Project IST 10069 AIT-WOODDES

Deliverable M2.1

Methodology
for developing real time embedded systems

Contractual Date of Delivery to the EC:	**T0+18**
Actual Date of Delivery to the EC:	**T0+21**
Deliverable Responsible:	**CEA**
Participant(s):	**CEA, I-Logix, Uppsala, OFFIS, PSA, MECEL, ICOM**
Workpackage:	**1.3**
Security:	**Public**
Delivery Type:	**Report**
Version:	**V1.0**
Total number of pages:	**130**

Abstract:

This intermediate deliverable defines the basis of the guidelines for the use of UML notations, profiles and modelling tools for the modelling of real time embedded systems. Its purpose is to elaborate a methodology framework, involving several stages as defined in M3, that can be used to describe the way from analysis and design down to validation and testing. While the work on UML Profile (WP1.2) is focussed on detailed formalisation of notation and concepts needed for real-time embedded systems modelling, this task will define which concept and diagram has to be used at the several steps of the development. It will bring into the methodology the experiences of the industrial applications conducted in WP4, in order to harmonise the guidelines with the project tools and to smooth the transition between the different stages of the framework.

Modelling guidelines will be written in order to be usable by any industrial user of the AIT-WOODDES methodology. Illustrative examples will show how the methodology is applied (WP4.3). In addition, implications of the methodology on the use of the project tools will be analysed and described by the tool vendors (i.e., in a dedicated user manual). In particular they will take care of the consistency between the guidelines and the support provided by the toolset.

Keyword list:

Methodology, Analysis model, Design model, Prototype model, Implementation model, Traceability , Consistency rules, Validation methodology, Validation models, Real-Time UML, UML profile, Active objects, State diagram, Timing notations, Concurrency model, Scheduling policy, Model consistency

Programme Name:	IST
Project Number:	IST-1999-10069
Project Title:	AIT-WOODDES
Partners:	Co-ordinator: PSA (F) Principal Contractors: MECEL (S), CEA-Leti (F), I-LOGIX (IL), ICOM (EL), UPPSALA (S), OFFIS (D)
Document Number:.................................	M2/CEA/WP1.3/V0.1
Work-Package:..	WP1.3
Deliverable Type:.....................................	Report
Contractual Date of Delivery:................	T0+18
Actual Date of Delivery:..........................	T0+21
Title of Document:...................................	Methodology for developing real time embedded systems
Author(s):	CEA, I-Logix, Uppsala, OFFIS, PSA, MECEL, ICOM
History:..	version 1.0
Availability:...	This report is limited to AIT-WOODDES consortium distribution

Table of Contents

TABLE OF CONTENTS I

1 THE AIT-WOODDES METHODOLOGY 1

1.1 GENERAL PROCESS OVERVIEW 1
1.2 A MODEL DRIVEN METHODOLOGY 5
1.3 CLASS VS. OBJECT 5
1.4 CASE STUDY: A CAR SPEED REGULATOR 6

2 PHASE – "BUILD ANALYSIS MODEL" 9

2.1 ACTIVITY – "BUILD PRELIMINARY ANALYSIS MODEL" 10
2.1.1 Activity – "compile dictionary" 12
2.1.2 Activity – "describe use case" 14
2.1.3 Activity – "describe high-level scenario" 19
2.1.4 Activity – "classify actors" 22
2.1.5 Summary 23
2.2 ACTIVITY – "BUILD DETAILED ANALYSIS MODEL" 24
2.2.1 Activity – "build structural basis" 26
2.2.2 Activity – "describe structure view" 29
2.2.3 Activity – "describe behavioural view" 55
2.2.4 Activity – "describe interactions view" 90

3 PHASE – "BUILD DESIGN MODEL" 104

3.1 USING UML FOR DESIGN 104
3.1.1 Artifacts 104
3.1.2 Roles\Workers 111
3.1.3 Phases and Activities 113

4 PHASE – "BUILD IMPLEMENTATION MODEL" 130

4.1 OVERVIEW OF THE AIT-WOODDES APPROACH 130
4.2 DIAGRAM TO USE FOR IMPLEMENTATION 130
4.2.1 The Component Diagram 131
4.2.2 The Deployment Diagram 132
4.3 MODEL CONSISTENCY RULES 133
4.4 DEPLOYMENT AND TARGETING 133
4.4.1 Preliminary Assessment 133
4.4.2 Tool Support 134

5 PHASE – "BUILD PROTOTYPE MODEL" 136

5.1 DIAGRAM TO USE 136
5.2 MODEL CONSISTENCY RULES 136

6 PHASE – "VALIDATE MODELS" 137

6.1 VALIDATION METHODS OF THE DEVELOPMENT PROCESS 137

6.2 VALIDATION METHODS FOR UML MODELS.. 138
6.2.1 Sanity Checks.. 138
6.2.2 Simulation... 139
6.2.3 Formal Verification... 140
6.2.4 Testing... 145
6.2.5 Other Methods.. 146
6.3 VALIDATION METHODS FOR IMPLEMENTATIONS.. 146
6.3.1 Sanity Checks.. 146
6.3.2 Simulation... 146
6.3.3 Testing... 147
6.3.4 Other Methods.. 148

APPENDIX 1 - UML BASICS 149

1 USE CASE MODELING 149

GLOSSARY.. 152

BIBLIOGRAPHY 153

1 THE AIT-WOODDES METHODOLOGY

This chapter is divided into three parts. The first part sets out the original principles on which the method was based. The second explains the approach used by this method to develop embedded real time control systems. Part three then describes the characteristics of a simple speed regulator application that serves to illustrate the concepts discussed all along other sections of this document.

1.1 General process overview

The process engineering concepts used all along this document are extracted from the Software Process Engineering Metamodel (SPEM) defined by the OMG [1]. In case of you are already used to apply an exiting process, the SPEM document defines in one annex a translation table for the terminology used in this document with different existing engineering process such as the RUP, OPEN, …

The AIT-WOODDES methodology relies on the software development cycle presented in [2]. The development cycle phases covered by the methodology are the following: analysis, design, prototyping, implementation, verification and validation. Progression from one of these classical phases to the next is achieved by a continuous and iterative process of refining UML models (Figure 1). As we can see on the process outline, the AIT-WOODDES profile will be used all along the development lifecycle. Indeed, the profile contains all the UML extensions definition introduced to ease the modelling of real-time systems with UML.

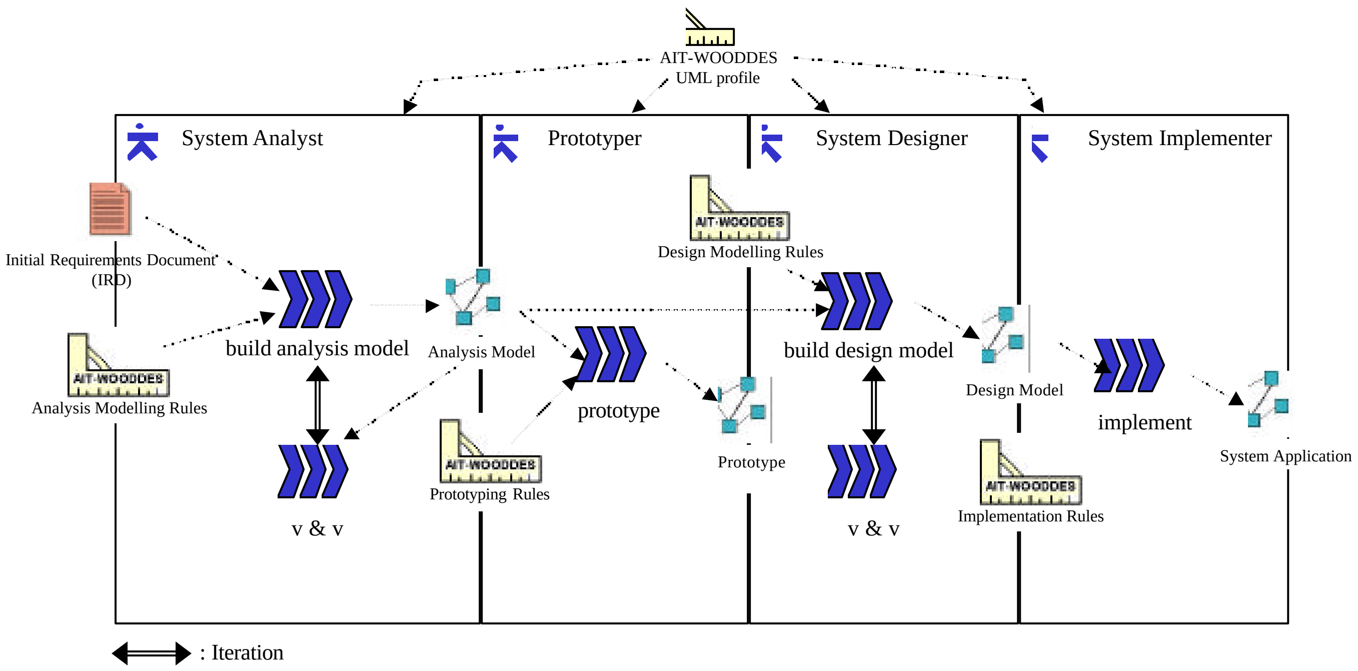


Figure 1: A continuous, iterative process, from specification to implementation phase.

Phase “build analysis model”

There are two important activities during the specification of the requirements – preliminary and detailed Analysis Modelling. Preliminary Analysis Modelling is concerned with specifying the overall functions of the application, in very general terms, as well as the interactions. Detailed Analysis Modelling then provides as thorough and accurate as possible an assessment of the functions to be performed by the system. At this stage, in addition to various structural and functional aspects, the designer specifies the real time behaviour of the application. To model the constraints pertaining to quantitative real-time features (e.g.: deadlines, periods, etc.) and qualitative one (e.g.: concurrency, parallelism, etc.), the engineer applies the methodology described in the following sections and relying on the UML profile defined in [3].

Phase "build design model"

The goal of the design phase is to choose a single "optimal" solution for the system described in the analysis. It identifies things such as concurrency models (which objects are active), scheduling policies, organization of software elements within deployable components, inter-processor communications, error-handling policies, etc.

The design phase is divided into three sub-phases: architectural, mechanistic, and detailed. The architectural design specifies overall strategic decisions such as concurrency model and distributions of components across processor nodes. The mechanistic design specifies the way individual objects collaborate sometimes by adding "glue objects" such as containers, iterators and smart pointers. The detailed design specifies the internal structure and behaviour of every class. This includes internal data structure and algorithm detail.

Phase "implement"

The implementation is the pre-final stage in the system development process that builds the application on a given target. Implementation relies primarily on architecture modelling and requires (as expressed in Figure 1) an iterative and incremental process. According to [14], modelling the system's architecture is an essential stage which is represented by five interrelated views: design, use case, process, deployment and implementation. Regarding implementation in our case, we consider that it encompasses:

- software development, i.e. production of source code, compilation into binary files and linking with external libraries e.g. of RTOS primitives for a given target platform;
- deployment of the application modules on the actual platform (possibly a distributed one), i.e. identifying and mapping the modules to the physical nodes that form the platform on which the system runs;
- validation of the final application that is running i.e. through debugging (e.g. with a kind of design tracer that traces back to the specification), and through testing (e.g. with a some test campaigner that executes test scenarios possibly derived from the specification).

For many real-time systems specially embedded systems, it is often the case that important architecture decisions are made early in the design cycle, but have then to be amended during the development process, generally when the implications of these architectural choices are perceived e.g. at prototyping or implementation. For this reason, it is important that architecture modelling covers:

- the mapping between system elements and physical elements e.g. tasks to processors, data components to storage devices;
- the hardware characteristics such as speeds, capacities etc.;
- the physical platform structure, processing / storage nodes, interfaces;
- Hardware / Software partitioning and interfaces.

Typical UML diagrams to represent these aspects are the **component and deployment diagrams**, although additional information has to be put via element properties and relationships. Typical aspects that should be addressed here are the identification of concurrent tasks, task communication, access to shared resources, etc.. In addition for a distributed application, deployment may often affect the way in which components are defined. For example, components cannot cross nodes or processes, and thus there may be a need to split a component into two or more components.

There are two more essential concerns that must be validated when constructing the implementation of a real-time system:

– How to test that schedulability and timeliness constraints are met?

– How to trace from requirements down to implementation?

A possible approach to do this is to derive **sequence diagrams** on which timing requirements can be annotated. Then tests can be constructed with the purpose of testing that such sequence diagram is always feasible at run time. More generally if these sequence diagrams are derived from (i.e. materialise) system use cases, then it should be easier to trace and to test the requirements at implementation level. Moreover it should also be easier to identify the parts of the system that are likely to be impacted by a specific set of requirement changes.

Phase "prototype"

At the end of analysis or design stages, the user enriches his model by supplementing the model of the application in order it can be executed. For example, the body of the object operations will be complemented in order the execution of the application could supply some analysable results.

The prototype is build in three stages: models transformation (e.g. to apply automatically some design patterns, …), generation of code (e.g. coding rules, …) and building of executable files (Figure 2).

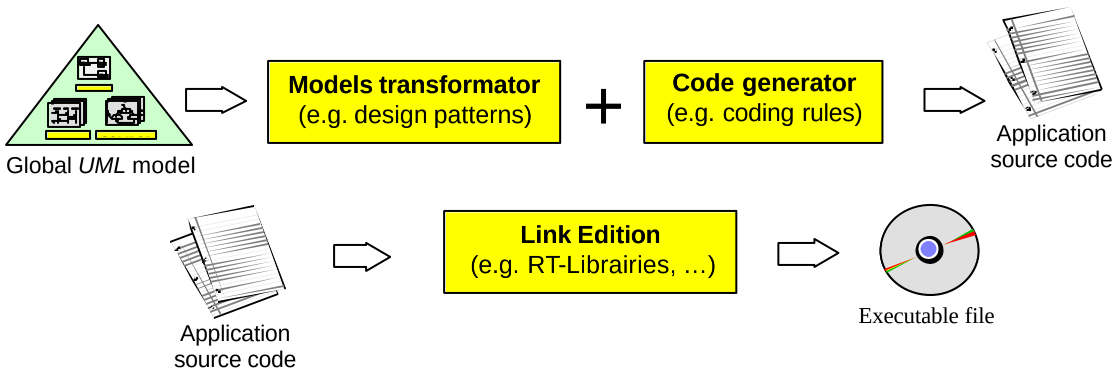


Figure 2: Support for Application Prototyping.

