Robocop Components
'Trust and Simulation'

Edwin Rikkers
Master's Thesis
May 2007

Mentors: Andries Stam and Marcello Bonsangue
Foundations of Software Technology Group
Faculty of Mathematics and Natural Sciences
LIACS Institute
University of Leiden

Contents

Foreword

I was looking for a thesis project which had something to do with software development. Marcello Bonsangue and Andries Stam, members of the 'Foundations of Software Technology' group at the LIACS Institute Leiden, introduced me with the Robocop framework for embedded devices. They explained what Robocop was all about and it looked very interesting to build an application on top of the Robocop framework.

The Robocop framework is part of the Trust4All project which develops a framework for building component based software on embedded devices. The framework provides mechanisms to obtain trust in the software application and its components. Because the Robocop framework is still under development and there is not much documentation about it, I was warned it would be quite a challenge to do a project about this. But the challenge did it for me and also the potential for this kind of middleware framework, since nowadays it is quite normal to take components developed by third parties and integrate them into a software system.

The main idea was to develop an application on top of the Robocop framework and to implement a mechanism to obtain trust in this application and its components. First I had to come up with an idea for such an application. After some thinking and looking at existing case studies I came up with the idea to build a 'Car Infotainment System' and extend this with my own ideas of how a such a system should look like.

Because it was not feasible to build a Car Infotainment System into a real car, to develop a working GPS system or to build a whole media system, I decided to simulate some of these components. The idea of a 'Robocop Simulation Component' was born. In this simulation component a mechanism to obtain trust had to be implemented as well. The result can be found in this thesis.

Chapter 1

Introduction

Components play a very important role in many software systems. Nowadays it is quite common to develop components of software systems separately and to integrate them at some point to build the complete software system. This is called the 'black box' representation of components, where only the functionality of the components is known to the developer, and not its implementation. Using software components makes upgrading and extending the software a lot easier. Often a middleware layer is responsible for providing mechanisms for extension and upgrading. A main problem with this kind of component based software development is the issue of trust. According to [1][2], trust is roughly defined as the degree of confidence users have in a software system and its components . A software system and its components must confirm to a certain quality to obtain trustworthiness. Components can have different kinds of quality to conform to, depending on the kind of component and its functionality. Research in this area, as in a lot of other areas, has only touched a few main issues [3].

Nowadays a lot of projects exist involving component based software development. One of them currently being carried out in the EU is the Trust4All project which concentrates on the development of open component based software on embedded devices. The Trust4All project is the latest project in a series of three projects, it extends the Space4U project [5], which extends the Robocop project [4]. The framework developed in these three projects for creating open component based for embedded devices is called Robocop. Robocop stands for 'Robust Open Component Based Software Architecture for Configurable Devices'.

In this thesis project we developed an application on top of the Robocop framework and defined a new kind of mechanism to obtain trust. This new kind of mechanism is called 'Modes of Operation'; the ability to change the properties of a component during runtime to obtain trust in the application and its components.

The case study we have implemented is a ‘Car Infotainment System’, which consists out of different components for features like driver information, navigation and media. Because of time contraints on the project it was not feasible to really build these components. The idea was to build a simulation component, which is able to simulate the Modes of Operation mechanism to obtain trust in a simulated component.

The background, details and main goals of the Trust4All project are described in Chapter 2 including the trust issue. The Modes of Operation mechanism is described in Chapter 3. This mechanism is implemented in the ‘Robocop Simulation Component’, which is described in Chapter 4. In Chapter 5 the Car Infotainment System is described including its architecture, the different components and how trust is obtained in this system. Conclusions about this thesis project are drawn in Chapter 6. Future work is recommended in Chapter 6 as well.

Chapter 2

Background

The Trust4All, Space4U and Robocop projects are a set of three EU-ITEA [7] projects which concentrate on defining an open component-based architecture for the middleware layer in high-volume embedded appliances. The focus on high volume devices means that the software frameworks developed have to be resource efficient and lightweight, which has been the primary challenge within these projects. The goal of the projects is to enable the construction of open configurable middleware for consumer devices, focusing on the provision of extra-functional properties, such as power awareness, resource awareness, fault management, software management and trust.

All three of these projects are examples of open innovation involving Nokia Research Center, Philips Research, Fagor, IKERLAN, academia and research institutes. The Robocop project started in 2001 and ended in 2003, when the Space4U project started. The current project is the Trust4All project, which started in July 2005, when the Space4U project ended.

Each project has been structured into three work packages, corresponding to the phases of a standard project life-cycle: requirements and specification, design and implementation, and, validation and evaluation. Within each work package there are tasks which are focused on the topics required to achieve the project goals. An explanation of the three projects [8] is given Sections 2.1 to 2.3. In Section 2.4 the architecture of the Robocop framework is described. Finally the trust issue of the Trust4All project is explained in Section 2.5.

2.1 The Robocop project

The Robocop project concentrated on building a proof-of-concept system that showed how configurable middleware could be made for high-volume devices. Its primary topics were resource awareness and software upgrade. It also studied the issue of business models for component based software. The Robocop tasks were:

- Core architecture: in this task, support for a common software component was designed and implemented.
- Secure download: in this task, a framework for inserting software into a device was created.
- Resource awareness: in this task, a framework allowed the management of resource usage within the middleware platform.
- Trading and IPR support: in this task, support for software trading, component modelling and supporting business oriented activities have been studied.

Robocop ended with 14 demonstrators running on a diverse set of platforms, such as Symbian, Linux, and PsOS.

2.2 The Space4U Project

Space4U was built on the result of Robocop, extending the scope of the framework and the system. It sought to address the issues relating to consequences of the system created in Robocop. The main questions were: How to handle externally provided software, how to protect a system from poor software, how to use resource usage to provide power awareness, and how can external systems improve the software inside devices? The task list for Space4U included:

- Core architecture: in this task, the Robocop core architecture and component model were extended and improved.
- Fault management: in this task, a framework and tooling to build systems that can respond effectively to a certain range of faults, were provided.
- Power awareness: in this task, a framework for power aware software was built on the Robocop resource awareness framework.
- Terminal software management: in this task, the Robocop download process and supporting tools to allow software integrity management, software visualization and resource usage prediction, were extended.

Space4U ended with 11 demonstrators showing the use of all frameworks. These demonstrators ran on several platforms, and interoperated (e.g. remote television control on a mobile phone).

2.3 The Trust4All project

While Space4U helped companies build reliable, managed software there are some aspects that must be addressed to enable widespread use of component based middleware in devices. The most important issue is to get user trust in the technical platforms provided by manufacturers. This involves interaction between application domains, security, availability and system awareness. The Trust4All project tasks are:

- Core architecture: in this task, the Space4U architecture is extended and adapted to incorporate trust characteristics.
- Trust model: in this task, trust as perceived by users and how it relates to system properties is modeled.
- Resource awareness: in this task is investigated, how components can be combined into trusted systems.
- Standardization: in this task is investigated, how to disseminate the project results through standard setting bodies.

The Trust4All project started in July 2005 and is still running. The Robocop framework for creating component based software for embedded devices developed in these projects is described in the next section.

2.4 Robocop Architecture

In the Robocop architecture the following entities and sub-entities are distinguished [9].

- **Computer systems**
  - Hosts are the systems where components are developed, certified, tailored and integrated.
  - Repositories are hosts where components are published.
  - Devices are systems that host and execute the software that provides the total functionality of the appliance.
- **Components** embody a subset of the functionality of the middleware layer of an appliance.
  - Models: anything that conveys information about the software artefact that realizes the functionality of the component and perform the actual computations. A Robocop component consists of a set of models.
  - The manifest: a table of contents for a component.
  - Package: the visual representation of a component on a host.

  - Service: The functionality offered by a component is logically modeled as a set of services.
  - Service Instance: the instantiated service at runtime.
  - Service Manager: creates service instances dynamically.
  - Interface: provides the means for applications and components to invoke operation on a service instance through function calls.
  - Interface reference: the identification of a specific interface on a specific service instance at run time.
- **Robocop Runtime Environment (RRE)** is the embodiment of the Robocop component framework on a device.
  - Component Support: implements the Robocop component model on a given platform.
  - Registry: maintains information about the registered entities.
  - Download support: optional part of the RRE implements the Robocop download facility.
  - OS abstraction layer: optional part of the RRE that implements a basic set of kernel type functions that allow a component to be written in an OS or kernel independent way.
- **UUID** stands for a Universal Unique Identifier. Any and all Robocop objects needing unique identification are assigned an UUID.

The development framework of Robocop defines the roles of and the relations between the various entities in the development, certification, trading, tailoring and integration of Robocop components and the Robocop Runtime Environment. Component builders develop components, the development consists of developing the constituent models and the associated manifest. How components are developed is left unspecified in Robocop ('black box' principle).

A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties. Specifying a component as a set of models is a key innovation in Robocop. There are various usages for this concept including trading, composition and execution time inspection of properties of components.

2.4.1 Object and Interfaces

The programming model offered by the Robocop architecture is based upon objects and interfaces. It is only possible to interact with objects through interfaces. Interfaces to objects can be obtained by creating service instances or can be returned by functions or operations. The functionality of services is mapped onto interface definitions. These specify the operations supported by the interface and the semantics of those operations. Interface definitions are identified by an Interface ID.

2.4.2 Services

Objects and interfaces give a fine-grained level op programming. Services provide a higher level of programming. Services specify which interface instances are supported when the service is instantiated. There are two types of interfaces:

- Provides interfaces: the interfaces the service provides, which are implemented by the component itself.
- Requires interfaces: the interfaces a service requires which are provided other components in the system. A requires interface is a binding point for interfaces.

The service also provides a framework for setting properties and a framework for creation. Services have a unique ID.

2.4.3 Components

From a programming perspective, services are the largest structure that is clearly identifiable. The components almost play no role in the programming model as seen by the client of the RRE. It is the responsibility of the RRE to determine the appropriate executable component when a service instance is requested. Therefore a component also needs a unique ID. The conceptual view of a component can be found in Figure 2.1. The component contains services which contain the provides and requires interfaces. The provides interfaces are the white dots and the requires interfaces the black dots coming out of the services.

Components, services and interfaces are defined in a RIDL-file. The RIDL-compiler takes this RIDL-file to generate the ANSI-C [12] files needed for this component and its service. The compiler is part of the Robocop framework. RIDL stands for 'Robocop Interface Description Language', it is used to express the conceptual elements of the Robocop framework.

The development of a component will be described by using the following RIDL-file. This example is taken from the 'Component Model Tutorial' [10].

```
interface IMemInfo { 49ead10b-823a-4105-a087-08fc9883f080 }
{
  long getTotal();
  long getFree();
};

service SvcSysInfo { 9fcdf688-e294-4a28-ba74-a237d879ac45 }
{
  provides {
    IMemInfo mem;
  };

  attributes {
    short memSig;
  };
};
```

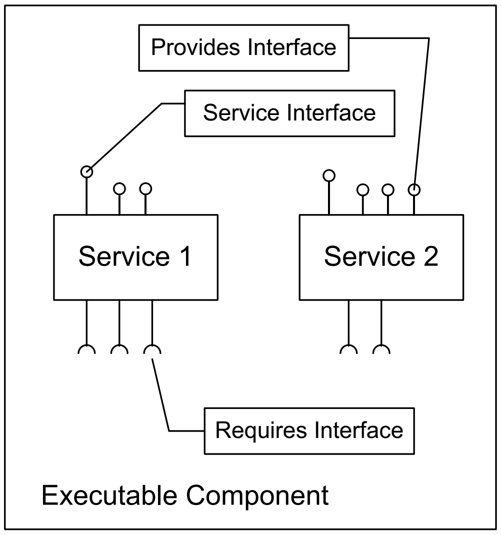


Figure 2.1: Conceptual view of a Robocop Executable Component

```
component CSysInfo { 702c7427-38bf-4092-a08b-fd2c14ce9871 }
{
  contains SvcSysInfo;
};
```

The RIDL-file defines one component `CSysInfo`, which contains one service `SvcSysInfo`. The service has one attribute `memSig` and it provides one interface `IMemInfo`. The `IMemInfo` interface contains two functions: `getTotal()` and `getFree()` to query the total and available memory. The RIDL-file is named `sysinfo.ridl`.

