Generating
RT-CORBA Components
from Service Specification

Abschlussarbeit

Oliver Müller
Generating
RT-CORBA Components
from Service Specification

Abschlussarbeit

Oliver Müller

Aufgabensteller : Univ.-Prof. Dr. Manfred Broy (TU München)
Betreuer : Prof. Ingolf Krüger (UC SanDiego)

Abgabedatum : 15. Oktober 2003

1email: muellero@in.tum.de
Hiermit erkläre ich, dass ich diese Abschlussarbeit selbständig und nur mit den angegebenen Hilfsmitteln angefertigt habe.

München, den 15. Oktober, 2003

______________________________
(Oliver Müller)
Abstract

The development of high-quality, distributed, reactive systems is still a daunting, error-prone task. Although many of today's CASE tools offer significant support for model-driven development and some even for formal analysis of the resulting models, the transition to executable code is typically performed manually. This manual step in the development process remains difficult, despite the existence of middleware technologies that relieve the developer from many implementation technicalities of distributed systems. Our goal is to bridge the gap between abstract models of reactive systems and their implementation. To that end, we present an approach to code generation starting from validated or verified models of distributed, reactive systems specified using the AutoFocus CASE tool, and targeting RT CORBA as a middleware platform.
# Contents

Preface

1 Introduction

2 Background and Motivation
   2.1 Reactive and Embedded Systems
   2.2 Distributed Real Time Computing
      2.2.1 Difficulty in Development Process
      2.2.2 Real Time Aspects
   2.3 Model Based System Development
   2.4 Use of CASE Tools
   2.5 Implementation Infrastructures: RT Middleware
   2.6 Contributions and Caveats
   2.7 Running Example: Abracadabra

3 Real Time CORBA
   3.1 From CORBA to RT CORBA
   3.2 RT CORBA Features
   3.3 Development Process and Artifacts
      3.3.1 Interface Definition using IDL
      3.3.2 Client and Server Implementation
   3.4 Available Implementations
      3.4.1 TAO
      3.4.2 ZEN
   3.5 Examples for Applications using RT CORBA

4 AutoFocus on Reactive Systems
   4.1 Software and System Architecture Specification
      4.1.1 Purpose
   4.2 Modeling Concepts and Execution Model
   4.3 Views and Models
      4.3.1 Structural View
      4.3.2 Behavioral View
   4.4 Validation and Verification Support
Preface

Software development is an incredible fast evolving field. My background is in mechanical engineering, which is one of the more traditional engineering disciplines. A lot of the used techniques & technologies are well profen and already have a long history. The area of software development thus brought up new ways of approaching problems for me. Technologies emerge and disappear and trends seem to be much more prevalent. Yet some principles have been established in almost all areas. The diversity of ways that are and will be used to tackle similar problems yields a very creative aspect. A good example for a pervasive problem that is being dealt with in many different manners is the area of distributed computing. Today this kind of computing is increasingly taking over control in areas where problems were traditionally solved in a mechanical way (for example in modern cars). Influencing the future development of major industries with such interesting topics prepares the ground for a fascinating research area.

Outline

The topic of this thesis is part of the ongoing research of Prof. Krüger both at the University of California San Diego (UCSD) and the Californian Institute of Telecommunications and Information Technologies Cal-(It)^2. In the following pages we will discuss the motivation and the implementation of a code generator that produces code for the real time CORBA middleware. To present the material this thesis is structured as follows:

First, we provide a detailed overview of the conceptual and technical background in Sec. 2. This section contains material on reactive and embedded systems (2.1), distributed real time computing (2.2), Model Based System Development (2.3), the use of CASE tools (2.4) and middleware. We further discuss the contribution of this work (2.6) and introduce a toy running example on which we illustrate both the modeling and code generation concepts we use (2.7).

Sec. 3 contains an overview of the implementation infrastructure provided by RT CORBA. We briefly talk about the CORBA architecture and the enhancements introduced with the RT CORBA standard (3.1) and give an overview of RT CORBA (3.2). In 3.3 we talk about the usage of RT CORBA and conclude this section presenting two available RT CORBA implementations.

In Sec. 4 the CASE tool AutoFocus is introduced, which provides a basic, yet comprehensive set of modeling notations and corresponding analysis tools for distributed, reactive systems. This section contains information on general specification of software systems (4.1) and on how those specifications are represented in AutoFocus (4.2, 4.3).

For deriving an implementation of the system specification, Sec. 5 first gives an overview
of the basic challenge (5.1). The necessary details of the translation from AutoFocus models into RT CORBA executables are provided in 5.2. Once having established the mapping, we introduce the design of the code generator in 5.3. The actual generation of code is discussed in the remainder of this section (5.4, 5.5).

To show the scalability of our approach we briefly discuss a more elaborate example of a system in sec. 6 and show the results of the code generation.

Finally in Sec. 7 we evaluate and discuss our approach. Sec. 8 contains our conclusions and an outlook to further work in this area.

Acknowledgments

I more than appreciate the opportunity I had to work on such an attractive topic with Professor Ingolf Krüger at the University of California San Diego (UCSD). The work provided more than I could have hoped for in my thesis. During this 6 month path I covered a lot of terrain, got to know many different approaches and ended up with a real product as a proof of concepts.

Of course having the chance to do this abroad at a renowned university in the US made the experience even more special. But mainly because of having Prof. I. Krüger as my adviser, colleague and friend I was able to work on a topic that more than satisfied my expectations. I want to thank him for his support. With his patience, his broad but still deep knowledge and his enthusiasm he was truly inspiring.

I'm fully aware of being fortunate to have the opportunities to conduct part of my education in a foreign country and therefore would like to thank Professor Manfred Broy at my university in Munich for initially opening up the possibility to complete my thesis at UCSD. Thanks also to Michael Meisinger who advised me from the side of the Technische Universität München and greatly helped me out organizing all the administrative issues related to my thesis abroad. Thanks also very much to my girlfriend and to my family who unfortunately did not see much of me during the last 6 month. Without their support my work would not have been possible.
Chapter 1

Introduction

Distributed, reactive systems are notoriously difficult to develop. The shift from monolithic to highly networked, heterogeneous, interactive systems has led to a dramatic increase in development and system complexity. At the same time the demands for safety, reliability and other quality attributes have also increased across application domains. Suggestions for addressing the underlying challenges exist at both ends of the development spectrum: For specifying the system under development notations and methodologies for better capturing the structural and behavioral requirements are available (including UML, and UML-RT, and SDL). Using suitable tools for computer aided software engineering (CASE) that take advantage of those notions and methodologies helps with the specification of the systems. On the other end there exists a variety of middleware technologies (including CORBA, RT CORBA, and .NET) that aid the development of distributed systems. The focus is shifted away from the distribution aspect towards the actual functional requirements. Most of the complexities of developing distributed implementations for heterogeneous hard- and software platforms are simplified and the likelihood of errors reduced.

However, a significant gap still exists between the modeling notations and tools for requirements capture, and the implementation based on a middleware platform. Even if all key requirements of the system have been modeled, and tools for checking the correctness of these models have been successfully applied, the step from the abstract model to a concrete implementation is a difficult, typically manual, and error-prone task.

In this thesis we present an approach at bridging this gap in the context of distributed, reactive, embedded systems. To that end, we have developed a code generator whose purpose is to take models of reactive systems and translate them into executable code for the RT CORBA middleware.

While code generators often are part of CASE tools such as Rational Rose, Rose RealTime, and Telelogic Tau, they typically target only one implementation platform, consisting of specific programming languages, operating systems, or devices. To the best of our knowledge, the code generator presented here is the first one to link a modeling and validation tool directly with the RT CORBA middleware.

Although the work in this thesis specifically targets RT CORBA as the middleware, the presented approach easily generalizes to other middleware platforms for distributed, reactive systems.
Chapter 2

Background and Motivation

The observations sketched above hold in technical domains, such as telecommunications, automotive, avionics and ubiquitous computing, but also in complex web-enabled business applications. Automotive software systems illustrate the underlying challenges particularly nicely. Software has become the enabling technology for almost all safety-critical and comfort functions offered to the customer in the automotive domain. 90% of all innovations in automotive systems are directly or indirectly enabled by software. Today’s luxury cars contain up to 80 electronic control units (ECUs) and five different, interconnected network platforms, over which some 700 software-enabled functions are distributed. The complexity induced by this large number of functions, their interactions, and their supporting infrastructure has started to become the limiting factor for automotive software development. The situation is aggravated further by demanding time-to-market requirements, short development cycles, and rapid change of technological infrastructures, customer demands, and product lines.

These challenges, shared to a large extent by all the application domains mentioned above, illustrate the need to go beyond ad-hoc system implementation if we want to establish confidence in the quality of the resulting software products in safety-critical, heterogeneous, distributed computing environments.

In the following sections we take a closer look at two complementary approaches to reducing the complexity of the development process for distributed, reactive systems. Model-based development addresses complexity by abstracting away from implementation details at early stages of software development. Middleware such as RT CORBA, provides an abstraction layer on the level of implementation; this layer handles, among others, message/event exchange in adequate data formats, task scheduling, component discovery, and even QoS\(^1\) properties transparently across different target devices and operating systems.

While each of these techniques provides means for both increasing the confidence in the resulting system, and reducing the complexity of the development task, a significant gap exists between the two ends of the development spectrum: capturing of requirements and abstract modeling on the one hand, and system implementation on top of a powerful middleware technology on the other.

\(^1\)Quality of Service
2.1 Reactive and Embedded Systems

Getting more and more pervasive, computer processors are integrated into all kinds of different hardware: automobiles, process control, medical devices etc. Systems that are made up of these kind of processors are said to represent reactive systems.

Unlike transformational systems the reactive systems keep running infinitely once they got started. From that moment on they are either waiting to react to a signal from the environment or are processing such a signal. Transformational systems, however, do not run in between invocations. They start to process a request as soon as it arrives and halt after they finished their execution. Compilers would be an example for such a system.

The kind of systems we address in this thesis are the technical systems. As opposed to business systems like for example a content management system that focuses on data storage and manipulation, the technical systems deal with the control of technical processes like for example an ESD (Electronic Skid Detection) for automobiles.