For prototyping purpose, real time features specification contained in the application model will be taken into account by a dedicated framework. Indeed, the tool used during this process phase will have to supply a set of tools (e.g. code generator, models transformers, …) and a set of libraries that will allow an automatic implementation of real-time concerns specified in the application models. For example, the ACCORD platform supplies a multi-tasking support through the Objecteering tool of Softeam. In this case, the real time implementation is automatically realized through a specific code generator and a set of pre-defined libraries.

Phase "verify & validate"

As indicated in Figure 1, the validation of a system is performed in parallel with all three stages of the development cycle. In addition, validation is also required after the implementation stage has finished. The overall purpose of performing validation is to locate and eliminate problems in the models as early as possible. This can be done using various methods. In section 3, we identify a set of methods that can be applied during the development cycle of a system to perform validation. The methods can roughly be divided into four groups:

- Sanity Checks – the purpose of the sanity checks are to validate that assumptions made during the development process are not violated at later stages. The following sanity properties are of particular interest: syntax and type checking, consistency checks, conformance checks, code sanity checks, and dead-code detection. Some sanity checks are performed on the UML models, and others on the actual implementation.
- Simulation – during simulation virtual executions of a system model (or implementation) is performed to investigate the dynamic behaviour of the system. This is useful for increasing the designer's confidence in the model and for debugging. We shall consider simulation methods that are performed randomly, user-guided, trace-guided, and trace-guided with simulated time, and performance evaluations. Simulations are mainly performed on the UML models produced during the system development.
- Verification – these methods are applied to formally establish that a system behaves according to properties expressed as logical formulae, sequence charts, or by observers in the model. The properties may be derived from requirement specifications, or design documents produced in the earlier development stages. The properties that can be verified include: safety, liveness, deadlock, live-lock, time-stops, and zeno behaviours. Normally, theses checks are performed on the UML models produced during the design and analysis stages.
- Testing – the methodology of testing is to execute a set of test-runs of an (implementation) model to identify that it behaves according its specification, and that the different sub components of a system behaves correctly when composed together. Various methodologies for testing exits, including glass-box testing, regression testing, conformance testing, run-time verification, and exhaustive test-case generation. Testing is mainly applied to the implementation model.

Some of the described validations methods can be applied manually. However, tools within the WOODDES consortium support most of them. In section 3, we shall describe the validation methods in more detail, and how the tools in WOODDES can be used to perform the suggested validations.

1.2 A model driven methodology

With the approach depicted within this document, modelling of a real time application is essentially based on three models that are in themselves partial, but consistent and complementary views of a larger, global model (Figure 3).

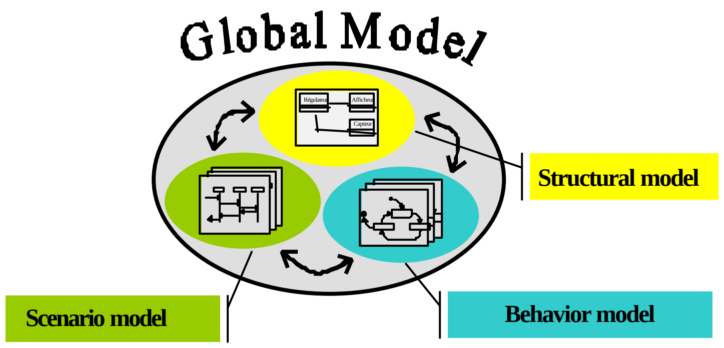


Figure 3: Construction of a Global Model.

The general idea behind application development is to always manipulate the same model, thus constantly and iteratively refining it at each phase in the modelling process. The use of a same formalism, i.e. UML, all along the development cycle eases to achieve a continuous and uniform development. Partial models contributing to the global model are as follows:

Structural model: This model defines the general architecture (topology) of the application in terms of classes and the relations between them. Its modelling function is limited to the "class" level of the application, i.e. to specifying both local class properties and those affecting the application as a whole (which means accounting for all necessary and possible interactions between classes). The structural model is described in particular by *UML* class diagrams.

Behaviour model: This model defines the behaviour of classes involved in the application. It is likewise concerned with the "class" level only, i.e. with specifying class behaviour. The behavioural model introduces two object views: that of the protocol, which specifies the global behaviour (also known as the "life cycle") of the object; and the "triggering" view, which accounts for reactive behaviour of objects such as reactions to received signals, periodic behaviour, etc. The model may also describe the behaviour of class operations.

Scenario model: This model is concerned with application "instances" and defines message passing between these various instances for the purpose of performing a given task. The interaction model is specifically described by *UML* use case and sequence diagrams. It has an additional facet to enable specification of application start-up and initialisation features of an application. Indeed, the application installation is likewise specified via the interaction model, using a specific sequence diagram dedicated to this aspect.

1.3 Class vs. Object

A distinction is made in subsequent discussions between model elements characterizing properties common to several objects (at the *"class"* level) and those characterizing specific class instance uses which are thus said to involve the *"instance"* or *"instancing"* level of the application).

During application modelling, mapping of a model to the real world in which the application will operate, means reasoning concretely, in terms of objects as "instances". Formalization and factorising of object properties, on the other hand, requires a more abstract type of reasoning (in terms of "class"). Note that class identification and formalization are the strong suit of the object oriented approach, since they facilitate both the modularity and reusability of application elements. Object-oriented methods thus tend to place particular emphasis on "classes". However, it should be kept in mind that a real application comprises only instances of classes, and this is the level at which full compliance with project specifications must ultimately be verified. A model defined by "classes" only, while reusable, cannot in any case be executed. Too much emphasis on class thus masks another vital modelling constraint: implementation, i.e. the "instance" level of the application. Subsequent chapters of this document will regularly underline the distinction between these differing, yet complementary viewpoints.

Remark: There is always some ambiguity in the term "object", which can refer either to:

- a set of properties shared by several entities (it then implicitly defines the generic concept of "class" by typing all entities whose properties are attached to that class); or to
- a specific entity with well-defined properties, which is considered as a particular "instance" of the class defined by the properties characterizing that entity.

In *UML*, the term "object" designates only the second of these two categories, i.e. a specific entity. The term "class instance" is also used.

1.4 Case Study: a car speed regulator

In order to illustrate the methodology described in this document, we have build an example extracted from automotive domain, a speed regulator system. This example is as simple as possible but it is also rich enough to be representative of the usual issues a user may be have to tackle.

Some automobiles are now equipped with an automatic control system that can regulate their speed to a preselected setpoint value reflecting the normal velocity at which the driver wishes to travel. While very simple, this example is nevertheless illustrative of the problems encountered in modelling onboard applications for motor vehicles. Such systems, which exhibit periodic behaviour, must also be designed for quick response and frequent interaction with the regulation environment.

The system is a speed regulator designed to maintain vehicle speed on a setpoint value selected by the driver. Speed is maintained with respect to changes in torque, which are signalled to the engine control system. To do so, the speed regulator uses the following regulating law to compute the changes:

$$dC = \frac{\arctan(k \times (Vorder - Vvehicle))}{2}$$

If $V_{order} > V_{vehicule}$, then $\delta C>0$ and speed increases. When the opposite occurs ($V_{order} \leq V_{vehicule}$), then $\delta C \leq 0$ and speed drops.

Vehicle speed is measured by a speedometer with its own display system. Measurement is performed by cycles at a frequency of 2 Hz. The speed display is refreshed at the same rate. The speedometer is assumed to indicate speed in integer form, in m/s.

The regulation system is started when the driver actuates the regulator on/off button, thus generating a start signal. This is only enabled at vehicle speeds of 50 km/h or more.

System stop is achieved in any of four ways: implicitly, by depressing the brake pedal; explicitly, by actuating the on/off button; by stopping the engine (via the starter); or when vehicle speed falls below the 50 km/h threshold. The anticipated response times for stop are: 0.5 s on braking or on/off switch actuation and 100 ms when the engine is halted or speed drops to less than 50 km/h.

Moreover, if the driver depresses the accelerator while the regulator is in service, regulation is interrupted until the accelerator is released. This interruption must take place in at most 200 ms, and system reinstatement must require no more than 250 ms. The regulator then reinstate control of vehicle speed and raises it progressively to whatever the setpoint value selected before acceleration.

Where deceleration and acceleration actions occur simultaneously, priority is given to deceleration. The regulation function is then no longer interrupted but, instead, is completely deactivated.

The speed regulator is equipped with a screen for display of speed setting and regulation system status (OFF, ON or STDY).

Input variables for the engine control system (which is outside the scope of these design specifications) are demand torque and change in torque (regulator output), which are used to generate a signal to the engine. This control signal (engine torque) is then applied to the various engine actuators (intake valves, throttle, etc.) to obtain the desired mechanical torque. The engine torque command is also computed on the basis of external, driver-controlled variables such as accelerator and gear positions. The resulting signal is then routed to the vehicle drive train, which transforms the mechanical torque value to speed. To enable prototyping of the regulation system environment, the engine control and drive systems have been grouped together as an "engine unit", for which the following equation is used to express speed as a function of the various input variables, thus completing the system control loop:

$$V = k \times (accP + dC) \times \left(\frac{c^3}{1500} + 1 \right) \times \left(\sqrt{\frac{50 - brkP}{50}} \right)$$

Symbols used for the variables included in this equation are as follows:

- k is a coefficient characterizing the simulated regulation environment. It represents engine efficiency, including that of the associated drive train.

- accP represents accelerator position and thus characterizes the demand torque (i.e. torque imposed by the vehicle driver);

- d**C** is a change in torque signalled by the regulator to the engine control system;

- c is a coefficient that characterizes a load applied to the vehicle. If the coefficient is positive, the load involved is, for example, a climb or an oncoming wind that slows the vehicle. If it is negative, it simulates downhill travel or a wind coming from behind the vehicle, which thus causes speed to accelerate.

- brkP represents the position of the brake pedal.

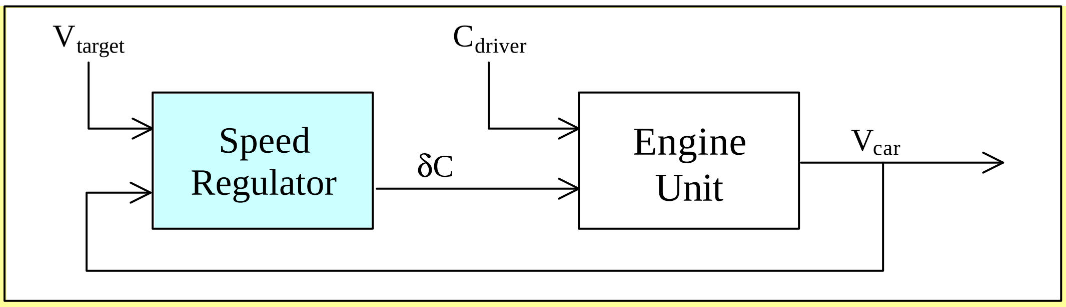


Figure 4 : Speed Regulation System Control Loop.

2 PHASE – "BUILD ANALYSIS MODEL"[1]

This phase[2] is the first one in the process life cycle. Its purpose is to describe what the system is expected to do. All the model elements involved in this phase are derived from the problem domain and from the physical constraints imposed on the system (e.g. Automotive, Telecom, …). The work product resulting from this stage are used at the design phase (see section 3) to specify how the "what[3]" is to be realized.

The specification phase involves two different roles (Figure 5):

- The **Domain Expert** role represents a professional who is an expert of the workflows and terminology that are practiced by the target audience of the application and is able to specify quantitative requirements for the application. S/he supplies the first input to the process: the Initial Requirements Document. The domain expert may be also interviewed in order to clarify any ambiguity in the requirement. Examples of Domain Experts can be a system engineer who is an expert of a certain telecom technology such as ATM or a car designer responsible for the human engineering of the car.
- The **Requirements Analyst** represents a professional whose field of expertise is to construct requirements analysis models using the methodology described here ([3]). S/he has to analyse the Initial Requirements Document and rewrites it to build the Analysis Model.

The analysis phase needs an input work product defined as the Initial Requirements Document and produced a work product called the Analysis Model:

- The **Initial Requirements Document (IRD)** work product is an informal[4] document where the client, the Domain Expert, describes what the system is expected to do. Usually it is a textual document written in natural language but it could be also any kind of document that may help in understanding of the user requirements.
- The **Analysis Model** work product is an abstraction of the physical solution focused on the domain view. It specifies the requirements in a very detailed and structured manner. It answers to the question: What is the system required to do?

The Analysis Model is itself made of two work products that supply two views of the requirements. Both views are different by their level of details:

- The **Preliminary Analysis Model (PAM)** work product is the rewriting of the Initial Requirements Document under the form of graphical models based on UML but it preserves the functional view

[1] The whole content of this section is dedicated to describe the AIT-WOODDES analysis stage. It is largely based on work done at CEA, in particular the PhD work of Sébastien Gérard [4].

[2] The "analysis" terms may be misunderstood. Indeed, "analysis model" refers to requirement description models, also called specification models. And it must be not confused with "model analysis" that may refer for example to model validation.

[3] i.e. the analysis work products that specify the system is expected to do.

[4] The "informal" term is used here in opposition of formal techniques using mathematics formalisms to specify a systems.

usually present in such documents. It is the result of the **"build preliminary analysis model"** activity.

- The **Detailed Analysis Model (DAM)** work product is an object oriented model describing the expected requirements pre-specified in the Preliminary Specification Model. It has to be complete, unambiguous, determinist and has to describe in detail all the user requirements. It is the result of the **"build detailed analysis model"** activity.