To generate the C-files needed for the component use the RIDL-compiler:

```
ridl -debug -skel sysinfo.ridl
```

The `skel` parameter tells the compiler to generate the skeleton files needed for the component and the service. Here is a simplified description of the generated files:

- **sysinfo.h and sysinfo.c**: These files contain the UUIDs and accessor functions for the interfaces and the services.
- **sysinfo__Impl.h and sysinfo__Impl.c**: In these files the IUnknown and IService interfaces are implemented.
- **CSysInfo_Impl.h and CSysInfo_Impl.c**: These files contain the implementation of the CSysInfo component.

- **SvcSysInfo_Impl.h and SvcSysInfo_Impl.c**: These files contain the implementation of the SvcSysInfo service.

The component can be built using Automake and Autoconf. To register the component to the Robocop Runtime Environment a `regscript.in` file is needed which contains the UUIDs of the component and its services. Details of this procedure can be read in the Component Model Tutorial [10]. After this, the functions `getTotal()` and `getFree()` have to be implemented to add functionality to the component. As can be seen, a developer only has to write the code for the implementation of the interfaces of the components defined. When using the RIDL-compiler almost no work is required from a developer.

2.5 Trust in Trust4All

The aim of Trust4All is to establish robust and reliable operation, upgrading and extension, and component trading within an open component based software application in such a way the user gets trust and keeps on having trust in the software system. But what properties do influence the trust of a software system? In this section these properties will be called quality attributes. Note that quality is a term which can have different meanings for different stakeholders (developers, users...); in this document the term quality will be used in its widest sense.

In the Trust4All project it is inadequate to just specify quality attributes on a high level, to simply state that an application should be secure and dependable is a too coarse grained requirement. The term quality is therefore divided into attributes and sub-attributes.

2.5.1 Quality Attributes

In Figure 2.2 a taxonomy of quality of attributes of the Trust4All project is shown. There are five main quality attributes: dependability, security, performance, robustness and interoperability [11]:

- **Dependability** is the ability to avoid service failures that are more frequent and more severe than is acceptable. The sub-attributes of dependability are:
  - **Availability** is readiness for correct service. Simply put, availability is the proportion of time a system is in a functioning condition.
  - **Reliability** is continuity of correct service. In general, reliability is the ability of a system to perform and maintain its functions in routine circumstances, as well as hostile or unexpected circumstances.
  - **Maintainability** is the ability to undergo modifications and repairs.
  - **Safety** is the absence of catastrophic consequences on the users and the environment.

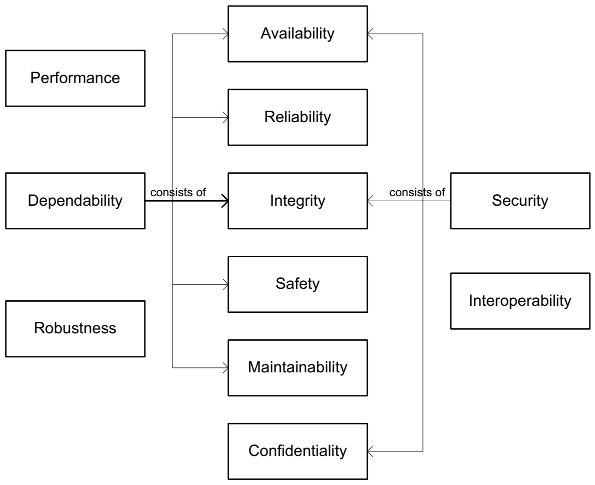


Figure 2.2: Taxonomy of quality attributes

  – **Integrity** is the absence of improper system alterations. In the context of security improper means unauthorized.

.

- **Security** is the absence of unauthorized acces, or handling of, to system state. Security quality attributes are confidentiality, availabilty and integrity. Availability and integrity are also part of depandability and are described there.
- **Performance** is the degree to which timing characteristics are adequate.
- **Robustness** is the degree to which an executable work product continues to work properly even under abnormal conditions or circumstances.
- **Interoperability** is the degree to which a system or one of its components is properly connected to and operates with something else.

2.5.2 Quality Attribute Control

In order to get a grip on quality attributes we need mechanisms to control them. Some of these mechanisms are described below [11]:

- **Containment** can be used to prevent propagation of errors from one subsystem to another to increase the reliability. Containment can also be used to provide confidentiality if the containment prevents access except via channels that require authorization.

- **Redundancy** can be used to increase both availability and reliability. Redundancy, in general terms, refers to the quality or state of being redundant, that is: exceeding what is necessary or normal; or duplication. Duplicates can be used for both availability and reliability of a system.
- **State transition** improves reliability by changing the system state to a different state that does not lead to a failure, or reduces the severity of a failure.
- **Guaranteed Resources** to provide the correct service.
- **Encryption** is used to prevent unauthorized disclosure of information to improve confidentiality.
- **Identification** is the degree to which the system identifies its externals before interacting with them.
- **Authentication** is the degree to which the system verifies the claimed identities of its externals before interacting with them. Thus, authentication verifies if a claimed identity is legitimate and belongs to the claimant.
- **Authorization** is the degree to which access and usage privileges of authenticated externals are properly granted and enforced.
- **Feature reduction** Functionality can be traded for quality attributes in a large number of situations. Quality attributes are improved by reducing the functionality provided. This also holds for the quality attributes related to trustworthiness. For example confidentiality can be improved by reducing the number of unauthorized entities in the system and availability can be improved by removing entities that use resources that are required for providing the correct service.
- **Modes of Operation** is the ability to change the set of conditions of a component during runtime to maintain trust in the software system and its components. A condition is a property of a component (e.g. ‘security mode’) and its value (e.g. ‘high’). By changing the condition, the value is changed in such a way that trust in the system is maintained. This new mechanism is one of the main subjects of this thesis.

Chapter 3

Modes of Operation

In this chapter, a new mechanism to obtain trust in a system is explained: ‘Modes of Operation’. In the first section the basic idea behind the Modes of Operation mechanism will be explained. The various quality attributes this mechanism can control will be described in the last section.

3.1 Basic Idea

A mode of operation is a set of conditions a component operates in. A condition can be things like:

- Security mode ‘high’.
- Video quality ‘low’.
- Text to speech conversion ‘off’.

A condition has a name called the ‘service property’ and a value called the ‘property value’. The mode of operation of a component is a set of conditions, and a condition is a service property with its property value.

The behavior of a process of some system can be described in an UML State Transition Diagram (STD) [13]. An STD is a behavorial model that describes the behavior of a process with states and transitions. A state is some state of a process and a transition is an action of a process to go from one state to another. The starting state of an STD is indicated by an initial state (black dot) with a transition to the starting state. To describe the behavior of a service of a component and the Modes of Operation mechanism involving this behavior, we use an extended version of the standard STDs. We extend the STDs with two notions of Paradigm [14]. Paradigm is a coordination specification language. Through Paradigm one can reformulate the sequential behaviour within various STDs in a more global phase-like manner. The STDs are divided into sub-STDs which each represent part of the total behavior of these STDs. In this way the

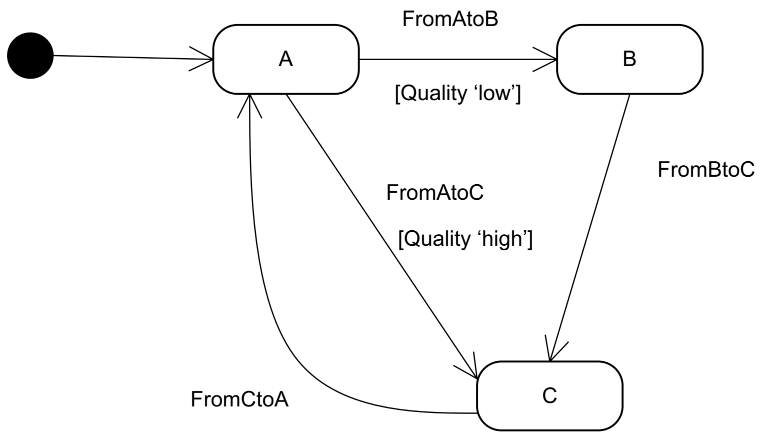


Figure 3.1: State Transition Diagram of process A of component A

modeling perspective can be lifted from detailed behavior to more abstract behavior in terms of going from one sub-STD to another.

The Paradigm notions of subprocesses and traps are used to describe the Modes of Operation mechanism in the STDs. A subprocess is an STD, restricted to a subset of the state and action spaces of the original STD, with the actions having the same behavioural effect. This means, any behaviour of a subprocess is a subsequence of some behaviour of the original STD. This is the reason a subprocess is informally referred to as a model of a certain phase within the set of possible behaviours of the original STD. A trap, being a subset of the subprocess' statespace, models a kind of final stage of the subprocess, once entered it cannot be left within that subprocess. It is only through the behaviours of the next phase, i.e. in another subprocess, that a trap can be left. We informally refer to such traps as overlaps between two subprocesses. If a trap contains all the states of a subprocess the trap is called trivial.

The STD of Figure 3.1 is a very simple STD with 3 states. It describes a very simple process with one service property called `quality` which can have two property values; `high` and `low`. In state `A` it can either go to state `B` with the transition `FromAtoB` or to state `C` with the transition `FromAtoC`. Transition `FromAtoB` has the label `Quality "high"`, which can only occur when the value of the service property `quality` is set on `high`. Transition `FromAtoC` has the label `Quality "low"`, which can only occur when the value of the service property `quality` is set on `low`. The service property `quality` divides the STD into the two subprocesses of Figures 3.3 and 3.2. If the property value is set to `high`, control takes the subprocess in Figure 3.3 which uses more resources.

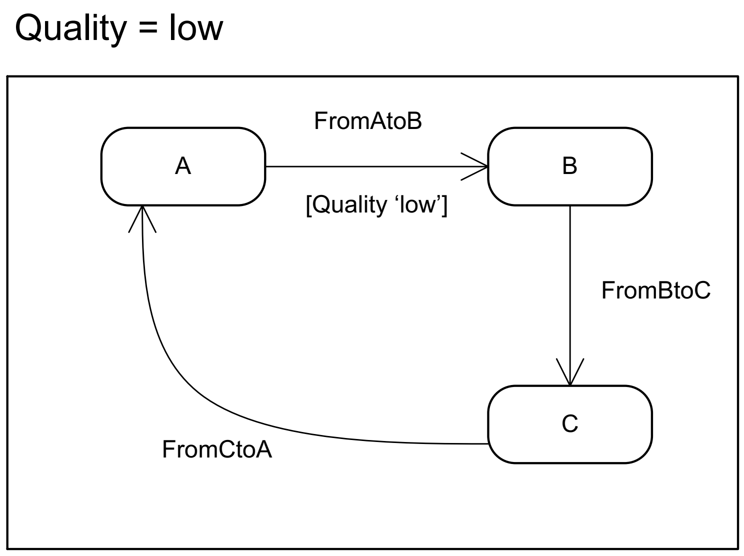


Figure 3.2: Subprocess of process A of component A with quality on ‘low’.

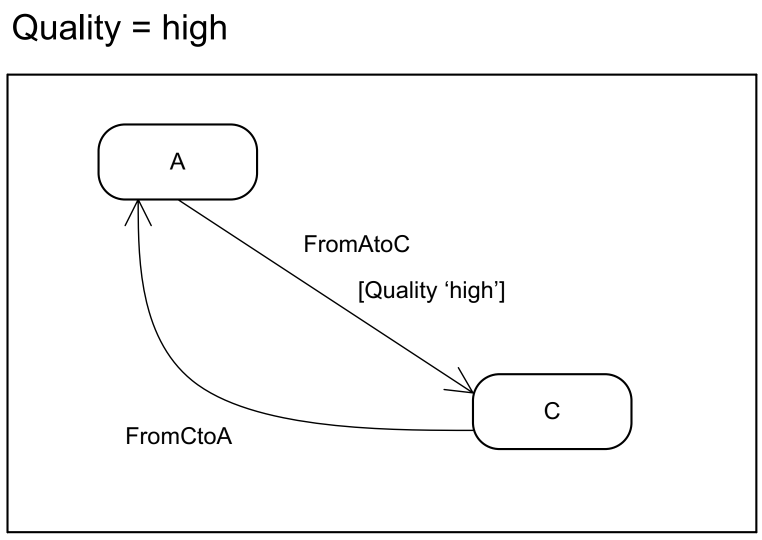


Figure 3.3: Subprocess of process A of component A with quality on ‘high’

If the property value is set to `low`, control takes the subprocess in Figure 3.2. which uses less resources. As one can see, the mode of operation, in this case consisting of one service property, decides what subprocess to take. In this way the Paradigm notion of subprocesses is used. When the property value changes, control must go from one subprocess to the other. According to the definition of traps, this can only happen in a trap. Since the traps of both subprocesses are trivial, this can happen in all the states of the subprocesses. Trivial traps are very important in the Modes of Operation mechanism, because it enables the user or system to change the mode of operation during runtime.