Embedded systems are a special form of technical system whose prevalence is already omnipresent. Embedded Systems are modules that are composed of hard- and software. They interact with their environment using sensors and actuators and mostly control technical processes (for example climate control systems). Once running they usually do not require any interaction with the users. Integrating those systems within a given hardware introduces a lot of additional constraints like e.g. limited space, limited processing power, or energy consumption. [21]

2.2 Distributed Real Time Computing

A large part of all software systems today can be considered distributed in a way that components that are not collocated interact with each other. This is conceptually similar to the paradigm of object orientation where different objects can access others via method calls. In addition to the challenges of designing the system to fulfill its functionality, distributed systems also have to overcome the difficulties of effective and reliable communication. Several approaches have emerged that all aim towards simplifying the development of such systems. A few examples of middleware that are used include Distributed Component Object Model (DCOM), Java Remote Method Invocation (RMI), CORBA (Common Object Request Broker Architecture) or Enterprise Java Beans (EJB). Even though each specific realization of such a middleware might solve the basic distribution problems in a different way and address slightly different problem domains, they all focus on the same idea: allowing an application to run on distributed nodes and enable the components to communicate with each other.

In many domains today (for example in the automotive sector) there is often another issue to tackle: real time requirements. Not only do different processing units have to communicate with each other in a correct and reliable fashion, they are furthermore required to do so within strict time limits. Those kind of real time requirements are available in almost all safety critical applications like air traffic control, telemedicine or military applications. Real time systems like that have to satisfy stringent qualities of service (QoS) like exact time constraints, minimum throughputs etc. It’s not sufficient for them to only fulfill best effort QoS. [33] [21]

As opposed to a non real time system, the correctness of real time systems does not only depend on the logical result but also on the time that is needed to produce the result. [43]. Considering such real time aspects in the development process of distributed systems
quickly brings up a variety of additional challenges that are neglected in most non-real time middlewares. Resources have to be under much tighter control and location transparency (one of the big advantages of traditional middleware) is not always a desirable property. Developers need to be able to explicitly control the allocation and the scheduling of resources like CPU, memory and networking resources to ensure an end-to-end quality-of-services support [34].

2.2 Distributed Real Time Computing

As already mentioned, the complexity of distributed systems is rising rapidly and steadily. Again, we are using modern cars as a representative example to illustrate a possible scenario. Starting out with ABS as one of the first computer controlled onboard system in automobiles, new systems are added with each upcoming version of an automobile. Since a large share of the new functionality deals with safety relevant issues, its use underlies strong regulations and has to be approved by governing authorities. Software reliability becomes much more significant and therefore the testing effort that is required for such systems increases dramatically.

In a growing number of new automobile series, the areas in which such embedded systems are applied today stretches from systems for infotainment (e.g. navigation, entertainment) over systems that help control essential functions of the car (e.g. controlling the transmission or the chassis) to systems that care for the user (e.g. voice recognition, seat adjustment). More and more functionality is partly or fully managed by ECUs that have to interact with other units. Considering the example of an apparently simple system like the central locking in a car reveals the complexity of its dependencies: it is, for example, required by law that the car unlocks in case of an accident. Therefore the locking system has to be coupled with appropriate sensors that detect such a situation. Furthermore it might be desirable that the locking system interacts with the climate control system. Or that it is able to recognize the driver that unlocks the car to arrange for individual adjustments of seat position etc. All participating control units are embedded and therefore closely tied to specific hardware. Their location is spread all over the car.

Taking such a rising complexity into consideration it is understandable that the distributed computing units in a car are not to be seen as independent islands but rather as parts of a highly networked and integrated system. Addressing the development of such systems in a structured and formal way is necessary if arduous and error prone ad-hoc manners are to be avoided.

2.2.1 Difficulty in Development Process

As already mentioned, the complexity of distributed systems is rising rapidly and steadily. Again, we are using modern cars as a representative example to illustrate a possible scenario. Starting out with ABS as one of the first computer controlled onboard system in automobiles, new systems are added with each upcoming version of an automobile. Since a large share of the new functionality deals with safety relevant issues, its use underlies strong regulations and has to be approved by governing authorities. Software reliability becomes much more significant and therefore the testing effort that is required for such systems increases dramatically.

In a growing number of new automobile series, the areas in which such embedded systems are applied today stretches from systems for infotainment (e.g. navigation, entertainment) over systems that help control essential functions of the car (e.g. controlling the transmission or the chassis) to systems that care for the user (e.g. voice recognition, seat adjustment). More and more functionality is partly or fully managed by ECUs that have to interact with other units. Considering the example of an apparently simple system like the central locking in a car reveals the complexity of its dependencies: it is, for example, required by law that the car unlocks in case of an accident. Therefore the locking system has to be coupled with appropriate sensors that detect such a situation. Furthermore it might be desirable that the locking system interacts with the climate control system. Or that it is able to recognize the driver that unlocks the car to arrange for individual adjustments of seat position etc. All participating control units are embedded and therefore closely tied to specific hardware. Their location is spread all over the car.

Taking such a rising complexity into consideration it is understandable that the distributed computing units in a car are not to be seen as independent islands but rather as parts of a highly networked and integrated system. Addressing the development of such systems in a structured and formal way is necessary if arduous and error prone ad-hoc manners are to be avoided.

2.2.2 Real Time Aspects

Systems that have to integrate real time constraints are usually found in the embedded area. As already mentioned their correctness depends both on the fulfillment of the functional requirements and on the meeting of real time constraints. That should not be misconceived with the statement that real time necessarily means fast execution. It is much more important to make reliable presumptions about the response time of the system. In those systems, constraints are applied to the different tasks that the system needs to execute as, for example, the computation of a certain value or a read/write access to fetch or store some data.

The nature of such constraints can be twofold: hard or soft real time constraints. The interpretation of the distinction between hard and soft real time systems is sometimes discussed controversially in literature. For this thesis we will use the term hard real time constraints to describe deadlines that cannot be missed without causing system failure. With the term of soft
2.3 Model Based System Development

real time constraints we distinguish three different types: time constraints that can be missed by a certain percentage of the time (1), time constraints that can be missed infrequently (with a maximum rate) (2) or the probability that events may be skipped occasionally (4). Since the specification of soft real time requirements is not as thorough as of hard real time constraints, it is usually more difficult to build systems with soft constraints.

To meet any real time requirements, the shared resource allocations have to be sequenced in a way that their accumulated execution time does not exceed the time constraint. The process of doing so is called Real time Scheduling. Algorithms that support such scheduling are often priority assignment algorithms. The tasks in a real time system are assigned priorities based on the timing constraint. Using this assignment it is possible to always have the highest priority task on a processor execute until completion. If during that execution a higher priority task enters the system, that task will be executed preempting the lower priority task.

Two types of scheduling algorithms are used. The first type are the fixed priority algorithms which assign a fixed priority to a task will not change. A well know example of this kind is the rate-monotonic scheduling (highest priority is assigned to the task with the shortest period [22]). The second type are the dynamic priority algorithms which may change the once assigned priority of a task if conditions in the system change. An example here would be the earliest-deadline-first algorithm (highest priority for the task with the shortest deadline [22]) [9].

In a distributed real time system the execution of tasks is not restricted to one node. Therefore it becomes possible for an overloaded node to transfer the execution of some of it’s tasks to a different node [3]. How the appropriate real time middleware deals with the scheduling will be addressed in section 3.

2.3 Model Based System Development

As an approach to dealing with the complexity of software, in particular during the analysis, specification, and design phases of the development process, model-based development techniques and notations have emerged; popular examples are the UML [30], ROOM [41], and SDL [11]. Each of these proposes managing the complexity of software development by separating the two major modeling concerns: system structure and system behavior. For both, a variety of special-purpose textual and graphical description techniques, which highlight either the structural or the behavioral system aspects, has been developed. Each model, represented in one of the description techniques, conveys one particular view (sometimes also called a projection) on the overall system. In this sense, models abstract away from some features of the system under consideration and thus highlight others.

In fact, some of the description techniques featured, say, in the UML are so abstract that code cannot readily be generated from them. An example of this is the use case notation, which is by definition geared toward informal requirements capturing. Other notations, such as class diagrams, are sufficiently close to implementation concepts and can easily be mapped to suitable programming languages. This explains why most CASE tools supporting the UML and similar notations provide little code generation support other than for turning class diagrams into prototypic class implementations in some programming language [36]; the behavior – at least large parts of it – of class instances typically has to be hand-coded, and manually modified whenever changes to the class diagrams occur.
2.4 Use of CASE Tools

Some modeling languages and corresponding CASE tools support structural and behavioral specifications at a level of detail enabling code generation; notable examples are Rational Rose RT, Rhapsody, Tau, and AutoFocus, among others. The authors of [5] provide an overview of these and numerous other CASE tools and the challenges they face especially in the automotive domain.

Of the mentioned CASE tools we concentrated on AutoFocus [14]. First, it constitutes a freely available research platform, which facilitates experimentation. Second, AutoFocus provides comprehensive support for consistency checking, verification, and validation of specifications. By means of consistency checking we can highlight specification errors, such as mismatches between syntactic component interfaces. AutoFocus also provides connectors for plugging in model-checkers (which enable automatic checking of properties for finite-state models), and theorem provers (which enable interactive verification even of models with infinite state spaces). Furthermore, extensive testing support exists for AutoFocus models.

2.5 Implementation Infrastructures: RT Middleware

Building a distributed system on top of heterogeneous computing and networking platforms introduces new difficulties to the development process: a communication infrastructure has to be enabled, data has to be marshaled and unmarshaled, exception handling has to be established – just to mention a few of the many challenges arising in this context. Since there is a lot of functionality required across all distributed applications, middlewares such as CORBA or J2EE [25] have been introduced, initially mainly targeting the business domain.

Because the heterogeneity of embedded systems is even more dramatic, and their quality requirements are even more demanding than those of most business systems, corresponding middleware approaches are becoming increasingly popular here as well. The specification and correct implementation of functional and nonfunctional (Quality-of-Service) properties, however, remains challenging despite middleware support [37, 39].

2.6 Contributions and Caveats

The main contribution of this thesis is the presentation of a code generator, which takes abstract, validated models of distributed, reactive systems as input and produces executables for the RT CORBA middleware implementing the properties checked for the abstract models. This closes a gap in the development process for reliable distributed and reactive systems by eliminating the manual transition from captured requirements to implementation on top of RT CORBA. The design of the code generator was engineered in a way that it can easily be adapted to different input languages and target middlewares.