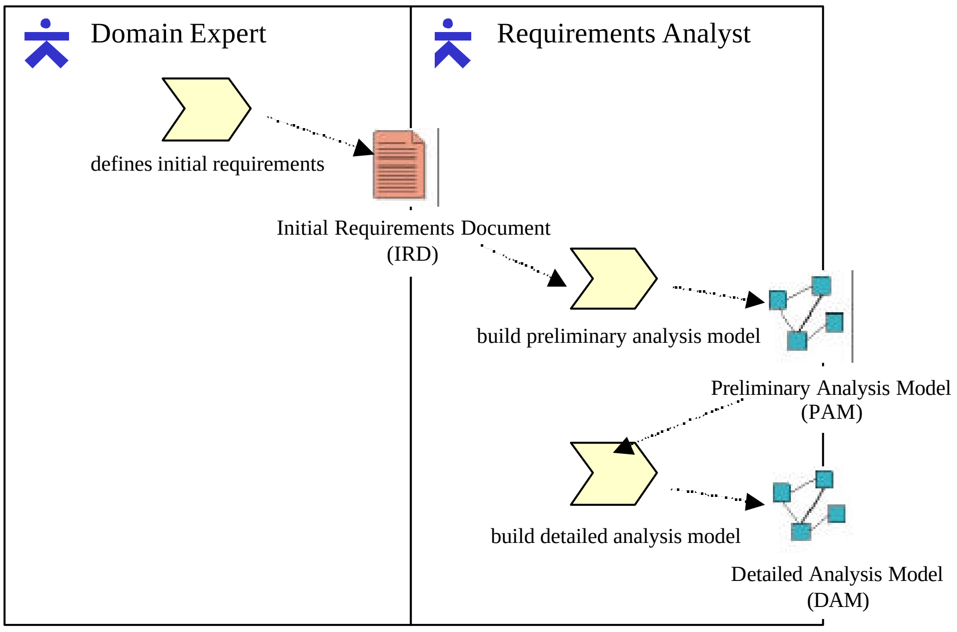

Domain Expert
Requirements Analyst
defines initial requirements
Initial Requirements Document (IRD)
build preliminary analysis model
Preliminary Analysis Model (PAM)
build detailed analysis model
Detailed Analysis Model (DAM)

Figure 5: The "build analysis models" Phase of the AIT-WOODDES Methodology.

2.1 Activity – "build preliminary analysis model"

This first activity plays a significant role in the project development cycle. It is the process activity at which product requirements (where they exist) can be reformatted as text and graphics that are easily accessible even to inexperienced users and become also formal. If it is not yet the case, this activity also offers the opportunity for the system analyst to become familiar with vocabulary and concepts specific to the application domain.

During this activity, the specific contributions of the AIT-WOODDES specification method, regarding other object-oriented approaches such as [5], [6] and [7], entail mainly the following points: a four-category dictionary structure; rules for naming model elements; introduction of a time constraint modelling concept; rule for differentiating various system actors. For more concerns relative to use cases modelling, the reader can have a look on the 6 contained some additional information pertaining to UML use cases, and s/he can also refer to the following documents[5]: [6], [8]…

The preliminary specification modelling activity refers to the following activities (Figure 6):

[5] This list is of course non-exhaustive.

- The **“compile dictionary”** activity is aimed at categorizing the domain key concepts of the IRD. The output is a dictionary;
- The **“describe use case”** activity is aimed at specifying the set of required functionalities the system expert does expect the system to do. The output is a use case diagram;
- The **“describe high-level scenario”** activity is aimed at describing the different scenarios that are possible for all use cases of the system. The output is a set of high-level sequence diagrams;
- The **“classify actors”** activity is aimed at specifying if an environment element of the system is acting on the system or vice versa.

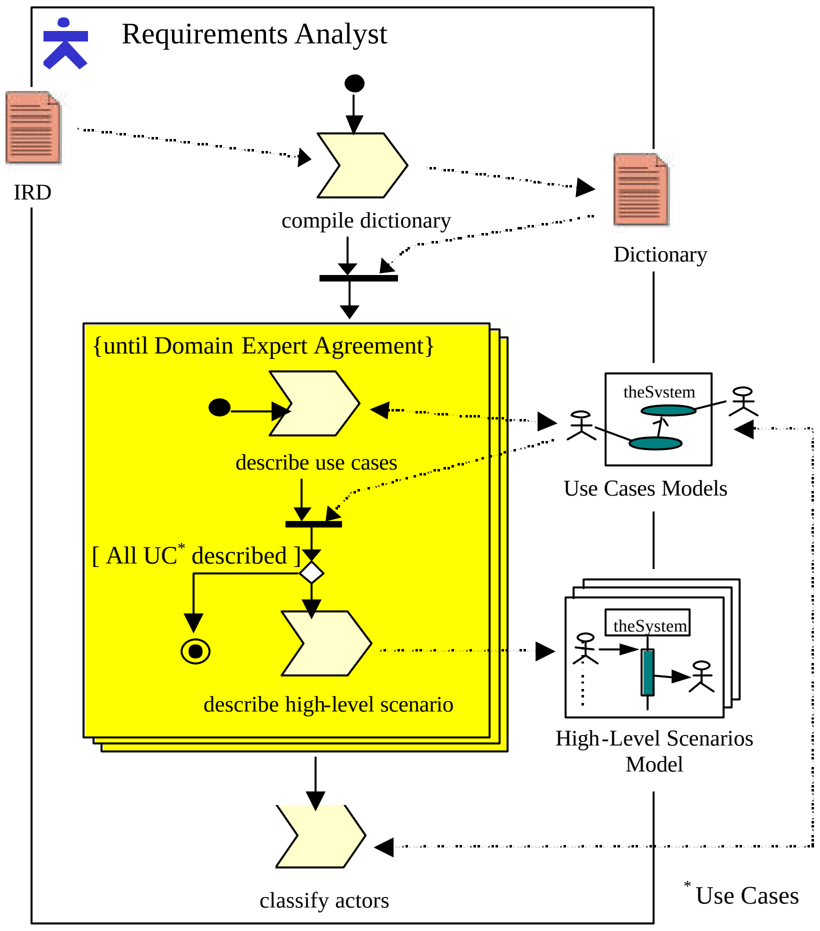


Figure 6: Activity – “build preliminary analysis model” of the AIT-WOODDES methodology.

Preliminary system requirements analysis modelling produces also the following work products (Figure 7):

- The **Dictionary** work product is a compiling of all the key concepts extracted from the IRD that are relevant to the domain and the application of interest. It takes the form of a table made of different columns defining categories the system analyst used to classify the domain key concepts of the IRD;
- The **Use Case** functional requirements via use cases and the environment elements interacting with the system through actors;
- The **High-Level Scenario** work products are sequence diagrams that describes for each use cases identified in the use case diagram a set of possible scenarios.

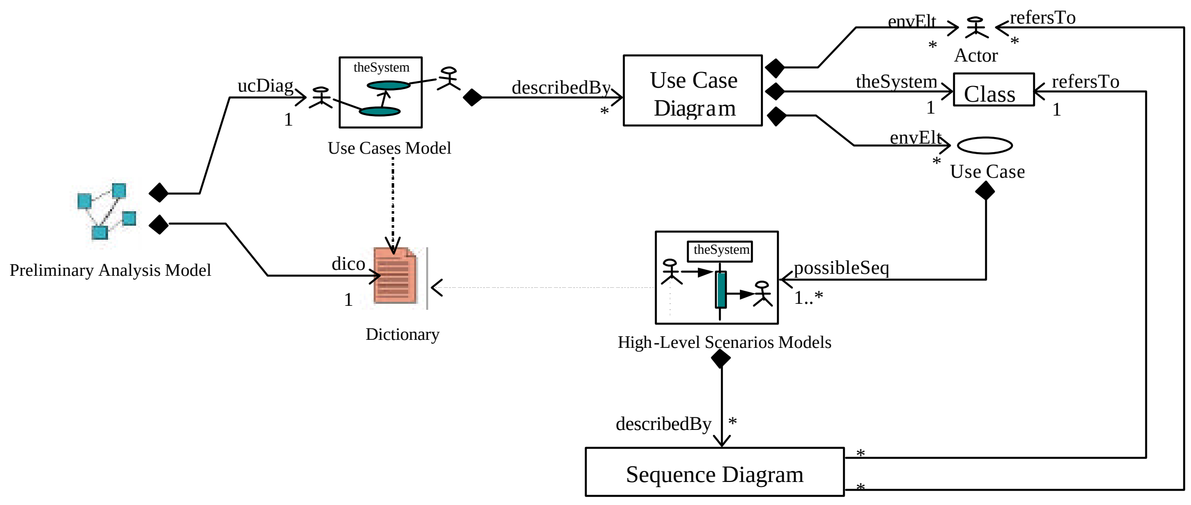


Figure 7: The Preliminary Analysis Model.

2.1.1 Activity – "compile dictionary"

The "compile dictionary" activity serves mainly two purposes:

First, compiling the dictionary teaches the system analyst about the domain covered by his application. This is because it entails domain-pertinent terms and operations. In learning their meaning, the developer becomes accustomed to the vocabulary of the domain and the development philosophy applied by its specialists. This step enhances the synergy needed between system analysts and domain experts to ensure suitable software development, while optimising chances that the final product will fully match user requirements.

Second, the dictionary helps identifying various object concepts – classes, attributes, operations, relations – involved in the application. To compile it, all of the terms used to describe the application in project specifications are grouped under three headings [9]:

- **Names**, which enable definition of concepts that correspond to classes or actors;
- **Qualifiers**, which can be associated with attribute or relationship concepts;
- **Verbs**, which are often translated as operations and impose relationships between two concepts.

In addition to the features found in dictionaries of other object-oriented approaches, the dictionary presented here provides a complementary column for description of the real-time constraints associated with domain concepts extracted from the specifications. In this dictionary, each name is associated with the verbs and qualifiers that relate to it in the specification.

Table 1 shows the dictionary compiled from speed regulator specifications:

Name (class or actor)	**Qualifier** (attribute or relationship)	**Verb** (operation)	**Real-Time Constraint**
Speed regulator		place (regulation) in service	1s
		shut down (regulation)	100 ms if engine halted 500 ms if braking or regulator turned off
	(relationship with speed display)	refresh (display)	
		interrupt regulation	200 ms
		reinstate regulation	250 ms
Speed	Setpoint	demand (setpoint)	
	Normal	achieve (normal speed)	2 Hz
Brake pedal		depress release	
Accelerator		depress release	
On/off button		activate deactivate	
Starter		turn ignition key remove key	
Display screen		display regulator status	
Control equation		calculate	
Regulator screen		display	
Speedometer			

Table 1: Dictionary for a Speed Regulation Application.

Finally, the dictionary may serve as basic input for a dedicated tool designed to trace user requirements throughout application development and successive modelling stages.

2.1.2 Activity – "describe use case"

In the same way as for most other *UML*-based object-oriented methods, the first graphic modelling activity is to define system use cases. Use cases make it possible for functional requirements to be captured in a structured way (see **6** for more details on this point).

Use case diagrams are used only at the system level to describe the behaviour at a high level of abstraction. At this moment of the development, the application to be modelled acts as a "black box"; and this box corresponds to what is referred to here as the "modelled system" or simply the "system" itself.

Constructing a use case diagram is useful at least for the following reasons [10] [11]:

1 - it enables domain specialists to specify a system from the professional's standpoint, yet precisely enough for the developer to transform it into a real, operational system;

2 - the elements making up use case diagrams (essentially actors and use cases) provide simple means of expression for use by all "players" involved in the life of a system (domain specialists, IT system developers and users). In this way, people with differing perspectives on this system can exchange ideas for the purpose of improving its development;

3 - use cases can serve as a basis for validating system implementation, for example by allowing to derive sequence diagrams that can be used for implementation testing against the initial requirements;

4 - it results in a clear identification of application boundary in order to well define what is the inside and what is the outside of the application to develop.

As described in the Figure 8, the "models use cases" activity refers to three steps: **"identify environment"**, **"identify services"** and **"identify relationships"**. The order of these different steps is immaterial and moreover the output work product resulting of this activity, i.e. the use case diagram, is build by iteration of the three steps. It is evident that the "identifies relationships" step may be executed once some actors and use case have been identified. The output of this step is then the use case model part of the PAM.

The **"Use Cases Model"** work product, as described in Figure 7, is a set of use cases (i.e. domain functions) that are clustered in a box, the system, and that are interconnected and also linked with the environment of the system via actors elements. All this models elements are depicted though the use of a use case diagram.

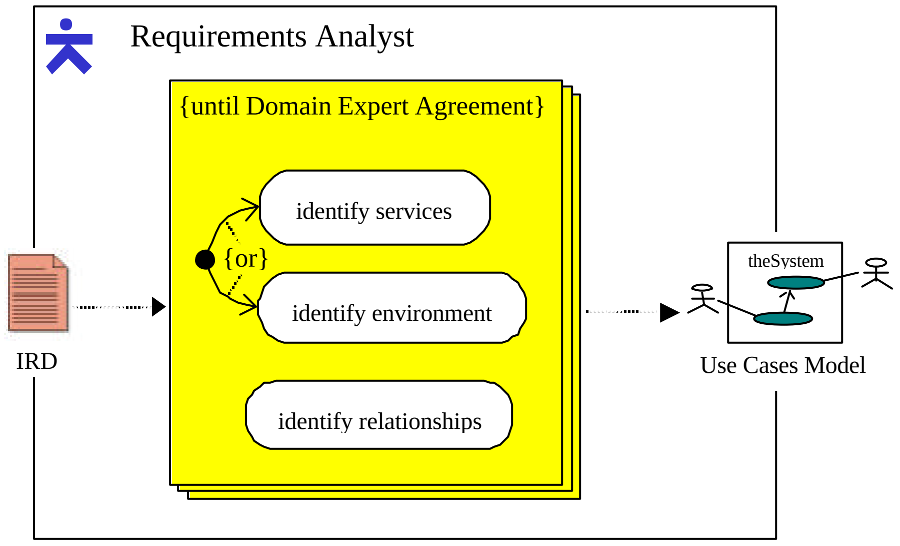


Figure 8: the "build use cases diagram" Activity of the AIT-WODDES Methodology.