The difference between the property values is the amount of resources they use. In this way, trust in the system can be influenced. The property value `low` of the service property `quality` needs less resources than the property value `high`. If the system runs in an ideal situation, the property value of `quality` is `high`. The ideal situation must be changed to a less ideal situation if by some influence of the environment, which can be some part of the system or the environment the system runs in, the property value of `quality` is set to `low`. For example, a more important component of the application needs resources that are not available while the value of the service property `quality` is `high`. The application now changes the mode of operation of the less important component by changing the value of the service property `quality` to `low`, to make more resources available for the more important part to run properly. This increases the availability of components to increase trustworthiness in the application. The Modes of Operation mechanism can influence a lot of other quality attributes as well.

3.2 Trust by Modes of Operation

One major quality attribute the Modes of Operation mechanism can control, is dependability. Some of the dependability sub-attributes it can control are:

- **Availability:** Turn to a lighter mode of operation so another component can be executed and the system stays ready for correct service. For example if a component wants to run its service but there are not enough resources available, another component may switch its mode of operation so more resources become available.

- **Reliability:** Turn to a lighter mode to maintain correct service. For example if a component is about to enter a critical state and switches to a lighter mode to maintain correct service.

- **Maintainability**: Turn to a mode where only the important components keep on running so the application is able to undergo repairs and modifications.

- **Safety:** Turn to a mode that tightens safety issues. For example when a critical state is entered the modes of operation of some components turn to safer modes so there will not occur harmful events to the user or the environment.

- **Integrity:** turn to a mode that tightens authorization, so integrity of the system is secured.

The Modes of Operation mechanism can also control the security and robustness quality attributes. An example of controlling security is when the application is in some critical state and the security of some components must be tightened, the mode of operation can switch to a more heavy security mode only allowing the most privileged users. An example of controlling robustness is when the conditions or circumstances of the application change to abnormal, the application switches to another mode of operation in order to keep the application running during these abnormal conditions or circumstances.

The Modes of Operation mechanism has two important characteristics, one of these is that it can change the property values during runtime. The software system decides, during runtime, how to change the Mode of Operation to maintain trustworthiness in the software system and its components. This is quite an innovative characteristic. The other charateristic is that the Modes of Operation mechanism is able to control quite a lot of quality attributes. These two characteristics make the Modes of Operation mechanism a powerful tool to obtain trust in software system.

Chapter 4

The Robocop Simulation Component

In this chapter, the Robocop Simulation Component (RSC) for a Car Infotainment System (CIS) is described. The RSC has been built to prototype some components of the CIS system. The RSC can be used to simulate components in other Robocop software systems as well. With the RSC a user can test a component before it is really implemented or test an already developed component in a software system, before really integrating it into the system. The RSC simulates a component by simulating the behaviors of the component and their resource usage.

In Section 4.1 the basic idea of the Robocop Simulation Component is described. How to use the RSC is explained in Sections 4.2 to 4.5. How the Modes of Operation mechanism is implemented in the RSC to obtain trust in a simulated component is explained in Section 4.6. In Section 4.7 conclusions are drawn by describing the major advantages and disadvantages of this kind of simulation.

4.1 Basic Idea

The basic idea is to describe the behaviors of a component in extended STDs. For each service a component provides, there is one STD to describe the behaviour of that service. Once the STDs have been built, they must be converted into a type that can be read more easily by an application. The Extensible Markup Language (XML) [15] is chosen, because this is quite easy to use and understand. Each STD has to be converted to a XML-file. The RSC takes this XML-file and simulates the behaviour described in the STD.

4.2 State Transition Diagram

The behavior of each service of a simulated component has to be described in a UML State Transition Diagram (STD). To add the Modes of Operation mechanism to the behavior described in the STD, the extended version of the STD, described in Chapter 3, is used.

4.3 XML Simulation File

The STDs have to be converted into XML-files conform some standard to make the XML simulator able to use the XML-files. This standard is called the 'XML Simulation Standard' (XSS) and will be described in this section. For the actual simulation of the process just converting the behavior described in the STD to an XML-file is not enough. The amount of resources a process needs also have to be simulated. The user should think about the amount of resources a particular transition (action of the process) needs, the variables that should be allocated and when they should be freed again. This is quite important because this simulates the resources a service needs, by which one can see how the software system reacts to it. Only the CPU and memory usage of a process is simulated.

To explain how to convert an STD and to add the resource usage into an XML-file conform the XSS, the STD of Figure 3.1 will be used. The name of the process of the component and the initial state name is contained in the tag `processes`, to let the simulator know where to start . In this case the process is called `ProcessA` and the inititial state name is `A`:

```
<processes name="ProcessA" initialStateName="A">
```

The next lines contain the transitions of the STD with all its information. A transition looks this:

```
<rules name="FromAtoB" cpu="5" mem="100">
  <sourceStateNames>A</sourceStateNames>
  <targetStateNames>B</targetStateNames>
  <property name="quality" value="high">
  <malloc name="var" size="200"/>
  <free name="temp"/>
</rules>
```

A transition contains the following:

- A tag `rules` with the unique name of the transition, CPU usage and memory space of that transition. The CPU usage stands for how many times the CPU simulation function has to be called. The CPU simulation function just that takes up some CPU-time. In this case the CPU function generates the first 1000 prime numbers, but the function can be changed to the user's preferences. The memory space is the amount of temporary space the transition needs in bytes. The memory space is allocated before taking the transition and freed directly after the transition reaches its target state. In case the transition takes up no CPU and memory space, then it is possible to simply fill in 0 here. In this case the name of the transition is `FromAtoB`, the cpu usage is 5 and the memory usage is 100kB.

- A tag `sourceStateNames` with the name of the source state. There can be multiple source states per transition, but in this implementation there can be only one to make things easier. In this case the source state is `A`.
- A tag `targetStateNames` with the names of the target states. A transition can have multiple target states. In this case the only target state is `B`.
- A tag `property` with the allowed property {name,value}-pair of the transition. If the property value is the same as the property value of the component, the transition is allowed, else it is forbidden. In this case the transition is allowed if the service property `quality` of the component has the property value `high`. This tag adds the Modes of Operation mechanism to the XML simulation file. A transition can contain multiple `property` tags.
- A tag `malloc` with the name and the size of a non-temporary variable. Memory for this variable is allocated when the transition reaches its target state and is freed if some other transition has this variable in its free field. A transition can contain multiple `malloc` tags, because a transition can have multiple non-temporary variables to allocate. In this case there is one non-temporary variable `var` and there must be 200 bytes of memory allocated for this variable.
- A tag `free` which contains a variable name to be freed from memory. A transition can contain multiple `free` tags. In this case there is one variable `temp` to be freed.

The XML-file must be conform to the XML Simulation Standard, else the Robocop Simulator is not able to read the XML-file. The complete XML-code for the State Transition Diagram of Figure 3.1 can be found in Appendix A.

4.4 Generation

After converting the STDs into XML-files, the XML-files must be parsed into ANSI C-code, because the Robocop Simulation Component is programmed in ANSI C. This is done with a small XML parsing library called Mini-XML [16]. Mini-XML is an XML parsing library which one can use to read XML and XML-like data files in an application without requiring large non-standard libraries. Mini-XML only requires an ANSI C compatible compiler and a 'make' program.

The generator function (part of the Robocop Simulator) will make a call to the `mxmlLoadFile` function (part of the mini-XML library) with the appropriate XML-file as parameter. This function returns a tree which contains all the information of the XML-file in a mini-XML tree structure. After the tree is generated the generator will convert this mini-XML tree into a linked list of transitions. Each transition contains the same information as a transition in the XML-file. The root of the linked list represents the initial state in the STD. The initial state in an STD is a pseudo-state to indicate the transition to the start state. The root contains the name of the target state (the start state) of the root transition. The RSC uses this linked list of transitions for the actual simulation of the service.

4.5 Simulation

For each simulated service of a component an instantiation of the Robocop Simulation interface is required. This interface is instantiated by the service of the component that needs simulation of some process. It is advised to name the instantiation after the simulated service. This instantiation has to be initialized by calling `init_sim` with the service instantiation and the appropriate name of the XML-file as parameters.

The initialization function calls the `generate_xml` function with the name of the XML-file as parameter. The `generate_xml` function produces the linked list of transitions and returns it to the initialization function. After initialization the current transition is the root transition, which is the transition to the start state. In the STD of Figure 3.1 this will be the transition from the initial state to the state `A`.

The simulation continues by calling the `simulate` function with the service instantiation as parameter. The `simulate` function simulates one step in the STD. When the `simulate` function is called and the CPU counter `cpu_count` has become 0 the transition has reached the target-state and the following things are done in the given order:

- Temporary memory space used for the transition is freed.
- Memory space for the non-temporary variables is allocated for the transition (the malloc field in the XML-file).
- Memory space for the non-temporary variables which are in de free list are freed (the free field in the XML-file).
- By calling the `get_Transition` function the next transition is chosen. The `get_Transition` function returns a legal transition from the chosen target state, which has now become the source state for the next transition.
- Temporary memory space for the new transition is allocated.
- The next target state is chosen randomly from the list of target states and put in the variable `chosen_target` of the newly returned transition.

If the CPU counter is not yet zero, the transition is still not finished; the action to go from one state to the next is still being done. To simulate this behaviour the CPU simulation function is called.

4.6 Simulating Modes of Operation

The Robocop Simulation Component also simulates the Modes of Operation mechanism to obtain trust in a simulated component. The services of a simulated component contain property-lists with the service property names and their property values. Just after initialization of the simulated service, service properties can be added with their initial property values. These lists of service properties represent the Modes of Operation of that component.

In the interface of the Robocop Simulation service there are two functions for controlling the mode of operation of a simulated process; the `add_property` and the `set_property` function. To add a service property to a simulated component the function `add_property` has to be called with as parameter the service instantiation, the name of the service property and the property value. In the STD of Figure 3.1 the property `quality` with the value `high` or `low` (depending on what the property value has to be initialized) should be added to the property list.

When simulation has begun, the property values can be changer by calling the `set_property` function with the same parameters as the `add_poperty` function, but the `value` parameter containing the new property value. In the STD of Figure 3.1 the value of the property `quality` will be set to `low` if resources become too low and to `high` if enough resources become available again. This should be controlled by some sort of resource manager which controls the resources.

A transition in the STD of Figure 3.1 is legal if the property {name,value}-pair of the transition matches the {name,value}-pair in the property of the simulated service. For example when the service poperty `quality` in the simulated process of 3.1 is `high` and the source state is `A`, the transition `FromAtoB` will not be chosen, instead the transition `FromAtoC` will be chosen.

4.7 Advantages and Disadvantages

The goal of simulation is to see how a software system reacts to a component, before the component is really developed or deployed. If the software system reacts improperly to the component, the design of the component may be altered before it is really implemented or the component may be discarded at all, because it is not suitable for the software system. There are advantages and disadvantages of simulating components. The advantages are:

- Test how the application reacts to the integrated simulated component in advance and see how feasible it is for the system to build the component and integrate it. This can diminish the costs of the overall development of the component.

- Designers and users can determine the correctness and efficiency of the component before it is actually designed.

- Test how a software system reacts to an already developed component before really integrating that component. Integrating a Robocop Component can be quite a task because the component should be converted into a Robocop component conform some standards. In the case of my Robocop Simulation Component integration of a simulated component is a lot easier because it is not nessecary to convert it into a Robocop component, because it already is a Robocop component.

The disadvantages are:

- It is hard to estimate things like cpu usage and memory usage in advance, if the component has not been developed yet.