The generated executables exploit the advanced event handling mechanisms of RT CORBA; implementations can make use of the existing features for scheduling and QoS within RT CORBA. Specification and implementation of elaborate, abstract real-time properties, however, is supported neither by the modeling tools available to us, nor by the existing RT CORBA implementations. We do not claim, therefore, that we translate real-time constraints into corresponding scheduling instructions for the middleware. Once support for
this is available within the middleware (as announced for TAO [40], an real time ORB), our code generator can be easily adapted accordingly.

Furthermore, we do not provide a formal proof of the correctness of the code generator itself. Instead, we show in detail how concepts of AutoFocus models (which do have a precise semantics, cf. [36]) map to corresponding concepts of RT CORBA, such that the resulting executables implement the AutoFocus specification.

2.7 Running Example: Abracadabra

We illustrate the concepts of RT CORBA, AutoFocus, and the actual process of code generation by means of a simplified version of the Abracadabra communication protocol[6, 7]. While this example is certainly of toy character it suffices for the purposes of explanation in this thesis.

![Figure 2.1: The Abracadabra components](image)

In modeling this protocol we assume given a system consisting of two distinct components \( X \) and \( Y \); we assume further that these two components communicate via messages sent along channels \( xy \) (from \( X \) to \( Y \)), and \( yx \) (from \( Y \) to \( X \)). Fig. 2.1 shows this component structure in graphical form.

The symmetric Abracadabra-protocol describes a scheme that allows any of the two components to establish a connection to the other component, send data messages once a connection exists, and tear down an existing connection it has initiated. If both components try to establish a connection simultaneously, the system is in conflict. Then, both components tear down their “attempted” connections to resolve the conflict.

In fig.2.1 a scenario of the Abracadabra communication is depicted. In fig.2.1(a) \( X \) sends a request \( xyz\text{-}sreq \) (“sending requested”) to \( Y \). Upon receipt of \( Y \)'s reply \( yxz\text{-}sack \) (“sending acknowledged”), \( X \) sends an arbitrary, finite number of \( xyz\text{-}d \) messages. Each data message is acknowledged individually by \( Y \); \( X \) waits for a \( yxz\text{-}dack \) message from \( Y \) before sending the next data message. Graphically, repetition is indicated by the loop box enclosing the recurring messages. To close the transmission \( X \) sends message \( xyz\text{-}e\text{-}req \) (“end requested”) to \( Y \); \( Y \) acknowledges transmission termination by means of a \( yxz\text{-}e\text{-}ack \) (“end acknowledged”) message. Fig. 2.2 (b) shows the symmetric case, where \( Y \) is the initiator. Conflict in the Abracadabra protocol occurs if both \( X \) and \( Y \) try to establish a connection simultaneously. The MSC in Fig. 2.2 (a) captures this case by means of causally unrelated messages, using the “parallel box” syntax of MSC-96. To handle conflicts both components request to tear down the connection they have tried to initiate, as indicated in Fig. 2.2.

Every system execution is an infinite sequence of steps, where each step’s behavior is described by one of the following: MSC \( SX \) (successful communication initiated by \( X \)), MSC
SY (successful communication initiated by Y), or MSC C (conflict, followed by conflict resolution).
Chapter 3

Real Time CORBA

CORBA (Common Object Request Broker Architecture) [27] has been successfully applied as a middleware solution for distributed system development in a variety of application domains since the early 1990s; it has had a major impact in the business domain, and has served as a reference architecture for more recent middleware technologies, such as J2EE’s NetBeans. Since it is independent from implementation languages, computing platforms and networking protocols it facilitates the development of new distributed applications as well as the integration of new components into legacy systems.

RT CORBA [28] has been introduced as an OMG\(^1\) standard as well, and has been included in the CORBA specification since version 2.4. Today it has a strong momentum as a middleware approach for distributed, reactive, and embedded systems with demanding quality requirements. RT CORBA provides, in particular, better support for event handling, task scheduling, and management of end-to-end QoS properties than core CORBA. In the following paragraphs we outline the basics of CORBA and RT CORBA, as well as their associated development processes and artifacts. For the rest of this chapter we will use the term CORBA when talking about CORBA without real time features and RT CORBA otherwise.

3.1 From CORBA to RT CORBA

CORBA, as standardized by the OMG, provides a transparent remote procedure call (RPC) mechanism, enabling software components running on distributed, heterogeneous soft- and hardware-platforms to inter-operate. In particular, CORBA achieves a strong form of decoupling between callers and callees in RPC communication [1]. In addition, CORBA provides standardized sets of basic middleware services, supporting persistence, component naming and discovery, event management, concurrency control, and transaction management, to name just a few examples.

RT CORBA, included in the 2.4 version of the CORBA specification, caters specifically to the requirements in the domain of embedded real time systems. Here, end-to-end QoS requirements for application with real time constraints are essential. RT CORBA allows application programmers to specify exact real time constraints and provides infrastructure for implementing them predictably across heterogeneous devices and their (real time) operating systems.

\(^1\)Object Management Group
3.2 RT CORBA Features

The main goal for the RT CORBA middleware was to achieve end-to-end predictable behavior throughout a distributed, heterogeneous system. For that it is inevitable to gain control over the following system resources: processing resources, communicating resources and memory resources.

As mentioned above, RT CORBA makes it possible to introduce a consistent priority scheme for a distributed system, thus allowing priorities to be propagated from one address space to another. RT CORBA furthermore introduces a standard thread pool model that allows the
definition of thread pools for different types of service. Using that model it becomes feasible to preallocate a certain number of threads, preventing the need for the more costly dynamic allocation. With usage of multithreading it also becomes possible to support preemption and prevent unbounded priority inversion [2]. Thread pools can also be partitioned into “lanes” of certain priorities, enabling developers to control the work that is performed at a given priority level. This partitioning also prevents situations where all available threads in a threadpool become occupied with low priority tasks. Since these low priority tasks will all have to use a certain lane, the other lanes will stay clear for higher priority tasks. The time overhead that is affiliated with preemption and reassignment that would have to be sacrificed in order to allow a higher priority task to execute is therefore avoided.

With RT CORBA’s buffer management, application programmers can specify if buffering client requests should be possible at all and, in case it is, predetermine the buffer size. The threading model introduced with RT CORBA also made it necessary to define a standard way of synchronizing operations. This is particularly important to ensure semantic consistency between applications and the middleware.

Two other significant features of RT CORBA touch upon network protocols and connections. First, RT CORBA allows the definition of client and server protocol policies with the intention of selecting and/or configuring the necessary network protocols. Therefore it is possible to configure custom protocols and obtain adequate end-to-end qualities on network level. Second, a major drawback of CORBA in the face of real time computing is overcome by eliminating the need for implicit binding. For real time systems it is desirable to avoid latency and jitter as much as possible. RT CORBA permits explicit binding and thus supports the pre-establishment of connections and enables private connections.

The RT CORBA specification is quite complex but provides features that are vital for making it possible to distribute real time applications with the middleware. For a much more complete description of the features and technical details, please refer to the OMG specification [28].

3.3 Development Process and Artifacts

To better understand the path of generating code using RT CORBA as a middleware, we briefly discuss the general process of designing an RT CORBA application. This process consists of the five steps outlined in Fig. 3.2.

CORBA relies on a distributed object oriented paradigm; interfaces serve as a contract between servers and their clients. CORBA comes equipped with an interface definition language (IDL). Mappings to a wide variety of programming languages (including C, C++, and Java) are available. A central step in application development, therefore, is to separate the system under consideration into a set of distributed components, and to specify their interfaces in the syntax of IDL.

Once the interfaces have been specified, an IDL compiler translates them into the target programming language for client and server components. It furthermore provides corresponding stubs and skeletons (CORBA terminology for client and server proxy implementations, respectively) in the respective target programming languages.

The developer then writes implementations for the methods specified in the component interfaces, filling in templates also provided by the IDL compiler. After these implementations have been compiled and linked for the respective target platform, the corresponding
executables can be deployed, and started on the target machines. Once started, CORBA components have access to the features provided by the ORB, for discovering other components, and performing remote method calls.

The preceding steps are common for both CORBA and RT CORBA applications. For RT CORBA the developer can additionally specify real-time constraints in the form of scheduling priorities – these priorities have to be predetermined so far, but can be added to a given implementation by adjusting scheduling parameters. The primitives that RT CORBA offers are sufficient to enforce real time requirements but the effective use of those is rather complicated and non intuitive. To implement a certain scheduling policy (for example rate monotonic scheduling), RT CORBA primitives have to be applied consistently throughout the whole system. The scheduling service helps the developer here and abstracts away from some low-level constructs [28].

The interface of the real time scheduling service as specified by the OMG is shown below (using IDL):

```idl
// File: RTCosScheduling.idl
#ifndef _RT_COS_SCHEDULING_IDL_
define _RT_COS_SCHEDULING_IDL_
#include <orb.idl>
#include <PortableServer.idl>
#pragma prefix "omg.org"
// IDL
module RTCosScheduling {
    exception UnknownName {};

    // locality constrained interface
```
interface ClientScheduler {
    void schedule_activity(in string name)
    raises(UnknownName);
};

// locality constrained interface
interface ServerScheduler {
    PortableServer::POA create_POA(
        in PortableServer::POA parent,
        in string adapter_name,
        in PortableServer::POAManager a_POAManager,
        in CORBA::PolicyList policies)
    raises( PortableServer::POA::AdapterAlreadyExists,
              PortableServer::POA::InvalidPolicy );

    void schedule_object(in Object obj, in string name)
    raises(UnknownName);
};
#endif // _RT_COS_SCHEDULING_IDL_

Real time CORBA introduces support for predictably managing these constraints across the entire implementation infrastructure. To that end, RT CORBA provides comprehensive thread pool management and similar techniques to enable observation of performance guarantees.

Despite the substantial relief that CORBA and RT CORBA provide for implementation and deployment of distributed, reactive systems, the development process using these technologies is still far from trivial [19]; the correct configuration of distributed components, and their mapping onto RT CORBA processes and tasks remains difficult. Also the infrastructure support cannot guarantee, per se, the correct functioning of the resulting application. To address these issues we automate the development process by generating code from abstract structural and behavioral models provided by CASE tools, such as AutoFocus. These models can first be verified or validated, and then reliably turned into corresponding implementations.

