC* column, while the contents of all active sets in the execution stack can be found in the third column. A Rialto program does also have its own execution environment, but since the environmental information is impossible to fit in a single cell only a brief status of the environment is shown in the *Environment* column. A detailed overview on the contents of the environment is made visible in a separate window when an environment cell is clicked. The second mouse button can be used to show several separate environment windows at once. The rightmost column contains the name, indented to reflect its hierarchical level, of the executed statement. All information available in the interpretation table can be printed or exported to a file. Currently, it is possible to export the table to a comma separated values (csv) file, a HTML file and to a LaTeX file. Since the export routine relies on a framework it should be easy to add new modules for exporting the table to any file format.


Interpreter output table (StepParallelismExample.jr)
File
View
Show Interpretation Table
Reset
Use stepping
Show policy execution
Find label...
Find Next

	Policy	PC	Stack (sc.active)	Environment	Comments
1		⊥	{∅}	∅	RialtoProgram pre
2		⊥	{1_0}	∅	RialtoProgram pre
3		1_0	{1_0}	∅	RialtoProgram pre
4		0_0	{0_0}	∅	RialtoProgram post
5		0_0	{0_0}	∅	RialtoPolicyDeclaration pre
6		23_0	{23_0}	No local declarations	RialtoPolicyDeclaration post
7	default	23_0	{23_0}	No local declarations	RialtoPolicyDeclaration pre
8	default	121_0	{121_0}	No local declarations	RialtoPolicyDeclaration post
9	default	121_0	{121_0}	No local declarations	RialtoPolicyDeclaration pre
10	default	s	{s}	No local declarations	RialtoPolicyDeclaration post
11	step	s	{s}	No local declarations	RialtoState pre
12	step	s_par	{s_par}	No local declarations	RialtoState post
13	step	s_par	{s_par}	No local declarations	RialtoParStatement pre
14	step	⊥	{sp2, sp1}	No local declarations	RialtoParStatement post
15	default	121_0	{121_0} {sp2, sp1}	No local declarations	Enter policy step pre
16	default	122_0	{122_0} {sp2, sp1}	No local declarations	Enter policy step post
			{122_0}		

Figure 3.14: The interpretation table

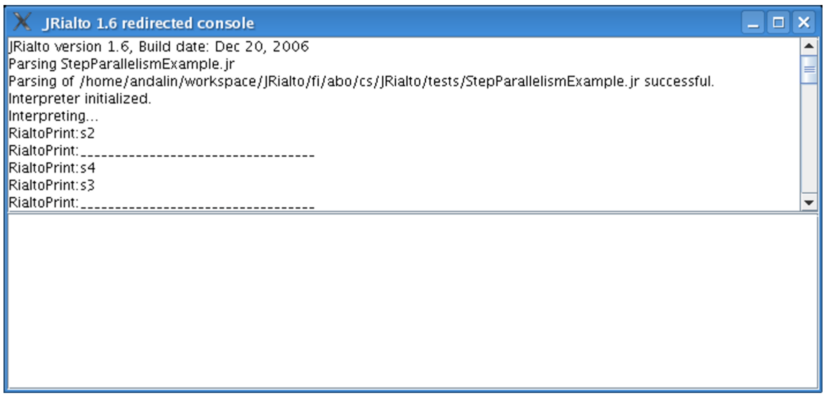


Scope	Type	Name	Value
LO	var Integer	tempCnt	1
LO	own Integer	cnt	3
G	own String	e1	eventStartCounter
G	policy	default	defined
G	own String	e2	eventTogglePause
G	own FifoQueueOfString	eventQueue	[ ]

Figure 3.15: Environment popup presenting the Rialto program's environment

We previously mentioned that a window containing environmental information could be brought up by a mouse click in the interpretation table. An example of an environment window is shown in figure 3.15. The contents are presented in a table in which scopes, types, names and values of variables are visible. Environment windows are most useful when several of them are aligned side by side on the screen. For instance, it is easy to compare the content of the environment before and after the execution of a statement by using a multiple window setup. Please note that the title of the window reflects from which row in the interpretation table the information originates.

The only window still not presented is the *Redirected console* window. From figure 3.16, we can see that the window contains two areas. Output that normally would have been sent to standard output (stdout) in console mode is presented in the upper area, while the lower area shows the output from standard error (stderr). The redirected console window is particularly useful when JRialto is launched from an integrated development environment or from the Java web start, which might imply that a normal text based console for viewing the output is not available.



```
JRialto version 1.6, Build date: Dec 20, 2006
Parsing StepParallelismExample.jr
Parsing of /home/andalin/workspace/JRialto/fi/abo/cs/JRialto/tests/StepParallelismExample.jr successful.
Interpreter initialized.
Interpreting...
RialtoPrint:s2
RialtoPrint:___________________________________
RialtoPrint:s4
RialtoPrint:s3
RialtoPrint:___________________________________
```

Figure 3.16: The graphical output console

4. Translating UML Models to Rialto

In this chapter, the translation process from UML models to Rialto is described. The translation to Rialto includes translation of models represented by a single diagram type, as well as models consisting of several different UML diagrams that together describe the model.

As shown in Figure 4.1, Rialto can be seen as an intermediate language for code synthesis from UML models. The automated translation process is able to capture the different models of computation represented inside a UML model, by using different scheduling policies in Rialto. When the model has been translated into a Rialto model, it can be optimized and source code can be generated for any target language supported by the code generator module.

4.1 The Unified Modeling Language

The Unified Modeling Language (UML) is a visual language that enables powerful design, execution and maintenance of software processes. It has become the standard in object-oriented modelling among software developers [24]. UML is developed by the Object Management Group (OMG), which is an international and non-profit computer industry consortium founded in 1989. Its hundreds of member organizations are representing almost every large organization in the computer industry and many smaller ones. Continuous work on developing the language and adapting it to today's needs is carried out.

The thought of a unified language for designing software systems arose in the late 1980s. At that time several different analysis and design techniques were used, each of them was using their own notation. UML was designed to bring together the best features of those existing analysis and design techniques in one language. As a result, the many diverse object-oriented notations and methods could also be eliminated.

Although most of the elements in UML are graphical, it is possible to create a UML model purely in a simple text editor. This is possible because the underlying

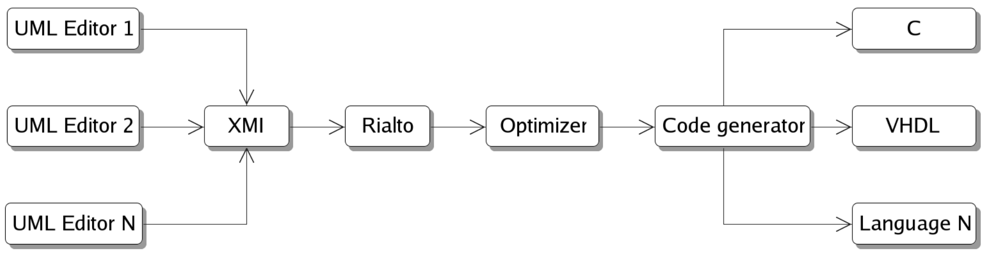


Figure 4.1: Rialto as an intermediate language for code synthesis from UML models.

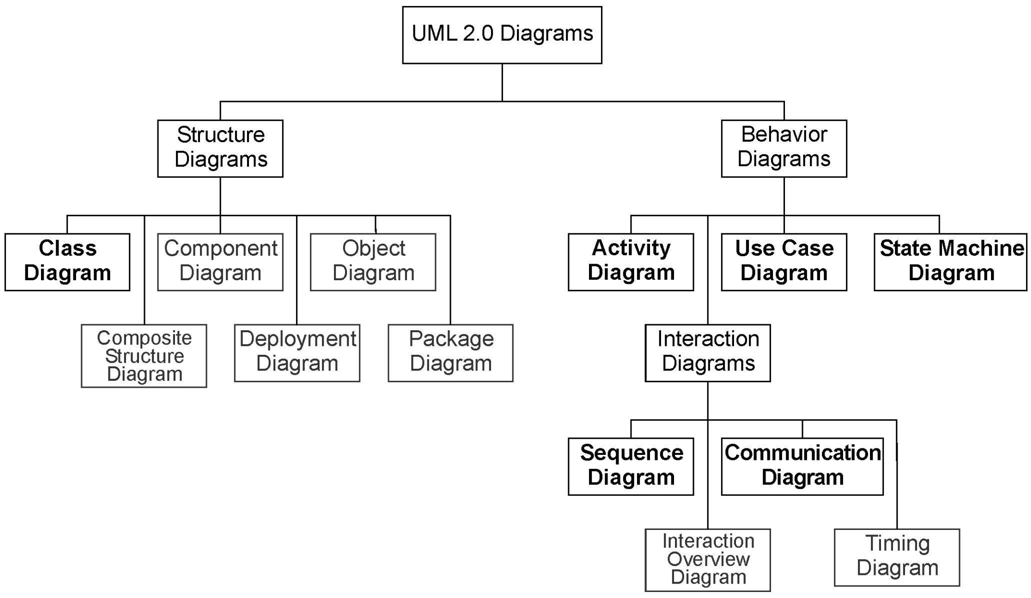


Figure 4.2: The diagrams of UML 2.0 and their structure (from [12]).

notation of UML is textual. The graphical elements are only a graphical representation of the model, making the model more intuitive and easier to understand. A modelling tool must conform to the XMI Schema in order to be conformant with UML. The exchange of models between different modelling tools can be guaranteed, as the XMI Schema is expressed in the extensible markup language (XML) instead of in a proprietary language. UML is not a programming language. However, UML is an excellent modelling language for creating platform independent models and by being able to translate these models into (platform specific) executable code, UML can in one sense be used as a programming language. The current release of UML is Version 2.0. UML 2.0 contains thirteen (13) diagrams classified as Behavior diagrams, Interaction diagrams and Structure diagrams [14]. Both behavior and interaction diagrams depict the behavioural features of a system, but interaction diagrams also describe the interaction between objects. Structure diagrams depict the time independent structure of the system. The most used structure diagram is the class diagram. UML diagrams are overlapping and sometimes a diagram can be totally synthesizable from another [3].

In this chapter the translation from six (6) of the UML 2.0 diagrams is covered. Communication diagrams, Activity diagrams, State Machine diagrams and Class diagrams are used for the translation of the UML model to Rialto 2.0 source code. Additionally Use Case diagrams and Sequence diagrams describe the external events that the JRialto interpreter can utilize for simulation of external events. The different elements and their translations will be presented one by one, but also the problems occuring in certain combinations of the elements will be discussed.

4.2 Communication Diagrams

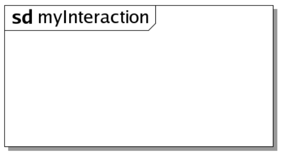
Communication diagrams show how the communication (messages) flows between different parts of the system, more specifically between objects in an object oriented application. The basic relationships between classes are also implied. Every communication diagram contains at least one interaction. Interactions are a mechanism for describing the interactions between different elements in a system [17]. Communication diagrams were known as collaboration diagrams in UML 1.x.

In Rialto, we currently use communication diagrams in order to resolve relationships between instances and generate the appropriate communication channels between these instances. A UML model containing solely communication diagrams is only useful for generating Rialto code stubs. For complete code generation the model must also contain at least an activity diagram or a state machine diagram that models the detailed behaviour of each lifeline. A lifeline is most often representing an instance of a class (an object), but other classifier instances can also be represented by a lifeline. A Communication diagram is always the starting point when translating UML models to Rialto. The interaction, inside any of the communication diagrams, with the same name as the name of the entire model will be chosen as the starting point for the translation process. If a model contains only one interaction, that interaction is chosen regardless of its name. The necessary Rialto program header including a program name is generated by the starting point interaction. The program name is equal to the name of the UML model. The default scheduling policy used for interactions is the `interleaving` policy, which is described in chapter 2. It is used because we consider interactions to have a loose underlying model of computation, similar to threads in programming languages. By specifying a custom interaction property `rialtoPolicy = policy_name`, any policy can be used for scheduling the interaction. Interactions are translated to states in Rialto as will be seen from the translation of an interaction in figure 4.3.

Lifelines are, as mentioned in the previous paragraph, an element representing a classifier instance inside an interaction. Interactions can involve an arbitrary number of lifelines that is representing the different components taking part in the interaction. In Rialto, each lifeline is represented as a state with a label equal to the name of the lifeline. Both lifelines with and without a classifier are allowed in UML, but are seldom useful since non-classified lifelines are translated into empty states in Rialto. Lifelines associated with a classifier or in other words lifelines of a certain type, are more interesting from a Rialto point of view. These lifelines are also translated to states in Rialto, but instead of creating an empty state, the state content will depend on the translation of the classifier associated with the lifeline. Classes, state machines and activities are all valid classifiers for the translation to Rialto. So far, the translation of each lifeline seen as a separate independent part of the interaction has been covered. In order to create a useful rialto model, the interaction between the lifelines must also be taken into account.