- The simulation model simplifies the component, so some key elements may be missing in the simulation. For example in the Robocop Simulation Component the simulation of operation calls, data exchange and bus-usage is missing.

Chapter 5

A Robocop case study: Car Infotainment System

The application developed on top of the Robocop Framework is a Car Infotainment System (CIS). A CIS is a software system in a car that gives information such as driver info (speed, fuel etc.), navigation and entertaiment (e.g. music and video) to the users of a car. The information and entertainment the CIS system of this project provides is displayed on 3 displays; one in the front and two in the back. The system consists out of several components such as a FrontDisplay, DriverInfo, GPS and Media component. The architecture of this system is described in Section 5.1. The components of the CIS system are described in Section 5.2. The main function is described in Section 5.3. In Section 5.4 an evaluation of the system is given with 3 scenarios.

5.1 Architecture

The CIS system is built on top of the Robocop 2.0 framework, and uses the GTK 2.0 library [17] for the graphical user interface. The architecture of the the CIS system can be found in the component diagram of Figure 5.1.

The CIS System consists of six components:

- The Car component simulates the behaviour of a car. The component does not use the Robocop Simulation Component (RSC), but has been programmed statically.
- The DriverInfo component takes the driver information from the Car component and puts the driver information in a GTK-box. The GTK-box is part of the GTK 2.0 library.
- The GPS component is a simulated component which uses the RSC. The component simulates three services: a Route Calculator, a Voice Synthe-

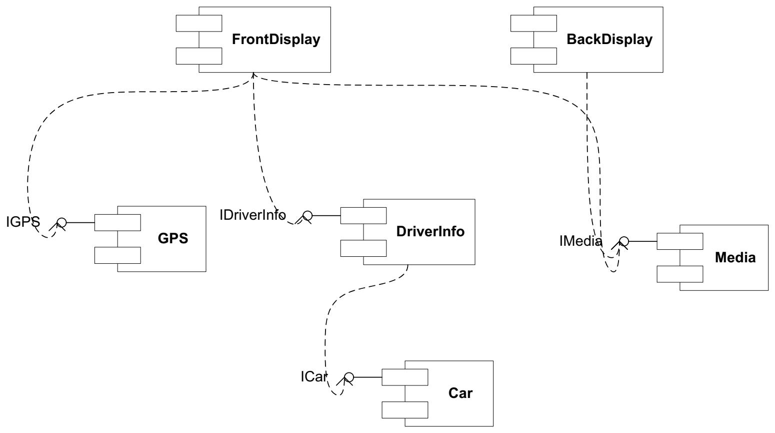


Figure 5.1: CIS Component Diagram

sizer and a Graphics Renderer. A GTK-box is created with the simulated behaviour of these services.

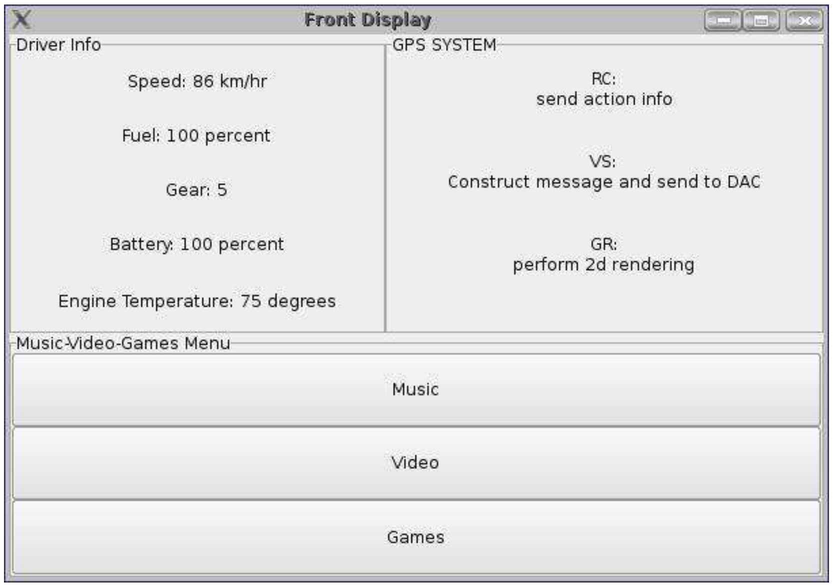
- The Media component is also a simulated component which uses the RSC. The Media component simulates the entertainment part of the CIS system. For purposes of demonstration and evaluation we have restricted the implementation of the Media component only with a simulation of the video system.

- The FrontDisplay component displays the driver information provided by the DriverInfo component, the navigation information provided by the GPS component and the media menu provided by the Media component. (Figure 5.2). The component also displays an options menu (Figure 5.3) to control certain properties of the CIS system.

- The BackDisplay component displays the media menu provided by the Media component (Figure 5.4).

The directory structure of the CIS system looks as follows:

- CIS: The top directory; contains the overal configure file and makefile.
  - `backdisplay`: Backdisplay Component.
  - `car`: Car Simulation Component.
  - `CIStestapp`: contains the main function.
  - `driverinfo`: Driverinfo Component.
  - `frontdisplay`: Frontdisplay Component.
  - `gps`: GPS Component (simulated).


Front Display

Driver Info

Speed: 86 km/hr

Fuel: 100 percent

Gear: 5

Battery: 100 percent

Engine Temperature: 75 degrees

GPS SYSTEM

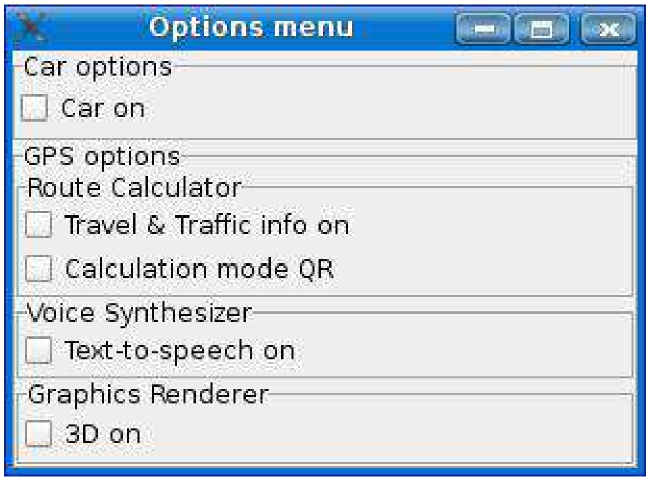
RC:
send action info

VS:
Construct message and send to DAC

GR:
perform 2d rendering

Music-Video-Games Menu

Figure 5.2: Front Display showing the driver information, gps system and the media menu.


Options menu

Car options

☐ Car on

GPS options

Route Calculator

☐ Travel & Traffic info on

☐ Calculation mode QR

Voice Synthesizer

☐ Text-to-speech on

Graphics Renderer

☐ 3D on

Figure 5.3: Options menu for controlling the diverse options of the CIS system.

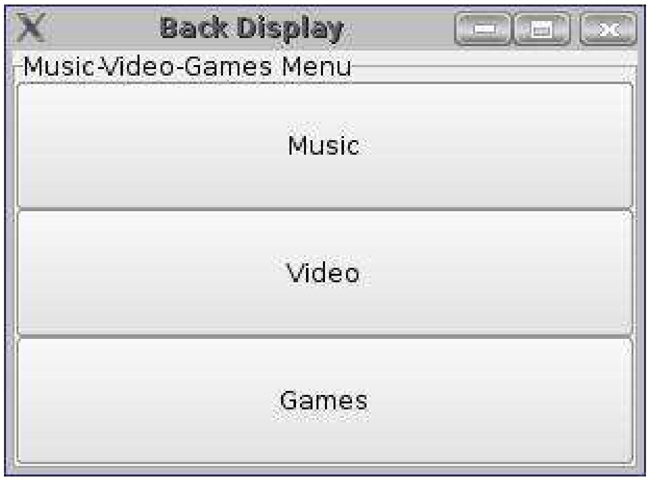


Figure 5.4: Back Display showing the media menu

- `media`: Media Component (music, video and games).
- `xmlsim`: Robocop Simulation Component.

The Robocop 2.0 framework does not provide multi-threading facilities, therefore we have chosen to implement the CIS as a single-threaded system. Because the State Transition model contains infinite loops, which is not common in Robocop, we have implemented a Round Robin scheduling strategy for the simulated services of the simulated components. Round-Robin (RR) is a scheduling algorithm, which assigns time slices to each process in equal portions and in order, handling all processes without priority. The CIS is a multi-process system, because it contains multiple processes running in parallel, which are scheduled by the RR scheduling algorithm.

Shortly described the system runs like this: The Frontdisplay component displays the driver information by passing an empty GTK-box to the Driverinfo interface. First it needs to initialize the Driverinfo box by calling the `init_DI` function:

```
CIS_IDriverInfo_init_DI( data->driverinfo, driverinfo_box );
```

After that the Driver Information is automatically updated by calling the `update_DI` function:

```
CIS_IDriverInfo_update_DI( data, driverinfo_box );
```

Both these functions get the driver information from the Car component by calling diverse functions such as `get_Speed`, `get_Fuel` etc. When the Frontdisplay calls the GPS system for the first time, it calls the `init_GPS` function and passes an empty GTK-box to initialize the system:

```
CIS_IGPS_init_gps( data->gps, gps_box );
```

After the initialization of the GPS system the Frontdisplay updates the GPS system by calling the `update_GPS` function of the GPS Interface:

```
the \texttt{get\_Menu} function from the Media Interface
CIS_IGPS_update_gps(data, gps_box);
```

The `update_GPS` function calls the `simulate` function to update the states of the simulated services.

The Frontdisplay component calls the `get_Menu` function from the Media Interface to get the media menu. The `get_Menu` function fills up the box with three buttons; Music, Video, Games. The two Backdisplays components display the Media Menu in the same way (Figure 5.4).

The Frontdisplay component also shows an options menu for controlling diverse options, such as turning the car `on` or `off` and diverse options for the GPS system such as calculation mode, speech-to-text, travel & traffic and rendering modes (Figure 5.3). This controls the mode of operation of the GPS System manually. By changing the mode of operation manually you can see the consequences directly, but in a real system this should be done by the Robocop Runtime Environment automatically.

5.2 The Components

In this section the components and give their RIDL-files are explained.

5.2.1 Car

The Car component (CCar) simulates the behavior of the car. It simulates the speed, current gear, fuel level, battery power and engine temperature. The RIDL-file of the car looks as follows:

```
#ifndef car_ridl_def
#define car_ridl_def

namespace CIS{ deb3b600-3147-43be-9a28-a361d4218593 }
{

  interface ICar{ c743c15d-af7f-45d2-b394-4a125aca6380 }
  {
    void StartCar();
    void StopCar();

    short get_Speed();
    short get_Fuel();
    short get_Gear();
    short get_Battery();
    short get_temperature();
  };
  service SCar{ 22d96db6-ec61-48c8-973f-fb3c124c40e9 }
  {
    provides{
      ICar car;
    };
    attributes{
      short speed;
      short fuel;
      short gear;
      short battery;
      short temperature;
      boolean caron;
    };
  };

  component CCar{ 137b71ca-ce00-4496-811a-a1584d2887f5 }
  {
    provides SCar;
  };
};

#endif
```

The Car component has a service called SCar which provides an interface ICar. The ICar interface has functions where you can access the simulated car information. There are also functions to turn the car on or off. The attributes of the service are the ‘parameters’ of the car.

5.2.2 Driver Information

The DriverInfo component gets the driver information from the Car component by calling the functions of the ICar interface. The DriverInfo component puts this information in a GTK-box which is displayed on the front display. The RIDL-file of the DriverInfo component looks like this:

```
#ifndef driverinfo_ridl_def
#define driverinfo_ridl_def

#include "car.ridl"

namespace CIS{ deb3b600-3147-43be-9a28-a361d4218593 }
{

  interface IDriverInfo{ f02131ab-da63-41c0-95da-ecc499ec8413 }
  {
    native init_DI();
    native update_DI();
  };


  service SDriverInfo{ e11e2c5d-bd85-4dbe-bbd0-cf3597f1175c }
  {
    provides{
      IDriverInfo driverinfo;
    };

    requires{
      ICar car;
    };
  };


  component CDriverInfo{ 2ec8d2fd-342a-4641-a772-cc51139a9777 }
  {
    provides SDriverInfo;
  };
};
#endif
```

The component provides an service called SDriverInfo. The service provides an interface called IDriverInfo which has functions to initialize and update the driverinfo GTK-box shown on the front display. The service requires the Car interface to get the driver information from.