2.1.2.1 Step – "identify environment"

Because a system never operates in isolation, the first job of the developer is to identify any "objects" external to, but interacting with, the system. Such objects are known as "actors".

Actors may be of different types: living beings (whether human or not); electronic devices (e.g. sensors) or software. In a use case diagram, they appear as matchstick figures[6]. However, stereotypes can be used to group them by category. Since each stereotype can be associated with a particular icon, it is possible to also supplement use case diagrams with domain-specific notations that enhance their information content.

There are three possible knowledge sources for identifying application concepts as actors: domain specialists, project specifications (which are usually drafted by domain specialists) and the dictionary, which "filters" the content of the specifications".

In the case described here, the name column of the dictionary (Table 1) contains the key application concepts drawn from these specifications. Concepts "external" to the system are then identified from this list, for modelling purposes (Table 2 shows these concepts in bold face type).

*… to generate a signal to the **engine**…*

*… The resulting signal is then routed to the vehicle **drive train**…*

*… the **engine control** and **drive systems** have been grouped together as an **engine unit**…*

*…The regulation system is started when the driver actuates the **regulator on/off button**…*

*…by depressing the **brake pedal**…*

*… by stopping the **engine** (via the **starter** …*

*… Moreover, if the driver depresses the **accelerator** …*

*… The speed regulator is equipped with a **screen** for **display** of speed setting…*

6

Table 2: Specification Excerpt Showing Concepts External to the System.

Concepts shown in bold face are then renamed, i.e. associated with a "system name" to be used in subsequent modelling and implementation phases. In order to help user in this job, the Modelling rule 1 :is defined. But in any case, this a mandatory rule and in case of an industry has already a convention notation, the users may replace it by their own. However, whatever the naming convention the user adopts, it has to include at least the following constraint: the “system name” contains no punctuation, diacritical marks or spaces. So in order to satisfy this constraint the following modelling rule is proposed.

Modelling rule 1 : System names may also be build following the naming convention rules[7]:

– if it is made up of several words, these one are concatenated on removal of spaces to become a single name;

– the first letter of each word in the name of an actor is written in upper case;

– diacritical and punctuation marks are replaced by "_".

Table 3 below lists the seven main application concepts interacting with the speed control system, yet considered to be external to it. These concepts are thus identified as system actors. The second column in this table contains the "system name" associated with each actor and that will be used within models during the next steps of the development.

Type of Actor	Assigned "System Name"
engine unit[8]	EngineUnit
speedometer	Speedometer
accelerator	Accelerator
brake pedal	BrakePedal
regulator on/off button	RegulatorOnOffButton
engine starter	EngineStarter
regulator screen	RegulatorScreen

Table 3: Speed Regulation System Actors with their "System Names".

As a further advice for finding actors, it can be said that:

– for every actor, there should be at least one user which will enact the role; a user here can be at any level of abstraction, for example, the driver or a software module;

– there should be a minimal overlap (ideally no overlap) of roles between actors. This should prevent having two actors that have essentially the same role;

[7] Each country language can add its own rules to manage their specific notational points (e.g. accents in French, …)

[8] As suggested in project specifications, engine control and drive systems have been grouped together here under the single heading "engine unit".

– all actors should be given relevant names (as in the table) and short textual descriptions of the role they play and how they use the system.

Based on the above table of system actors, it is possible to create the first *UML* diagram, i.e. the use case diagram. In this diagram, the system is depicted at the center, as a white rectangle containing the name of the developed application. Identified actors are then positioned around the rectangle (system) with which they interact in the application (Figure 9).

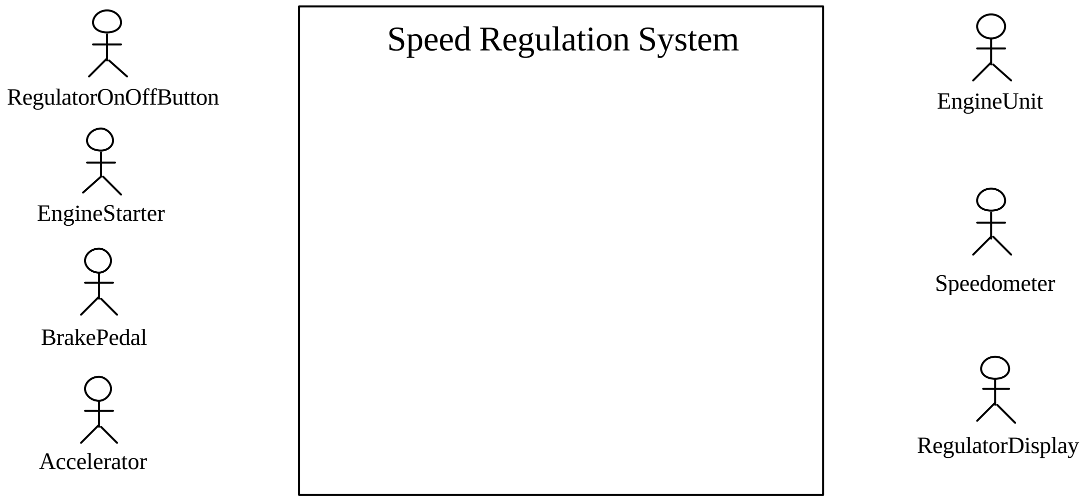


Figure 9: Depiction of Actors in a Use Case Diagram.

At the end of this first phase, in the use case diagram description, the boundary between the system and the outside environment has been clearly identified. Since the system itself is also clearly delineated, it is now possible to identify its use cases, i.e. the main functions it is expected to perform.

2.1.2.2 Step – "identify services"

Identification of main system functionality relies on the same knowledge sources as for actors – domain specialists, project specifications and the application dictionary.

Services identified in the specifications are generally associated with the verbs listed in the dictionary for key concepts of the system (in this case the regulator). In our example, there are five key functions: maintain vehicle speed at setpoint value, start, stop, interrupt and reinstate speed regulation (Table 4).

*… designed to **maintain** vehicle speed on a setpoint value…*
… The regulation system is **started**…
… System **stop** …
… regulation is **interrupted** until…
… The regulator then **reinstates** control…

Table 4: Specification Excerpt Showing Major System Functions.

These functions therefore constitute the system use cases and result, for the example of interest here, in the use case diagram given below:

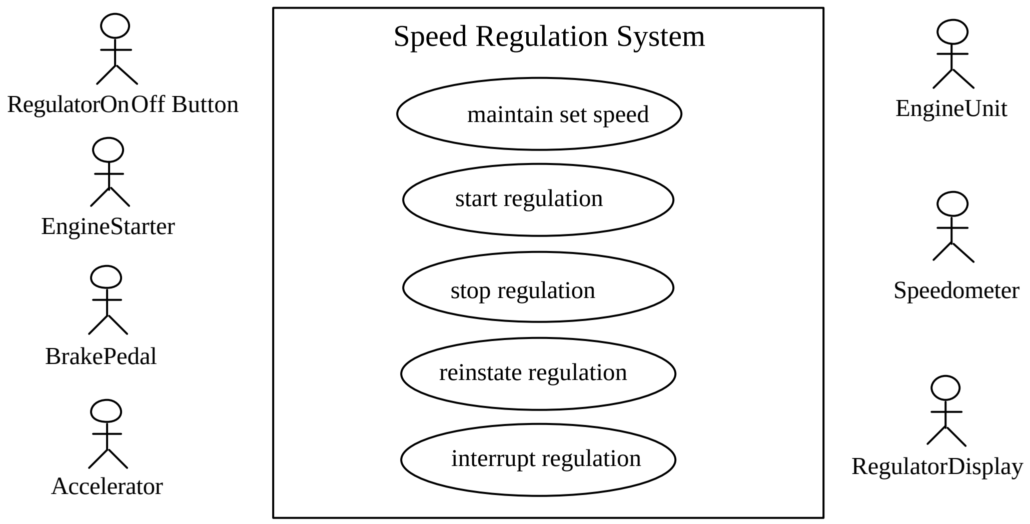


Figure 10: Depiction of Use Cases in a Use Case Diagram.

2.1.2.3 Step – “identify relationships”

The third and last step in the construction of a use case diagram consists of identifying relationships between use cases and actors. Such relationships ensure communication between actor and system for the purpose of performing the functionality associated with the use case linked to the actor. These are not "oriented" relationships and do not specify communication (types of exchanged data or commands).

Further study of project specifications and of verbs identified in the dictionary (Table 1, column 3) enables the addition of relevant links to the speed regulator use case diagram.

… Speed is maintained with respect to changes in torque, which are ***signalled to*** *the engine control system …*

… Vehicle speed is measured ***by*** *a speedometer …*

… The regulation system is started when the driver actuates the regulator on/off button , thus ***generating a start signal*** *…*

… System stop… ***by depressing*** *the brake pedal …*

*… System stop…****by actuating*** *the on/off button …*

… System stop… ***by stopping*** *the engine…*

…System stop… when vehicle speed ***falls below*** *50 km/h …*

…if the driver ***depresses*** *the accelerator… regulation is interrupted …*

… is interrupted ***until*** *the accelerator* ***is released*** *…*

… with a screen for display of speed setting and regulation system status …

Key :

Expression associated with an actor

Expression associated with a use case

Expression associated with a relationship between an actor and a use case

Table 5: Specification Excerpt Showing Links between Actors and Use Cases.

This results in the final version of the use case diagram depicted in Figure 11.

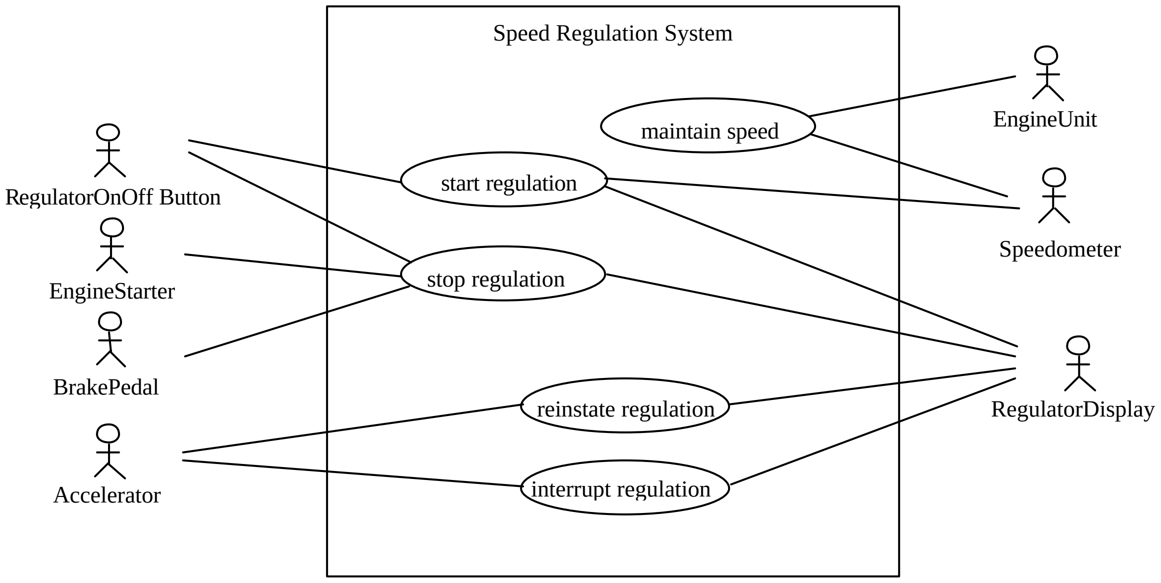


Figure 11: Use Case Diagram for a Speed Regulation System.

In the final use case diagram, the points at which a system communicates with its environment are identified, but the types, meanings, and protocol of such interactions are not yet specified. The next section describes how this process takes place.

2.1.3 Activity – "describe high-level scenario"

For each use case specified in the use case diagram work product, the various possible scenarios are then identified and described in detail. Such scenarios are to be considered as use case instances. The purpose of this phase is to obtain all available information relating to the function represented by the use case, either from domain specialists or in documents included in the project specification package. These data, which describe the relevant scenario in text form are then translated into sequence diagrams. This phase focuses on describing the sequence of messages exchanged by the system with the actors making up its environment.

Example: For the *"stopRegulation"* use case, there are three possible scenarios: "*speed regulation stopped by actuating the regulator on/off button*", "*speed regulation stopped by braking*" and "*speed regulation stopped when engine is halted*".

In the first scenario, "*regulation stopped via on/off button*", the environment sends a message to the system indicating that the on/off button has been actuated. The system then stops the regulation process and updates its display in less than a half second. The sequence diagram in Figure 12 describes the "*regulation stopped by on/off button*" of the use case "*stop regulation*". This scenario must be carried out within 500 ms.