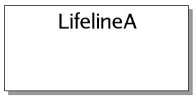
The interaction between lifelines is described using links. Links may be in-

Interaction



```
program
  begin
    myInteraction: state
      policy interleaving;
      begin
    endstate;
end;
```

Lifeline



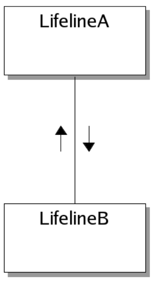
```
LifelineA: state
  begin
endstate;
```

Lifeline of a certain type

LifelineA:myClass

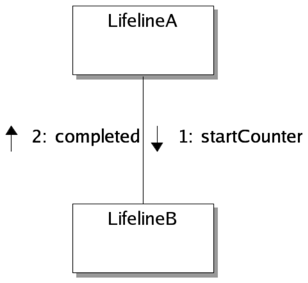
```
LifelineA: state
  // Type specific code
endstate;
```

Communication channel



```
own LifelineAq : FifoQueueOfString;
own LifelineBq : FifoQueueOfString;
par
  LifelineA: state
    begin
  endstate;
||
  LifelineB: state
    begin
  endstate;
endpar;
```

Message



```
own LifelineAq : FifoQueueOfString;
own LifelineBq : FifoQueueOfString;
par
  LifelineA: state
    begin
      LifelineBq.add(startCounter);
  endstate;
||
  LifelineB: state
    begin
      LifelineAq.add(completed);
  endstate;
endpar;
```

Figure 4.3: Translation of communication diagram elements to Rialto.

stances of the associations between classes in a class diagram, or temporary links between lifelines that enable them to send messages to each other. A link can be seen as communication channel between two lifelines as shown in figure 4.3. Lifelines in an interaction are considered to exist in parallel. To reflect the parallelism between the lifelines, the parallel statement in the Rialto language can be used. The communication channel itself is realized using one queue per lifeline. For instance, the end of a link connected to a lifeline `LifelineA` will be modelled as a FIFO queue with the name `LifelineAq`. The sending of a certain message is performed by adding the message to the appropriate queue. The queue to which the message should be added is the queue representing the lifeline that is the receiver of the message, while a statement in the state body of the sender is adding it to the mentioned queue. A message in UML consists of a sequence number and a name, as well as optional attributes, arguments and return values. Currently only the name and the sequence number (to some extent) are considered in Rialto. The string that is added to a queue is equal to the name of the message, while the sequence number decides in which order multiple messages are added to that queue. For instance, if `LifelineA` would like to send the message `start` to `LifelineB`, the statement `LifelineBq.add(start)` in the state body of `LifelineA` would be created.

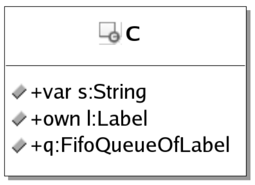
4.3 Class Diagrams

A UML Class diagram shows a static view of the classes in a model. Class diagrams is the perhaps most often used diagram type in UML. Software developers use class diagrams in order to create a design model containing all classes to be included in the software project. As the project moves on from the design phase to the implementation phase, code stubs for several programming languages can be generated from the class diagrams. This is a fast and effective way to generate code and as a result, the code will also be more conformant to the design, but a major drawback is the lack of synchronization between the class diagram and the generated source code. If the design evolves during the implementation of the software, the risk of a gap between the real implementation and the design is imminent. The possibility for collaboration among classes, through message passing, is shown as relationships between the classes [19].

In Rialto, class diagrams are not emphasized as much as the other diagram types described in this chapter. The simple reason for this is the fact that class diagrams describes the static structure among classes, while we in Rialto are more interested in the behavioural properties of a model. A class diagram alone cannot be used for generating Rialto code; however, class diagrams can be used to produce a more complete Rialto model, when used in combination with communication diagrams, activity diagrams and state machine diagrams. The model of the Dining Philosophers problem presented in section 4.7 should give an idea about the benefit of including class diagrams as part of a model. In figure 4.4, the UML class features supported by Rialto are listed together with their corresponding translations.

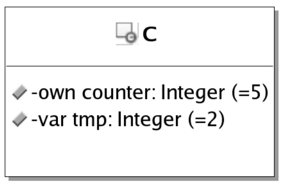
Class attributes are useful for declaring variables in a Rialto program. The

Class attributes



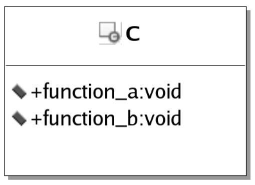
```
C: state
  own q : FifoQueueOfLabel;
  own l : Label;
  var s : String;
  begin
endstate;
```

Class attributes with default value



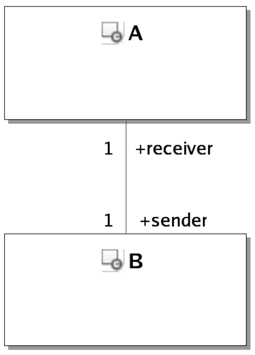
```
C: state
  own counter_def : Boolean;
  own counter     : Integer;
  var tmp_def     : Boolean;
  var tmp         : Integer;
  begin
    if counter_def == false then
      counter := 5;
      counter_def := true;
    else endif;

    if tmp_def == false then
      tmp := 2;
      tmp_def := true;
    else endif;
endstate;
```

Class methods



```
externmethod function_a();
externmethod function_b();
...
C: state
  begin
endstate;
```

Association



```
A: state
  begin
    ...
    senderq.doSomething();
    ...
endstate;

B: state
  begin
    ...
    receiverq.doSomething();
    ...
endstate;
```

Figure 4.4: Translation of class diagram elements to Rialto.

type property of an attribute is used to determine the type of the Rialto variable to be created. The name of the Rialto label will be decided by the attribute name. The attribute name does also indicate if the Rialto variable is of type `var` or of type `own` according to the following rule. If the attribute name begins with the string var or own followed by a space a variable of the corresponding type is created. If var/own is omitted a variable of type own will be created. The scope of an attribute (package, public, protected or private) is not taken into consideration in the translation to Rialto. It is possible to specify a default value for each attribute in UML. The translation of default values are more complicated, as Rialto currently does not have support for user specified initial values of variables. The workaround necessary for supporting default value translations can be found in figure 4.4.

The only intended use for class methods in Rialto are declarations of external methods. External methods are methods residing in a native library. They are explained more in detail in section 3.4.3. Currently only external methods without parameters and no return value are supported in Rialto. The name of a method specified, as a class member will be directly used as the name for the external method declared in Rialto. The scope of a method is ignored.

4.4 State Machine Diagrams

In the two previous sections of this chapter, we focused on the possibility to describe the high-level behaviour of a model with communication diagrams, as well as the possibility to use class diagrams for describing the static structure of a part in the modelled system. In this section, we will focus on how the detailed behaviour of a part in the system can be modelled with UML state machine diagrams. States are fundamental elements in both UML state machine diagrams and in the Rialto language, hence the translation of states machine diagrams to Rialto are important, but relatively easy. The translation of states and other elements used in state machine diagrams will be presented.

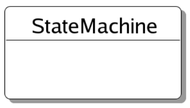
UML state machine diagrams are a diagram type that can be used to describe the behaviour of a model element such as an object or an interaction. A state machine diagram contains state machines that describe the flow between states. State machines are suitable for describing embedded systems, since embedded systems often reacts on stimuli that change the state of the system [3]. While the system is in a certain state, work may or may not be going on. For instance, when a traditional phone is in the hung up state no activity is going on, but when the phone is in another state because it is engaged in a call, there is lots of activity in the phone. State machines can contain three fundamental building blocks: state, transition and event. A state represents a possible state of the system at some point in time. From figure 4.5, we can see that several types of states exist. Each type will be explained more in detail together with their corresponding Rialto translation. Systems that are modelled with state machines are driven by events, both external and internal. The systems respond to the events and quite often, the response results in a transition to another state of the system. Transitions are defined as the movement from one state to another, whereas events are responsible for triggering the transitions.

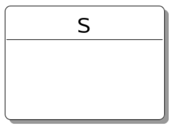
State machine diagrams can be combined with communication diagrams, class diagrams and activity diagrams in a model. By combining the diagram types, it is possible to describe complex systems in an intuitive way and still be able to translate the UML model automatically to a Rialto model. A state machine can for instance describe the behaviour of an instance (lifeline) in a communication diagram or in a more hierarchical model, describe the behaviour of a class. When the system enters a state, some activity can take place or even when the system is in a state, there can be continuous activity in the state. Activity diagrams can be used to specify the behaviour of the activity to be executed. Models that consist of only state machine diagrams can also be used for modelling in Rialto.

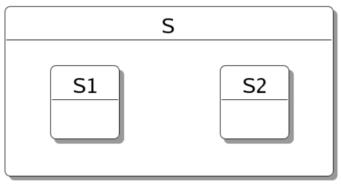
Events are dispatched and processed one at a time in a UML state machine. The semantics of event processing is based on the run-to-completion assumption according to [3]. The run-to-completion (RTC) assumption says that an event can only be dequeued and dispatched if the processing of the previous event is fully completed. It is possible to use different dequeueing orders for the events, because no order of dequeueing is defined in the UML standard. Events are dequeued in a first in - first out order in Rialto by default. The reason for this is that all events are stored in FIFO queues; hence could a different type of queue provide another order of dequeueing. A state machine is translated to a state scheduled by the `rtc` policy in Rialto (figure 4.5). By specifying the custom state machine property `rialtoPolicy = policy_name`, any policy can be used for scheduling the state machine.

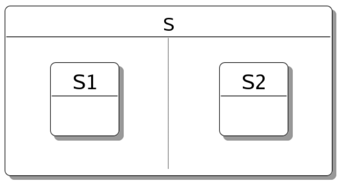
Earlier in this section, we mentioned that different types of states exist in UML. The simplest type of state is called a simple state and it is naturally represented by a `state` statement in Rialto, whose label is equal to the name of the simple state. A simple state cannot be broken down further into states. A composite state on the other hand, can be further broken down into substates. If a composite state *S* is broken down to two substates *S1* and *S2*, the corresponding Rialto code will contain two states labelled `S1` and `S2` inside the state `S`. In order to model states that exist in parallel, orthogonal regions are used in UML. The Rialto `par` statement is suitable for describing orthogonal regions in Rialto. However, if one or several of the orthogonal regions contain more than one state it is necessary two create an additional Rialto state block since the `par` statement syntax does only allow one state block per parallel section. An example of translation of orthogonal regions can be found in figure 4.5. The last state machine element presented in the mentioned figure is a termination point. The interpretation of the entire model is terminated whenever a termination point is reached. In Rialto, such behaviour can be achieved by inserting a `null` statement as the last statement in the program. The transition from a state to a termination point will result in a jump to the `null` statement and since the statement does not reside inside a state, the interpretation will stop after the execution of the statement.

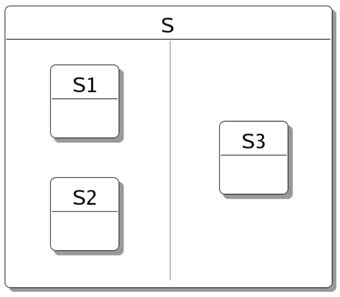
Before we continue with the translation of events and normal transitions, we will cover a special type of transition, namely the unconditional transition. An unconditional transition, also known as triggerless transition, is a transition without any triggering event. It is normally used between actions in activity

State Machine		`StateMachine: state`
		`  policy rtc;`
		`  begin`
		`endstate;`

State		`S: state`
		`  begin`
		`endstate;`

Composite state		`S: state`
		`  begin`
		`    S1: state`
		`      begin`
		`    endstate;`
		`    S2: state`
		`      begin`
		`    endstate;`
		`endstate;`

Parallel state (Orthogonal regions)		`S: state`
		`  begin`
		`    par`
		`      S1: state begin endstate;`
		`    ││`
		`      S2: state begin endstate;`
		`    endpar;`
		`endstate;`

Parallel state (multiple states in a region)		`S: state`
		`  begin`
		`    par`
		`      state`
		`        begin`
		`          S1: state begin endstate;`
		`          S2: state begin endstate;`
		`      endstate;`
		`    ││`
		`      S3: state begin endstate;`
		`    endpar;`
		`endstate;`

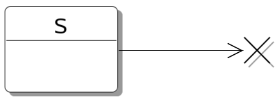
Termination point		`program myProgram`
		`...`
		`S: state`
		`  begin`
		`    goto myProgram_terminate:`
		`endstate;`
		`...`
		`myProgram_terminate: null;`
		`end;`

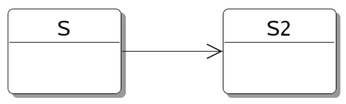
Figure 4.5: Translation of state machine elements to Rialto.

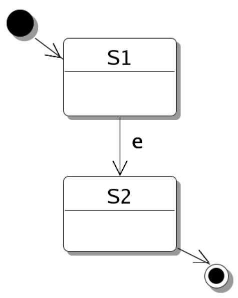
diagrams and not in state machines. However, it is relevant to include a Rialto translation of unconditional transitions in state machines since a state machine could be scheduled by a custom policy instead of the `rtc` policy. Unconditional transitions should never be used in a state machine scheduled by the `rtc` policy. The unconditional transition from a state *S* to another state *S2* is represented in Rialto by the statement `goto S2;` in the body of state `S`.