5.2.3 GPS

The GPS (Global Positioning System) component fills up a GTK-box with navigation information for the user. This GTK-box is displayed on the front display. The three services of the GPS component are simulated by the Robocop Simulation Component: the Route Calculator, the Graphics Renderer and the Voice Synthesizer. The GPS component has one service called SGPS, and this service requires 3 IXmlSim Interfaces for the simulated services. The RIDL-file of the GPS component looks like this:

```
#ifndef gps_ridl_def
#define gps_ridl_def

#include "xmlsim.ridl"

namespace CIS{ deb3b600-3147-43be-9a28-a361d4218593 }
{

  interface IGPS{ 32b07b6d-1d5a-4e2c-bac6-4b0bde0b7305 }
  {
    native init_gps();
    native update_gps();
  };

  service SGPS{ 87ae75f1-d9b3-48d4-8e58-d14b73b02587 }
  {
    provides{
      IGPS gps;
    };
    requires{
      IXmlSim rc_sim;
      IXmlSim vs_sim;
      IXmlSim gr_sim;
    };
  };

  component CGPS{ 43e53c10-2f7e-4413-8a21-43ec65edd141 }
  {
    provides SGPS;
  };
};
#endif
```

The service SGPS provides one interface IGPS which has functions to initialize and update the navigation information. The initialize and update functions are used by the Frontdisplay component.

The GPS component is a simulated component. The development of the simulated services is described by starting from the design of the UML State Transition Diagrams. The component has three services to be simulated: the Route Calculator, the Voice Synthesizer and the Graphics Renderer. The behaviors of these services are described in the UML State Transition Diagrams of Figures 5.5, 5.6 and 5.7.

In Figure 5.5 you can see that in some states 'get T&T info' (Travel and Traffic information) transition can be chosen. If the T&T is on, it chooses that transition else it takes the other transition. For the Calculation Mode there are two options: Quickest Route (QR) and Shortest Route (SR). This gives the following mode of operation set for the Route Calculator:

- Service property: `T&T`, property values: `on`, `off`
- Service property: `Calculation Mode`, property values: `sr`, `qr`

As can be seen in Figure 5.6 there is an option to do a text-to-speech conversion, so the voice synthesizer has the following mode of operation set:

- Service property: `Text-to-Speech`, property values: `on`, `off`

As can be seen in Figure 5.7 there is an option to choose between 2D and 3D rendering, so the graphics renderer has the following mode of operation set:

- Service property: `Rendering Mode`, property values: `2d`, `3d`

Now that the UML State Transition Diagrams are finished and the service properties are identified, the diagrams are converted into XML-files following the XML Simulation Standard described in Chapter 4. The XML-files can be found in Appendices B, C and D. Before simulation can begin, we must initialize the GPS component, this is done by the IGPS interface. After initialization the service properties with their initial property values are added to the property lists of the services.

```
//route calculator
CIS_IXmlSim_init_sim( data->rc_sim, "routecalculator.xml",
  &trans_RC );
CIS_IXmlSim_add_moo( data->rc_sim, "travelandtraffic", "off");
CIS_IXmlSim_add_moo( data->rc_sim, "calculationmode", "sr");
```

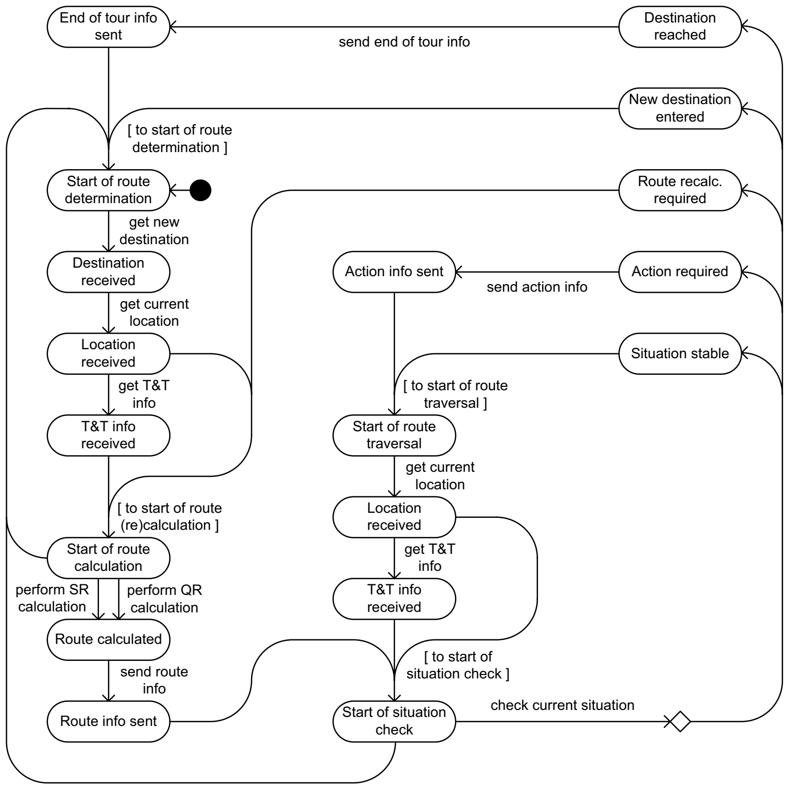


Figure 5.5: UML State Transition Diagram for the Route Calculator service of the GPS Component

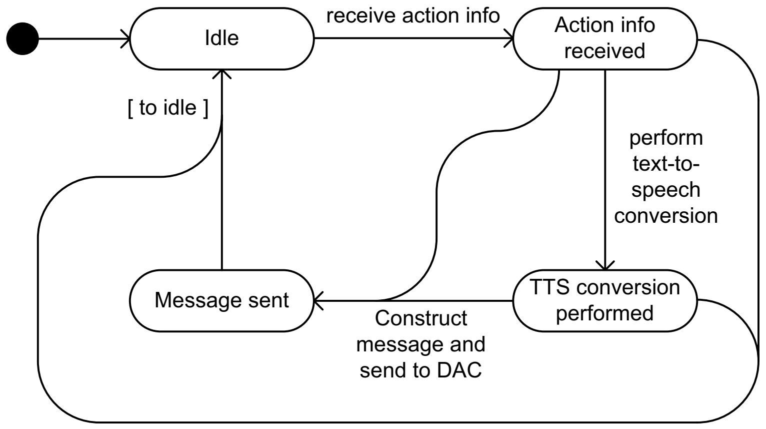


Figure 5.6: UML State Transition Diagram for the Voice Synthesizer service of the GPS Component

```
//voice synthesizer
CIS_IXmlSim_init_sim( data->vs_sim, "voicesynthesizer.xml",
  &trans_VS );
CIS_IXmlSim_add_moo( data->vs_sim, "texttospeech", "off");

//graphics renderer
CIS_IXmlSim_init_sim( data->gr_sim, "graphicsrenderer.xml",
  &trans_GR );
CIS_IXmlSim_add_moo( data->gr_sim, "3d", "off");
```

The `trans_RC`, `trans_VS` and `trans_GR` parameters are pointers to one of the transitions in the linked list of transitions of the simulated service. The names of these transitions are put into a GTK-box which is shown by the FrontDisplay component. After the simulated services have been set up, simulation can begin by calling the `simulate` function:

```
CIS_IXmlSim_simulate( data->rc_sim, &trans_RC );

CIS_IXmlSim_simulate( data->vs_sim, &trans_VS );

CIS_IXmlSim_simulate( data->gr_sim, &trans_GR );
```

The `simulate` function simulates one step in the UML State Transition Diagram of the simulated service. How this works has been described in Chapter 4.

5.2.4 Media

The Media component (CMedia) is the entertainment part of the CIS System. The RIDL-file of the Media component looks like this:

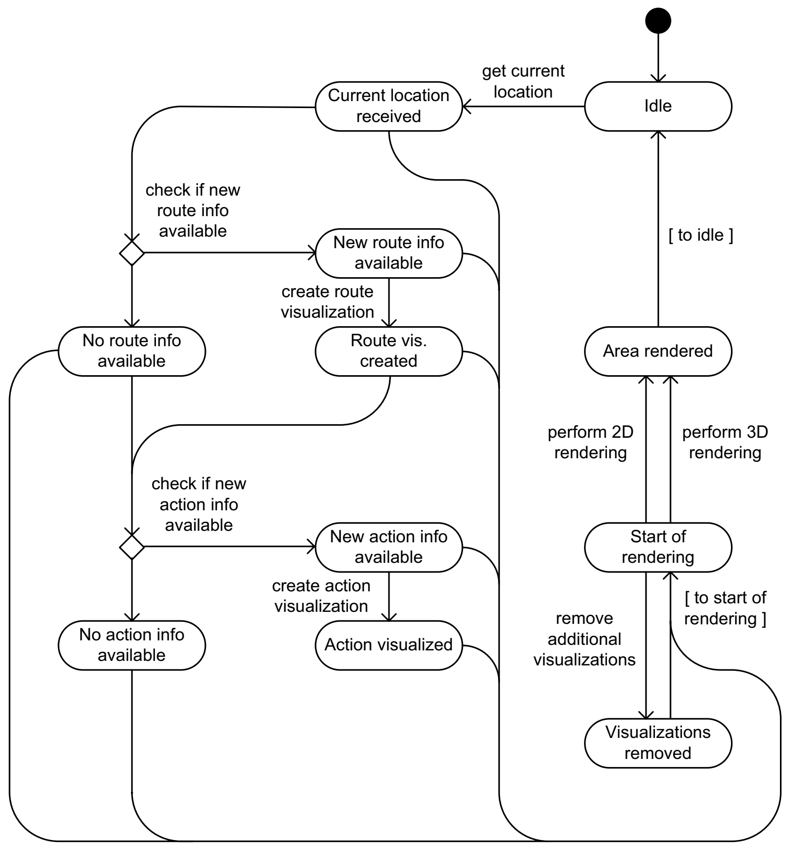


Figure 5.7: UML State Transition Diagram for the Graphics Renderer service of the GPS Component

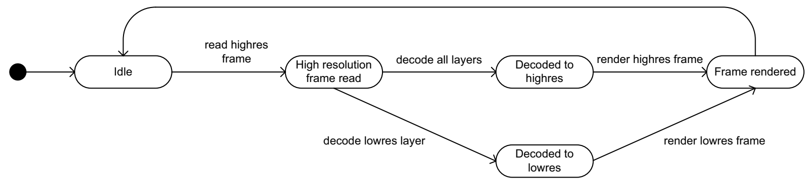


Figure 5.8: UML State Transition Diagram for the simulated Video System of the Media component

```
#ifndef media_ridl_ref
#define media_ridl_def

#include "xmlsim.ridl"

namespace CIS{ deb3b600-3147-43be-9a28-a361d4218593 }
{

  interface IMedia { 3eec37e0-da0d-4503-ac03-3509b5dd9457 }
  {
    native get_Menu();
  };

  service SMedia { 9b8c50ab-9f06-4cc5-8f1a-689d20c79330 }
  {
    provides{
      IMedia media;
    };

    requires{
      IXmlSim video;
    };
  };

  component CMedia { 6b81d6e8-1916-4cf0-b13b-3fd93fc8683e }
  {
    provides SMedia;
  };
};
#endif
```

The Media component provides a service called SMedia. This service provides an interface IMedia which has a function to get the Media Menu. The Media service requires a IXmlSim interface for the video system simulation. The behavior of the video system is described in the State Transition Diagram (STD) of Figure 5.8.

The STD is converted into the XML-file of Appendix E. The Robocop Simulation Component simulates the video system using the XML-file. The Media component displays the simulation of the video system. The video system has one service property `resolution` with can have the value `high` or `low`.