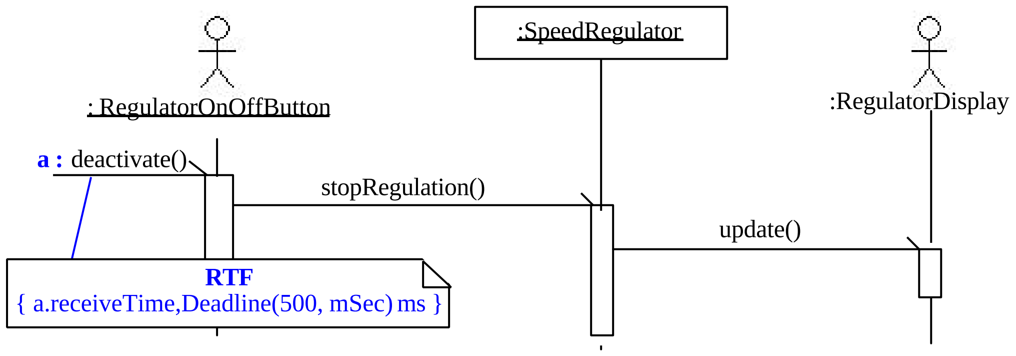


*Figure 12 : "*regulation stopped via regulator On/Off button*" Instance of "*stopRegulation*" Use Case.*

Note that real time constraints set by project specifications already appear at the preliminary analysis level. The type of notation used here is that recommended by *UML*, i.e. a constraint (in brackets) owning a timing expression. However, specialized real time constraint semantics are provided via the keyword "***RTF***" in brackets, followed by keywords and numerical characterizing the type of time constraint (deadline, period, priority, etc.). Parameters of the ***RTF***-type tagged value are expressed using, among others, *UML*-defined time functions like ***receiveTime*** and ***sendTime***, together with additional keywords (see section 2.2.3.5 p.79 and [3] for more details on these concepts). In the example shown in Figure 12, the tagged value contained in the time constraint attached to the *deactivate* message has two parameters:

– the first is the reference date used as the starting point for expressing tagged value time properties. In the example given here, the reference date is the date on which the deactivate message is received;

– the second parameter, "Deadline", is a possible *RTF* parameter specifying the deadline for a treatment. In this case, it corresponds to the maximum time allowed for the task triggered by reception of the deactivate message. This applies to all of the treatments performed in the sequence initiated by said message. In the scenario shown in Figure 12, the time constraint, i.e. "500 ms deadline", takes effect from receipt of the triggering event –*deactivate*- and lasts until the end of whatever the longest of the triggered actions.

The time constraint could also have been expressed using basic *UML* mechanisms. To do so, in a sequence diagram, *UML* proposes message tagging and use of specific time functions (such, for example, as ***receiveTime***, ***sendTime***) to express message time characteristics for the *UML* constraints shown in brackets. The number of time functions is unlimited, thus allowing the user to invent any new ones (***elapsedTime***, ***executionStartTime***, ***queuedTime***, etc.) he may need. The example given in Figure 13 uses two specific time functions, ***executionEndTime*** and ***receiveTime***, which designate the completion time of the treatment triggered by a message and the time on which a message was received respectively. The time constraint shown in brackets in the following example thus specifies, in a form equivalent to the notation introduced in ([4] and [12]), that all of the treatments performed by the task initiated by the *deactivate* message must be completed in 500 milliseconds or less.

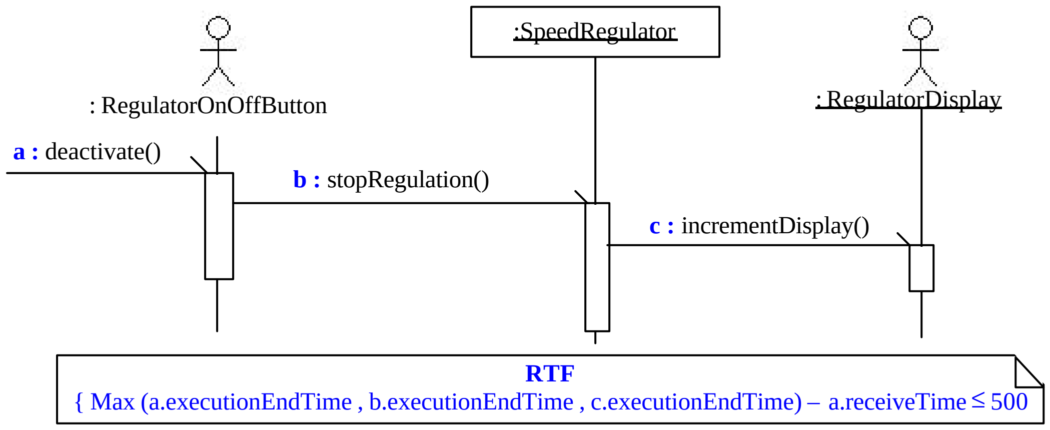


Figure 13 : Specification of a Time Constraint using Time Tags.

However, there are two advantages to using such ***{RTF=(…)}***)-type time constraint to specify time properties of an application in sequence diagrams:

– the same type of constraint will be used to specify time properties in both sequence diagrams and state diagrams, throughout the application development cycle. This provides a uniform approach to time constraint specification that can only benefit model consistency;

– the ***RTF*** may have different time characteristics, such as earliest starting time, a period, a property, etc. Its various possibilities are described in greater detail in the notation document [12].

In order to maintain a uniform approach to time constraint specification, those constraints expressed using message tags and real-time functions will be converted into an ***RTF***-type constraint. For example, the time constraint shown in Figure 13 is strictly equivalent to that depicted in Figure 12. The constraint defined by the ***RTF*** value in this example sets only one parameter: the deadline associated with the treatment. The deadline is based, in this case, on an absolute date that is specified implicitly, since, by default, the reference date is the date of reception of whatever the event triggering the sequence. In the example shown in Figure 12, the reference date is thus the date on which the *deactivate* event signal is received.

The second scenario is similar, except that this time, the event triggering the sequence is "*depressing brake pedal*".

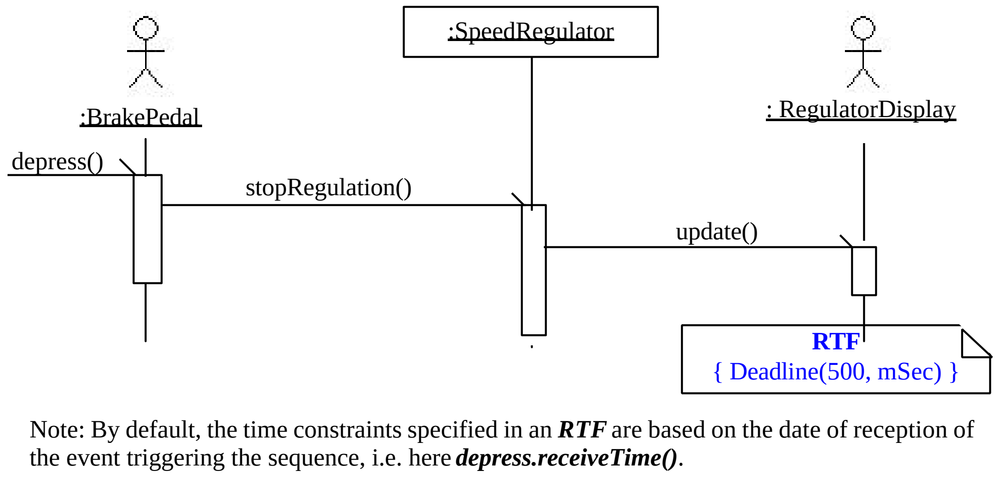

:SpeedRegulator
:BrakePedal
: RegulatorDisplay
depress()
stopRegulation()
update()
RTF
{ Deadline(500, mSec) }
Note: By default, the time constraints specified in an ***RTF*** are based on the date of reception of the event triggering the sequence, i.e. here ***depress.receiveTime()***.

Figure 14 : "speed regulation stopped by braking" Instance of the "stopRegulation" Use Case.

2.1.4 Activity – "classify actors"

The work product of the previous activity, the high-level sequence diagrams, provides the opportunity for specifying the type of role – passive or active – played by the actor with respect to the system. This information will be used in the subsequent modelling phase to automatically construct a generic component architecture and also constructs automatically from preliminary analysis models the bases of the detailed analysis models. It is in fact interesting to associate two specific packages with a same component. Indeed, by distinguishing between interaction taking place in the system-to-environment direction (i.e. how the system acts on its environment or retrieves data from the latter) from that occurring from environment to system (i.e. how the environment stimulates or controls the system), it is possible to improve the structure of the component interface with its environment. The two additional packages are thus referred to as "provided interface package" and "required interface package" respectively.

To model this characteristic of an actor, both following actor stereotypes are introduced: ***« active »*** and ***« passive »***: an active actor may be depicted as and a passive one as .

An actor cannot be active and passive at the same time. If analysis of its function reveals a situation of this type, two actors, one for each (active and passive) role must be incorporated into the model. Identification of the (active or passive) role of an actor with respect to the system requires analysis of the sequence diagrams in which it is involved and application of the following rules:

Modelling rule 2 : If an actor's role in a sequence diagram is to output messages and receive responses to these messages, said actor is considered to be active and is therefore stereotyped as such. If, on the contrary, it serves only to receive messages sent by the system, it is said to be passive (even if it responds to the messages received) and is stereotyped accordingly.

Where analysis of links between actor and system shows that the actor is both passive and active, a split is necessary into two actors: one playing the active role and the other the passive role.

This results in the following use case diagram:

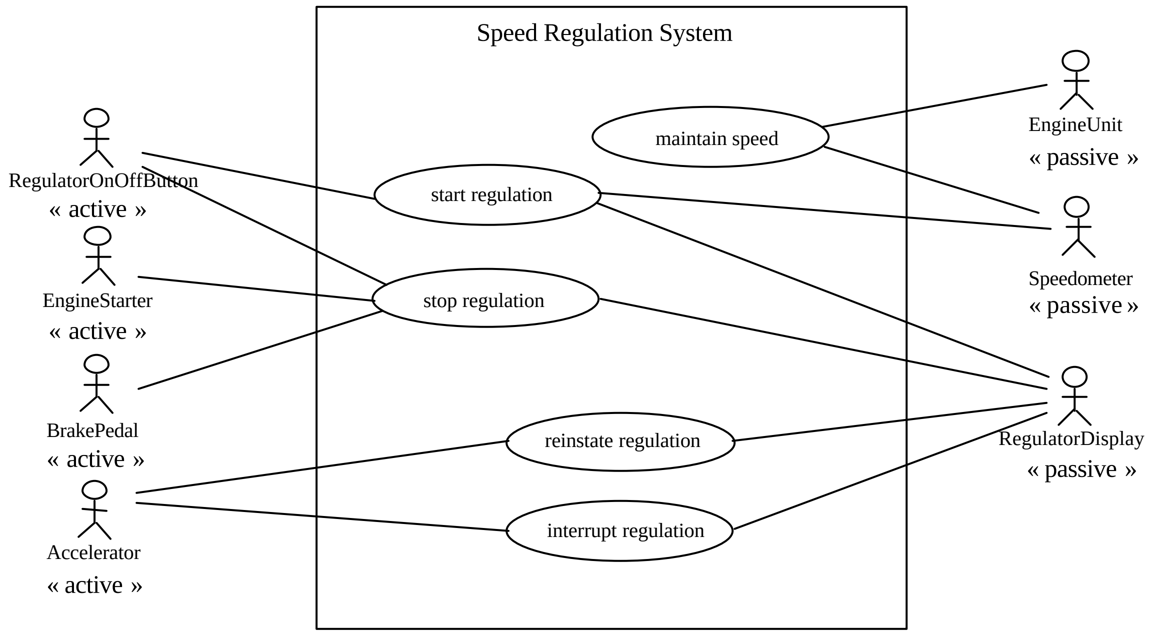


Figure 15 : Specification of Actors' Active or Passive Roles in the Use Case Diagram.

2.1.5 Summary

At the end of this phase, the global model for the application of interest thus comprises:

– a dictionary;

– a use case diagram;

– a set of high level sequence diagrams.

In the “models preliminary analysis” phase, dictionary construction enables the system designer to familiarize himself with the vocabulary of the domain for which his application is being developed. The dictionary serves as a source of information for identifying concepts to be included in the models.

The collaboration diagram then permits specification of relevant actors and use cases, together with the relationships between them. At this stage, identification of actors likewise facilitates determination of the system boundary. Once this boundary has been clearly defined, it is easier to identify the functions to be carried out by the system, i.e. its use cases. It is then possible to link the system with its environment by specifying the relationships between actors and use cases.

Finally, the high level sequence diagrams describe the different possible scenarios for each use case. These diagrams specify the sequences of messages exchanged with the environment for the purpose of performing a given function (the use case) within a given context. At this modelling level and, whenever possible, the diagrams should have an ***RTF*** (Real Time Feature)-type constraint introduced in [12]. This feature contains the time properties to be attached to the message sequence and incorporated into the constructed system.

Preliminary analysis thus enables rewriting (and, for some aspects, initial drafting) of a set of more formal specifications for the application. It also provides a series of application models that combine a functional viewpoint (which is often that of domain specialists) with the object-oriented approach that will prevail in the rest of the development process. Because this phase involves few object concepts and relies largely on "intuition", it can be performed as easily by a domain specialist (learning time for the relevant concepts and diagrams is short) as by an experienced developer. It identifies communication links between the different actors participating in system development, thus enhancing the quality of the development process and ensuring that the finished product will match desiderata expressed at the outset of the project. Finally, it lays the groundwork required for subsequent modelling phases such as the actors identification that will be translated in detailed analysis concept, in this case interface classes.

2.2 Activity – "build detailed analysis model"

The clear, unambiguous model of user needs that results from preliminary analysis still reflects primarily a functional approach. The aim of the subsequent phase is to build a global application model based on user requirements expressed essentially as use cases and sequence diagrams. This phase is concerned with finding answers to the following questions:

"What will my system be able to do (i.e. what internal functions will it afford)?"

To answer this question, it is proposed to organize global model construction around three partial, but complementary and consistent "sub-models" (see section 1.2): a structural model, a behaviour model and an interaction model. The rest of this section (i.e. **2**) is devoted to the detailed analysis stage through a description of these sub-models which, when taken together, constitute the complete application specification (structure in section 2.2.1, behaviour in 2.2.3 and interaction in section 2.2.4), and places special emphasis on their points of consistency.

The modelling of an application relies also on the three following sub-models described in the rest of the section. But the order following the one the model are here presented is not mandatory. Of course, because following object oriented approaches, statecharts are used to describe the behaviour of objects, it seams difficult to start to model an application with its behaviour without previously define some objects in the application. But the user can also choice to start by constructing either sequence diagrams or class diagrams. After a first draft of one of these diagrams has been achieved, the order to manipulate the sub-models has no importance. Indeed, each one will be continuously refined in order to reach a point satisfying the requirements. Moreover, the three models being connected through an overlapping of manipulated concepts, when a user updates a sub-model, he has to update both other sub-models. For example, if a user adds a transition in the statechart of a class and that she/he defines for this new transition a new call event, she/he will have to add the corresponding operation in the class structure he/she has modified the behaviour.