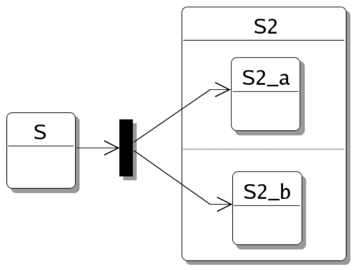
Normal transitions, as opposite to unconditional transitions, contain a triggering event. A triggering event is an event that fires the transition when the event occurs. A transition can also contain an optional guard that must be satisfied before the transition can be fired. When a transition is fired, an optional effect can be executed. The effect can be a call to a procedure, but it can also be a more complex effect described by an activity diagram. As we can see in the second translation from the top in figure 4.6, a transition is represented in Rialto by the `trap` statement. The trap contains an expression and an actions section. The expression represents the event and the optional guards associated with the transition, hence the expression must hold in order to execute the actions section. The actions section contains the transition effect followed by a `goto` statement, which does the actual state transition in Rialto, encapsulated in an atomic unit. The transition should be executed immediately after the effect execution is completed and this can be guaranteed by the atomic unit, since no policy is let in between the execution of statements in an atomic unit.

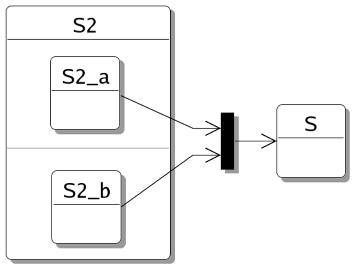
An initial state and a final state are also included in the previously mentioned translation. They are two special pseudostates in the UML. Initial states are the start of a flow, while final states represent a normal completion of the state machine. An initial node is taken into consideration in Rialto by placing the state, which the initial node is pointing at, as the first state in the Rialto model. The final state is replaced by a goto statement in the end of the state pointing at the final state. The destination of the goto statement depends on the model; the goto will result in a termination of the program if the model only contains one state machine, otherwise it will result in a transition to a state in the model that resides in a higher hierarchical level.

Forks and joins are elements to be used to split a transition into multiple paths and later combine the transition into a single transition. The need for a splitted transition arises if the transition should move the system into two or more concurrent states. The join element is useful if the system moves into a single state from the three concurrent states. For instance in a situation where the system is in three concurrent states performing a parallel computation, and the system should move into one single state after each of the computations are performed, the join element will wait until all concurrent states has requested an transition and then fire the outgoing transition from the join element. The last element to be covered is the history state. History states are useful to re-enter a composite state at the same point at which is was last left. If the history element is not used for a certain composite state and a transition moves the system out of that state, all information about the internal state of the composite state is lost.

Unconditional transition		`S: state`
		`  begin`
		`    goto S2;`
		`endstate;`
		`S2: state begin endstate;`

Intial Node, Final State and Transition		`var S1queue : FifoQueueOfString;`
		`S1: state`
		`  begin`
		`    trap S1queue.peek() == e`
		`      do [S1queue.poll(); goto S2;]`
		`    endtrap;`
		`endstate;`
		`S2: state`
		`  begin`
		`    goto SMachine_final;`
		`endstate;`

Fork		`S: state begin goto S2; endstate;`
		`S2: state`
		`  begin`
		`    Fork: par`
		`      S2_a: state begin endstate;`
		`    ||`
		`      S2_b: state begin endstate;`
		`    endpar;`
		`endstate;`

Join		`S2: state`
		`  begin`
		`    par`
		`      S2_a: state begin endstate;`
		`    ||`
		`      S2_b: state begin endstate;`
		`    endpar;`
		`    Join: goto S;`
		`endstate;`
		`S: state begin endstate;`

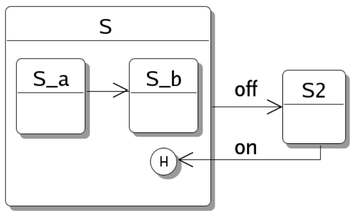
History		`S: state begin`
		`    trap q.peek() == off do`
		`      [q.poll();suspend S;goto S2;]`
		`    endtrap;`
		`    S_a: state ... endstate;`
		`    S_b: state ... endstate;`
		`endstate;`
		`S2: state begin`
		`    trap q.peek() == on do`
		`      [q.poll();resume S;goto S;]`
		`    endtrap;`
		`endstate;`

Figure 4.6: More translations of state machine elements to Rialto.

4.5 Activity Diagrams

After describing the translation of state machine diagrams in the previous section, we will now cover another diagram type that is related to state machine diagrams, namely activity diagrams. Activity diagrams can be used for a great variety of problems from business process modelling to primitive assembly instruction modelling, since activity diagrams can describe the detailed behaviour of an effect or a part of the system. A description of the translation of various activity diagram components to Rialto is presented in this section.

Activity diagrams are a UML diagram type to be used to describe flow in a variety of ways. Activity diagrams contain activities. We remember that state machines (presented in section 4.5) also describes flow in a system, however state machines and activities differs on the type of flow they describe. Whereas activities describe flow between areas of work, state machines describe flow between states. The purpose of activities is to model flows driven by internal processing rather than external events. An activity contains action nodes, object nodes, data stores, control nodes, control edges and a number of other elements [15].

Only a subset of the activity elements has a Rialto translation. The elements related to objects and their flows are not taken into consideration currently. Instead, we focus on the flow of control between actions. Models consisting of only activities can be used for translation to valid and executable Rialto programs. An example of this is the model of a JPEG encoder presented in Chapter 5. Activities can also be used as a part of a more complex model, in which they for instance describe an effect associated with a transition in a state machine. The behavioural specification for an instance (lifeline) in a communication diagram is another possible area of use for activities.

The translation of an activity to Rialto is demonstrated in figure 4.7. We can see that an activity is represented by a state, with an label equal to the name of the activity, in Rialto and an activity parameter is used to declare an extern variable for usage in the activity. Still more important is the `policy step: Policy` activity parameter, which specifies the policy to be used for scheduling the activity. If this activity parameter is omitted the policy `step` is used since it implements the model of computation used for activities in UML. The MOC states that the computation should proceed in steps, where a statement or a block is executed in each concurrent thread at each step [3]. The `step` policy is introduced in chapter 2, while a complete listing of the policy is available in appendix B.3. As mentioned, activity parameters can be used to make a variable, defined externally in a native library, available to the activity. Sometimes it is also desirable to be able to declare a new Rialto variable in the activity and to achieve this we use data stores. Data stores are central buffer nodes that can be used to contain data. The name of the data store should be equal to a normal variable declaration in Rialto, for instance will the name `var i : Integer` create an integer *i* of type *var*.

Actions together with control flows are the fundamental building blocks in activities. An action is a unit of work that needs to be carried out. The extent of an action is depending very much on the abstraction level of our model, from

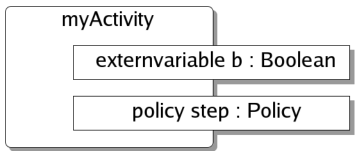
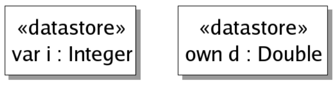
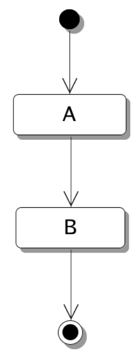
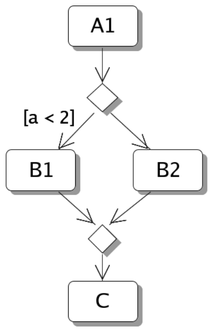
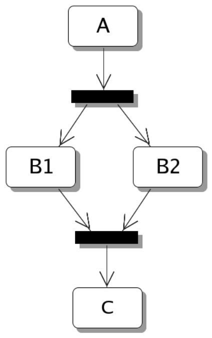
Activity and Activity Parameter	 myActivity externvariable b : Boolean policy step : Policy	`externvariable b: Boolean;` `myActivity: state` `  policy step;` `  begin` `endstate;`
Action		`Action: state` `  policy default;` `  begin` `endstate;`
Data Store		`own d : Double;` `var i : Integer;`
Intitial Node, Activity Final and Control Flow		`A: state` `  begin` `    goto B;` `endstate;` `B: state` `  begin` `    goto actFinalNode;` `endstate;` `actFinalNode: state` `  begin` `    goto ...;` `endstate;`
Decision and Merge		`A: state` `  begin` `    if a < 2 then` `      goto B1;` `    else` `      goto B2;` `    endif;` `endstate;` `B1: state begin goto C; endstate;` `B2: state begin goto C; endstate;` `C: state begin endstate;`
Fork and Join		`A: state begin goto Fork;` `endstate;` `Fork: state` `  begin` `    par` `      B1: state begin endstate;` `    ||` `      B2: state begin endstate;` `    endpar;` `    goto C;` `endstate;` `C: state begin endstate;`

Figure 4.7: Translation of activity diagram elements to Rialto.

a single program instruction to a time consuming calculation. Actions can also be seen as internal states of the activity, hence they are represented by states (labelled according to the name of the actions) in Rialto. Each action can contain an effect that describes the work to be performed in the action. The effect is inserted unmodified in the body of the corresponding Rialto state. In figure 4.7 we see a simple example where two actions A and B are linked together using control flow edges. Additionally, an activity initial node is included to represent the entry point to the flow and an activity final node is there to mark the end of the control flow, i.e. the completion of the activity. There can be only one initial node per activity but several final nodes. The initial node has an outgoing control flow to the action A, which results in a Rialto translation where the state originating from A is placed in the beginning of the activity state body. The control flow between A and B should move the control from A to B when the effect of A is completed, according to the UML superstructure definition [12]. It says that control flow edges can be seen as transitions triggered by implicit completion events. This behaviour is simply achieved by a goto statement at the end of a completed action. The activity final node is represented as a state, which will receive control upon B's completion. The behaviour of the activity final state depends on the overall model translated.

The control flow in an activity diagram does not always follow a single predefined path; quite often, it is necessary to choose different paths or branches depending on some condition. A decision node is used to create branches in activity diagrams. The decision node has one incoming control flow and several outgoing control flows. A condition that can be expressed in any language, determines which outgoing control flow should be chosen. The decision is represented by if statements in Rialto. Another node, the merge node, is used to bring a number of alternative flows together. It is important to use merge nodes only to bring together flows created by decisions nodes, and not the parallel flows created by fork nodes.

Fork nodes are used when it makes sense to allow a number of actions to run in parallel. A fork node will split the incoming flow into several parallel outgoing flows. The parallel flows can be seen as threads running in parallel. Each of the threads can contain an arbitrary number of actions and actions with different execution times. Because of this, a join node must be used to synchronize the threads by waiting for each thread to complete before the control will be given to the action after the join node. Fork and decision nodes that have only one outgoing control flow, as well as join and merge nodes with only one incoming control flow are superfluous, hence they are ignored in the translation.

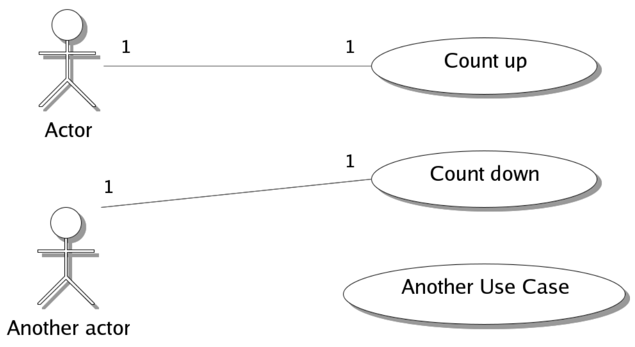


Figure 4.8: A use case diagram describing event simulation

4.6 Simulation of Events using Use Case Diagrams

Sometimes it can be useful to see how the modelled system reacts on events during a simulation of the system. Events generated and consumed internally in the system are covered by state machines according to the rules presented in section 4.4, but what about the events generated outside the modelled system (external events). How can they be considered? The mechanism for handling external events presented in section 3.4.3 is using an event file to describe the external events. The file specifies at which point of time a certain event will occur, as well as in which part of the system the event will occur. The simulation of external events is currently not defined in the Rialto language and it should therefore be considered as a feature of the JRialto implementation. In this section, we will present a way in which UML use case diagrams and sequence diagrams can be used for the generation of the event file.

Use case diagrams are often used to overview the usage requirement for a system and to effectively share information about the system with project stakeholders [19]. A use case diagram shows how use cases and actors are associated to each other, which means that the only notational elements needed are actors, associations and use cases. An actor is a person or system that interacts with a use case, while a use case describes a sequence of actions that a system needs to perform to produce a result of value to the actor. Associations provide a link between the actor and the use case with which the actor interacts. The other diagram type used in this section is the sequence diagram. Sequence diagrams are used to model interaction between class instances by showing the sequential messages that are exchanged between different instances. Instances and messages are modelled in the same way they are modelled in communication diagrams, i.e. by lifelines and messages. However, they are presented by a different graphic notation in communication diagrams and sequence diagrams.