5.2.5 FrontDisplay

The Frontdisplay component (CFrontDisplay) represents the front display of the CIS system. The component provides a service called SFrontDisplay which requires the IDriverInfo interface to show the Driverinfo, the IGPS interface to show the behaviour of the GPS Sytem, the IMedia interface to show the media menu and the ICar interface to turn the car on or off. The RIDL-file looks like this:

```
#ifndef frontdisplay_ridl_def
#define frontdisplay_ridl_def

#include "driverinfo.ridl"
#include "gps.ridl"
#include "media.ridl"

namespace CIS{ deb3b600-3147-43be-9a28-a361d4218593 }
{

  service SFrontDisplay { 619db924-caf5-4803-8edd-6be4c29e7fe5 }
  {
    requires{
      ICar car;
      IDriverInfo driverinfo;
      IGPS gps;
      IXmlSim rc_sim;
      IXmlSim vs_sim;
      IXmlSim gr_sim;
      IMedia media;
    };
  };

  component CFrontDisplay { 65c51828-ea00-4e78-9a4d-218ff155d3c6 }
  {
    provides SFrontDisplay;
  };
};
#endif
```

The front display is displayed from the main function by the following call:

```
result = CIS_SFrontDisplay_start( s_fd );
```

This function calls another function that implements the front display.

5.2.6 BackDisplay

The BackDisplay component shows the back display of the CIS system. This is the RIDL-file of the BackDisplay:

```
#ifndef backdisplay_ridl_def
#define backdisplay_ridl_def

#include "media.ridl"
```

```
namespace CIS{ deb3b600-3147-43be-9a28-a361d4218593 }
{

  interface IBackDisplay { b097845c-9f1f-4918-8e32-e3c0eb24c68d }
  {
    void show_BackDisplay();
  };

  service SBackDisplay { cb1e74d6-b613-461f-9484-d8e77d919f4d }
  {
    provides{
      IBackDisplay backdisplay;
    };
    requires{
      IMedia media;
    };
  };

  component CBackDisplay { 2785c22e-c4b0-4b8e-b994-a7c77177122b }
  {
    provides SBackDisplay;
  };
};
#endif
```

The component provides a service called SBackDisplay which requires the IMedia interface to show the media menu.

5.3 The Main Function

The implementation of the `main` function can be found in Appendix E. As a first step, all the services have to be created. After this, all the interfaces the services provide and require have to be created. Then, the services that require interfaces have to be bound to the interfaces they require. When this is all done, the Car service is started, this initializes the Car component. Then the system boots by starting the front display and the two back displays.

5.4 Evaluation

To test the CIS system and especially how it reacts to the simulated components 3 scenarios are tested. These scenarios were tested on a normal Desktop PC with 512 MB of main memory.

- **SCENARIO 1:** 'The video component needs more memory than is available when running in high quality mode.'
The video system needs 800 MB of main memory if the video quality is set on `high`. In this case when running the application and starting the video, setting the video quality on `high` will cause an `OUT OF MEMORY`-warning. When the video quality is set back to `low` again, the simulation continues normally.

- **SCENARIO 2:** 'A component does not work properly, because another component takes up too many resources.'
In this scenario the video system needs 400 MB of main memory if the video quality is set on `high`. The GPS System is running with all its options on:

- Travel and Traffic (T&T) `on`.
- Calculation mode `SR` (see page 31).
- Text-to-speech conversion `on`.
- Rendering mode `3d`.

If the quality of the video system is set on `high`, the video system and the GPS system will give `OUT OF MEMORY`-warnings. By switching the video quality back to `low`, the `OUT OF MEMORY`-warnings will disappear and the system will run properly again.

- **SCENARIO 3:** 'Three video systems running in high quality mode requiring 600MB of memory per video system.'
All three systems give an `OUT OF MEMORY`-warning beginning with the system last started. When switching back to `low` quality, needing 100MB of memory per video system, the video systems run properly again.

Which component of the system gives an `OUT OF MEMORY`-warning depends on when the component is started and at what point in time it needs to allocate memory. Memory will be allocated to a component only when it is available, else it will give an `OUT OF MEMORY`-warning and it will retry to allocate the same amount of memory in the next run of the simulation. By changing the mode of operation of the component, it can happen that the component needs less memory and runs properly again.

To add realistic resource usage to the XML-files is quite a challenge if the component is not developed yet. Good simulation depends on realistic resource usage. It looks like simulation of an already developed component is better, because more realistic behaviors and values of resource usage are known.

Chapter 6

Conclusions and Future Work

The main goal of this thesis project was developing a Car Infotainment System (CIS) and the Modes of Operation mechanism to obtain trustworthiness in it. Already during the design phase it did not seem feasible to build all actual components of the CIS system. Because of this, the idea came up to develop a Robocop Simulation Component (RSC) which can simulate the behaviours of a component using XML-files. Now the main goal was building the RSC-tool and implementing the Modes of Operation mechanism to obtain trust and demonstrating this using the CIS system.

The RSC seems to be quite a powerful tool for simulating Robocop components. In the RSC the quality of the simulation depends very much on how realistic the values of the resource usage are. Simulating an already developed component looks easier than simulating a non-existing component, because the behaviors and resource usage of the developed component are already known. The RSC is suitable for simulating Robocop components in every Robocop application.

Once the XML-files have been produced, using the RSC-tool is quite simple. Only the appropriate function calls and service properties have to be added by the user. The service properties implement the actual Modes of Operation mechanism in the RSC. Changing the mode of operation is done manually, in this way a user can see directly how a change in the mode of operation influences the system. In real embedded systems this should be done automatically.

The innovative Modes of Operation mechanism developed to obtain trustworthiness in a component based software system is also quite powerful, because it can change the mode of operation of the system during runtime to obtain trust in the system. Also it can control quite a lot of quality attributes to obtain trust in a software system and its components.

Understanding the Robocop framework and its features was quite a challenge, because the framework is still under development and there is not much documentation available. Before developing a Robocop component or integrating an already developed component, it can be useful to test how the Robocop application reacts to the integration of a simulation of that component using the RSC-tool. In such a way a user can see if it is useful to develop the component for the Robocop application or to integrate the already developed component in the Robocop application. This is both cost and time effective.

Here are recommendations for future work on the subject of this thesis:

- During the project a new version of the Robocop framework was released with support for multi-threading. The RSC-tool must be adapted so it can run on the new Robocop framework in such a way the new multi-threading facilities can be exploited.
- The RSC-tool can be extended with a a protocol to send messages between simulated services. This makes it possible that services can communicate with each other which allows new features to be implemented.
- Develop a Resource Manager Component (RMC) that can switch the mode of operation during runtime automatically. This is more realistic than the current manual switching of the mode of operation.
- Adapt the CIS application in such a way that it compiles and runs on the new Robocop framework.
- In the CIS application the Music and Games buttons don't have functionality. Functionality can be added by simulating both components using the RSC-tool.
- Implement other mechanisms to obtain trust in the RSC-tool. Some of these mechanisms are mentioned in section 2.5.2

Bibliography

[1] I. Crnkovic and M. Larssons (Eds.): Building Reusable Component Based Software Systems. Aktech House Publisher, 2002.

[2] F.B. Schneider, S.M. Belovin and A.S. Inovye: Building Trustworthy Systems. IEEE Internet Computing 3(6): 64-72, 1999.

[3] C. Szypersky: Component Software - Beyond Object Oriented Programming. 2nd Edition, Addision Wesley, 2002.

[4] ITEA Robocop Project: Robust Open Component Based Software Architecture for Configurable Devices Project, http://www.hitech-projects.com/euprojects/robocop/ (23-04-2007).

[5] ITEA Space4U Project: Software platform and component environment 4 you, http://www.hitech-projects.com/euprojects/space4u/ (23-04-2007).

[6] ITEA Trust4All Project: Trustworthiness in embedded software, http://www.hitech-projects.com/euprojects/trust4all/ (23-04-2007).

[7] ITEA: Information Technology for European Advancement, http://www.itea-office.org/ (27-04-2007).

[8] Nokia Research website: Trust4All / Robocop / Space4U, http://research.nokia.com/research/projects/trust4all/index.html (23-04-2007).

[9] Space4U Deliverable 1.2: Report on the concept of framework with functionality extensions. ITEA Space4U Consortium Confidential.

[10] Arnaud Gouder: Component Model Tutorial Version 1.0.

[11] Johan Muskens, Ruben Alonso, Zhen Yhang, Koen Egelink, Arantxa Larranaga and Arnaud Gouder: Trust4All Trust Framework and Mechanisms. Version 1.0.

[12] Dennis Ritchie: The C programming language. Prentice Hall, 1988.

[13] Jason T. Roff: UML 'A Beginners Guide'. Osborne Publishing, 2002.

[14] L. Groenewegen and E. de Vink: Operational Semantics for Coordination in Paradigm. In F. Arbab and C. Talcott, editors, Proc. Coordination 2002, volume 2315 of LNCS, pages 191-206, 2002.

[15] Robert B. Mellor: XML 'Learning by Example'. Franklin Beedle & Associates, 2002.

[16] Michael Sweet: Mini-XML, a small XML parsing library,
http://www.easysw.com/ mike/mxml/ (23-04-2007).

[17] Syd Logan: Gtk+ Programming In C. Prentice Hall, 2001.

Appendix A

Example XML file

Here you can find the complete XML-file for the UML statechart diagram of figure 3.1.

```
<?xml version="1.0" encoding="UTF-8"?>
<ParADE:ParadeModel xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:ParADE="http://www.liacs.nl/ParADE/core"
name="Simulation Example">
<processes name="Example" initialStateName="A">
  <rules name="FromAtoB" cpu="5" mem="100">
    <sourceStateNames>A</sourceStateNames>
    <targetStateNames>B</targetStateNames>
    <property name="quality" value="high">
    <malloc name="var" size="200"/>
    <free name="temp"/>
  </rules>
  <rules name="FromAtoC" cpu="10" mem="200">
    <sourceStateNames>A</sourceStateNames>
    <targetStateNames>C</targetStateNames>
    <property name="quality" value="low">
    <malloc name="temp" size="200000"/>
  </rules>
  <rules name="FromBtoC" cpu="5" mem="100">
    <sourceStateNames>B</sourceStateNames>
    <targetStateNames>C</targetStateNames>
    <free name="var" size="200"/>
  </rules>
</processes>
</ParADE:ParadeModel>
```