3.3.1 Interface Definition using IDL

One of the main merits of the CORBA middleware is to provide application programmers with interfaces so they can access location transparently. The Interface Definition Language (IDL) defined by the OMG is the abstraction mechanism that separates interfaces from their implementation. It is a purely declarative language and thus does not support the specification of behavior or data state. These interfaces defined in IDL serve as contract between servers and their clients. The description of the interfaces is programming language independent and thus allows for scenarios where clients and servers are implemented in different languages. A mapping of IDL constructs to programming language entities is standardized for a large variety of languages like Java, C, C++, Smalltalk, COBOL and Ada. The IDL interface definitions are compiled using an IDL compiler and are translated into language specific type definitions and APIs. Once servers are implemented, the clients can access their interfaces without knowing which language they are implemented in [23].
3.3.2 Client and Server Implementation

In a CORBA system the components that offer a service are called servers and components using a service are called clients. Nevertheless a component can both act as a server to other components or as a client of other servers.

If a client wants to use some sort of service offered by the server, it first has to obtain an object reference for a CORBA object that it then can use. Holding on to such an object reference the client can then invoke requests and receives replies in the same manner it would do using a local implementation.

Since CORBA objects are only virtual, it first has to be incarnated by an instance of a servant class. That is, a class that implements the IDL interface and is written in a real programming language like C++. Once the CORBA object is incarnated, clients can access its services. To start up a CORBA server it is necessary to perform some preparing steps like initializing the ORB and the POA2, create servants that incarnate CORBA objects and start listening to requests [23].

3.4 Available Implementations

TAO and ZEN [20] are two implementations of the RT CORBA standard, covering most of its features; in the following sections both of them are described.

3.4.1 TAO

The ACE Orb (TAO) [40] is a high performance C++ implementation for RT CORBA which is being developed at the University of Washington. It is based on the Adaptive Communication Environment (ACE) [38]. ACE is an open-source object-oriented framework that provides facilitated access for a lot of software tasks related with communication issues. Making extensive use of patterns for concurrent software development, it abstracts away from many proprietary OS-features and offers a lot of reusable C++ wrapper facades and framework components (see figure 3.3). It is furthermore available throughout a broad range of operating systems which makes it particularly useful for developing applications that have to run on many different OS platforms.

Building TAO on top of ACE has the advantage that it becomes available on all platforms supported by ACE. It is compliant with most of the features and services defined in the CORBA 3.0 specification (which includes the RT CORBA specification). Figure 3.3 illustrates the main components of TAO: The IDL-Compiler implements the mapping from IDL to C++ and generates the stubs and skeletons needed for CORBA applications. The ORB Core which provides predictable communication and enables the implementation of different concurrency models. The Portable Object Adapter that provides constant-time lookup of servants, and an Inter-ORB Protocol Engine that supports both static and dynamic programming models.

TAO also supports many standard CORBA services such as the naming service that can be used to perform lookup of object references or the real time event service that decouples multiple suppliers and consumers by using events rather than remote function calls.

The Scheduling service as defined by the OMG was not yet fully implemented in the version used for the code generation ([26]) but TAO already allows the specification of real time
Figure 3.3: TAO on top of ACE
requirements via the real time event service([44]). It is planned to integrate a scheduling service with the ORB in coming versions.

3.4.2 ZEN

In a more recent effort another implementation of the RT CORBA standard is being implemented at the university of Washington. ZEN is based on the Real Time Specification for Java (RTSJ) and was driven by experiences with TAO [45]. Using a variety of patterns and techniques one of the main goals of its development is to make it highly configurable and to substantially reduce the memory footprint of a usually rather heavy CORBA implementation. Although the design of ZEN looks very promising, its development process is fairly young. We decided to use the TAO implementation because of its mature state.

3.5 Examples for Applications using RT CORBA

RT CORBA is successfully applied in various safety critical applications ranging from military defense systems over industrial process control to real time avionics [4]. All those applications share the need for stringent Quality-of-Service guarantees. Another example for an application built with RT CORBA is Atriah, a massively-multiplayer online persistent world game. TAO is here used as the implementation of RT CORBA. The middleware is responsible for the communication between all participating servers. The requirements for the Atriah-application nicely illustrate the needs for features provided by this middleware: real-time QoS constraints, fault-tolerance, fail over and load balancing.

For a rich collection of examples the TAO website\footnote{http://www.cs.wustl.edu/~schmidt/} offers a lot of interesting material.
Chapter 4

AutoFocus on Reactive Systems

4.1 Software and System Architecture Specification

4.1.1 Purpose

Deriving a solid and thorough specification of a software system from the requirements is a complex task. Keeping it stable is mostly impossible since the requirements are often very erratic. Considering the goal of eventually ending up with a correct implementation of the system, the specification needs to be verified. That is, it needs to be checked if the system under construction complies with the specification.

For distributed real-time systems that are often safety critical, it is even more important to already validate models even before implementation. The use of appropriate methodologies and notions makes it possible to specify those models that are essential when it comes to the point of automatic code generation. In the following sections we will take a closer look at AutoFocus, the modeling tool already mentioned in the CASE tool section (2.4). We have selected AutoFocus as the CASE tool for our experiments because it is freely available as a research platform and provides strong modeling, verification and validation capabilities. AutoFocus was developed at the Technical University of Munich [15, 17], based on the semantic model of Focus [8]. Its description techniques cover system structure, data types, event traces, and state machines for reactive systems.

4.2 Modeling Concepts and Execution Model

In AutoFocus, a system consists of a set of named components, communicating by means of message exchange via directed channels. Components encapsulate data (by means of local variables), internal structure (each component can be hierarchically decomposed into a network of sub-components), and behavior. Components use named ports to communicate with their environment. Input ports allow a component to receive messages from its environment, output ports allow the component to send messages to the environment. Each port specifies the type of messages it can relay. The set of its input and output ports constitutes a component’s interface.

Channels, which are also named and typed, connect corresponding ports of communicating components. The types of a channel and the ports it connects have to match.

Components are either terminal or hierarchically decomposed (but not both). Termi-
4.2 Modeling Concepts and Execution Model

Figure 4.1: Typical Modeling Session in Autofocus
nal components have an associated state machine (technically it is an extended finite state transducer), consisting of a finite set of control states, and transitions between states. If a component is hierarchically decomposed, all its input ports must be connected by channels to corresponding input ports of its subcomponents, and symmetrically for its output ports.

Execution of an AutoFocus model is governed by a discrete global clock. During each clock cycle, each component reads all its input ports, selects and performs an enabled transition, and writes its output (if any) to the corresponding output ports. This output is transmitted to the respective communication partner’s input port between the current and the next clock cycle; in other words, there is a delay of one time unit (or tick) between the sending and the receipt of a message. This execution scheme prevents causal loops in AutoFocus models. Port contents that are not consumed by the receiver during a clock cycle are thrown away [32].

4.3 Views and Models

AutoFocus provides graphical description techniques and corresponding editors for specifying system structure and behavior. In the following paragraphs we highlight the models we use in the process of code generation using the Abracadabra protocol introduced in Section 2.7. We refer the reader to [15] for a comprehensive explanation of other AutoFocus views and models, such as data types and event traces.

4.3.1 Structural View

In AutoFocus the system’s components and their interconnection are depicted using System Structure Diagrams (SSDs). Figure 4.2 shows an SSD for the Abracadabra protocol. Labeled rectangles represent components, labeled arrows represent channels, and filled and outlined circles on either end of a channel denote input and output ports, respectively. Terminal components are labeled with an encircled “A” to indicate that they have an automaton associated with them.

![Figure 4.2: Structural view of the Abracadabra System](image)

4.3.2 Behavioral View

To model component behavior, AutoFocus offers State Transition Diagrams (STDs) that can be used to capture control states, state transitions and local data. Fig. 4.3 shows an STD for the Abracadabra example. Each node in this STD corresponds to a control state; the initial state is labeled by a black dot. Transitions are represented by labeled arrows (between distinct states) or labeled lines (if the source and target state coincide). Transition labels are of the form “[precondition][input pattern][output pattern][postcondition]”. Here, the precondition
is a predicate on the data state of the component. The input pattern indicates values that have to be present at the corresponding input ports for the transition to be enabled. The input pattern is a list of entries, where each entry is of the form “p?v”, where “p” denotes a port name, and “v” the expected value at this port. A transition fires only if both the precondition holds, and the input pattern is satisfied. The output pattern indicates which values are written to the component’s output ports when this transition is taken. It is a list of entries, where each entry is of the form “p?v”, where “p” denotes a port name, and “v” the value to be written to this port. The postcondition indicates what values the data variables are expected to have after the transition is completed.

Similar to AutoFocus components, states can be hierarchically decomposed; a decomposed state $DS$ consists of an entire STD $S$ itself, such that each incoming and outgoing transition of $DS$ is connected to a corresponding state of $S$.

4.4 Validation and Verification Support

AutoFocus provides sophisticated support for model simulation, verification, and validation [31]. The CASE tool has a built-in code generator targeting Java, which allows step-by-step execution and simulation of the model under consideration. While this is in general inadequate for validating real-time properties, this can be a valuable means for exploring the captured functional properties during model elaboration.

There are also connectors to tools for model-checking (SMV), bounded model checking (SATO), and theorem proving (VSE) [42]. In the Abracadabra example, for instance, we could use SMV to check that no data is exchanged between the two components between detection of a conflict and its resolution [16]. By means of these model-checking or theorem proving tools we can formally verify functional requirements of the system under consideration.
Furthermore, AutoFocus provides a number of automated and semi-automated testing approaches, which facilitate conformance tests between expected and actual interaction sequences in executions of the system under development.

These simulation, verification, and validation tools within the AutoFocus CASE tool framework enable the developer to either formally prove correctness of a model with respect to certain requirements, or at least gain additional confidence that the model captures the intended requirements. This is the motivation for taking AutoFocus models as the input for our code generation approach, which we describe, in detail, in the following section.
Chapter 5

Design and Implementation of the AFRTC Code Generator

5.1 Overview

As examined in the preceding chapters we now have all necessary tools that need to be applied for modeling and implementing distributed real time systems. In this chapter we will now explain the process of generating code and take a closer look at the design and implementation details of the code generator.

Figure 5.1: Toolchain
The steps that are necessary for modeling and code generation are depicted in Fig. 5.1. Starting point is the CASE tool (for us AutoFocus) which is used to establish a formal specification and validation of the system under consideration. This specification is exported by AutoFocus to an XML file. Now the code generator is activated the first time and the XML file is used to populate a symbol table. The code generator furthermore produces an IDL file containing the interface specifications for all terminal components of the AutoFocus model. The IDL file is then fed into the IDL compiler which is part of the RT CORBA implementation; this compiler produces all the relevant stubs, skeletons, and “empty” CORBA component implementation prototypes. Here the code generator comes into play again: It is fed the output of the IDL-compiler and populates the files with code obtained from translating the behavioral aspects contained in the AutoFocus model, and with code that takes care of other runtime configurations discussed later. The final output is code runnable on the RT CORBA middleware and ready for distribution.