In the context of external event handling in JRialto, use case diagrams and sequence diagrams are used for describing different scenarios consisting of a certain sequential event sequence. The behaviour of each use case is specified by a sequence diagram interaction that is linked to the particular use case. The generation of an event file from a use case diagram is performed by analyzing the

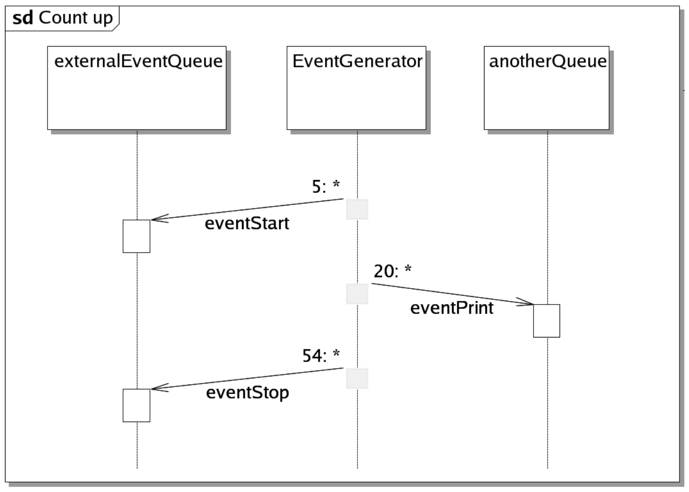
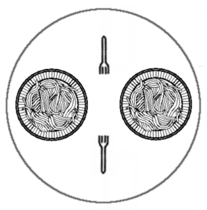


Figure 4.9: A sequence diagram describing the details of a certain use case

associations from each actor within the diagram. From this follows that actors or use cases without any associations are ignored. For instance, the use case *Another Use Case* in the use case diagram in figure 4.8 would be ignored for event generation. One of the actors, *Actor*, has an association to the use case *Count up*. The behaviour of *Count up* is specified by a sequence of events in the sequence diagram in figure 4.9. Each sequence diagram that describes a use case should contain at least two lifelines. One of the lifelines must be named `EventGenerator`, which is a predefined lifeline that generates all events. Apart from this lifeline, at least one lifeline representing a queue defined in the Rialto program should as well be present in the diagram. The lifelines must be named same way as an FIFO queue existing in the Rialto program. The name of an event to be generated is equal to the name of the message. Another important feature of messages is also taken into consideration, namely the sequential order among messages. By using the sequential id of each message as a time tag for the generated event, we can specify the point of time when an event should occur to the system. Furthermore, messages can only have `EventGenerator` as their source, while their destination can be any of the other lifelines. The generation of events from the sequence diagram in figure 4.9 produces the following output to the event file:

```
// Events from 'Count up' use case
{5,    eventStart,   externalEventQueue};
{20,   eventPrint,   anotherQueue};
{54,   eventStop,    externalEventQueue};
```

From the output, we can see that all three messages in the sequence diagram are represented as events. For instance, the message `eventStart` is now represented by an event named the same way. Please see section 3.4.4 for a detailed presentation of the content of event files. A corresponding section with event information is generated for each use case with which an actor interacts.



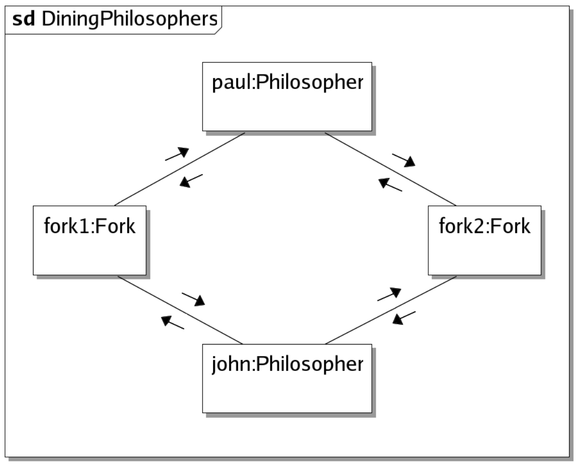


Figure 4.10: Dining philosophers problem overview and the corresponding UML communication diagram

4.7 Automated UML to Rialto Translation

In the previous sections of this chapter, we have presented translation of the different diagrams to Rialto. The focus has been on translation of single independent diagrams, rather than translation of different combinations of diagrams. In this section, we will focus on how Rialto code can be automatically generated from not only a single diagram, but also from combinations of different diagrams. A well-known problem in computer science is used to illustrate and make the translation process easier to understand.

The *Dining philosophers problem* is an often-used example of a common computing problem in concurrency. It concerns synchronization of multiple processes, something that is very essential in computer programming. In computer programming, locking of shared resources is often used to guarantee that only one thread or process at a time is accessing the resource. In the dining philosophers problem there are a number of philosophers spending their time eating and thinking around a table. Each philosopher has a plate of spaghetti in front of him and a fork on each side of the plate. In other words, there is one fork between each plate. As the philosophers do not speak to each other, the risk that every philosopher holds a left fork and forever waits for a right fork (and vice versa) arises. This situation, where a system cannot move on to another state, is called a deadlock. Obviously, the philosophers will suffer from starvation due to the deadlock, but starvation can also occur independently of a deadlock situation if a livelock situation occurs. Livelock means that the system can always advance to a different state, but the processes involved are not really making any progress although they are changing. For instance, livelock can occur if all philosophers pick up their left fork at exactly the same time, but in order to avoid deadlock, all philopsophers will put down their forks for a certain time and then try to pick up the left fork again. As they all waited the same amount of time, they will also this time end up waiting another certain amount of time.

In the example used here for code generation, the number of philosophers is two. An overview of this particular setup of the dining philosophers problem is illustrated to the left in figure 4.10. The corresponding communication diagram representation is to the right in figure 4.10. In the diagram, we can see that we have two philosophers and two forks represented by four different objects. The philosophers, called Paul and John, are instances of the class *Philosopher*, while each fork is an instance of the *Fork* class. The communication channels between the instances are also defined here. Each philosopher is connected to the fork on his left hand side, as well as to the fork on his right hand side. Observe that no communication channel is available between the two philosophers, which originate from the fact that the philosophers do not speak or communicate in any other way with each other. The automatic translation process uses the interaction in this communication diagram as its starting point. The Rialto program stub including the name of the program is generated here. We can see that the interaction in the communication diagram is named DiningPhilosophers and that is why the program generated will be named DiningPhilosophers. The complete code listing for the code generated is available in Appendix C.2. A policy is chosen according to the rules described in section 4.2. The interleaving policy will be used for scheduling this interaction because the interleaving policy is the default one for communication diagrams and in this case, no custom policy has been specified in the interaction. The declaration of the interleaving policy is included in the generated Rialto source code as well.

In order to know the behaviour and structure of each of the instances found in the interaction, parsing of the classes and state machines associated with each instance is necessary. We let philosopher Paul illustrate the parsing process. From figure 4.10, we can see that Paul is an instance of the class Philosopher and thus Paul's static structure is described in the mentioned class. The Philosopher class, illustrated in a class diagram in figure 4.11a, contains variable declarations and associations to the Fork class. The variable declarations eat, sleep, gotIt and ftaken, are of type string as their intended use are as events in our model. The `String` data type is used for event representation in Rialto 2.0. Furthermore, no policy is associated to classes because they represent static structure and not behaviour. The associations between the Philosopher and the Fork classes are not considered at this stage, but they will play an important role in the translation of the state machines to Rialto.

This far we have considered the translation of different instances and the ways they are interacting at a more abstract level, as well as the static structure of the classes from which they are instantiated. The most important part, the exact behaviour of the instance, is still to be covered. The behaviour of an instance is in this case considered the same as the behaviour of the instantiated class. Paul is a philosopher; hence, his behaviour is explained by the state machine describing the behaviour of the philosopher class. The graphical presentation of the mentioned state machine can be found in figure 4.11c. The state machine has four states representing the states a philosopher can be in, which are sleeping, eating, waitLeft and waitRight. Additionally there are appropriate transitions between these states. It should be observed that the events triggering the state transitions

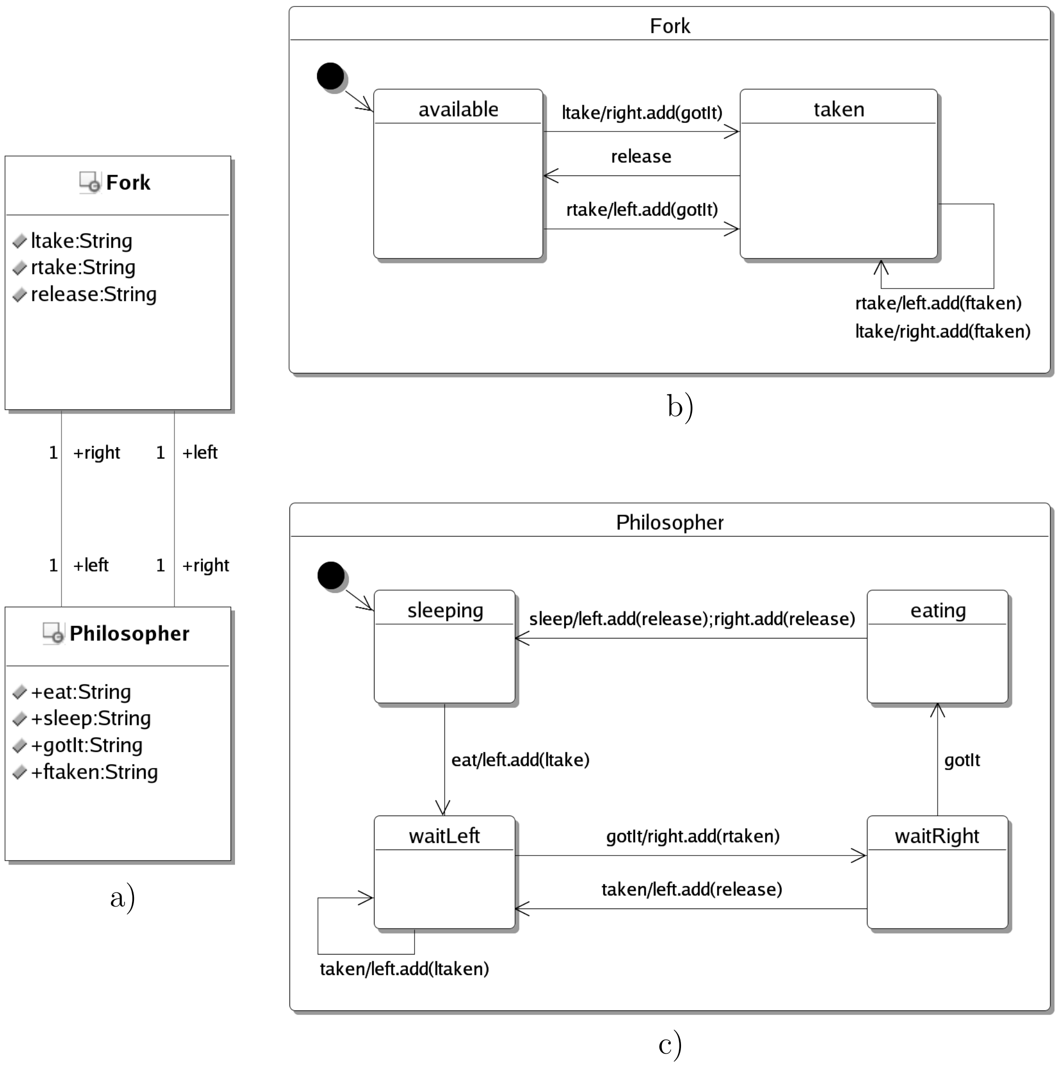


Figure 4.11: The static structure described in a a) class diagram and the behaviour of b) a fork and c) a philosopher represented by state machines.

are the same as the events declared in the class diagram. The translation of the state machine follows the procedure described in section 4.4 and hence this state machine will be scheduled using the run-to-completion (rtc) policy.

As previously mentioned, associations between classes are taken into account when translating the state machine for a class. To illustrate it, the transition from the sleeping state to the waitLeft state will be analyzed. The transition is triggered by the eat event, but besides the transition there is also an effect taking place as a result of the event. The effect is `left.add(ltake)` (it can be found in the diagram to the right of the event). It means that the event ltake should be put into a queue named left. The philosopher class has an association also named left (figure 4.11a) and it associates to class Fork. By combining this information with the information found in the communication diagram, we see that the queue left in the state machine actually refers to the queue attached to fork2. Because of this, the queue name left is replaced with the name of the queue attached to fork1, which is the queue fork2q.

As we can see in figure 4.11b, the behaviour of a fork is also modelled using a state machine. As a fork can only be taken or available, its state machine requires only those two states. In the dining philsophers model presented here both classes were modelled by one state machine each. In the general case, there is no reason to limit the modelling to only one state machine per class and not even to limit the modelling to state machines. The behaviour could be modelled by activities as well, or in an arbitrary combination of state machines and activities. The translation procedure for philosopher Paul that was explained here above holds for each of the instances, i.e. fork1, fork2 and John are also translated using the same procedure. The result, after all instances have been parsed, is a complete and executable Rialto program conforming to the Rialto 2.0 syntax.

5. Case Study - JPEG Encoder

In this chapter, the case study performed as a part of this thesis work will be presented. The goal was to produce a JPEG compressed image from an uncompressed bitmap, using Rialto 2.0. The conceptual idea is to use Rialto for scheduling the different stages in the JPEG algorithm, which are provided by a JPEG library. This involves automated translation of the JPEG encoder model from UML to Rialto, as well as the possibility to use native method calls for real simulation of the model in JRialto. In the following section the underlying JPEG theory necessary in this case study is covered.