Appendix B

Route Calculator XML file

```
<?xml version="1.0" encoding="UTF-8"?>
<ParADE:ParadeModel xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:ParADE="http://www.liacs.nl/ParADE/core"
name="Global Positioning System">
  <processes name="Route Calculator"
initialStateName="Start of route determination">
    <rules name="get new destination" cpu="5" mem="1000000">
      <sourceStateNames>Start of route determination</sourceStateNames>
      <targetStateNames>Destination received</targetStateNames>
      <malloc name="dest" size="1000000"/>
    </rules>
    <rules name="get current location" cpu="5" mem="1000000">
      <sourceStateNames>Destination received</sourceStateNames>
      <targetStateNames>Location received</targetStateNames>
      <malloc name="loc" size="1000000"/>
    </rules>
    <rules name="get T&amp;T info" cpu="30" mem="400000000">
      <sourceStateNames>Location received</sourceStateNames>
      <targetStateNames>T&amp;T info received</targetStateNames>
      <malloc name="tt" size="400000000"/>
      <property name="travelandtraffic" value="on"/>
    </rules>
    <rules name="[ to start of route (re)calculation ]" cpu="0" mem="0">
      <sourceStateNames>T&amp;T info received</sourceStateNames>
      <targetStateNames>Start of route calculation</targetStateNames>
    </rules>
    <rules name="[ to start of route (re)calculation ]" cpu="0" mem="0">
      <sourceStateNames>Location received</sourceStateNames>
      <targetStateNames>Start of route calculation</targetStateNames>
      <property name="travelandtraffic" value="off"/>
    </rules>
    <rules name="[ to start of route (re)calculation ]" cpu="0" mem="0">
      <sourceStateNames>Route recalc. required</sourceStateNames>
      <targetStateNames>Start of route calculation</targetStateNames>
    </rules>
    <rules name="perform SR calculation" cpu="75" mem="400000000">
      <sourceStateNames>Start of route calculation</sourceStateNames>
      <targetStateNames>Route calculated</targetStateNames>
      <malloc name="route" size="400000000"/>
  <property name="calculationmode" value="sr"/>
</rules>
<rules name="perform QR calculation" cpu="50" mem="100000000">
  <sourceStateNames>Start of route calculation</sourceStateNames>
  <targetStateNames>Route calculated</targetStateNames>
  <malloc name="route" size="100000000"/>
  <property name="calculationmode" value="qr"/>
</rules>
<rules name="send route info" cpu="5" mem="1000000">
  <sourceStateNames>Route calculated</sourceStateNames>
  <targetStateNames>Route info sent</targetStateNames>
</rules>
<rules name="[ to start of situation check ]" cpu="0" mem="0">
   <sourceStateNames>Route info sent</sourceStateNames>
  <targetStateNames>Start of situation check</targetStateNames>
</rules>
<rules name="[ to start of situation check ]" cpu="0" mem="0">
  <sourceStateNames>T&amp;T info received while traversing</sourceStateNames>
  <targetStateNames>Start of situation check</targetStateNames>
</rules>
<rules name="[ to start of situation check ]" cpu="0" mem="0">
  <sourceStateNames>Location received while traversing</sourceStateNames>
  <targetStateNames>Start of situation check</targetStateNames>
  <property name="travelandtraffic" value="off"/>
</rules>
<rules name="check current situation" cpu="5" mem="1000000">
  <sourceStateNames>Start of situation check</sourceStateNames>
  <targetStateNames>Situation stable</targetStateNames>
  <targetStateNames>Action required</targetStateNames>
  <targetStateNames>Route recalc. required</targetStateNames>
  <targetStateNames>New destination entered</targetStateNames>
  <targetStateNames>Destination reached</targetStateNames>
</rules>
<rules name="[ to start of route traversal ]" cpu="0" mem="0">
  <sourceStateNames>Situation stable</sourceStateNames>
  <targetStateNames>Start of route traversal</targetStateNames>
  <free name="tt"/>
  <free name="loc"/>
</rules>
<rules name="[ to start of route traversal ]" cpu="0" mem="0">
  <sourceStateNames>Action info sent</sourceStateNames>
  <targetStateNames>Start of route traversal</targetStateNames>
  <free name="tt"/>
  <free name="loc"/>
</rules>
<rules name="get current location" cpu="5" mem="1000000">
  <sourceStateNames>Start of route traversal</sourceStateNames>
  <targetStateNames>Location received while traversing</targetStateNames>
  <malloc name="loc" size="1000000"/>
</rules>
<rules name="get T&amp;T info" cpu="5" mem="100000000">
  <sourceStateNames>Location received while traversing</sourceStateNames>
  <targetStateNames>T&amp;T info received while traversing</targetStateNames>
  <property name="travelandtraffic" value="on"/>
  <malloc name="tt" size="100000000"/>
</rules>
<rules name="send action info" cpu="5" mem="500">
```

```
      <sourceStateNames>Action required</sourceStateNames>
      <targetStateNames>Action info sent</targetStateNames>
    </rules>
    <rules name="[ to start of route determination ]" cpu="0" mem="0">
      <sourceStateNames>New destination entered</sourceStateNames>
      <targetStateNames>Start of route determination</targetStateNames>
      <free name="dest"/>
      <free name="loc"/>
      <free name="route"/>
      <free name="tt"/>
    </rules>
    <rules name="[ to start of route determination ]" cpu="0" mem="0">
      <sourceStateNames>End of tour info sent</sourceStateNames>
      <targetStateNames>Start of route determination</targetStateNames>
      <free name="dest"/>
      <free name="loc"/>
      <free name="route"/>
      <free name="tt"/>
    </rules>
    <rules name="[ to start of route determination ]" cpu="0" mem="0">
      <sourceStateNames>Start of route calculation</sourceStateNames>
      <targetStateNames>Start of route determination</targetStateNames>
      <free name="dest"/>
      <free name="loc"/>
      <free name="route"/>
      <free name="tt"/>
    </rules>
    <rules name="[ to start of route determination ]" cpu="0" mem="0">
      <sourceStateNames>Start of situation check</sourceStateNames>
      <targetStateNames>Start of route determination</targetStateNames>
      <free name="dest"/>
      <free name="loc"/>
      <free name="route"/>
      <free name="tt"/>
    </rules>
    <rules name="send end of tour info" cpu="5" mem="100">
      <sourceStateNames>Destination reached</sourceStateNames>
      <targetStateNames>End of tour info sent</targetStateNames>
    </rules>
  </processes>
</ParADE:ParadeModel>
```

Appendix C

Voice Synthesizer XML file

```
<?xml version="1.0" encoding="UTF-8"?>
<ParADE:ParadeModel xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:ParADE="http://www.liacs.nl/ParADE/core" name="Global Positioning System">
  <processes name="Voice Synthesizer" initialStateName="Idle">
    <rules name="receive action info" cpu="5" mem="100">
      <sourceStateNames>Idle</sourceStateNames>
      <targetStateNames>Action info received</targetStateNames>
      <malloc name="info" size="1000000"/>
    </rules>
    <rules name="perform text-to-speech conversion" cpu="40" mem="100000000">
      <sourceStateNames>Action info received</sourceStateNames>
      <targetStateNames>TTS conversion performed</targetStateNames>
      <property name="texttospeech" value="on"/>
      <malloc name="speech" size="500000000"/>
      <free name="info"/>
    </rules>
    <rules name="Construct message and send to DAC" cpu="10" mem="10000">
      <sourceStateNames>TTS conversion performed</sourceStateNames>
      <targetStateNames>Message sent</targetStateNames>
      <free name="speech"/>
    </rules>
    <rules name="Construct message and send to DAC" cpu="10" mem="10000">
      <sourceStateNames>Action info received</sourceStateNames>
      <targetStateNames>Message sent</targetStateNames>
      <property name="texttospeech" value="off"/>
      <free name="info"/>
    </rules>
    <rules name="[ to idle ]" cpu="1" mem="0">
      <sourceStateNames>Action info received</sourceStateNames>
      <targetStateNames>Idle</targetStateNames>
    </rules>
    <rules name="[ to idle ]" cpu="1" mem="0">
      <sourceStateNames>TTS conversion performed</sourceStateNames>
      <targetStateNames>Idle</targetStateNames>
    </rules>
    <rules name="[ to idle ]" cpu="1" mem="0">
      <sourceStateNames>Message sent</sourceStateNames>
      <targetStateNames>Idle</targetStateNames>
    </rules>
  </processes>
</ParADE:ParadeModel>
```

Appendix D

Graphics Renderer XML file

```
<?xml version="1.0" encoding="UTF-8"?>
<ParADE:ParadeModel xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:ParADE="http://www.liacs.nl/ParADE/core" name="Global Positioning System">
  <processes name="Graphics Renderer" initialStateName="Idle">
    <rules name="get current location" cpu="5" mem="100">
      <sourceStateNames>Idle</sourceStateNames>
      <targetStateNames>Current location received</targetStateNames>
      <malloc name="location" size="1000000"/>
    </rules>
    <rules name="check if new route info available" cpu="5" mem="100">
      <sourceStateNames>Current location received</sourceStateNames>
      <targetStateNames>No route info available</targetStateNames>
      <targetStateNames>New route info available</targetStateNames>
    </rules>
    <rules name="create route visualization" cpu="100" mem="1000000">
      <sourceStateNames>New route info available</sourceStateNames>
      <targetStateNames>Route visualization created</targetStateNames>
      <malloc name="routevis" size="5000000"/>
    </rules>
    <rules name="check if new action info available" cpu="5" mem="100">
      <sourceStateNames>Route visualization created</sourceStateNames>
      <targetStateNames>No action info available</targetStateNames>
      <targetStateNames>New action info available</targetStateNames>
    </rules>
    <rules name="check if new action info available" cpu="5" mem="100">
      <sourceStateNames>No route info available</sourceStateNames>
      <targetStateNames>No action info available</targetStateNames>
      <targetStateNames>New action info available</targetStateNames>
    </rules>
     <rules name="create action visualization" cpu="10" mem="100">
      <sourceStateNames>New action info available</sourceStateNames>
      <targetStateNames>Action visualized</targetStateNames>
      <malloc name="actionvis" size="5000000"/>
    </rules>
    <rules name="[ to start of rendering ]" cpu="0" mem="0">
      <sourceStateNames>Current location received</sourceStateNames>
      <targetStateNames>Start of rendering</targetStateNames>
    </rules>
    <rules name="[ to start of rendering ]" cpu="0" mem="0">
      <sourceStateNames>No route info available</sourceStateNames>
      <targetStateNames>Start of rendering</targetStateNames>
    </rules>
    <rules name="[ to start of rendering ]" cpu="0" mem="0">
      <sourceStateNames>New route info available</sourceStateNames>
      <targetStateNames>Start of rendering</targetStateNames>
    </rules>
    <rules name="[ to start of rendering ]" cpu="0" mem="0">
      <sourceStateNames>Route visualization created</sourceStateNames>
      <targetStateNames>Start of rendering</targetStateNames>
    </rules>
    <rules name="[ to start of rendering ]" cpu="0" mem="0">
      <sourceStateNames>No action info available</sourceStateNames>
      <targetStateNames>Start of rendering</targetStateNames>
    </rules>
    <rules name="[ to start of rendering ]" cpu="0" mem="0">
      <sourceStateNames>New action info available</sourceStateNames>
      <targetStateNames>Start of rendering</targetStateNames>
    </rules>
    <rules name="[ to start of rendering ]" cpu="0" mem="0">
      <sourceStateNames>Action visualized</sourceStateNames>
      <targetStateNames>Start of rendering</targetStateNames>
    </rules>
    <rules name="[ to start of rendering ]" cpu="0" mem="0">
      <sourceStateNames>Visualizations removed</sourceStateNames>
      <targetStateNames>Start of rendering</targetStateNames>
    </rules>
    <rules name="perform 2d rendering" cpu="50" mem="100000000">
      <sourceStateNames>Start of rendering</sourceStateNames>
      <targetStateNames>Area rendered</targetStateNames>
      <targetStateNames>Visualizations removed</targetStateNames>
      <property name="3d" value="off"/>
      <free name="routevis"/>
      <free name="actionvis"/>
      <free name="location"/>
    </rules>
    <rules name="perform 3d rendering" cpu="100" mem="400000000">
      <sourceStateNames>Start of rendering</sourceStateNames>
      <targetStateNames>Area rendered</targetStateNames>
      <targetStateNames>Visualizations removed</targetStateNames>
      <property name="3d" value="on"/>
      <free name="routevis"/>
      <free name="actionvis"/>
      <free name="location"/>
    </rules>
    <rules name="[ to idle ]" cpu="0" mem="0">
      <sourceStateNames>Area rendered</sourceStateNames>
      <targetStateNames>Idle</targetStateNames>
    </rules>
  </processes>
</ParADE:ParadeModel>
```

Appendix E

Video XML Simulation File

```
<?xml version="1.0" encoding="UTF-8"?>
<ParADE:ParadeModel xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:ParADE="http://www.liacs.nl/ParADE/core" name="Video System">
  <processes name="Video" initialStateName="Idle">
    <rules name="read highres frames" cpu="10" mem="1000">
      <sourceStateNames>Idle</sourceStateNames>
      <targetStateNames>High resolution frame read
</targetStateNames>
      <malloc name="frames" size="5000000"/>
    </rules>
    <rules name="decode all layers" cpu="100" mem="600000000">
      <sourceStateNames>High resolution frame read
</sourceStateNames>
      <targetStateNames>Decoded to highres</targetStateNames>
      <property name="resolution" value="high"/>
      <malloc name="layers" size="100000000"/>
      <free name="frames"/>
    </rules>
    <rules name="decode lowres layers" cpu="50" mem="100000000">
      <sourceStateNames>High resolution frame read</sourceStateNames>
      <targetStateNames>Decoded to lowres</targetStateNames>
      <property name="resolution" value="low"/>
      <malloc name="layers" size="100000000"/>
      <free name="frames"/>
    </rules>
    <rules name="render highres frames" cpu="10" mem="500000">
      <sourceStateNames>Decoded to highres</sourceStateNames>
      <targetStateNames>Frame rendered</targetStateNames>
      <malloc name="frames" size="1000000"/>
    </rules>
    <rules name="render lowres frames" cpu="5" mem="100000">
      <sourceStateNames>Decoded to lowres</sourceStateNames>
      <targetStateNames>Frame rendered</targetStateNames>
      <malloc name="frames" size="100000"/>
    </rules>
    <rules name="to idle" cpu="0" mem="0">
      <sourceStateNames>Frame rendered</sourceStateNames>
      <targetStateNames>Idle</targetStateNames>
      <free name="layers"/>
      <free name="frames"/>
    </rules>
  </processes>
</ParADE:ParadeModel>
```