Although we have geared these steps toward the combination of AutoFocus and RT CORBA, specifically to its TAO incarnation, our toolchain can be adapted easily to other input languages and target middleware platforms.

5.2 Mapping of AutoFocus concepts to C++ RT CORBA

As our target RT middleware we have selected TAO, developed at the University of Washington, University of California at Irvine, and Vanderbilt University. As a consequence, our target programming language is C++. In the following paragraphs we describe the mapping between the AutoFocus modeling concepts introduced in Sec. 4 (components, ports, channels, and state machines) and C++/RT CORBA.

Components: Mapping components with classes was the most apparent and straightforward issue. Considering the basic characteristics of objects as entities encapsulating data and behavior, the components obtained from the AutoFocus model directly match the class concept and RT CORBA’s component model. Every AutoFocus component is mapped to a class containing the component’s behavior and data.

Channels and Ports: In order to establish a mapping for channels and ports it is important to carefully consider the execution model of AutoFocus again. Recall that all AutoFocus components are driven by a discrete, global clock. During each clock cycle, each component reads its input ports, selects and executes an enabled transition, and writes its output ports. Between clock cycles the content of output ports of sender components is transferred to the corresponding input ports of the respective receiver.

To maintain this execution model in the RT CORBA implementation we introduce a scheduler, which plays the role of the discrete, global clock. We equip each RT CORBA component implementing an AutoFocus component (and the corresponding IDL interface) with two methods: `tick()` and `move()`. During each clock cycle the scheduler calls upon each component’s `tick()` method; as a result, each component selects an enabled transition, executes it, and changes state. Once all components have finished their state change, the scheduler calls upon each component’s `move()` method; this results in each component writing the content of its output ports into the respective receiver’s input ports. If an output port was not written
during `tick()` it is empty; consequently the “receiving” component’s corresponding input port is empty after the subsequent `move()`.

\textit{AutoFOCUS model}

\begin{center}
\begin{tikzpicture}
  \node (x) at (0,0) [shape=rectangle, draw] {X};
  \node (y) at (2,0) [shape=rectangle, draw] {Y};
  \draw[->] (x) -- (y);
\end{tikzpicture}
\end{center}

\textit{C++ representation}

\begin{center}
\begin{tabular}{|c|c|}
\hline
X & Y \\
\hline
private _x_out & private _y_in \\
move(); & writePort_y_in(value); \\
tick(); & move(); \\
tick(); & tick(); \\
\hline
\end{tabular}
\end{center}

Figure 5.2: AutoFocus/C++ Mapping

To explain how ports and channels are implemented we consider without loss of generality an AutoFocus component \( C \) with one input port \( i \), and one output port \( o \). \( C \) controls only its output ports, its input ports are controlled by its environment. Thus, we implement \( i \) as a method `writePort_i` in \( C \)'s IDL interface specification; the component sending its messages to \( C \) via \( i \) calls upon `writePort_i` to move messages from its corresponding output port into \( i \). Output port \( o \), on the other hand, is implemented as a private member “\( o \)” of \( C \) – only \( C \) writes on it. Each channel is implemented as a directed association between components.

Fig. 5.2 shows the mapping from AutoFocus ports and channels to C++ concepts for our Abracadabra example.

\textbf{State Machines:} We implement the state machine as specified by an AutoFocus STD within the respective component’s `tick()` method. To that end, `tick()` contains a C++ `switch` statement, discriminating the current control state of the component under consideration. For each control state \( p \) we introduce a sequence of `if` statements, whose conditions correspond to the preconditions on \( p \)'s outgoing transitions. Within each such `if` statement we test for the occurrence of the right inputs on the ports specified for this transition within the AutoFocus STD. If they are present, the output ports are written according to the transition’s output pattern, the post-condition is established (by assignments to the component’s data variables), and the control state is set to the transition’s target state. Fig. 5.3 illustrates this for two states and transitions of the Abracadabra example. A segment of the corresponding content of method `tick()` is given, below:

```c++
switch (this->state){
    case _SENDINGDATA:
```
5.2 Mapping of AutoFocus concepts to C++ RT CORBA

![State Transition Diagram]

Figure 5.3: State Transition

```cpp
if (size==0)
{
    x_out.val = EREQ;
    x_out.hasValue = true;
    this->state = _Acknowledging_Teardown;
    break;
}
if (size>0)
{
    if ((pX_in.hasValue)&&(pX_in.val == DACK))
    {
        x_out.val = D;
        x_out.hasValue = true;
        size = size-1;
        this->state = _sendingData;
        break;
    }
}
break;
...
```

Environment: Ports that do not belong to a component within the AutoFocus model (see Fig. 4.2) connect the system with its environment. For transferring information into the system programmatically, the write ports need to be accessed somehow. The read-ports can be used to receive information about the state of the system. We implement such ports by introducing an explicit environment component in the RT CORBA code. This environment
component can be used to drive the system with an external program for simulation purposes, or to embed the system into a larger context. Such an environment component is very similar to regular AutoFocus components in that it has read- and write-ports. Nevertheless it does not have an automaton of its own.

<table>
<thead>
<tr>
<th><strong>AutoFocus</strong></th>
<th><strong>C++</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>components</td>
<td>classes</td>
</tr>
<tr>
<td>channels</td>
<td>associations</td>
</tr>
<tr>
<td>output ports</td>
<td>private field</td>
</tr>
<tr>
<td>input port</td>
<td>field with write access</td>
</tr>
<tr>
<td>data transportation</td>
<td>move() function</td>
</tr>
<tr>
<td>timing scheme</td>
<td>scheduler</td>
</tr>
<tr>
<td>data processing</td>
<td>tick() function</td>
</tr>
</tbody>
</table>

Table 5.1: Autofocus concepts mapped to C++
5.3 Design of the Code Generator

5.3.1 Use Cases

Knowledge of mapping between the concepts of AutoFocus and C++ is required for automating a transition of the models now specified using an XML output and docking onto the middleware. Fig. 5.4 depicts the main use case of the code generator. Ideally, the entire generation process does not require any interaction with the user. Every code generation starts with parsing the specification of the system (AutoFocus offers a convenient XML-output) and builds up a symbol table (use case parse XML Specification). Generating IDL (generate IDL interfaces) and generating the behavior (generate behavior) are further use cases that are always included in the main use case. The compile IDL use case is special because it requires the usage of the IDL-compiler as an external tool. In our current implementation this step has to be performed manually.

Filling code into the skeletons generated by the IDL compiler (fill generated skeletons) again is a mandatory step and as such included in the main use case. One use case extends the code generation, the specification of real time constraints (use case specify constraints).

Figure 5.4: Main Use Case for Code Generation
5.3 Design of the Code Generator

5.3.2 Subsystem Design

The overall design of the code generator is depicted in fig. 5.5. It consists of 4 subsystems, the `generate` subsystem, the `populate` subsystem, the `constraint` subsystem and the subsystem `utilities`. Within the implementation, those names are abbreviated.

For the activities necessary to construct the IDL file (as illustrated in fig. 5.1 on page 22), the generate package does the main job. It furthermore is responsible for generating the behavior of the components. Filling the skeletons with the appropriate code is done by the populater subsystem. For adding real time constraints to the system, the constraint subsystem has to be activated. The utilities subsystem is in charge of a lot of the tasks repeatedly needed by all other subsystems like text file modifications.

In the following paragraphs we will briefly discuss the design of those subsystems, bringing use cases into play to illustrate the functionality of its classes.

First we will discuss the subsystems of the generator that are in charge for constructing the executables which then can get deployed in a distributed fashion. Fig. 5.7 (page 30) and 5.9 (page 32) depict the use case for the general process of code generation.

*Generate Subsystem* In the generation subsystem depicted in fig. 5.6, the user activates a `Coordinator` that handles most of the tasks by itself. It creates an instance of `RTCGen` which is the centerpiece of the generator. This RTCGen in turn builds up a symbol table for the AutoFocus XML-file containing the system specification. The construction of that symbol table (`AFST`) is described in 5.4. From that point on this symbol table can be used to query the specification.

With all the relevant information available the RTCGen can now generate the behavior of the later components and construct the interfaces using IDL. The `AutomatonMaker` is then
5.3 Design of the Code Generator

Figure 5.6: Generate-Subsystem

entrust with the task of creating the behavior. It queries the symbol tree and builds the automaton for the component in question. In order to construct the IDL the RTCGen uses an instance of the IDLManager.

Figure 5.9 shows how the generated IDL file is passed to the IDL manager who takes care of activating the IDL compiler to process the IDL files.

**Populate Subsystem** As mentioned above, the current version of the code generator does without an automated IDL compiler invocation. Thus, the user has to invoke the IDL compiler after the generation of the IDL files. Then the coordinator takes over again and finishes the job. Using an instance of the Splitter class of the populater subsystem (fig. 5.8), the output of the IDL compiler, which is all contained in a single file, is split into multiple files. Each header file (*.h) gets its own file destination as well as each implementation file (*.cpp).

In a final step a Populater iterates over all separated files and completes them. For header files important attributes and include statements are added. The implementation files that were generated by the IDL compiler already contain empty function definitions. The populater accesses the behavior that was generated earlier and populates the function definitions. The components are now ready for distribution.

The ImplementationUnit class acts as a representation of C++ implementations of functions, constructors and destructors. It allows to subdivide class implementations and makes it easier for the populater to pick the code it needs to fill the skeletons.

Last class in the populater subsystem is the Coordinator. It’s purpose is to coordinate all the steps on the way from a complete system specification from AutoFocus to an executable system running on TAO RT CORBA middleware.

**Utilities Subsystem** A number of reoccurring tasks throughout all other subsystems are taken care of by the utilities subsystem (fig. 5.10). The FileModifier offers a variety of services related with modifying text files. For populating the skeletons for example it is necessary to
Figure 5.7: Usecase Generating Distributed System(1)

Figure 5.8: Populator Subsystem
replace very specific parts of the file or to insert text at a predetermined location. The FileCopier is a simple helper class that is responsible for copying tasks. For easier formatting of the generated C++ output of the generator, the class OutputFormatter can be used as a wrapper. Reaching the end of a textfile while parsing it throws an EofException that is needed by classes of the populator subsystem.