5.1 JPEG Encoder Theory

Multimedia data compression has become increasingly important in today's digital world. The various types of multimedia, image, video, audio, text, etc., requires different types of compression algorithms in order to compress the content in an efficient way. In the field of image compression two important international bodies exist, International Organization for Standardization (ISO) and International Telecommunication Union - Telecommunications Sector (ITU-T). ISO focuses on issues such as storage and retrieval of an image, whereas ITU-T focuses on how the image is transmitted. JPEG (Joint Photographic Expert Group) is a standard for still image compression developed jointly by ISO and ITU-T in 1992. The JPEG standard is officially referred to as **ISO/IEC IS 10918-1:** *Digital Compression and Coding of Continuous-tone Still images* and also as **ITU-T Recommendation T.81**. The JPEG theory in this section is based on [25].

JPEG is the first international image compression standard for continuous-tone images. An image is continuous-tone if each colour at any point in it can be produced as a single tone, for instance colour photographs. Its purpose is to support many different applications that need compression of continuous-tone images, for instance a photo editing application. Support for most image sizes in any colour space and adjustable compression ratios are necessary for such a standard. Another criterion is also important if a standard should become widespread with many practical implementations on platforms with different computational power, that criterion is the performance. As a result JPEG has manageable computational complexity even on small platforms with low computational power. To be able to adapt to different requirements on compression quality and speed as well as to the constraints that different transmission channels have, the JPEG defines four modes of operations. The lossless mode, *Sequential Lossless Mode*, is relying on the principles of predictive encoding to be able to compress an image in a single scan without losing any relevant information. In other words, the decoded image is an exact replica of the original image. The three other modes *Sequential DCT-based Mode*, *Progressive DCT-based Mode* and *Hierarchical Mode* provide lossy compression based on the Discrete Cosine Transform (DCT). The reasons why none of these three modes can provide lossless compression are the precision loss in the digitally computed DCT and rounding errors introduced in the quantization process. In *progressive DCT-based mode* the image is compressed in multiple scans, each scan enhances the image quality. A coarse version of the image is transmitted in the first scan, while the latter scans progressively improve the image quality. The advantage of this approach is noticed while decompressing an image for viewing, for instance in a web browser. The user can see a course approximation of the image even if the entire image is not transferred yet and then choose to continue loading the image for full quality or otherwise stop loading it. *Hierarchical DCT-based mode* does provide different resolutions of the image inside one compressed bit stream. This kind of pyramidal multiresolution approach becomes useful when a high-resolution image is viewed on a device with a low-resolution display. The simplest form of the sequential DCT-based JPEG algorithm is called the baseline JPEG algorithm. In the rest of this chapter we will focus on the baseline JPEG algorithm because the JPEG library used in this case study is an implementation of that algorithm.

5.1.1 Baseline JPEG Compression

The baseline JPEG compression algorithm is widely used for both commercial and educational purposes. It is fairly easy to understand and implement in a number of programming languages. Baseline JPEG is defined for images with up to four components. Greyscale images are represented by one component, while RGB (Red, Green and Blue) images use three colour components. The samples inside every component are 8-bit each. A problem with compressing an RGB-image is the significant correlation between red, green and blue colours. The solution here is to convert the image into a decorrelated colour space, for instance the luminance-chrominance colour space YCbCr. The chrominance channels, Cb and Cr, contain much redundant information meaning that it is possible to subsample the channels without reducing the image quality.

When an uncompressed bitmap should be compressed, the first step is to convert the colour space to YCbCr, as mentioned above, if the source bitmap is not already in that colour space. There are several ways to do this conversion by simple mathematical operations as shown in [25, Page 60]. The chrominance channels can now be sub-sampled horizontally and/or vertically in order to remove the redundant information and reduce the data size of these components. It is important to point out that subsampling is optional. Each colour component should now be divided into minimum coded units (MCU), which are data blocks of 8 x 8 samples.

As the baseline JPEG algorithm follows the principles of block-based transform coding, the algorithm should be applied in each of the 8 x 8 data blocks in all components. As shown in Figure 5.1, the following operations are applied to each block (in order): DCT, Quantization and Entropy encoding. The Discrete Cosine Transform used is the two dimensional DCT because an image actually is a two-dimensional signal. The 2-D DCT can be calculated by applying one-dimensional DCT first row-wise and then column-wise. In practical imple-

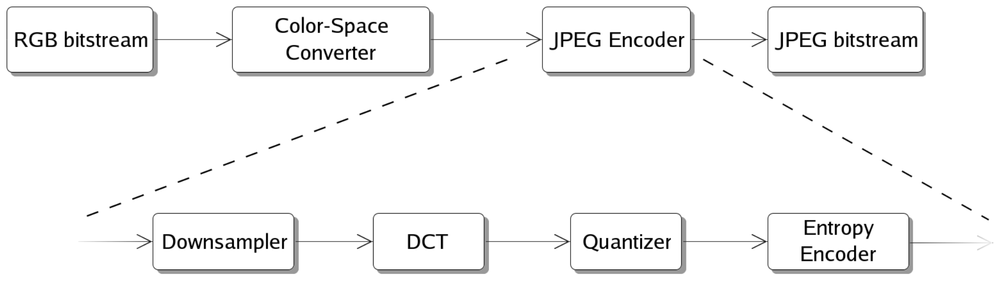


Figure 5.1: Baseline JPEG compression

mentations, more sophisticated and fast algorithms are often used. The results of the DCT calculation are 64 DCT coefficients that should be quantized. The quantization step is primarily responsible for the loss of information and thereby reduction in image quality. The level of quantization is adjustable and in fact the level is exactly what can be found as a JPEG quality factor setting in some photo editing applications.

In the entropy encoding all values are first reordered in Zigzag ordering. The values are now run-length encoded (RLE). Run-length encoding is a very simple form of data compression, in which sequences of the same data value are represented as the value and a number indicating the length of the sequence. The last step is to Huffman encode the run-length encoded values. Huffman encoding is a lossless data compression algorithm that compresses the data by replacing sequences of data with a symbol. The symbol tables, called Huffman tables, contain the lookup symbol and the corresponding data sequence. Finally, the huffman encoded sequence is written to the bit-stream.

5.1.2 JPEG File Interchange Format

The JPEG standard defines the content of the compressed bit stream, but it does not specify any inherent file format in the standard. The absence of a standardized file format resulted in creation of several file formats to store JPEG compressed images. One of these file formats is JFIF. JFIF stands for JPEG File Interchange Format and is one of the most used file formats for storing of JPEG compressed images. It is a small file format focusing on exchange of JPEG bit streams between platforms and applications, but without any advanced features. The simplicity is also its greatest strength and the file format is now available for a wide variety of platforms. It is strongly recommended to use the baseline JPEG mode with JFIF, although any of the four JPEG operation modes are supported. By using the baseline mode, maximum compatibility with all applications and platforms is achieved. Details on the exact syntax of JFIF can be found in [7].

5.2 Modelling the Encoder using Rialto

There are several ways to model a JPEG encoder in Rialto. The model used in this case study is simple but still useful for showing how Rialto can be used for modelling and simulation of real world problems. It makes use of both UML to Rialto translation (covered in Chapter 4) and simulation of external events using Java Native Interface (covered in Section 3.4.3, page 24).

The different inputs to JRialto as well as the output are illustrated in Figure 5.2. To be able to simulate the encoder four inputs are necessary. Obviously, the UML 2.0 Model is the most important input as it models the entire problem; in fact, this single input is sufficient if the purpose of the simulation is not to produce concrete output. The three other inputs enable real simulation in JRialto. An uncompressed bitmap is provided to the system as an image source and the JPEG library supplies a native implementation of the necessary methods, while the JNI Wrapper acts as a bridge between the library and JRialto. The JPEG library used in this case study is a simple implementation of the baseline JPEG algorithm in the C language. It takes care of all system specific operations as well as the computational operations and provides a public interface consisting of methods without return value and parameters. Implementation details on the library can be found in [9]. In order to use native libraries together with JRialto a wrapper must be used, otherwise JRialto cannot call upon the native functions. The requirement to use a wrapper comes from the fact that JRialto is implemented in Java language. The Java Virtual Machine features an interface, JNI, which provides access to native method calls.

The UML 2.0 Model in this case consist of one activity diagram (see Figure 5.3), describing not only the behavioural properties of the encoder but also its static structure. Many of the actions in the activity diagram can easily be derived from the baseline JPEG algorithm covered in the previous section, while other actions and elements need a more thorough explanation. The activity *encodeImage* encapsulates all actions necessary to perform the encoding of the uncompressed bitmap. Activity parameters declare the external variables that are available to the activity, in this case they reside in the native JPEG library. One of the activity parameters, *policy default : Policy*, specifies that the activity should be scheduled using policy default, instead of the RTC policy normally used for activity diagrams. The default policy is used here because our encoder model is sequential. In order to collect statistics from the execution a counter is incremented in each of the actions. These counters are declared as Rialto integers by the *data store* boxes (found in the bottom of the activity).

Before the actual encoding can begin, the uncompressed image file has to be opened for reading. *Read BMP File* tells the JPEG library which file to open by writing the filename to a variable and then call upon the native method `r_read_file()`. All counters are also reset in this action. After the image is opened, the colour space of the entire image is converted from RGB to YUV. The image is split into the three components Y, U and V. We are now ready to request a component by executing `r_get_next_component()`. This method updates the variables `component_available` and `downsample_enabled` to reflect the current

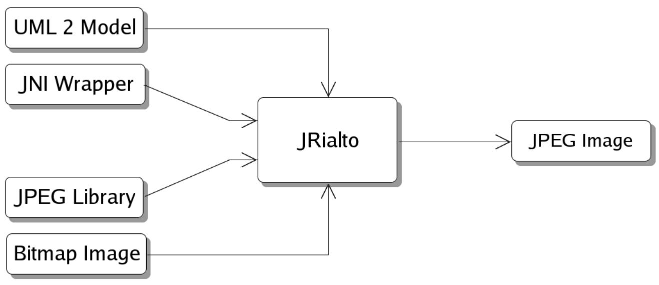


Figure 5.2: JPEG Encoder simulation inputs and output

situation. For instance if we use 4:2:2 downsampling, `downsample_enabled` has value *false* while processing the Y-component and value *true* while processing the other two components. Rialto will schedule the next action to be executed based on these two variables. If there is no component left for processing, the actions *WriteJFIF* and *Print Counters* are executed to write out the compressed bit stream to a file and finally print encoding statistics to standard out, before terminating the activity. Otherwise, if a component is available, downsampling is performed on the component only if it should be downsampled and afterwards the first block of the selected component is read.

The *Read block* action calls upon the native method `r_read_block()`. `r_read_block()` tries to read a block from the component, and sets `block_available` to *true* if a block could be read. In the case that all blocks in the selected component already have been read, `block_available` will contain value *false* and the next component is chosen by executing *Get A Component* again. Otherwise, if a block could be read, the procedures defined in the baseline JPEG standard should be applied to this block. In our activity diagram, these procedures correspond to the actions *DCT*, *Quantize*, *Zig Zag* and *RLE and Huffman*. In the JPEG library used, the last mentioned action is also responsible for writing the encoded block to the compressed output stream. The next block is now read by executing *Read Block* again.

Finally, in the situation where the last block of the last component has been encoded, the compressed bit stream is encapsulated in a JFIF header and written to the specified file. Encoding statistics are written to standard output. The full Rialto source code listing for this case study is available in Appendix C.3.