The specification phase involves two different roles (Figure 16):

The **Domain Expert** role represents a professional who is an expert of the workflows and terminology that are practiced by the target audience of the application and is able to specify quantitative requirements for the application. S/he supplies the first input to the process: the Initial Requirements Document. The domain expert may be also interviewed in order to clarify any ambiguity in the requirement. Examples of Domain Experts can be a system engineer who is an expert of a certain telecom technology such as ATM or a car designer responsible for the human engineering of the car.

All along the detailed requirements modelling activity, this actor has to performs the following activities:

- The **"build structural basis"** activity is aimed at building from the preliminary analysis model (PAM) a first version of the DAM' structural model. This first draft is also detailed/refined by successive iterations of the three following activities until the analysis model reaches a detailed and complete level satisfying the domain expert.

Once this first activity is achieved, the detailed requirements analyst operates iteratively the three following activity until s/he validates[9] all systems requirements has sufficiently detailed.

- The **"describe interactions view"** activity is aimed at describing the different detailed scenarios that are possible. This activity produces a set of sequence diagrams.
- The **"describe structure view"** activity is aimed at specifying the structure of the application under the form class diagrams.
- The **"describe behavioural view"** activity is aimed at specifying the behaviour of the application. The work products resulting from this activity is the behavioural model.

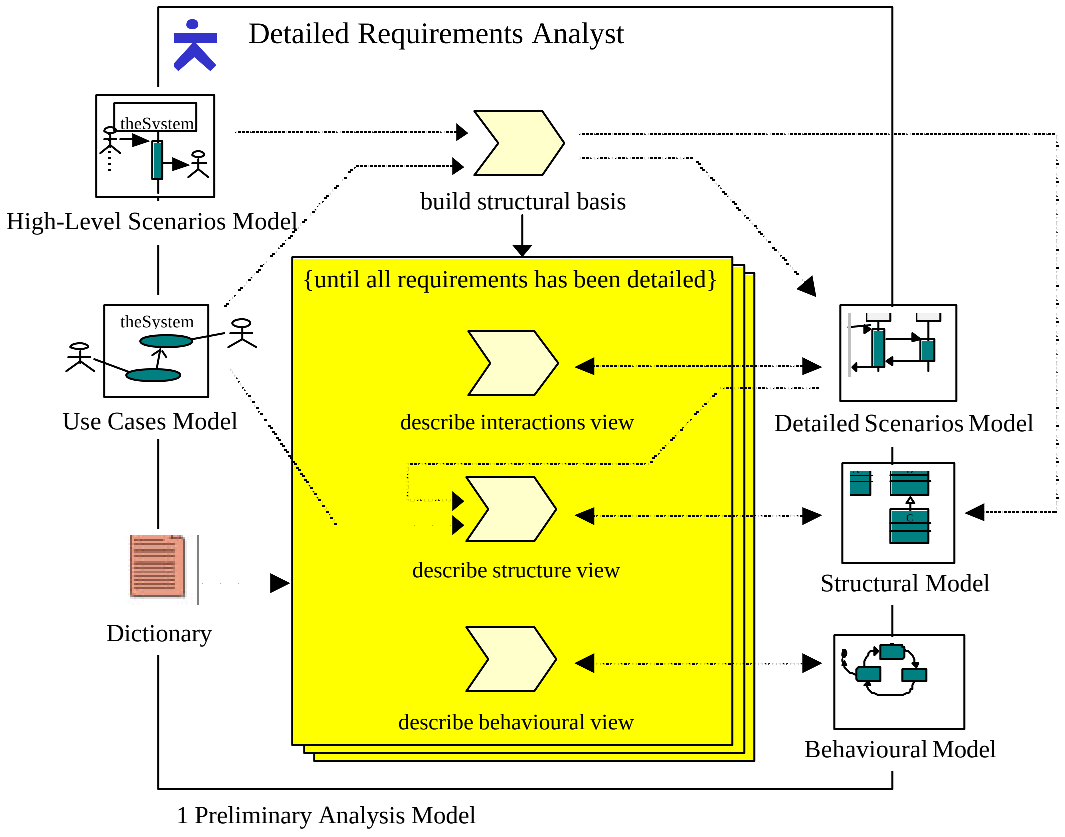

Detailed Requirements Analyst
theSystem
High-Level Scenarios Model
build structural basis
{until all requirements has been detailed}
theSystem
Use Cases Model
describe interactions view
Detailed Scenarios Model
describe structure view
Structural Model
Dictionary
describe behavioural view
Behavioural Model
1 Preliminary Analysis Model

Figure 16: Activity – "build detailed analysis model" of the AIT-WOODDES methodology.

This phase is directly fed by output work products stemming from the preliminary requirements analysis phase (the previous one) and it provides a work product called the Detailed Analysis Model (DAM, see Figure 5). As it is described in Figure 17, this work product is composed of three models that describe three complementary and consistent views (or aspects) of the system under modelling (see the section 1.2 p.5 for more detail on this subject).

- The **Detailed Scenarios Model** work product describes the interactions, i.e. message exchange sequences, that take place within the application. It is modelled under the form of sequence

[9] This point is tackled in section describing verify&validate activity.

diagrams describing the set of interactions and may be accompanied by specialized activity diagrams dedicated to clarify links/constraints between the different identified scenarios.

- The **Structural Model** work product describes the structural aspect of the system. Indeed, it expresses how the different requirements are organized. It is modelled thanks to class diagrams.
- The **Behavioural Model** work product describes the behavioural aspect of the system. It specifies the logic behaviours of a system via the use of states-transitions diagrams and algorithmic behaviours via activity diagrams.

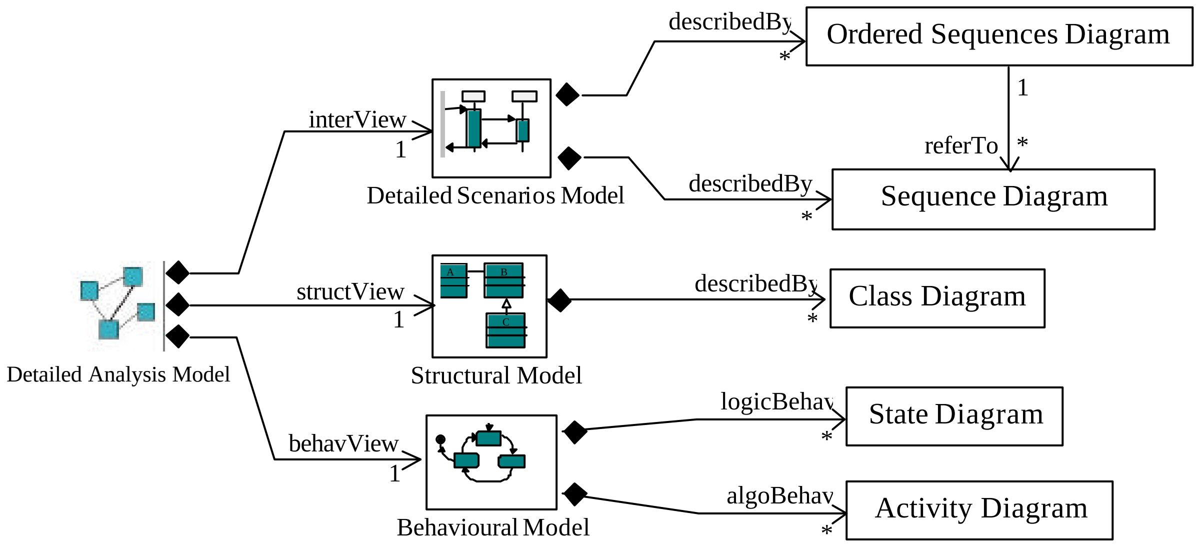


Figure 17: The Detailed Analysis Model.

2.2.1 Activity – "build structural basis"

The actor of this activity is the detailed requirements analyst. Nevertheless, this activity may be easily automated in the tool supporting the approach if this latter offers possibilities to add functionalities allowing to manipulate in read/write mode the application model.

The following paragraphs present a structural template based on a layered model. This model enables structuring the system in such a way as to facilitate the development of a component by emphasizing on specification of its interfaces (Provided and Required). Indeed, one insist on both following issues regarding to component development:

– How to use a component through the definition of a clear provided interface?

– What are the requirements of a component to be used and how to plugging it in the environment needed for its running through the definition of a clear required interface?

To facilitate construction of an application in terms of reusable components, one advocates a generic software architecture that places emphasis on separation between the core of the system and its interface with the environment. This architecture is divided into three separate packages (Figure 18). The first is defined by the ***ProvidedInterface*** package describing active interaction points (i.e. where and how the environment "stimulates" the system). The lowermost package called ***RequiredInterface*** describes the points at which the system interacts with its environment and models the environmental interface required for the system to operate. Indeed, this package defines the specification the environment needed to run the developed component will have to implement (see section **4** on prototyping for more details on this point). The middle layer describes the core of the system and is also called the ***Core*** package. It incorporates the actual application model.

For reasons of organization (to avoid confusing different concepts) all of the signal definitions are grouped together in separate stereotyped *«Signal»* packages[10] depending on their scope. As described in Figure 18, each application high level package, i.e. *ProvidedInterface*, *Core* and *RequiredInterface*, posses its own signal definition package.

This template of structure proposed a last specific package on the left hand side on the . This package stereotyped «DomainTypes» is introduced in order to collect all the domain types the analyst could need to introduce to describe the system requirements. Indeed, very often, the analyst is obliged to make some design choice to model the requirement. For example, in the speed regulator example, s/he may need to specify a parameter of an operation with a speed type. In this case, the first reflex of the analyst may be to say: "Let's have an integer or a float…". In our approach, s/he will define a new type in this package, a Speed type in our case. Therefore s/he does not need to make such design choice if s/he does not want it. Moreover, it is better such design choices to be avoided during the requirements analysis phase.

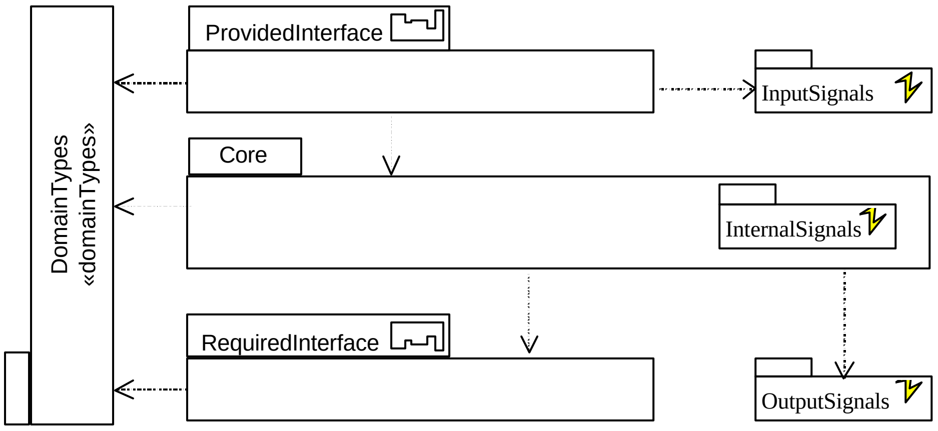


Figure 18: Generic High-Level Architecture of an AIT-WOODDES Application.

Use of two specific interface packages, *stereotyped «ProvidedInterface»* and « *RequiredInterface* », and the choice of orientations for links between them are intended to ensure clear structuring and identification of dependencies between the developed system and its environment. Such design is necessary to facilitate system integration into an existing context and enhance the reusability of the application analysis models.

Regarding to these specific packages constituent of the template structure we propose, the following modelling rules apply:

[10] A «Signal»n stereotyped package is also depicted via the icon .

Modelling rule 3 : A signal-stereotyped package must contain only signal-type elements.

Modelling rule 4 : The package stereotyped «DomainTypes» must only contain model elements stereotyped «type».

Modelling rule 5 : In the requirements analysis phase, stereotyped packages *« ProvidedInterface »* and *« RequiredInterface »* contain only interface classes.

Remark: during next stages of the development (design, implementation and prototyping), each interface class must be implemented by a class whose name is the same as that of the interface class preceded by "***I_***". The provided interface package may not be used by any other system package.

During this step, some specific classes of the application may identified automatically from the preliminary analysis data: these are the system-to-environment interface classes, also called interface classes. By applying the following modelling rule, the analyst populates the two interface packages of the application in a systematic way. If the tool supporting the method allows it, this step could be automated.

Modelling rule 6 : Each actor identified in the use case model built during preliminary requirements analysis results in identification of a corresponding interface class in the model generated during detailed analysis. All of the classes introduced in this way are interface classes[11] (and are therefore either given an "interface" stereotype or depicted as circles with class name labels (Figure 21)) and are assigned the same names as the actors they refine.

Furthermore, if an actor is stereotyped as *« active »*, the corresponding class is positioned in the provided interface package. If, on the other hand, it is stereotyped as *« passive »*, it is included in the required interface package.

Modelling rule 7 : An actor element in the PAM is linked via a dependency link stereotyped «refine» with its matching interface class in the DAM. Moreover, the direction of the dependency is from the interface class towards the actor.

In the example of the speed regulating, key concepts for the provided interface package are as follows (see Figure 21) :

– *EngineStarter*: interface between the engine starter and the regulation system;
– *RegulatorOnOffButton*: interface between the regulator on/off button and the speed regulator;
– *Accelerator*, interface between the accelerator and the speed regulator;
– *BrakePedal*, interface between the brake pedal and the speed regulator;

The following classes are derived from the use case diagram (**Figure 11**) for the required interface package:

[11] *UML* definition : "An interface is a named set of operations that characterize the behavior of an element."

– *EngineUnit*, interface between the regulation system and the engine unit;

– *RegulatorDisplay*, interface between the regulation system and the regulator display screen;

– *Speedometer*, interface between the speed regulator and the speedometer.