**Constraint Specification** Although this use case is not yet implemented in our version, a possible design for a subsystem implementing this use case is presented in fig.5.11. In this design, a ConstraintManager is responsible for enforcing user defined constraints. One of the next steps of development include the graphical specification of real time constraints in form of annotated event traces. Such event traces could be obtained with AutoFocus while executing the system in a testrun. In the constraint subsystem, an EventTracer is used to capture EventTraces. The constraint manager then uses a ConstraintSpecifier to annotate the event traces with real time Constraints.

After capturing all constraints, the constraint manager has to perform the scheduling analysis of the system tasks and to evaluate the constraints. Figures 5.12 (page 34), 5.13 (page 34) and 5.14 (page 35) illustrate a possible use case of the constraint subsystem.

For an overall view of all implemented subsystems including their classes refer to figure 5.15 on page 35.
Figure 5.9: Use case Generating Distributed System (2)

Figure 5.10: Utilities Subsystem
Figure 5.11: Constraint Subsystem
5.3 Design of the Code Generator

Figure 5.12: Constraint Use Case (a)

Figure 5.13: Constraint Use Case (b)
5.3 Design of the Code Generator

Figure 5.14: Constraint Use Case (c)

Figure 5.15: Design of Code Generator
5.4 Populating the Symbol Table

A central element of our code generator is its symbol table; it is queried frequently for information associated with identifiers contained in the AutoFocus model. All the information contained in the specification constructed with AutoFocus is made accessible through that table. A simple example for a possibly interesting query would be to determine all the components to which a given component is connected via output ports and channels.

AutoFocus exports its models in the form of an XML file which simplifies parsing efforts and further processing steps. This file contains all the elements of information required for populating the symbol table. The basic structure is always the same:

```xml
<project name = "AbracadabraProtocolShort">
   <components>
      <component id="_Mainsystem" name="Mainsystem">
         <channels>
            ...
         </channels>
         ...
         <subcomponents>
            <component id="_X" name="X">
               <ports>
                  ...
               </ports>
               <automaton name="X">
                  ...
               </automaton>
               ...
            </component>
            ...
         </subcomponents>
      </component>
      ...
   </components>
</project>
```

The Mainsystem in this file is the overall system, by using Subcomponents the system is then hierarchically structured. The Channels describe the connections of all components. For each component all Ports are listed and the automaton is described.

The AutoFocus XML output is now parsed to extract all necessary information and collect it in the symbol table. In fact, using JDOM [18] for parsing and to build a Java object model from it, together with XPath [35] for querying this object model, we are able to use the XML file essentially as the symbol table. For performance purposes and for convenient access to the information we have built a simple Java wrapper around the XML files exported from AutoFocus. Once the symbol table for the AutoFocus model is constructed, all information
can be directly accessed through the wrapper and the XML file is no longer needed. In the
design of the code generator, this symbol table is one of the central elements. As can be
seen in figure 5.6, the AFST (AutoFocus Symbol Table) is mainly used by ServerGen and
RTCGen both for deriving structural and behavioral information.
Fig. 5.16 shows the symbol table, which we called AFST, as an UML class diagram.

5.5 Code Generation

The mapping of AutoFocus concepts to C++ (see sec. 5.2) and a complete symbol table of the
specified system (sec. 5.4) allows us now to exploit the code generator described in sec. 5.3
and finally generate the code we need for the distributed system. A brief overview of the
items that need to be generated is included to clarify the objective of the generation process.

5.5.1 Overview of Generated Items

The main goal of the code generator is to automatically derive an executable implementation,
coming from an AutoFocus specification and running on RT CORBA middleware. Fig. 5.17
illustrates the process again and highlights the final product: a system consisting of

1. component implementations and their component mains,

2. the environment component and its main routine,

3. the Scheduler with the server main and

4. the communication framework (which is added as a library).

The scheduler is necessary for establishing AutoFocus’s execution model (as described in
4.2) and will also be implemented as a CORBA component. For controlling and monitoring
the system inputs and outputs an Environment Component is generated.
The server main-files of the scheduler and all components (including the environment component)
are responsible for starting up the CORBA component itself. This can include
registering the component with the naming-service, subscribing for events the component is
interested in and registering events with the event channel the component will issue. Together
with the component implementations, these are the work products that are the desired final
outcome of the Code Generator and which will enable the distribution of the system on the
middleware.

For constructing components we need to generate C++ code for the following implement-
ation units (see also section 5.2 about mapping AutoFocus to C++):

- behavior (tick()-method)
- event handling (handleEvent()-method)
- finding partner components (findPartners()-method)
- for the ports (write()-methods and attributes)
- for transferring data over the channels to associated components (move()-method)
- for the constructor
- for the destructor
```java
public class AFST {
    +getAllSubcomponents:List
    +getAllTerminalSubcomponents:List
    +getAttributes:Map
    +getAutomaton:Element
    +getChannels:List
    +getComponent:Element
    +getComponents:List
    +getConnect:Element
    +getDtdFiles:List
    +getEntryPartners:Set
    +getEntryPorts:List
    +getExitPartners:Set
    +getExitPorts:List
    +getExitTransitionSegments:List
    +getInitialState:Element
    +getInitialSubState:Element
    +getInitValue:Element
    +getInputs:List
    +getInterfacePoints:List
    +getOutputs:List
    +getPartners:Set
    +getPartners:Set
    +getCorrespondingInputPort:Element
    +getComponentForPort:Element
    +getPorts:List
    +getPorts:List
    +getPostconditions:List
    +getPrecondition:Element
    +getPredicate:Element
    +getState:Element
    +getSubcomponents:List
    +getSubcomponents:List
    +getSubStates:List
    +getSuperstate:Element
    +getTargetComponent:Element
    +getSourceComponent:Element
    +getTargetState:Element
    +getTerminalSubcomponents:List
    +getText:String
    +getTransitionSegments:List
    +getVariables:List
}
```

Figure 5.16: AutoFocus Symbol Table
5.5 Code Generation

Figure 5.17: Generated Items
5.5.2 Event based Communication

For implementing the communication between AutoFocus components (this corresponds to implementing the associations induced by the existence of channels) we exploit RT CORBA's event service. Using this service it is possible to decouple the communication between the distributed components and the scheduler while avoiding synchronous request invocations (see Observer pattern in [10] for details on this kind of architectural decoupling). Suppliers can send messages to a whole group of consumers within a single call. In this sense the event service has the role of a mediator that decouples suppliers from consumers [23]. Consumers register for certain types of events without having to know which supplier will send them, the event channel mediates on behalf of the suppliers.

![Event-based Communication Diagram]

Figure 5.18: Event based Communication

During start up each component $C$, it waits until it receives a notification (an event) from the scheduler to look for its partner components, i.e. the ones acting as receivers for messages sent by $C$. To that end, the scheduler calls upon each component's $findPartners()$ method. Within this method each component invokes the CORBA naming service to obtain references to its partner components. At the end of its lookup phase, each component sends an event to the scheduler, indicating that it is now ready to participate in clock cycles. Similarly, all components use the event service to signal that their $tick()$ and $move()$ method executions are finished.

An important feature of RT CORBA event service is the capability of event filtering and correlation. In some cases consumers have to wait for multiple events to arrive or may not be interested in all events provided by the event channel. While event filtering could also be applied using several distinct event channels that are each responsible for different events, correlation of events was not readily possible in the classical CORBA event service. If a consumer had registered for certain events, it would receive them one by one even if it was only interested when all events occurred. Since each transmission of an event to a consumer usually will trigger a remote call, the sending of each event individually causes a significant communication overhead that is not appropriate for real time systems [13].
Because RT CORBA’s event service is capable of collecting related events, and of forwarding aggregate events (stating that all or some of a specified set of events have occurred), the scheduler simply waits for a signal from the ORB that all clients have found their partners, finished their \texttt{tick()}, or completed their \texttt{move()}, respectively.

5.5.3 Generation of Server Mains

For component startup each component needs a main routine. One of the tasks for those main routines is to subscribe for events and to announce the events to the event channel that will be published by the component. Using the communication framework described later (figure 5.19 on page 44), the C++ code needed to register the distributed components for single events of interest and events of publication will look like this:

```
EventManager * eManager = _X_Servant.getEventManager();

eManager->registerEventListener((EventListener *)&_X_Servant);

// events of interest
EventType iType1 = TICK;
EventType iType2 = MOVE;
EventType iType3 = FIND_PARTNERS;
eManager->addSingleEventInterest(iType1);
eManager->addSingleEventInterest(iType2);
eManager->addSingleEventInterest(iType3);

// events to publish
EventType pType1 = STARTUP_X;
EventType pType2 = FINISHED_X;
EventType pType3 = MOVE_RETURN_X;
eManager->registerEventTypeToPublish(pType1);
eManager->registerEventTypeToPublish(pType2);
eManager->registerEventTypeToPublish(pType3);

eManager->prepareEventConfiguration();
```

An instance of the \textit{Event Manager} is used for subscribing to events of interest and registering events that will be produced. An example of using the event correlation is taken from the generated code for the \textit{Scheduler} component. For starting up the whole distributed system for each component an ORB and a POA have to be initialized. A servant has to be created and a corresponding CORBA object incarnated. Components also have to register with the naming service publish the events they are interested in. The scheduler should be informed when every component is ready. If 90%
of all components are ready or 0% does not make a difference to the scheduler. It therefore constructs a set containing all startup events of all components and registers with the event channel for the conjunction of all startup events contained in the set. Thus it will only be notified when all events have occurred:

```c
... 
EventManager->registerEventListener((EventListener *)&scheduler_Servant);

// events of interest
EventType iType1 = STARTUP_X;
EventType iType2 = STARTUP_Y;
EventType iType3 = STARTUP_MAINSYSTEM;

EventTypeSet * startSet = new EventTypeSet();
startSet->insert(iType1);
startSet->insert(iType2);
startSet->insert(iType3);
eManager->addConjunctiveEventInterests(*startSet);

eManager->prepareEventConfiguration();
... 
```

### 5.5.4 Generating Components

In section 5.2 we already talked about the derivation of an IDL specification for a given AutoFocus component. The IDL for one component will consists of providing `tick()`, and `move()` methods, as well as one `writePort_i()` method for each input port `i`.