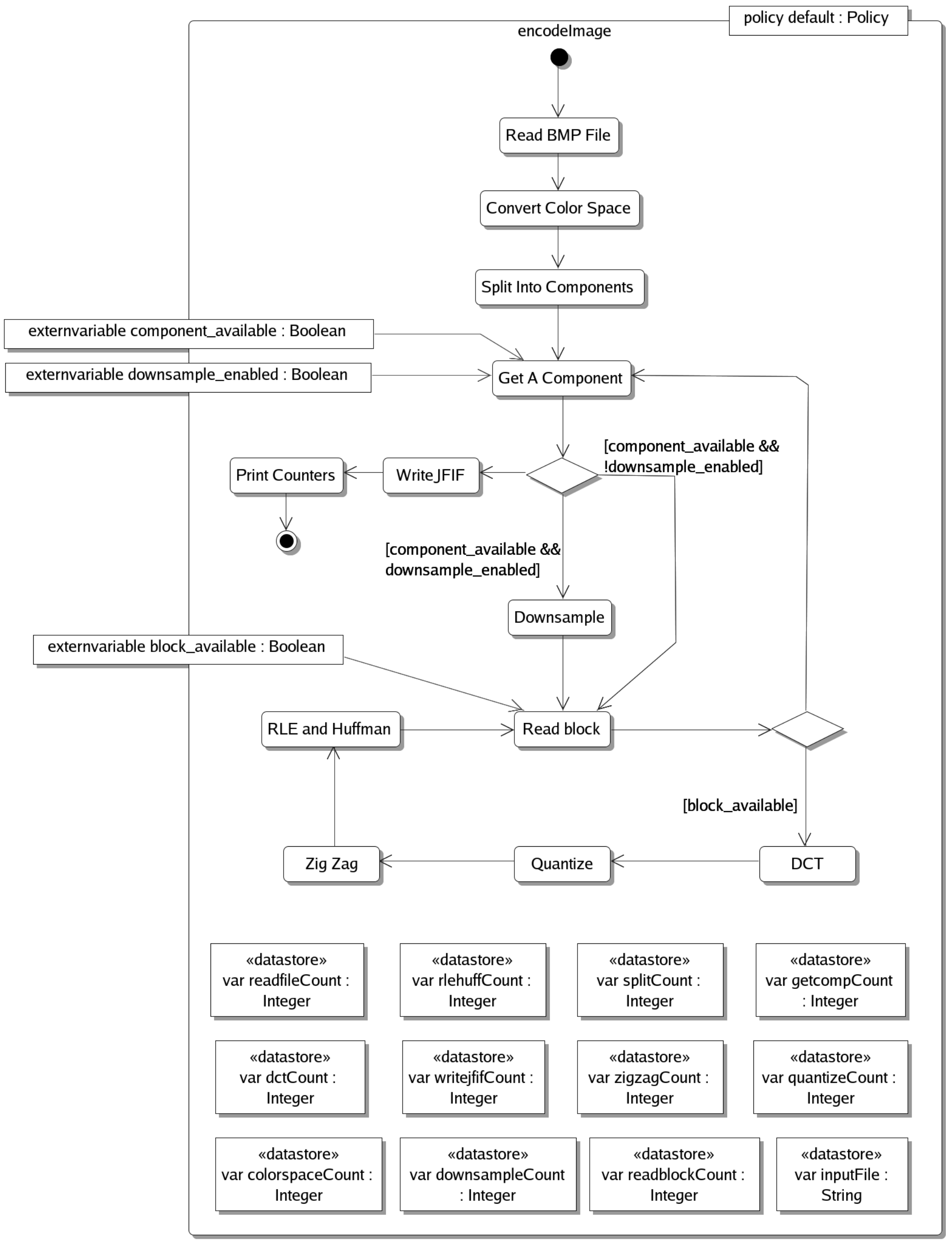

policy default : Policy
encodeImage
Read BMP File
Convert Color Space
Split Into Components
externvariable component_available : Boolean
externvariable downsample_enabled : Boolean
Get A Component
[component_available && !downsample_enabled]
Print Counters
Write JFIF
[component_available && downsample_enabled]
Downsample
externvariable block_available : Boolean
RLE and Huffman
Read block
[block_available]
Zig Zag
Quantize
DCT
«datastore» var readfileCount : Integer
«datastore» var rlehuffCount : Integer
«datastore» var splitCount : Integer
«datastore» var getcompCount : Integer
«datastore» var dctCount : Integer
«datastore» var writejfifCount : Integer
«datastore» var zigzagCount : Integer
«datastore» var quantizeCount : Integer
«datastore» var colorspaceCount : Integer
«datastore» var downsampleCount : Integer
«datastore» var readblockCount : Integer
«datastore» var inputFile : String

Figure 5.3: Activity diagram describing the encoding of an image

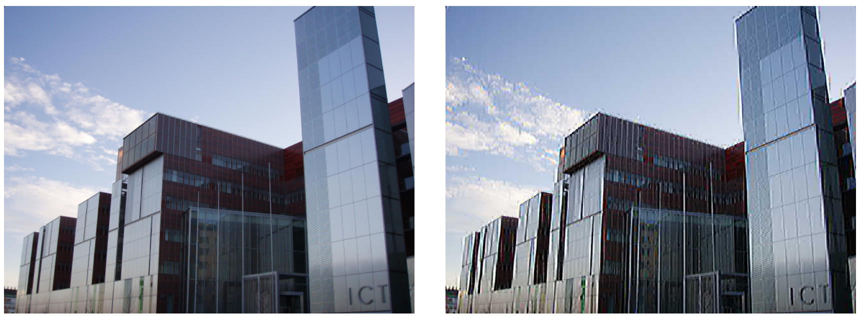
5.3 Simulation Results

The simulation was performed on a computer equipped with an Intel Pentium D at 3.2 GHz and 2 GB of system memory. The operating system running on the computer was Linux (Fedora Core 4). JRialto was running on Suns Java virtual machine (Sun JDK 1.5.6), while the JPEG library was compiled with GNU GCC version 4.0.2. The uncompressed image is 640x480 pixels and it uses the 24-bit RGB colour space.

Encoding the image using JRialto roughly doubled the time consumption compared to encoding the image using a native application written in C, and statically linked together with the JPEG library. In the previous section, we mentioned the usage of counters in our model. The counters are useful to collect statistics about the encoding process and thereby making it possible to verify, not only by viewing the compressed image, that the encoding is scheduled correctly. The output below is the textual output written to standard output by JRialto.

```
Parsing JPEG_Rialto.jr
Parsing of ../CaseStudy/JPEG_Rialto.jr successful.
External library '../CaseStudy/JPEG_Rialto.so' loaded successfully
Interpreter initialized.
Interpreting...
RialtoPrint:Encoding_Statistics
RialtoPrint:____________________
RialtoPrint:ReadBMPFile executed 1 times
RialtoPrint:ConvertColorSpace executed 1 times
RialtoPrint:SplitIntoComponents executed 1 times
RialtoPrint:GetAComponent executed 4 times
RialtoPrint:Downsample executed 2 times
RialtoPrint:ReadBlock executed 9603 times
RialtoPrint:DCT executed 9600 times
RialtoPrint:Quantize executed 9600 times
RialtoPrint:ZigZag executed 9600 times
RialtoPrint:RLEandHuffman executed 9600 times
RialtoPrint:WriteJFIF executed 1 times
Interpretation successful.
```

From the output, we can find out the name of the Rialto program executed. We can also see that an external native library (the JPEG library) was successfully loaded before the interpretation begun. The lines prefixed *RialtoPrint* are the lines that are produced by the print statements in our encoder model. From those lines we can easily see how many times every single action in the model was executed. We know that our image has three components, but why has *Get A Component* been executed four times? This happens because *Get A Component* must always be executed one time more than the number of components in the image, in order to find out that there are no more components available. The same reason makes *Read Block* execute one time more per component than there are blocks in each component. An image of size 640 x 480 pixels in RGB colour space consists of 80 x 60 = 4800 blocks per component. In our simulation, the chrominance components in the compressed image were sub-sampled horizontally



a) b)

Figure 5.4: a) The original and b) the baseline JPEG compressed image

reducing the required amount of data to 66% of the original amount (4:2:2 subsampling format). As a result, the two chrominance components only consist of 2400 blocks each and thus the total amount of blocks in the image will be 4800 + 2400 + 2400 = 9600 blocks. We know that for instance the DCT should be applied once to every block, resulting in a total of 9 600 executions of the DCT action. The statistics shows that this is exactly the number of times that the DCT action actually has been executed.

A visual verification of the result can be seen in Figure 5.4. It is possible to see that the compressed image is of lower quality than the original image due to the lossy baseline JPEG compression. However, the human eye can also detect that the compressed image is encoded correctly, as no major defect exists in the image.

6. Conclusions and Future Work

In this final chapter, we summarize the contents of this thesis. Conclusions on the progress made and the current state of Rialto are presented, as well as the lessons learned during the development of the language implementation from scratch. Proposals for future work concerning both the Rialto language and the JRialto implementation are discussed.

6.1 Summary

This work has presented an initial implementation of the second version of the Rialto language and covered how UML models can be translated to Rialto models. Rialto 2.0 is an evolution from version 1.0; hence, they have the same approach for describing systems with several underlying models of computation. The novelty of the Rialto approach lies in the way the semantics of a model of computation is captured. The computational model is described in so-called scheduling policies. The Rialto approach aims to be able to describe systems consisting of hierarchies of computational models, where a subsystem described using a certain model of computation is encapsulated by a state block. Each state block is attached with a scheduling policy corresponding to the computational model of the subsystem. Scheduling policies can be expressed in the Rialto language, which means that existing policies can be modified and new policies can be created without modifying the interpreter. The policies in this thesis that have been expressed in Rialto show that the language currently is capable of describing computational models correctly. However, the writing of a policy is still quite complicated and cumbersome.

Rialto is intended to be used as an intermediate language between platform independent modelling languages and platform specific executable code. It is important to be able to reuse existing modelling tools for the creation of the models to be executed in Rialto, since the development of a new tool would be very time consuming but also out of scope for the Rialto project. The unified modeling language is a natural choice of language to support in Rialto, since a great number of graphical editors and modelling tools for UML already exist today. We have showed how UML models can be translated to Rialto models and thereby showed on the possibility also to use Rialto for code synthesis from UML models. Behavioural diagrams are the most interesting diagram type in UML from a Rialto point of view, because the behavioural properties of a system are by far more important to us than the static structure. Each of the different UML diagrams used for translation into Rialto, rely on their own computational model. The computational model is captured in the translation process by the assignment of a corresponding Rialto scheduling policy to the state block that encapsulates the diagram content. Any custom policy can be specified to schedule the state block if it is necessary.

Our Rialto implementation is called JRialto since it is developed in Java. The software development was performed in a two-person team. Besides the core interpreter, JRialto does also make it possible to simulate the modelled systems and verify, from the graphical user interface, that the model behaves as intended. The graphical user interface was also of great importance for debugging the interpreter during the development process. Now afterwards, we can state that Java is a suitable and productive language for the implementation of an interpreter for a new language. Java makes it easy and not too time consuming to start producing early prototypes of the final product, while its support for object-oriented programming ensures a straightforward use of modern object-oriented design patterns and techniques. Software developed purely in Java is executable on most computer platforms without any modifications to the source code, which is of clear benefit to a small resource project. In the following section, some future work and improvements to both Rialto and JRialto will be presented.

6.2 Future Work

The implementation of Rialto 2.0 presented in this thesis should, as earlier mentioned, be considered as an initial implementation. At this stage, JRialto should only be used for developing and evaluating the features of Rialto. Additional features should be incorporated into the language itself, as well as into the next version of JRialto.

Currently it is very cumbersome to write a scheduling policy in Rialto 2.0. Even a simple policy requires a considerable amount of code lines, including quite complex control structures, to work properly. Implementation specific knowledge about the stack is also necessary in order to be able to write successfully a policy. By using a *policy abstraction layer*, the creation of new policies would be easier. The layer should be able to hide implementation specific details and provide the user with a relatively easy and intuitive interface. Besides the policy abstraction layer, support for user defined initial values of variables should be included in Rialto. The support for initial values in Rialto would result in a more intuitive translation of UML class attributes. Other necessary improvements to the included data types are support for arbitrary strings (including white spaces and special characters) and the possibility to store container data types inside container data types, i.e. a set should be able to contain a set.

Regarding JRialto, the automatic translation of UML models to Rialto could use an external library or tool for the translation, instead of using a translation implementation inside JRialto. That kind of solution would not require instant changes to JRialto in order to support new versions of UML and perhaps even other visual modelling languages. However, a well-specified interface for the translation module is necessary. Two other major features that should be implemented in JRialto are the optimization and code writer modules, currently only available in the implementation of Rialto 1.0.

A. Usage of Documentation Comments in JRialto

A.1 Introduction

The JRialto source code is, in addition to traditional comments inside the source files, also documented using documentation comments. These comments are delimited by `/** ... */` in the source code. Documentation comments provide an efficient way to document the parts of the source code that should be available to the public. Furthermore, the API documentation can automatically be generated from the documentation comments with the javadoc tool [23]. The generated documentation is easily browsable and therefore an excellent support while developing software. The documentation comments can contain normal language and predefined tags as well as custom tags that help the javadoc tool generate correct and properly formatted documentation.

A.2 Complexity Tag

A new custom tag `@complexity` is introduced in the JRialto source code documentation. The complexity tag briefly describes the CPU time usage and memory consumption of a method or constructor. The intention with this tag is not to provide the developer with an exact metric, but to give a picture of the relative complexity among methods used in JRialto. Integer values [0, 8] are allowed to describe the complexity.

A.3 Guidelines

The guidelines presented here are based on the document *How to Write Doc Comments for the Javadoc Tool* [20] published by Sun Microsystems. In addition, guidelines for the usage of the complexity tag are presented. The type of source

Table A.1: Complexity levels

Complexity	CPU intensive	Memory intensive	Documentation text
0	-	-	No complexity classification available
1 - 2	No	No	Not CPU or Memory intensive
3 - 4	Yes	No	CPU intensive
5 - 6	No	Yes	Memory intensive
7 - 8	Yes	Yes	CPU and Memory intensive

code documentation used should reflect the purpose and intended use of the documentation. Short and concise implementation comments are sufficient in the private scope sections of the source code, but in more public code sections additional documentation comments are of great benefit as they can be separated from the source code files if necessary. The following guidelines are applied on JRialto source files:

- Public classes, fields and methods should be documented
- Protected classes, fields and methods should be documented
- Private classes, fields and methods should not be documented using documentation comments

A different set of tags are suitable for public and protected classes/interfaces, methods and fields. The list below contains the recommended tags and a short description. A detailed reference on these tags can be found in *How to Write Doc Comments for the Javadoc Tool* [20].