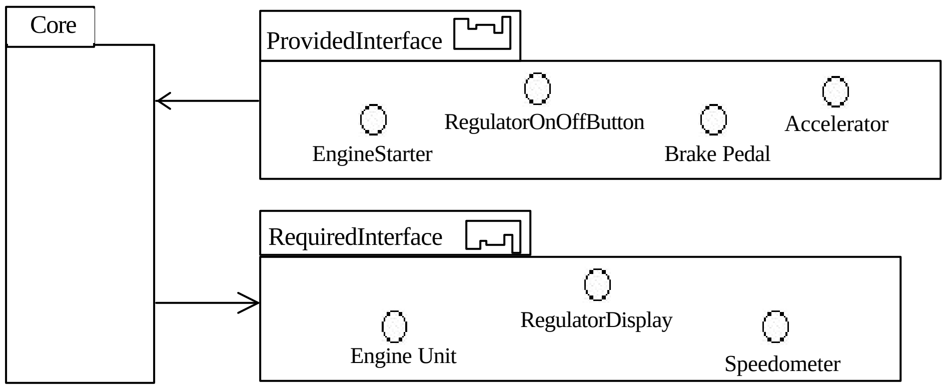


Figure 19: Interface Packages Population.

2.2.2 Activity – “describe structure view”

Structural model description is based on a generic architecture described in subsection **Erreur ! Source du renvoi introuvable.** below. As depicted in the Figure 20, this modelling activity refers to eight iterative steps. But except for the initial definition of application classes, the relative order in which these steps take place is irrelevant: they are all part of a global, iterative approach aimed at refining the application structural sub-model. The transition from specification to design is itself a gradual one, since it involves continuous increments. The specification model is completed with more detailed model elements including software issues to model how the system realizes what it has to do.

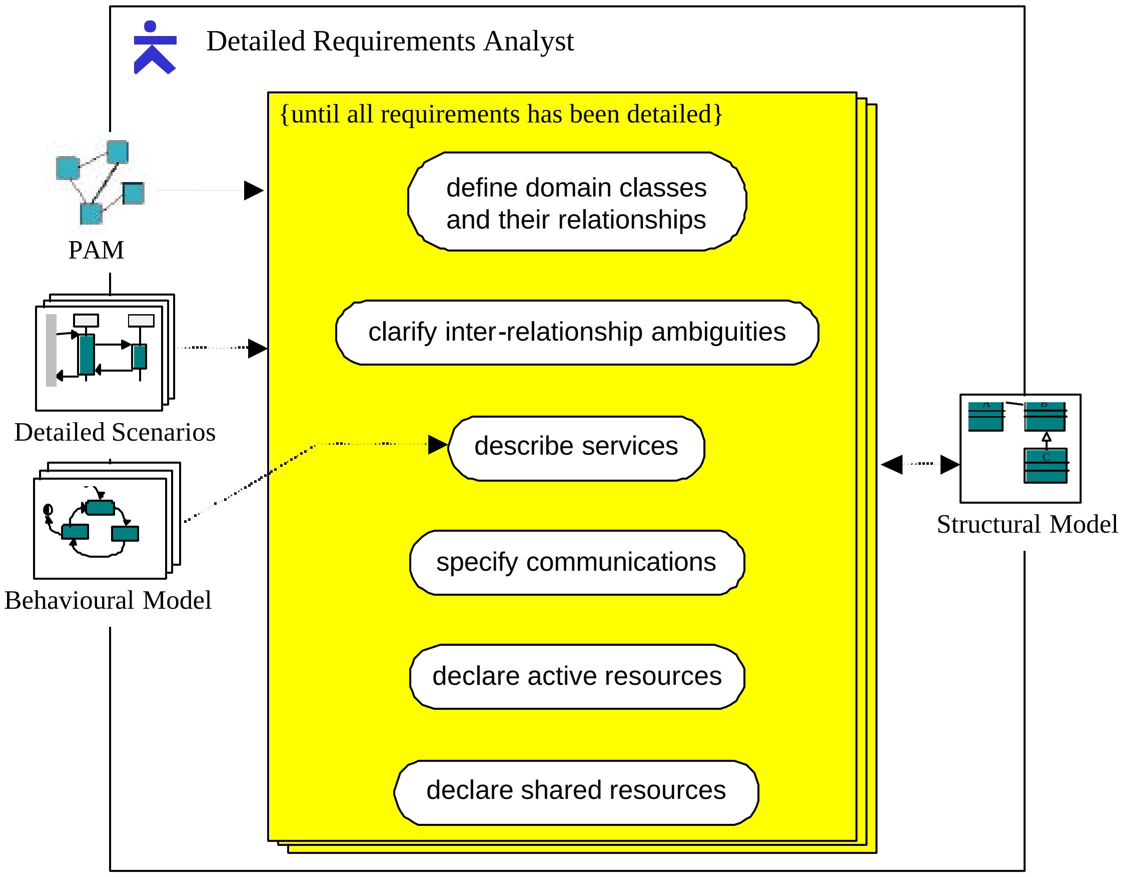

Detailed Requirements Analyst
{until all requirements has been detailed}
define domain classes and their relationships
clarify inter-relationship ambiguities
describe services
specify communications
declare active resources
declare shared resources
PAM
Detailed Scenarios
Behavioural Model
Structural Model

Figure 20: Activity – the "describe structure view" Activity of the AIT-WOODDES Methodology.

The following paragraphs give a step by step description of the model building process, illustrated by the speed regulator example. Throughout this presentation, a number of modelling rules are provided to formalize the approach. In this stage of development, the main contribution of the method is to afford specialization of certain *UML* concepts such as "active object", and, again, to formalize *UML* via the modelling rules.

2.2.2.1 Step – "define domain classes and their relationships"

For this step, no extensions are introduced there, but instead rules for using existing *UML* concepts are providing to the user to help her/him modelling her/his application.

From the dictionary and IRD, three main system core classes – ***Regulator***, ***ControlEquation*** and ***Speed*** – are identified in a first analysis. ***ControlEquation*** and ***Speed*** classes are given special attention, since these two concepts are prevalent in automated systems and automotive applications respectively. It thus seems logical to design them for reuse in other applications. One obtains also a fist draft for the structural submodel of the application:

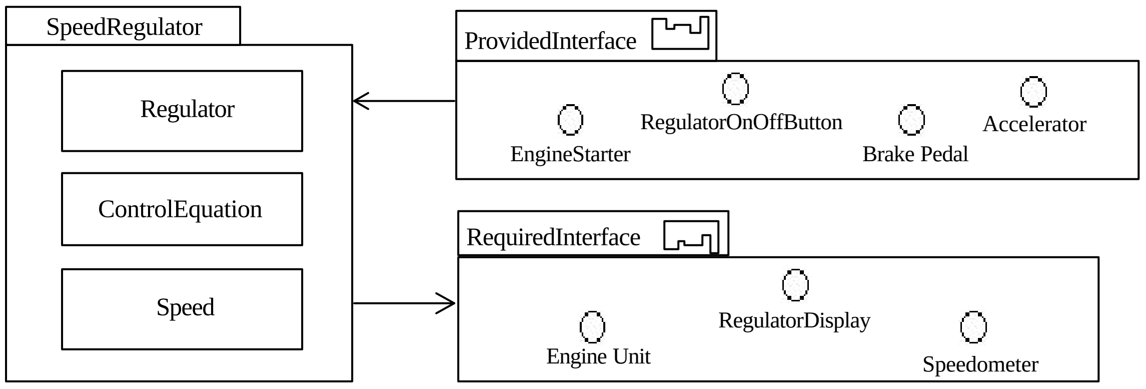


Figure 21: A first list of Classes for the Speed Regulator System.

Once system interface classes and basic core classes have been identified, these classes must be linked to one another. This is achieved in three steps:

- linking provided interface classes to system core classes;
- linking system core classes to required interface classes;
- interconnecting system core classes.

This work of defining relationships in the application may also respect as much as possible the following modelling rule:

Modelling rule 8 : Analysis of the dependencies between application concepts takes place in three steps:

- linking the provided interface package to the core package. Provided interface classes are the points at which the environment "stimulates" the system, and system core classes used by the provided interface classes are the points at which the system responds to these stimuli;
- linking the core package to the required interface package. This step serves to specify the points at which the system acts on its environment;
- interconnecting system core classes.

Because the direction of a class-to-class association impacts both the reusability of that class and the modularity of the application, orientation choices made in the structural model are of great significance. Initial analysis of the speed regulator model reveals the following relationships:

a. *Links between provided interfaces and system core classes:*

There is an association link between the accelerator interface and the regulator (Figure 22), oriented from the *Accelerator* to the *Regulator* class, with a cardinality of *0..1* (meaning that each *Accelerator* class instance is associated with a single *Regulator* instance) (Figure 23).

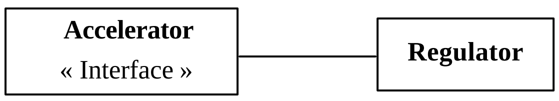


Figure 22: Association.

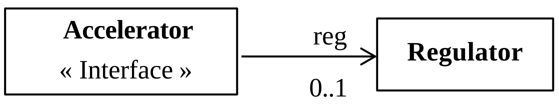


Figure 23: Orientation and Cardinality.

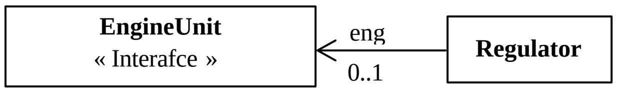
This orientation of the *Accelerator*-to-*Regulator* relationship can be justified as follows:

- If the orientation were *Regulator*-to-*Accelerator* (and, in such cases, the accelerator interface would not recognize the regulator concept), the accelerator interface could not address the regulator to signal a change in state (e.g. to inform the regulator to interrupt regulation when the driver depresses the accelerator). Such an architecture could, however, function correctly, if the regulator were designed to periodically request accelerator status, whether or not the driver is depressing it. The main drawback of this architecture would be the activity overload induced by periodic verification of accelerator status, since driver action on the accelerator is only sporadic.
- With an *Accelerator*-to-*Regulator* orientation, the accelerator interface can address the regulator to signal any action by the driver (depressing or releasing the accelerator). In this case, since the activity is known to be sporadic, and unlike what occurs in the previous scenario, there would not normally be an overload.

Similar association relationships exist between brake pedal interface and regulator, engine starter interface and regulator and regulator on/off button and regulator, the preceding remarks also apply to each of these cases.

b. *Links between system core classes and required interface classes are as follows:*

The speed regulator makes direct use of the engine unit interface to determine the torque change required to maintain speed at the setpoint value imposed by the driver. To do so, the *Regulator* class has an association link oriented toward the *EngineUnit* interface class with a cardinality of *1..**. The choice of direction for this association is guided by modularity considerations. This is because use of a regulation system only makes sense if the vehicle has an engine unit. The engine unit must, however, be able to operate without a speed regulator (this device is not (yet) a standard feature of all automotive vehicles). If this orientation were reversed, i.e. from *EngineUnit* to *Regulator* class, the application would need to allow for *EngineUnit* classes with and without speed regulators.



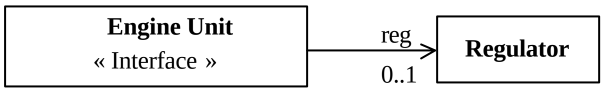


Figure 24: EngineUnit Operating Independently from Speed Regulateur. *Figure 25: Speed Regulator-Dependent EngineUnit.*

Important : Choice of relationship orientations has significant impact on the reusability of the application. Once two classes are linked by a relationship, whichever class is used by the other is no longer reusable alone.

Choice of cardinality *1..** can be justified as follows: A *Regulator* instance must control at least one power source enabling displacement of the vehicle, i.e. with a minimum cardinality of *1*. A vehicle may, however, have several power sources (hybrid engines). A *Regulator* instance must therefore be able to control all power sources of the speed-regulated vehicle, thus requiring a cardinality of *1..**.

The speed regulator also makes direct use of the interface with the display screen to update its status display (*ON*, *OFF* or *STDY*). The *Regulator* is thus oriented toward the *RegulatorDisplay* with a cardinality of *0..1* (Figure 26). Since the screen is not required for regulator operation, minimum cardinality is *0* and the maximum is *1*, to account for the fact that there is no more than one display screen per regulator.

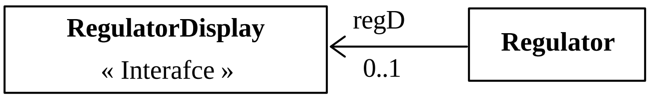


Figure 26: Regulator—RegulatorDisplay.

The Speed class makes direct use of the speedometer to acquire actual vehicle speed. This class is thus oriented toward the *Speedometer* interface, with a cardinality of *1..** (Figure 27). It seems obvious that, to operate, such a speed regulation system must be able to acquire this speed value, meaning a minimum cardinality of *1*. It is possible, however, for the system to obtain said data from several sources and to compute a particular speed on the basis of these acquisitions. It thus requires a maximum cardinality of * to enable reuse of the class in another application. Note that, in object-oriented modelling, there is no direct, clearcut relationship between orientation and the direction in which data flows. Here, for example, data flow from the speed class to the speedometer interface takes place mainly in the *Speedometer*-to-*Speed* (speed value acquisition) direction.

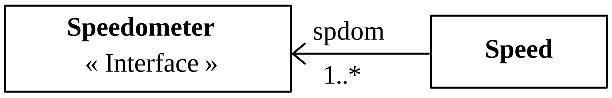


Figure 27: Speed—Speedometer *Relationship.*

c. *Analysis of relationships between system core classes:*

The speed regulator controls the speed class for the purpose of initiating updates and uses the speed value to initialise its own operation. Speed regulation cannot, in fact, be activated, if actual vehicle speed exceeds a minimum level. There is thus an association relationship, oriented from *Regulator* to *Speed*, with a cardinality of *1* (Figure 28). This orientation enables reuse of the Speed class, which can operate independently of the regulator. The cardinality requirement results from the fact that the regulator must know at least one speed value, i.e. actual speed of the vehicle at the time.