Because IDL provides interface inheritance, we collect commonalities of AutoFocus component implementations in corresponding base interfaces. For instance, we introduce the `TickableComponent` interface as the base interface for all components participating in clock cycles and implementing the `tick()` and `move()` methods.

The scheduler’s interface contains a single method `run()` for starting the execution of clock cycles.

The following segment of the IDL file generated for the Abracadabra example illustrates the mapping from AutoFocus to IDL.

```c
... 
interface AbstractComponent{
  void findPartners();
};

interface TickableComponent : AbstractComponent{
  void tick();
  void move();
};

interface Scheduler : AbstractComponent{
  void run();
... 
```
interface Component1 : TickableComponent{
    void writeMessage_in(...);
    void writeTrigger_in(...);
};

...  

interface Environment : TickableComponent{
};

... 

The IDL compiler, invoked with the generated IDL file, generates stub and skeleton code that directly interacts with the RT CORBA middleware. For each component, we need to provide two files: A header file which contains the C++ interfaces, attributes and possibly additional language dependent information, and an implementation file that implements those interface definitions.

There are two alternatives for writing code for the skeletons that are generated by the IDL compiler: implementing the generated skeletons by inheriting from the IDL compiler output or delegating from such a derived class to a class that purely implements the component. When integrating the code into the generated skeletons, CORBA code gets mixed up with the logic. Switching to a different middleware requires more manual adaptation since all the CORBA related parts have to be adapted.

On the other hand, using the skeleton as a proxy and delegating to the real implementation yields an additional upcall that takes more time. Considering the trade-off of the two possibilities we decided to use inheritance rather then delegation since we are generating code for systems with possibly stringent time constraints. Preventing the overhead of an additional upcall might mean the difference between meeting a time constraint and breaking it. Based on the output of the IDL compiler, we populate the method definitions for findPartners(), tick(), and move() for each component: this is done based on the information contained in the symbol table, and according to the schema outlined in Sec. 5.2.

The main portion of the code for the C++ headers is produced by the IDL compiler. We have to take care of additional information like including libraries or other files and guarding against multiple includes of the same header files. At some places we also have to add namespace definitions in order to remain consistent with other generated files.

5.5.5 Generating the runtime system

With all components generated, we still need to provide a runtime system for them in order to receive an executable system. A scheduler component is in charge for establishing the right execution model and closely interacts with components in the communication framework described in this section.

The C++ Communication Framework

Because the generated components share substantial code for registering with the event service, and for sending and receiving of events, we
have developed a helper framework for event management within RT CORBA. Together with the middleware itself this constitutes the runtime environment for the generated distributed components. Figure 5.19 depicts an overview of the communication framework. Although not

```c++
    server->set (handle0, // RT_Info handle
           RtecScheduler::HIGH_CRITICALITY, // Criticality
           10, // Worst case time (in 100 nanosecs)
           10, // Typical time (in 100 nanosecs)
           10, // Cached time (in 100 nanosecs)
           50000 * 10, // Period in 100 nanosecs (= 20 Hz)
           RtecScheduler::LOW_IMPORTANCE, // Importance
           0, // Quantum (unused)
           1, // Threads
           RtecScheduler::OPERATION, // Info type
```

Figure 5.19: C++ communication framework
The BootstrapManager is responsible for initial bootstrapping tasks that are necessary for the distributed components. The main purpose of this entity is to take care of starting up a component and of initializing the naming service. It is implemented as a singleton to ensure that there is only one instance per address space. The EventManager is in charge of registering the individual components for events used during tick() and move(); it uses an EventManagerImplementation that takes advantage of the RT CORBA event service [29] [13]. The CORBA consumer and the CORBA supplier serve as proxies for event sending and reception.

**Scheduler** The Scheduler’s main tasks include the coordination of the control flow of all components and the control of system initialization. Components receive the order to perform their actions with a tick event triggered by the scheduler. With a move event, data is transferred from the out-ports to corresponding in-ports. Only if all components have completed their actions will the scheduler be informed, reacting to that event in the appropriate manner.

The scheduler code for handling those events is contained in its `handleEvent()` function:

```c++
void AbracadabraProtocolShort_Scheduler_i::handleEvent(EventType type){
    if (isSTARTUP(type)){
        // ...
        eManager->submitEvent(event, FIND_PARTNERS);
        // ...
    }
    else if (isTICKRETURN(type)){
        // ...
        makeMove();
        // ...
    }
    else if (isMOVERETURN(type)){
        // ...
        giveTick();
        // ...
    } else if (isENVIRONMENT_TICK(type)){
        giveTick();
    }
}
```

For instructing the components to perform a tick, the scheduler implements a `giveTick()` function. An event is constructed and the EventManager is instructed to deliver the events to all registered consumers.

```c++
void AbracadabraProtocolShort_Scheduler_i::giveTick(){
    allTicksReturned = false;
    Event event;
    event.name = "tick";
```
5.6 Implementation Details

Using the design we presented in the preceding sections we implemented the code generator using Java as our implementation language. Although it may seem to be more consistent to implement the generator itself in the same language it is generating the code for (C++) we decided to use Java for a variety of reasons:

- Constructing the symbol table (described in sec. 5.4) a lot of XML-parsing had to be implemented. Using JDOM for the parsing greatly facilitated the efforts here. In order to gain access to the individual elements in the AutoFocus-XML file the use of XPath also proved a very valuable tool that perfectly served our purpose.

- Before we implemented the code generator we built an example CORBA application in ad-hoc manner, slowly developing the communication framework introduced in 5.5.5. For both we used C++ since this is what the TAO implementation was built for. The support we could get here from our IDE\(^1\) (we used Visual Studio .NET) was already surprisingly good but still did not offer comparable support for establishing unit tests, perform refactorings etc. we were used from Java. Exploiting the features the Java language provides together with the support of the Eclipse-IDE streamlined the development process noticeably.

- Unit testing in C++ is less straight forward than in Java. From the beginning on we implemented new functionality by following the test-first principle. Not only gives us that confidence in the current implementation, it furthermore made it possible to carry out big refactorings that would otherwise not have been possible. The test cases also nicely illustrate the use of the functionality provided by the code generator and can therefore be used as additional documentation for the source code.

  \(^1\)Integrated Development Environment

The communication framework (fig. 5.19) was implemented in C++ for reasons already mentioned (TAO RT CORBA implementation). Nevertheless it is not restricted for C++ middleware implementation. Most of the CORBA specific functionality is encapsulated and
can therefore be replaced with different middlewares that provide similar services (a lookup-service like CORBA naming and an event service). For a cleaner separation we implemented the bridge pattern [10], thus decoupling the EventManager interface from CORBA specific implementations.

In the next section we will talk about a possible way of interfacing our system via a web interface.

5.7 Providing access to the system via webservices

As stated above it is possible to interact with the system via it’s Environment Component. When all components are generated, they make extensive use the communication framework 5.19. The environment component can now be used to insert some information into the system and to get information out again. To facilitate the use of the system, we implemented the port access for the environment using webservices. Wrapping the C++ code with this technique allows us to easily provide an user interface to the system. In a prototype we implemented a web interface using ASP.net and C#. Within the .NET-environment [24] those technologies offer a very convenient way to make use of our web service. The implementation for that web interface is of course strongly dependent on the kind of system and user preferences and thus is not being generated.
Chapter 6

Extended Example

6.1 Purpose

In the preceding sections we always referred to the Abracadabra case study (2.7). To illustrate the scalability of our approach we will briefly discuss the generation of a more complex system: The Remote Sensing Station (RSS). This case study was part of the examples in the AutoFocus CASE tool and shows a much higher degree of complexity than our initial example. In this section the models for the RSS are presented with the focus on the system structure and its hierarchy. It is not, however, the intent of this synopsis to explore any of the functional or nonfunctional requirements.

6.2 Case Study Example

Figure 6.1: Remote Sensing Station in Context
The system under consideration stretches over three hierarchical layers. The uppermost layer, as seen in fig. 6.1, contains two entities, namely one Scheduler-component (not to be confused with the scheduler we need for our runtime system) and one RSS-component. The Scheduler is a terminal subcomponent (one that does not contain other components but has an automaton) while the RSS is only a hierarchical hull that can be further decomposed into subcomponents (indicated by the encircled $D$ in the upper left corner).

Figure 6.2: Remote Sensing Station

On the next level we find 7 components (fig. 6.2). All but one are terminal subcomponents and contain automata. The BundleDist-component, however, is again only a hierarchical hull.

On the lowest level of the hierarchy (within the BundleDist-component) there are 3 more components that cannot be decomposed any more (fig. 6.3).

When generating code for this system, our code generator takes care of all hierarchical levels, correctly matching ports and channels for components on different levels. Taking for example the $statusA$-channel of type BundList in fig. 6.3 (upper right corner) its out-port is connected with the StatusSplitter and its in-port with the environment of this hierarchical level. Note that this is not the environment we use to describe interactions of the system on its boundaries. In this case it is the BundleDist component seen in fig. 6.2. Here the channel is also visible, it leaves the BundleDist component (see lower left corner) and ends up in component $A$.

When building up the components for the generated system, the code generator takes that into consideration and generates only the essential functions and attributes that are associated...
6.2 Case Study Example

Figure 6.3: hierarchical decomposed BundleDist

For the above example the generator produces 12 RT CORBA components, all with appropriate port-attributes and `writePort()` functions. All channels are correctly mapped and the hierarchical levels are dealt with in the right way. It shows that our code generator has no problems coping with any elements of complexity that are part of AutoFocus specification. For more information on this example the reader is referred to the AutoFocus-webpage.

\[1\text{http://autofocus.in.tum.de/}\]
Chapter 7

Evaluation and Conclusion

Our primary goal in this project is to bridge the gap between validated specifications for distributed, reactive systems, produced using corresponding CASE tools, and implementations of these specifications on top of adequate middleware technologies. Using AutoFocus as the CASE tool, we have established a mapping of its modeling concepts onto corresponding elements of C++ and RT CORBA. In our experiments with the code generator we have found that it indeed eliminates a significant amount of the burden in the transition from captured requirements to RT CORBA implementations.

The toy example we have used in this paper, the Abracadabra protocol, was modeled using two AutoFocus components; this results in six RT CORBA classes: one for each component implementation, one for the scheduler, one for the environment, and one each for the abstract and tickable components (implementing the corresponding base interfaces of the IDL). Abracadabra is an example with a flat component hierarchy to begin with; we have also applied the code generator to an AutoFocus model with three layers of hierarchy (and corresponding hierarchical decomposition), resulting in 12 distinct RT CORBA components of which eight corresponded to terminal AutoFocus components; the remaining four components are again those for the scheduler, the environment, and the abstract and tickable base interfaces. The event handling framework, linked to every RT CORBA executable, contributes another 10 classes.