1. Classes and Interfaces

   @author Name of the author(s). The creator should be listed first.

   @see Adds a reference to some other part of the API documentation.

   @since Specifies the product version. Always specify a `@since` tag for classes and interfaces.

   {@link} In-line tag to create a link.

2. Constructors

   @param Parameter list, parameters should be in the same order as in the argument declaration list.

   @throws Exceptions thrown, should be listed alphabetically by the exception names.

   @see Adds a reference to some other part of the API documentation.

   @since Specifies the product version. Use only when a constructor has been added to the class after the class was created.

   @complexity Description of the complexity of the constructor.

3. Methods

   @param Parameter list, parameters should be in the same order as in the argument declaration list.

   @return Description of the return value. Should be omitted for methods that returns void.

   @throws Exceptions thrown, should be listed alphabetically by the exception names.

@see Adds a reference to some other part of the API documentation.

@since Specifies the product version. Use only when a method has been added to the class after the class was created.

@complexity Description of the complexity of the method.

4. Fields

@since Specifies the product version. Use only when a field has been added to the class after the class was created.

@see Adds a reference to some other part of the API documentation.

{@value} Represents the value of a field. This tag is not important.

Deciding the Complexity of a Method

Because of the non-exact nature of the complexity tag, it is impossible to provide any exact set of rules for the classification process. By examining existing source code documentation in the JRialto project, the programmer should get a good starting point for classification of new methods. The by far most important rule when deciding the complexity is that a method cannot have a lower complexity level than any method it is calling upon. Classification example:

0 A method that has not been classified yet or a method to which complexity classification is irrelevant.

1 All get/set methods that do not call any method or does not use any loop. If statements and case switches are allowed.

2 All get/set methods that use relatively simple looping (not nested loops) or only a few method calls to methods with complexity 1.

3 Methods using nested loops, advanced algorithms or a great amount of method calls. Memory consumption should be low.

4 Methods that could cause a reduced responsiveness in the application due too heavy CPU usage. Memory consumption should however remain at a low level.

5 Methods handling memory allocation for new objects. Constructors that are more complex could also be in this category.

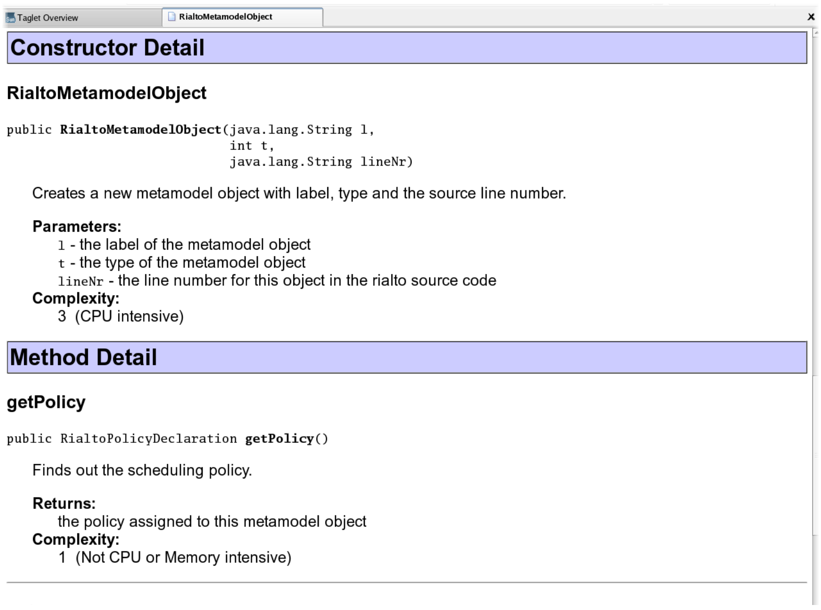
6 Very frequent memory access and/or memory allocation for large data structures.

7 Complex methods consuming both memory and CPU time.

8 Very complex methods consuming both memory and CPU time. This complexity level should not be used to often.

Example of a Documented Class

```
package fi.abo.cs.JRialto.TreeNodes;

import java.util.Vector;
...

/**
 * The RialtoMetamodelObject class represents the basic
 * metamodel object in Rialto programs.
 *
 * @author John Johnson
 * @since 1.6
 * @see RialtoTreeNode
 */
public class RialtoMetamodelObject implements RialtoTreeNode
{
  /**
   * The type of the node represented as an integer.
   * The value is {@value} for RialtoMetamodelObject
   * @since 1.7
   */
  public final int nodeType = 1;


  /**
   * The label assigned to this object.
   * A label is valid only for statements.
   */
  protected RialtoLabel label;

  // Reference to the tree node
  private DefaultMutableTreeNode thisNode;


  /**
   * Creates a new metamodel object with label, type
   * and the source line number.
   *
   * @param l the label of the metamodel object
   * @param t the type of the metamodel object
   * @param lineNr the line number for this object in
                  the rialto source code
   * @complexity 3
   */
  public RialtoMetamodelObject(String l, int t, String lineNr)
  {
    ...
  }
  /**
   * Finds out the scheduling policy.
   *
   * @return the policy assigned to this metamodel object
   * @complexity 1
   */
  public RialtoPolicyDeclaration getPolicy()
  {
    DefaultMutableTreeNode dmt = getTreeNode().getParent();
    return dmt.getUserObject().getPolicy();
  }
  ...
}
```



Constructor Detail

RialtoMetamodelObject

```
public RialtoMetamodelObject(java.lang.String l,
                             int t,
                             java.lang.String lineNr)
```

Creates a new metamodel object with label, type and the source line number.

Parameters:
`l` - the label of the metamodel object
`t` - the type of the metamodel object
`lineNr` - the line number for this object in the rialto source code
Complexity:
3 (CPU intensive)

Method Detail

getPolicy

```
public RialtoPolicyDeclaration getPolicy()
```

Finds out the scheduling policy.

Returns:
the policy assigned to this metamodel object
Complexity:
1 (Not CPU or Memory intensive)

Figure A.1: API documentation generated from the example class

B. Listing of Policies

B.1 Default Policy

```
policy default
  own indefault: Boolean;
  var l: Label;
  begin
    l := sc.prevProgCtx.getLabelFromActiveSet();
    if indefault == true then
      indefault := false;
      sc.bottom().getActiveSet().add(l);
      if sc.size() > 2 then
        sc.popFromPrevProgCtx();
      else
      endif;
      return __;
    else
      indefault := true && !sc.inPolicyMode();
      return l;
    endif;
end;
```

B.2 Interleaving Policy

```
policy interleaving
  own instep         : Boolean;
  own stackSize      : Integer;
  own lubLabel       : Label;
  var labelFound     : Boolean;
  var step           : FifoQueueOfLabel;
  var l              : Label;
  var labelToRun     : Label;
  var currentStepSet : SetOfLabel;
  var resultLabels   : SetOfLabel;
  var sconfig        : StateConfig;
  var labelAsString  : String;

  begin
    if instep then
      goto interleaving_end;
    else
      goto interleaving_init;
  endif;
```

```
// Initialisation state
interleaving_init:
  instep := true;
  resultLabels := sc.prevProgCtx.getActiveSet();
  labelToRun := sc.prevProgCtx.getRandomLabelFromActiveSet();
  step := calculateStep(currentPc);
  sc.prevProgCtx.getActiveSet().clear();
  lubLabel := step.poll();
  sc.prevProgCtx.getActiveSet().add(lubLabel);
  step.poll();

  interleaving_loop: if step.empty() == false then
    labelAsString := step.poll();
    if labelAsString == __ then
      if labelFound == true then
        sconfig.getActiveSet().add(currentStepSet);
        sc.pushAbovePrevProgCtx(sconfig);
        sconfig.clear();
        resultLabels.remove(currentStepSet);
        labelFound := false;
      else
        currentStepSet.clear();
        goto interleaving_loop;
      endif;
    else
      l := labelAsString;
      currentStepSet.add(l);
      labelFound := (labelFound == false && l == labelToRun)
                    || labelFound == true;
      goto interleaving_loop;
    endif;
  else
  endif;

  sconfig.getActiveSet().add(resultLabels);
  sc.pushToBottom(sconfig);
  stackSize := sc.size();

  if lubLabel == labelToRun then
    return labelToRun;
  else
    return __;
  endif;


// End interleaving step state
interleaving_end:
  instep := false;
```

```
    sconfig := sc.popFromPrevProgCtx();
    sconfig.getActiveSet().remove(lubLabel);
    currentStepSet := sconfig.getActiveSet();
    sc.bottom().getActiveSet().add(currentStepSet);

    if stackSize - sc.size() == 1 then
      sc.popFromPrevProgCtx();
    else
    endif;

    if sc.size() > 2 then
      sconfig := sc.popFromBottom();
      currentStepSet := sconfig.getActiveSet();
      sc.bottom().getActiveSet().add(currentStepSet);
    else
    endif;

    return __;
end;
```

B.3 Step Policy

```
policy step
  own instep         : Boolean;
  own stackSize      : Integer;
  own lubLabel       : Label;
  var step           : FifoQueueOfLabel;
  var l              : Label;
  var currentStepSet : SetOfLabel;
  var sconfig        : StateConfig;
  var labelAsString  : String;

  begin
    if instep then
      goto step_end;
    else
      goto step_init;
    endif;


    // Initialisation state
    step_init:
      instep := true;
      step := calculateStep(currentPc);
      sc.prevProgCtx.getActiveSet().clear();
      lubLabel := step.poll();
      sc.prevProgCtx.getActiveSet().add(lubLabel);
      step.poll();

      step_loop: if step.empty() == false then
        labelAsString := step.poll();
        if labelAsString == __ then
          sconfig.getActiveSet().add(currentStepSet);
          sc.pushAbovePrevProgCtx(sconfig);
          sconfig.clear();
          currentStepSet.clear();
        else
          l := labelAsString;
          currentStepSet.add(l);
        endif;
        goto step_loop;
      else
      endif;

      sconfig.clear();
      sc.pushToBottom(sconfig);
      stackSize := sc.size();

      if lubLabel == labelToRun then
        return labelToRun;
      else
        return __;
      endif;

    // End of step state
    step_end:
      instep := false;
      sc.popFromPrevProgCtx();
      sconfig.getActiveSet().remove(lubLabel);
      currentStepSet := sconfig.getActiveSet();
      sc.bottom().getActiveSet().add(currentStepSet);

      if stackSize - sc.size() == 1 then
        sc.popFromPrevProgCtx();
      else
      endif;

      if sc.size() > 2 then
        sconfig := sc.popFromBottom();
        currentStepSet := sconfig.getActiveSet();
        sc.bottom().getActiveSet().add(currentStepSet);
      else
      endif;

      return __;
end;
```

B.4 Synchronous Dataflow Policy

```
policy sdf
 // Policy sdf reads the execution sequence
 // from the global variable sdfSequence
 own insdf        : Boolean;
 own remainSeq    : FifoQueueOfLabel;
 own currentLabel : Label;
 own eosLabel     : Label;
 own myLabel      : Label;
 var bottomLabel  : Label;
 var bottomSet    : SetOfLabel;
 var sconfig      : StateConfig;

 begin
   if insdf then
     goto sdf_continue;
   else
     goto sdf_init;
   endif;


   sdf_init:
     insdf := true;
     remainSeq := sdfSequence;
     currentLabel := sdf_continue;
     myLabel := sc.prevProgCtx.getLabelFromActiveSet();
     sc.pushToBottom(sconfig);


   sdf_continue:
     sc.prevProgCtx.getActiveSet().clear();
     bottomSet := sc.bottom().getActiveSet();
     if bottomSet.empty() == false then
       bottomLabel := bottomSet.getObjectFromSet();
       sc.bottom().getActiveSet().remove(bottomLabel);
     else
     endif;

     if currentLabel == bottomLabel
          || currentLabel == sdf_continue then
       bottomLabel := remainSeq.poll();
       if remainSeq.empty() then
         goto sdf_end;
       else
         currentLabel := bottomLabel;
       endif;
     else
     endif;
```

```
      sc.prevProgCtx.getActiveSet().add(myLabel);
      sc.pushAbovePrevProgCtx(sconfig);
      sc.prevProgCtx.getActiveSet().add(bottomLabel);
      return __;


    sdf_end:
      insdf := false;
      sc.popFromBottom();
      sc.prevProgCtx.getActiveSet().add(bottomLabel);
      return bottomLabel;
end;
```

C. Source Code Listing

C.1 Step Example

```
program Step
  policy interleaving
    // policy body omitted
  end;

  policy step
    // policy body omitted
  end;


  begin
    s: state
      policy step;
      begin
        s_par: par
          sp1: state
            policy default;
            begin
              s2: state
                policy default;
                begin
                  goto s3;
              endstate;

              s3: state
                policy default;
                begin
                  goto s2;
              endstate;
          endstate;

        ||

          sp2: state
            policy interleaving;
            begin
              sp2_par: par
                sp2_parleft: state
                  policy default;
                  begin
                    s4: state
                      policy default;
                      begin
                        goto s5;
                    endstate;

                    s5: state
                      policy default;
                      begin
                        goto s4;
                    endstate;
                endstate;

              ||

                sp2_parright: state
                  policy default;
                  begin
                    s6: state
                      policy default;
                      begin
                        goto s7;
                    endstate;

                    s7: state
                      policy default;
                      begin
                        goto s6;
                    endstate;
                endstate;
              endpar;
          endstate;
        endpar;
    endstate;
end;
```