Appendix F

The Main function

```
#include <stdlib.h>
#include <rre.h>
#include "gtk/gtk.h"
#include "login.h"
#include "car.h"
#include "driverinfo.h"
#include "media.h"
#include "backdisplay.h"
#include "xmlsim.h"
#include "gps.h"
#include "frontdisplay.h"
#include "video.h"

int main(int argc, char *argv[])
{
gtk_init (&argc, &argv);

/* Define a standard result catcher */
RcResult result=RC_OK;

/* Define a RcIServiceRef variable */
RcIServiceRef svc=NULL;

//the interface references
CIS_ILoginRef i_login;
CIS_ICarRef i_car;
CIS_IDriverInfoRef i_di;
CIS_IMediaRef i_media1;
CIS_IMediaRef i_media2;
CIS_IMediaRef i_media3;
CIS_IVideoRef i_video1;
CIS_IVideoRef i_video2;
CIS_IVideoRef i_video3;

CIS_IXmlSimRef i_rc_sim;
CIS_IXmlSimRef i_vs_sim;
CIS_IXmlSimRef i_gr_sim;

CIS_IXmlSimRef i_reader;
CIS_IXmlSimRef i_decoder;
CIS_IXmlSimRef i_renderer;
CIS_IXmlSimRef i_interpolator;

CIS_IGPSRef i_gps;

//the service references
CIS_SLoginRef s_login;
CIS_SFrontDisplayRef s_fd;
CIS_SCarRef s_car;
CIS_SDriverInfoRef s_di;
CIS_SBackDisplayRef s_bd1;
CIS_SBackDisplayRef s_bd2;
CIS_SMediaRef s_media1;
CIS_SMediaRef s_media2;
CIS_SMediaRef s_media3;
CIS_SVideoRef s_video1;
CIS_SVideoRef s_video2;
CIS_SVideoRef s_video3;

CIS_SXmlSimRef s_rc_sim;
CIS_SXmlSimRef s_vs_sim;
CIS_SXmlSimRef s_gr_sim;

CIS_SXmlSimRef s_reader;
CIS_SXmlSimRef s_decoder;
CIS_SXmlSimRef s_renderer;
CIS_SXmlSimRef s_interpolator;

CIS_SGPSRef s_gps;

//integer to check if there has been logged in
int logged_in = 0;


/* CREATING THE SERVICES */

//Create the Login Service
result = rre_getServiceInstance( &CIS_SLogin_GUID, &svc);
//Cast to SLogin
result = RcIService_QueryInterface( svc, &CIS_SLogin_GUID,
(void **) &s_login);

//Create the FrontDisplay Service
result = rre_getServiceInstance( &CIS_SFrontDisplay_GUID, &svc);
//Cast to SFrontDisplay
result = RcIService_QueryInterface( svc, &CIS_SFrontDisplay_GUID,
(void **) &s_fd);

//Create the Car Service
result = rre_getServiceInstance( &CIS_SCar_GUID, &svc);
//Cast to SCar
result = RcIService_QueryInterface( svc, &CIS_SCar_GUID,
(void **) &s_car);
;
//Create the DriverInfo Service
result = rre_getServiceInstance( &CIS_SDriverInfo_GUID, &svc);
```

```
//Cast to SDriverInfo
result = RcIService_QueryInterface( svc, &CIS_SDriverInfo_GUID,
(void **) &s_di);

//create the BackDisplay service 1
result = rre_getServiceInstance( &CIS_SBackDisplay_GUID, &svc);
//cast to SBackDisplay 1
result = RcIService_QueryInterface( svc, &CIS_SBackDisplay_GUID,
(void **) &s_bd1);

//create the BackDisplay service 2
result = rre_getServiceInstance( &CIS_SBackDisplay_GUID, &svc);
//cast to SBackDisplay 2
result = RcIService_QueryInterface( svc, &CIS_SBackDisplay_GUID,
(void **) &s_bd2);

//create the Media Service
result = rre_getServiceInstance( &CIS_SMedia_GUID, &svc);
//cast to SMedia;
result = RcIService_QueryInterface( svc, &CIS_SMedia_GUID,
(void **) &s_media1);
//create the Media Service
result = rre_getServiceInstance( &CIS_SMedia_GUID, &svc);
//cast to SMedia;
result = RcIService_QueryInterface( svc, &CIS_SMedia_GUID,
 (void **) &s_media2);
//create the Media Service
result = rre_getServiceInstance( &CIS_SMedia_GUID, &svc);
//cast to SMedia
result = RcIService_QueryInterface( svc, &CIS_SMedia_GUID,
(void **) &s_media3);

//create the Video Service
result = rre_getServiceInstance( &CIS_SVideo_GUID, &svc);
//cast to SMedia;
result = RcIService_QueryInterface( svc, &CIS_SVideo_GUID,
(void **) &s_video1);
//create the Video Service
result = rre_getServiceInstance( &CIS_SVideo_GUID, &svc);
//cast to SMedia;
result = RcIService_QueryInterface( svc, &CIS_SVideo_GUID,
 (void **) &s_video2);
//create the Video Service
result = rre_getServiceInstance( &CIS_SVideo_GUID, &svc);
//cast to SMedia
result = RcIService_QueryInterface( svc, &CIS_SVideo_GUID,
 (void **) &s_video3);


//cerating the xml simulation services and casting them to SXmlSim
//create the xml simulation service
result = rre_getServiceInstance( &CIS_SXmlSim_GUID, &svc);
result = RcIService_QueryInterface( svc, &CIS_SXmlSim_GUID,
(void **) &s_rc_sim );

result = rre_getServiceInstance( &CIS_SXmlSim_GUID, &svc);
result = RcIService_QueryInterface( svc, &CIS_SXmlSim_GUID,
(void **) &s_vs_sim );

result = rre_getServiceInstance( &CIS_SXmlSim_GUID, &svc);
result = RcIService_QueryInterface( svc, &CIS_SXmlSim_GUID,
(void **) &s_gr_sim );

result = rre_getServiceInstance( &CIS_SXmlSim_GUID, &svc);
result = RcIService_QueryInterface( svc, &CIS_SXmlSim_GUID,
(void **) &s_reader );

result = rre_getServiceInstance( &CIS_SXmlSim_GUID, &svc);
result = RcIService_QueryInterface( svc, &CIS_SXmlSim_GUID,
(void **) &s_decoder );

result = rre_getServiceInstance( &CIS_SXmlSim_GUID, &svc);
result = RcIService_QueryInterface( svc, &CIS_SXmlSim_GUID,
(void **) &s_renderer );

result = rre_getServiceInstance( &CIS_SXmlSim_GUID, &svc);
result = RcIService_QueryInterface( svc, &CIS_SXmlSim_GUID,
(void **) &s_interpolator );

//create the GPS Service
result = rre_getServiceInstance( &CIS_SGPS_GUID, &svc);
//cast to GPS Service
result = RcIService_QueryInterface( svc, &CIS_SGPS_GUID,
(void **) &s_gps);


/* GETTING THE INTERFACES */

//get the Login Interface
result = CIS_SLogin_getProvides_login( s_login, &i_login );

//get the Media interfaces
result = CIS_SMedia_getProvides_media( s_media1, &i_media1 );
result = CIS_SMedia_getProvides_media( s_media2, &i_media2 );
result = CIS_SMedia_getProvides_media( s_media3, &i_media3 );

//get the Video interfaces
result = CIS_SVideo_getProvides_video( s_video1, &i_video1 );
result = CIS_SVideo_getProvides_video( s_video2, &i_video2 );
result = CIS_SVideo_getProvides_video( s_video3, &i_video3 );

//get the xml simulation interfaces
result = CIS_SXmlSim_getProvides_xmlsim( s_rc_sim, &i_rc_sim );
result = CIS_SXmlSim_getProvides_xmlsim( s_vs_sim, &i_vs_sim );
result = CIS_SXmlSim_getProvides_xmlsim( s_gr_sim, &i_gr_sim );

result = CIS_SXmlSim_getProvides_xmlsim( s_reader, &i_reader );
result = CIS_SXmlSim_getProvides_xmlsim( s_decoder, &i_decoder );
result = CIS_SXmlSim_getProvides_xmlsim( s_renderer, &i_renderer );
result = CIS_SXmlSim_getProvides_xmlsim( s_interpolator, &i_interpolator );

//get the GPS interface
result = CIS_SGPS_getProvides_gps( s_gps, &i_gps);
```

```
//get the driver info interface
result = CIS_SDriverInfo_getProvides_driverinfo(s_di, &i_di);

//get the car interface
result = CIS_SCar_getProvides_car( s_car, &i_car );


/* BINDING */

//set de car interface to the driverinfo service
result = CIS_SDriverInfo_bindTo_car(s_di, i_car);

//set the driverinfo interface to the frontdisplay service
result = CIS_SFrontDisplay_bindTo_driverinfo(s_fd, i_di);

//set the media interface to the backdisplay service
result = CIS_SBackDisplay_bindTo_media(s_bd1, i_media1);
result = CIS_SBackDisplay_bindTo_media(s_bd2, i_media2);

//set the xml simulation interface to the gps service
result = CIS_SGPS_bindTo_rc_sim( s_gps, i_rc_sim );
//set the xml simulation interface to the gps service
result = CIS_SGPS_bindTo_vs_sim( s_gps, i_vs_sim );
//set the xml simulation interface to the gps service
result = CIS_SGPS_bindTo_gr_sim( s_gps, i_gr_sim );

//set the media interface to the fron display service
result = CIS_SFrontDisplay_bindTo_media(s_fd, i_media3);

//set the car interface to the front display service
result = CIS_SFrontDisplay_bindTo_car(s_fd, i_car);

//set the gps interface to the front display service
result = CIS_SFrontDisplay_bindTo_gps(s_fd, i_gps);

//set the xml simulation interface to the front display service
result = CIS_SFrontDisplay_bindTo_rc_sim( s_fd, i_rc_sim );
result = CIS_SFrontDisplay_bindTo_vs_sim( s_fd, i_vs_sim );
result = CIS_SFrontDisplay_bindTo_gr_sim( s_fd, i_gr_sim );

//set the simulation interfaces to the video service
result = CIS_SVideo_bindTo_reader( s_video1, i_reader );
result = CIS_SVideo_bindTo_reader( s_video2, i_reader );
result = CIS_SVideo_bindTo_reader( s_video3, i_reader );

result = CIS_SVideo_bindTo_reader( s_video1, i_decoder );
result = CIS_SVideo_bindTo_reader( s_video2, i_decoder );
result = CIS_SVideo_bindTo_reader( s_video3, i_decoder );

result = CIS_SVideo_bindTo_reader( s_video1, i_renderer );
result = CIS_SVideo_bindTo_reader( s_video2, i_renderer );
result = CIS_SVideo_bindTo_reader( s_video3, i_renderer );

result = CIS_SVideo_bindTo_reader( s_video1, i_interpolator );
result = CIS_SVideo_bindTo_reader( s_video2, i_interpolator );
result = CIS_SVideo_bindTo_reader( s_video3, i_interpolator );
```

```
//set the video interface to the media service
result = CIS_SMedia_bindTo_video( s_media1, i_video1 );
result = CIS_SMedia_bindTo_video( s_media2, i_video2 );
result = CIS_SMedia_bindTo_video( s_media3, i_video3 );


/*STARTING SERVICES*/

//CIS_ILogin_login( i_login, &logged_in );

//if( logged_in == 1 ){
//start the car service
result = CIS_SCar_start(s_car);

//start the frontdisplay service
result = CIS_SFrontDisplay_start( s_fd );

//start the backdisplay services
result = CIS_SBackDisplay_start( s_bd1 );
result = CIS_SBackDisplay_start( s_bd2 );
//}

gtk_main();

return (EXIT_SUCCESS);
};
```