In the following paragraphs we address a number of issues arising from our approach that help to put our results in perspective.

**Scalability:** Because we generate code only for the terminal AutoFocus components, the number of RT CORBA components produced is linear to the number of AutoFocus components. To exploit the hierarchy potentially contained in the AutoFocus model in the generated code also, we could provide deployment and configuration information indicating that all hierarchical children of a given component end up on the same computational node in the network. However, we also see good reasons for distributing the terminal components independently of their original logical grouping onto target architectures; hence the approach we have followed.

**Independence from Modeling Language and Target Middleware:** Clearly, the generated code follows closely the execution model dictated by AutoFocus; this is the price we pay for ensuring with minimal effort that the implementation is truthful to the specification
provided in the AutoFocus model. However, because of its modularity the code generator can be flexibly adapted to other input modeling languages and target middleware platforms. If we had to give up on RT CORBA’s event service, for instance, we would simply have to extend the event handling framework with corresponding capabilities. The switch to the .NET platform, to name just one example, would be very straightforward, because of its event handling facilities – of course, we would have to give up on many of the performance guarantees RT CORBA provides for us. Directly targeting specific real-time operating systems that also provide some form of RPC-like communication model would be similarly easy.

**Extensions for Real-Time Constraints:** So far, AutoFocus provides no support for specifying real-time properties and constraints. Furthermore, TAO, the RT CORBA platform we have worked with, relies on predetermined priorities for guaranteeing task schedulings that observe the real-time constraints. Support for dynamic scheduling under real-time constraints has yet to be implemented and evaluated. The idea for introducing specification of abstract real-time properties into AutoFocus is to combine exploit the notation for event traces contained in AutoFocus, to associate timing constraints with segments of these event traces, and to map them into scheduling priorities for TAO. This is an element for future work and evaluation.
Chapter 8

Summary and Outlook

Code generation is a means for bridging the gap between abstract modeling techniques, for which quality assurance techniques exist, and concrete implementations that fulfill the captured and validated requirements. In the preceding sections we have presented a specific code generator that provides a link between the CASE tool AutoFocus and RT CORBA as a middleware infrastructure for distributed system implementation.

Using the Abracadabra communication protocol as a toy example we have illustrated both the features of RT CORBA and the specification of distributed, reactive systems based on the structural and behavioral models provided by AutoFocus; this has prepared our discussion of the mapping between the AutoFocus modeling concepts and corresponding implementation elements in C++ and RT CORBA. This mapping is implemented in the code generator we have developed. To the best of our knowledge this is the first code generator targeting RT CORBA.

We have discussed the results of the code generator for the Abracadabra example, and have addressed issues of scalability, flexibility with respect to input language and target implementation platform, as well as handling of real-time constraints.

Although we believe that the current version of the code generator already goes a long way toward supporting the development of reliable, distributed, reactive systems, there is ample opportunity for future work. As examples we mention the introduction of optimizations in the code generation process, exploitation of hierarchy in the deployment of the generated components, and true support for mapping abstract real-time and other Quality-of-Service specifications for corresponding implementation strategies on top of RT CORBA.
Glossary
<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACE</td>
<td>ADAPTIVE Communication Environment</td>
</tr>
<tr>
<td>AMI</td>
<td>Asynchronous Message Invocation</td>
</tr>
<tr>
<td>APIC</td>
<td>ATM Port Interface Controller</td>
</tr>
<tr>
<td>COS</td>
<td>CORBA Object Service</td>
</tr>
<tr>
<td>COTS</td>
<td>commercial-off-the-shelf</td>
</tr>
<tr>
<td>DOC</td>
<td>Distributed Object Computing</td>
</tr>
<tr>
<td>DPCP</td>
<td>Distributed Priority Ceiling Protocol</td>
</tr>
<tr>
<td>DRE</td>
<td>distributed real-time embedded (...systems)</td>
</tr>
<tr>
<td>EDF</td>
<td>Earliest deadline first</td>
</tr>
<tr>
<td>EET</td>
<td>Extended Event Trace (Autofocus)</td>
</tr>
<tr>
<td>GCS</td>
<td>global critical section</td>
</tr>
<tr>
<td>HRT</td>
<td>Hard real-time for critical operations</td>
</tr>
<tr>
<td>ISR</td>
<td>interrupt service routine</td>
</tr>
<tr>
<td>LCS</td>
<td>local critical section</td>
</tr>
<tr>
<td>LFU</td>
<td>least-frequently used</td>
</tr>
<tr>
<td>LRU</td>
<td>least-recently used</td>
</tr>
<tr>
<td>MSC</td>
<td>Message Sequence Charts</td>
</tr>
<tr>
<td>OFP</td>
<td>operational flight program</td>
</tr>
<tr>
<td>PERT</td>
<td>Program Evaluation and Review Technique</td>
</tr>
<tr>
<td>PERTS</td>
<td>A Prototyping Environment for Real-Time Systems</td>
</tr>
</tbody>
</table>

Table 8.1: Glossary(part 1)
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCP</td>
<td>Priority Ceiling Protocol</td>
</tr>
<tr>
<td>PIM</td>
<td>Platform Independent Model</td>
</tr>
<tr>
<td>POA</td>
<td>Portable Object Adapter</td>
</tr>
<tr>
<td>PSM</td>
<td>Platform Specific Model</td>
</tr>
<tr>
<td>QoS</td>
<td>Quality of service</td>
</tr>
<tr>
<td>RAIi</td>
<td>Resource Acquisition is Initialization</td>
</tr>
<tr>
<td>RIDL</td>
<td>Real-time IDL (in TAO)</td>
</tr>
<tr>
<td>RIO</td>
<td>real-time I/O</td>
</tr>
<tr>
<td>RM</td>
<td>rate-monotonic</td>
</tr>
<tr>
<td>RMA</td>
<td>Rate Monotonic Analysis</td>
</tr>
<tr>
<td>RMS</td>
<td>Rate Monotonic Scheduling</td>
</tr>
<tr>
<td>RTU</td>
<td>Real Time Upcall</td>
</tr>
<tr>
<td>SRT</td>
<td>Soft real-time for non-critical operations</td>
</tr>
<tr>
<td>SSD</td>
<td>System Structure Diagram (Autofocus)</td>
</tr>
<tr>
<td>TAO</td>
<td>The ACE ORB</td>
</tr>
</tbody>
</table>

Table 8.2: Glossary(part 2)
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>The Abracadabra components</td>
<td>7</td>
</tr>
<tr>
<td>2.2</td>
<td>Abracadabra-protocol dynamics</td>
<td>8</td>
</tr>
<tr>
<td>3.1</td>
<td>CORBA Architecture overview</td>
<td>10</td>
</tr>
<tr>
<td>3.2</td>
<td>Developing CORBA Applications</td>
<td>12</td>
</tr>
<tr>
<td>3.3</td>
<td>TAO on top of ACE</td>
<td>15</td>
</tr>
<tr>
<td>4.1</td>
<td>Typical Modeling Session in Autofocus</td>
<td>18</td>
</tr>
<tr>
<td>4.2</td>
<td>Structural view of the Abracadabra System</td>
<td>19</td>
</tr>
<tr>
<td>4.3</td>
<td>Behavioral view</td>
<td>20</td>
</tr>
<tr>
<td>5.1</td>
<td>Toolchain</td>
<td>22</td>
</tr>
<tr>
<td>5.2</td>
<td>AutoFocus/C++ Mapping</td>
<td>24</td>
</tr>
<tr>
<td>5.3</td>
<td>State Transition</td>
<td>25</td>
</tr>
<tr>
<td>5.4</td>
<td>Main Use Case for Code Generation</td>
<td>27</td>
</tr>
<tr>
<td>5.5</td>
<td>Subsystems for Code Generation</td>
<td>28</td>
</tr>
<tr>
<td>5.6</td>
<td>Generate-Subsystem</td>
<td>29</td>
</tr>
<tr>
<td>5.7</td>
<td>Usecase Generating Distributed System(1)</td>
<td>30</td>
</tr>
<tr>
<td>5.8</td>
<td>Populator Subsystem</td>
<td>30</td>
</tr>
<tr>
<td>5.9</td>
<td>Usecase Generating Distributed System(2)</td>
<td>32</td>
</tr>
<tr>
<td>5.10</td>
<td>Utilities Subsystem</td>
<td>32</td>
</tr>
<tr>
<td>5.11</td>
<td>Constraint Subsystem</td>
<td>33</td>
</tr>
<tr>
<td>5.12</td>
<td>Constraint Use Case (a)</td>
<td>34</td>
</tr>
<tr>
<td>5.13</td>
<td>Constraint Use Case (b)</td>
<td>34</td>
</tr>
<tr>
<td>5.14</td>
<td>Constraint Use Case (c)</td>
<td>35</td>
</tr>
<tr>
<td>5.15</td>
<td>Design of Code Generator</td>
<td>35</td>
</tr>
<tr>
<td>5.16</td>
<td>AutoFocus Symbol Table</td>
<td>38</td>
</tr>
<tr>
<td>5.17</td>
<td>Generated Items</td>
<td>39</td>
</tr>
<tr>
<td>5.18</td>
<td>Event based Communication</td>
<td>40</td>
</tr>
<tr>
<td>5.19</td>
<td>C++ communication framework</td>
<td>44</td>
</tr>
<tr>
<td>6.1</td>
<td>Remote Sensing Station in Context</td>
<td>48</td>
</tr>
<tr>
<td>6.2</td>
<td>Remote Sensing Station</td>
<td>49</td>
</tr>
<tr>
<td>6.3</td>
<td>hierarchical decomposed BundleDist</td>
<td>50</td>
</tr>
</tbody>
</table>
List of Tables

5.1 Autofocus concepts mapped to C++ .............................. 26
8.1 Glossary (part 1) .................................................. 55
8.2 Glossary (part 2) .................................................. 56
Bibliography


[34] Irfan Pyarali and Douglas Schmidt. Techniques for enhancing real-time corba quality of service, 2002.


[37] D. Schmidt, F. Kuhns, R. Bector, and D. Levine. The design and performance of an i/o subsystem for real-time orb endsystem middleware.