C.2 Dining Philosophers Example

```
program DiningPhilosophers
  policy interleaving
    // policy body omitted
  end;

  policy rtc
    // policy body omitted
  end;

  begin
    dinner: state
```

```
policy interleaving;
own fork1q : FifoQueueOfString;
own fork2q : FifoQueueOfString;
own johnq  : FifoQueueOfString;
own paulq  : FifoQueueOfString;
begin
  par
    // Fork 1
    fork1: state
      policy rtc;
      own ltake   : String;
      own rtake   : String;
      own release : String;
      begin
        ltake   := ltake;
        rtake   := rtake;
        release := release;

      fork1_available: state
        begin
          trap fork1q.peek() == ltake
            do [fork1q.poll(); paulq.add(gotIt);goto fork1_taken;]
          endtrap;
          trap fork1q.peek() == rtake
            do [fork1q.poll(); johnq.add(gotIt);goto fork1_taken;]
          endtrap;
      endstate;

      fork1_taken: state
        begin
          trap fork1q.peek() == release
            do [fork1q.poll(); goto fork1_available;]
          endtrap;
          trap fork1q.peek() == ltake
            do [fork1q.poll(); paulq.add(ftaken);]
          endtrap;
          trap fork1q.peek() == rtake
            do [fork1q.poll(); johnq.add(ftaken);]
          endtrap;
      endstate;
    endstate;
  ||
    // Fork 2
    fork2: state
      policy rtc;
      own ltake   : String;
      own rtake   : String;
      own release : String;
      begin
```

```
        ltake   := ltake;
        rtake   := rtake;
        release := release;

      fork2_available: state
        begin
          trap fork2q.peek() == ltake
            do [fork2q.poll(); johnq.add(gotIt);goto fork2_taken;]
          endtrap;
          trap fork2q.peek() == rtake
            do [fork2q.poll(); paulq.add(gotIt);goto fork2_taken;]
          endtrap;
      endstate;

      fork2_taken: state
        begin
          trap fork2q.peek() == release
            do [fork2q.poll(); goto fork2_available;]
          endtrap;
          trap fork2q.peek() == ltake
            do [fork2q.poll(); johnq.add(ftaken);]
          endtrap;
          trap fork2q.peek() == rtake
            do [fork2q.poll(); paulq.add(ftaken);]
          endtrap;
      endstate;
    endstate;
  ||
    // Philosopher John
    john: state
      policy rtc;
      own eat    : String;
      own sleep  : String;
      own gotIt  : String;
      own ftaken : String;
      begin
        eat := eat;
        sleep := sleep;
        gotIt := gotIt;
        ftaken := ftaken;

        john_sleeping: state
          begin
            trap johnq.peek() == eat
              do [johnq.poll(); fork1q.add(ltake);
                  goto john_waitLeft;]
            endtrap;
        endstate;
    john_eating: state
      begin
        trap johnq.peek() == sleep
          do [johnq.poll(); fork1q.add(release);
              fork2q.add(release); goto john_sleeping;]
        endtrap;
    endstate;

    john_waitLeft: state
      begin
        trap johnq.peek() == gotIt
          do [johnq.poll(); fork2q.add(rtaken);
              goto john_waitRight;]
        endtrap;
        trap johnq.peek() == ftaken
          do [johnq.poll(); fork1q.add(ltaken);]
        endtrap;
    endstate;

    john_waitRight: state
      begin
        trap johnq.peek() == gotIt
          do [johnq.poll(); goto john_eating;]
        endtrap;
        trap johnq.peek() == ftaken
          do [johnq.poll(); fork1q.add(release);
              goto john_waitLeft;]
        endtrap;
    endstate;
  endstate;
||
  // Philosopher Paul
  paul: state
    policy rtc;
    own eat    : String;
    own sleep  : String;
    own gotIt  : String;
    own ftaken : String;
    begin
      eat := eat;
      sleep := sleep;
      gotIt := gotIt;
      ftaken := ftaken;

      paul_sleeping: state
        begin
          trap paulq.peek() == eat
            do [paulq.poll(); fork2q.add(ltake);
                goto paul_waitLeft;]
                endtrap;
            endstate;

            paul_eating: state
              begin
                trap paulq.peek() == sleep
                  do [paulq.poll(); fork2q.add(release);
                      fork1q.add(release); goto paul_sleeping;]
                endtrap;
            endstate;

            paul_waitLeft: state
              begin
                trap paulq.peek() == gotIt
                  do [paulq.poll(); fork1q.add(rtaken);
                      goto paul_waitRight;]
                endtrap;
                trap paulq.peek() == ftaken
                  do [paulq.poll(); fork2q.add(ltaken);]
                endtrap;
            endstate;

            paul_waitRight: state
              begin
                trap paulq.peek() == gotIt
                  do [paulq.poll(); goto paul_eating;]
                endtrap;
                trap paulq.peek() == ftaken
                  do [paulq.poll(); fork2q.add(release);
                      goto paul_waitLeft;]
                endtrap;
            endstate;
        endstate;
      endpar;
  endstate;
end;
```

C.3 JPEG Encoder Example

```
program JPEG_Rialto
  externmethod r_dct();
  externmethod r_downsample();
  externmethod r_get_next_component();
  externmethod r_quantize();
  externmethod r_read_block();
  externmethod r_read_file();
  externmethod r_rgb_to_yuv();
externmethod r_rlc_vlc_encode();
externmethod r_split_to_components();
externmethod r_write_jfif();
externmethod r_zigzag();

externvariable block_available     : Boolean;
externvariable component_available : Boolean;
externvariable downsample_enabled  : Boolean;

begin
  encodeImage: state
    policy default;
    var colorspaceCount : Integer;
    var dctCount        : Integer;
    var downsampleCount : Integer;
    var getcompCount    : Integer;
    var quantizeCount   : Integer;
    var readblockCount  : Integer;
    var readfileCount   : Integer;
    var rlehuffCount    : Integer;
    var splitCount      : Integer;
    var writejfifCount  : Integer;
    var zigzagCount     : Integer;
    var inputFile       : String;
    begin
      ReadBMPFile: state
        policy default;
        begin
          [
          inputFile := myimage;
          readfileCount := 0;
          colorspaceCount := 0;
          splitCount := 0;
          getcompCount := 0;
          downsampleCount := 0;
          readblockCount := 0;
          dctCount := 0;
          quantizeCount := 0;
          zigzagCount := 0;
          rlehuffCount := 0;
          writejfifCount := 0;
          ];
          [
          r_read_file();
          readfileCount := readfileCount + 1;
          ];
          goto ConvertColorSpace;
      endstate;
```

```
ConvertColorSpace: state
  policy default;
  begin
    [
    r_rgb_to_yuv();
    colorspaceCount := colorspaceCount + 1;
    ];
    goto SplitIntoComponents;
endstate;

SplitIntoComponents: state
  policy default;
  begin
    [
    r_split_to_components();
    splitCount := splitCount + 1;
    ];
    goto GetAComponent;
endstate;

GetAComponent: state
  policy default;
  begin
    [
    r_get_next_component();
    getcompCount := getcompCount + 1;
    ];

    if component_available && !downsample_enabled then
      goto Readblock;
    else
    endif;
    if component_available && downsample_enabled then
      goto Downsample;
    else
      goto WriteJFIF;
    endif;
endstate;

WriteJFIF: state
  policy default;
  begin
    [
    r_write_jfif();
    writejfifCount := writejfifCount + 1;
    ];
    goto PrintCounters;
endstate;
```

```
Downsample: state
  policy default;
  begin
    [
    r_downsample();
    downsampleCount := downsampleCount + 1;
    ];
    goto Readblock;
endstate;

DCT: state
  policy default;
  begin
    [
    r_dct();
    dctCount := dctCount + 1;
    ];
    goto Quantize;
endstate;

Quantize: state
  policy default;
  begin
    [
    r_quantize();
    quantizeCount := quantizeCount + 1;
    ];
    goto ZigZag;
endstate;

ZigZag: state
  policy default;
  begin
    [
    r_zigzag();
    zigzagCount := zigzagCount + 1;
    ];
    goto RLEandHuffman;
endstate;

RLEandHuffman: state
  policy default;
  begin
    [
    r_rlc_vlc_encode();
    rlehuffCount := rlehuffCount + 1;
    ];
    goto Readblock;
endstate;
```

```
        Readblock: state
          policy default;
          begin
            [
            r_read_block();
            readblockCount := readblockCount + 1;
            ];

            if block_available then
              goto DCT;
            else
              goto GetAComponent;
            endif;
        endstate;

        PrintCounters: state
          policy default;
          begin
            [
            print Encoding_Statistics;
            print ____________________;
            print ReadBMPFile, executed , readfileCount, times;
            print ConvertColorSpace, executed, colorspaceCount, times;
            print SplitIntoComponents, executed, splitCount, times;
            print GetAComponent, executed, getcompCount, times;
            print Downsample, executed, downsampleCount, times;
            print ReadBlock, executed, readblockCount, times;
            print DCT, executed, dctCount, times;
            print Quantize, executed, quantizeCount, times;
            print ZigZag, executed, zigzagCount, times;
            print RLEandHuffman, executed, rlehuffCount, times;
            print WriteJFIF, executed, writejfifCount, times;
            ];
            goto encodeImage_FinalNode;
        endstate;

        encodeImage_FinalNode: state
          begin
            goto JPEG_Rialto_terminate;
        endstate;
    endstate;

    JPEG_Rialto_terminate: null;
end;
```

Bibliography

[1] Object Management Group Website - `http://www.omg.org`.

[2] Axel Jantsch. *Modeling Embedded Systems and SoCs.* Morgan Kaufmann Publishers, 2001.

[3] Dag Björklund. *A Kernel Language for Unified Code Synthesis.* PhD thesis, 2005.

[4] Dag Björklund, Johan Lilius and Ivan Porres. A Unified Approach to Code Generation from Behavioral Diagrams. Technical report, 2003.

[5] Markus Dahlgård. Modelling Techniques and a Compiler Front End Implementation for the Rialto modelling language. Master's thesis, 2007.

[6] Dick Grune, Henri E. Bal, Ceriel J.H. Jacobs and Koen G. Langendoen. *Modern Compiler Design.* Wiley, 2000.

[7] Eric Hamilton. *JPEG File Interchange Format.* `http://www.w3.org/Graphics/JPEG/jfif3.pdf`, 1992.

[8] Erik Gamma, Richard Helm, Ralph Johnson and John Vlissides. *Design Patterns.* Addison-Wesley, 1995.

[9] Johan Ersfolk. Code Generation for Heterogeneous Platforms. Master's thesis, 2007.

[10] Johan Lilius and Lionel Morel. Rialto 2.0: a language for heterogeneous computations. Technical report, 2006.

[11] Object Management Group. *Meta Object Facility 2.0 XMI Mapping Specification, v2.1.* `http://www.omg.org/cgi-bin/doc?formal/05-09-01`, 2005.

[12] Object Management Group. *UML Superstructure Specification, v2.0.* `http://www.omg.org/cgi-bin/doc?formal/05-07-04`, 2005.

[13] Peter Marwedel. *Embedded Systems Design.* Springer, 2006.

[14] Scott W. Ambler. *Introduction to the Diagrams of UML 2.0.* `http://www.agilemodeling.com/essays/umlDiagrams.htm`, 2006.

[15] Scott W. Ambler. *UML 2 Activity Diagrams.* `http://www.agilemodeling.com/artifacts/activityDiagram.htm`, 2006.

[16] Scott W. Ambler. *UML 2 Class Diagrams.* `http://www.agilemodeling.com/artifacts/classDiagram.htm`, 2006.

[17] Scott W. Ambler. *UML 2 Communication Diagrams.* `http://www.agilemodeling.com/artifacts/communicationDiagram.htm`, 2006.

[18] Scott W. Ambler. *UML 2 State Machine Diagrams.* `http://www.agilemodeling.com/artifacts/stateMachineDiagram.htm`, 2006.

[19] Simon Bennett, John Skelton and Ken Lunn. *UML.* McGraw-Hill International, 2005.

[20] Sun MicroSystems. *How to Write Doc Comments for the Javadoc Tool.* `http://java.sun.com/j2se/javadoc/writingdoccomments/`, 2006.

[21] Sun Microsystems. *Java 2 Platform Standard Edition 5.0 API Specification.* `http://java.sun.com/j2se/1.5.0/docs/api/`, 2006.

[22] Sun Microsystems. *Java Native Interface 5.0 Specification.* `http://java.sun.com/j2se/1.5.0/docs/guide/jni/`, 2006.

[23] Sun Microsystems. *Javadoc Tool Home Page.* `http://java.sun.com/j2se/javadoc/`, 2006.

[24] Max Söderström. vUML - A Tool for Verifying Collaborations of UML Statecharts. Master's thesis, 2004.

[25] Tinku Acharya and Ping-Sing Tsai. *JPEG2000 Standard for Image Compression.* Wiley, 2005.