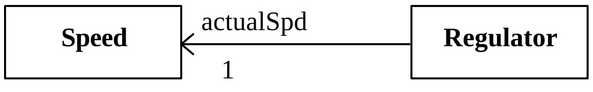


Figure 28: Regulator—Speed *Relationship.*

The speed regulator is also linked to the control equation by an association relationship oriented from *Regulator* to *ControlEquation*, with a cardinality of *1..** (Figure 29). This orientation accounts for the fact that the regulator requires the control equation to regulate speed, but that the equation can very well operate alone and even be used by another class in a system other than that regulating speed. Cardinality is *1..**, firstly because, to operate, the regulator requires at least one control equation and, secondly, because this device can use different control equations as a function of operating mode. It may have different control equations for good weather and rainy weather, for example.

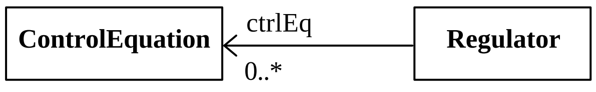


Figure 29: Regulator—ControlEquation *Relationship.*

The control equation is linked to speed by a relationship oriented from *ControlEquation* to *Speed* with a cardinality of *1* (Figure 30): the equation requires a speed value, but the speed class can function without the equation (reusability of *Speed*). Since, to compute its output, the control equation requires only the actual vehicle speed, the required cardinality is 1.

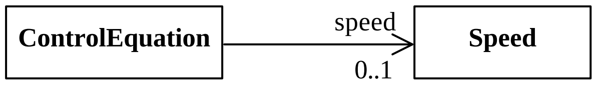


Figure 30: ControlEquation—Speed *Relationship.*

Global analysis of these links provides the global structural model shown in Figure 31. In this diagram, the interface classes which, up until now, were distinguished from other classes using the *« Interface »* stereotype, now appear as circles labelled with their class names. This new representation is linked to the specific icon attached to the stereotype. The main advantage of such notation is to simplify the diagram. But it has the drawback of "hiding" the internal characteristics of the stereotyped class, i.e. its operations and attributes.

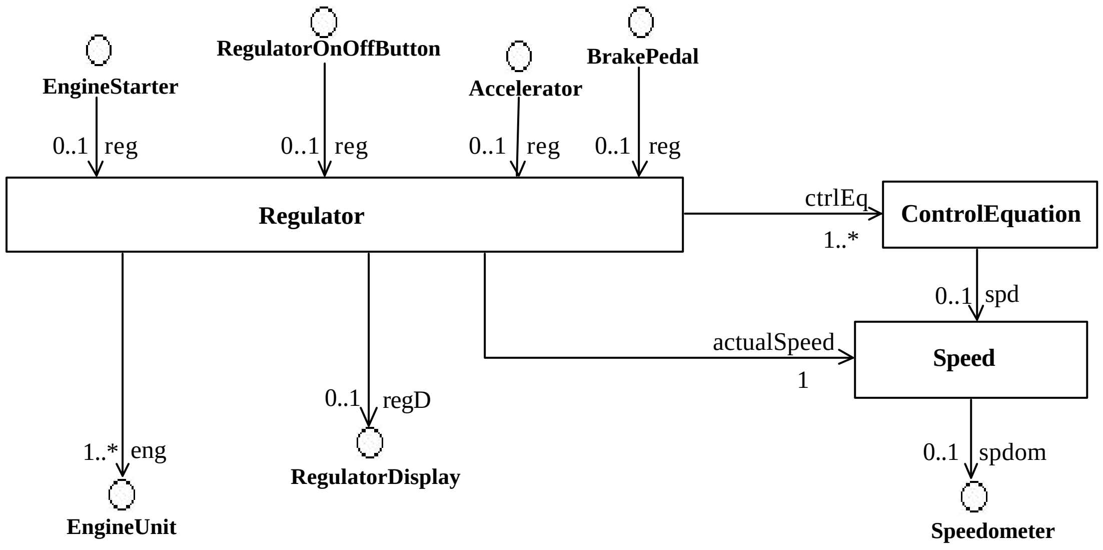


Figure 31: Global Structural Model of the Speed Regulator with Class Relationships.

2.2.2.2 Step – "clarify interrelationships ambiguities"

The model shown in Figure 32 illustrates a typical interrelationship. In this model, classes *A* and *B* both have a relationship with a same class, *C*. Class *A* is linked to class *C* via *roleCA*, with a cardinality of *cardRCA*; and class *B* is linked to class *C* via *roleCB*, with a cardinality of *cardRCB*. The problem here is thus to determine the links likely to exist between one or more *C* instances known to an instance of *A*, via *roleCA*, and one or more *C* instances known to an instance of *B* via *roleCB*. To answer this question, there are two possibilities: either there is a relationship between the two associations or no such relationship exists.

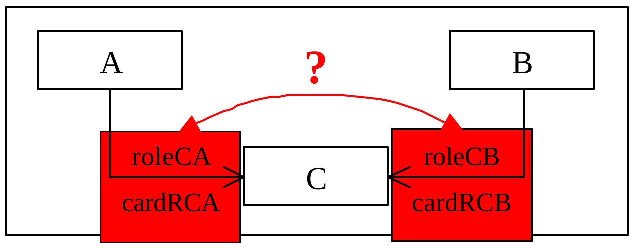


Figure 32: Interrelationship Constraint between Two Roles.

The simplest situation is the one in which no relationship exists between *roleCA* and *roleCB*. This means that no class *A* instance shares use of a class *C* instance with a class *B* instance. In this case, the structural model must contain an *{ xor }*-type constraint between the two associations (Figure 33). There is thus no possible ambiguity in the structural model specification, at least not for this particular point.

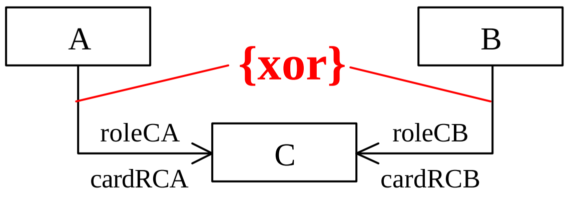


Figure 33: {xor} *Constraint between Two Associations.*

Modelling rule 9 : In a class diagram, if two (or more) classes have relationships with a third and, provided there is no sharing of used class instances by user class instances, associations linking the user classes with the used class are mutually exclusive. This is reflected in the model by the implicit presence of a mutually exclusive constraint, *{xor}*, attached to the associations involved.

To comply with Modelling rule 9 :, it is considered by default, in cases where a class is used by several other classes, that the "shared" class instances are not actually shared, i.e. that the user instances do not use the same instances of the shared class. This implies that, in modelling, the *{xor}* constraint is automatically inserted by the development tool. In the opposite case, the user places a *{=}*-type constraint between the roles pointing to the common instances.

In the second type of situation, a relationship constraint is required between two associations whose ends point to a same class. The class model specification may then evidence ambiguities (problem inherent in class diagrams). Based on the model shown in Figure 32, even in a simple case where each association end has a cardinality of 0..1, several different instantiations of a same class model are possible.

This means that, to obtain a fully defined model, ambiguities inherent in the class diagram must be clarified by adding specification elements to the underlying structural model.

One way of settling conflicts arising from shared use of certain application instances is to impose the number of allowable instances for each class. Modelling of such specifications merely involves associating the class whose number of instances is to be specified, with itself. The source end of this association is a class property, with a cardinality of *. This end is commonly known as an ***instance*** and its cardinality specifies the number of instances in the class[12]. Generally speaking, a cardinality of ***n..m*** with n = m and (n,m) (N x N*∝) means that the application has at least n and at most m class instance(s).

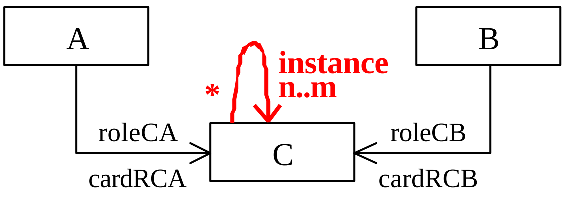


Figure 34: Specification of the Allowable Number of Class Instances

This solution nevertheless limits the reuse of classes whose number of instances is a constant. The model requires that class ***C*** be instantiated at least n times and at most m times. Consequently, any application wishing to use class ***C*** will not be able to use more than m instances of that class.

In the speed regulator example, *Regulator* and *ControlEquation* are in a class *A* and *B* situation with respect to use of the *Speed* class. This specification, which consists of imposing a single *Speed* class instance, seems reasonable for *Regulator* and *ControlEquation* classes (it may preclude coexistence of two instances in a consistent form). However, the *Speed* class is typically a generic concept of interest to other types of applications, for which it could be advantageous to simultaneously make use of one or more instances without specifying their number. In such cases, the number of instances used should preferably be a property of the complete set of classes used for the application considered and not of a single class. The solution is to express constraints on model elements via an invariant attached to the package containing the set of classes and not directly on each separate class. In this way, the constraint becomes part of the package context and not of the model element. The structural model shown in Figure 32 thus reads: Within the scope of the *SpeedRegulator* package, there is a single *Speed* class instance. As a result, all *ControlEquation* and *Regulator* instances will make use of the same speed object.

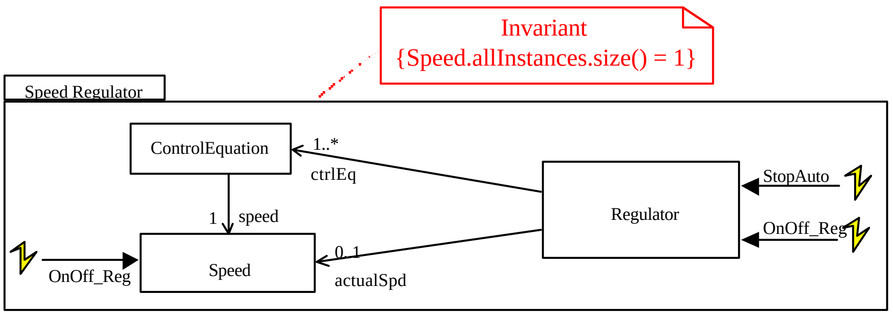


Figure 35: Specification of an Instantiation Constraint Attached to a Package.

Since *Regulator* and *ControlEquation* classes use a single *Speed* instance, this solution is suitable for the case considered here. However, if the structural model includes an additional user class for another *Speed* class instance (different from the one used by *Regulator* and *ControlEquation* instances), it no

[12] The class association also serves to access class C instances, as materialized by the expression "C::getInstances(i) ", where i ≤ m enables accessing of the ith instance of C.

longer suffices to specify an allowable number of *Speed* class instances. Specification of exactly two *Speed* class instances does not, in effect, enable a distinction between the *Speed* instance used by an *ObstacleDetector* instance and that used by *Regulator* and *ControlEquation* instances. The solution to this new problem is to add constraints between the *Speed* class roles (Figure 36). The first constraint, *{xor}*, specifies that a *Speed* instance cannot be simultaneously linked to a *Regulator* and an *ObstacleDetector* instance. This constraint can also be written in OCL form:

{ ControlEquation.allInstances → forAll(a | Regulator.allInstances →
forAll(b | a.speed → intersection(b.actualSpd) → isEmpty))}.

The next constraint,

{ ControlEquation.allInstances→forAll(a | Regulator.allInstances→forAll(b | a.speed=b.actualSpd))},

specifies that *ControlEquation* and *Regulator* instances must share a same *Speed* instance when they call on the services of this class.

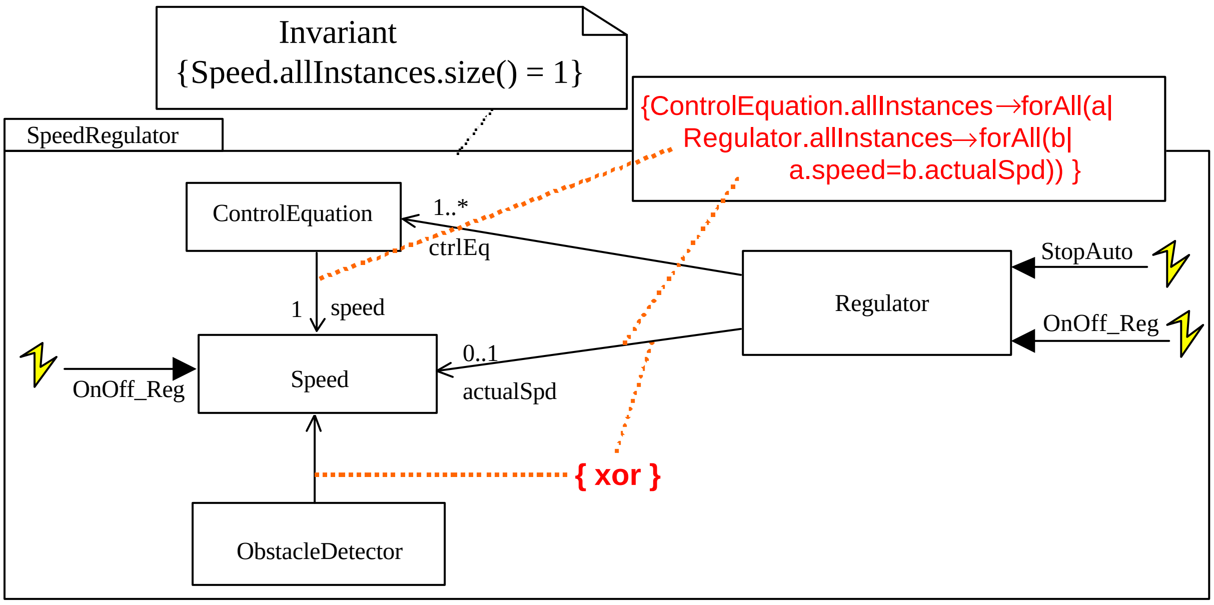


Figure 36: Interrelationship Constraints.

This can be expressed using another OCL invariant (Figure 37) :

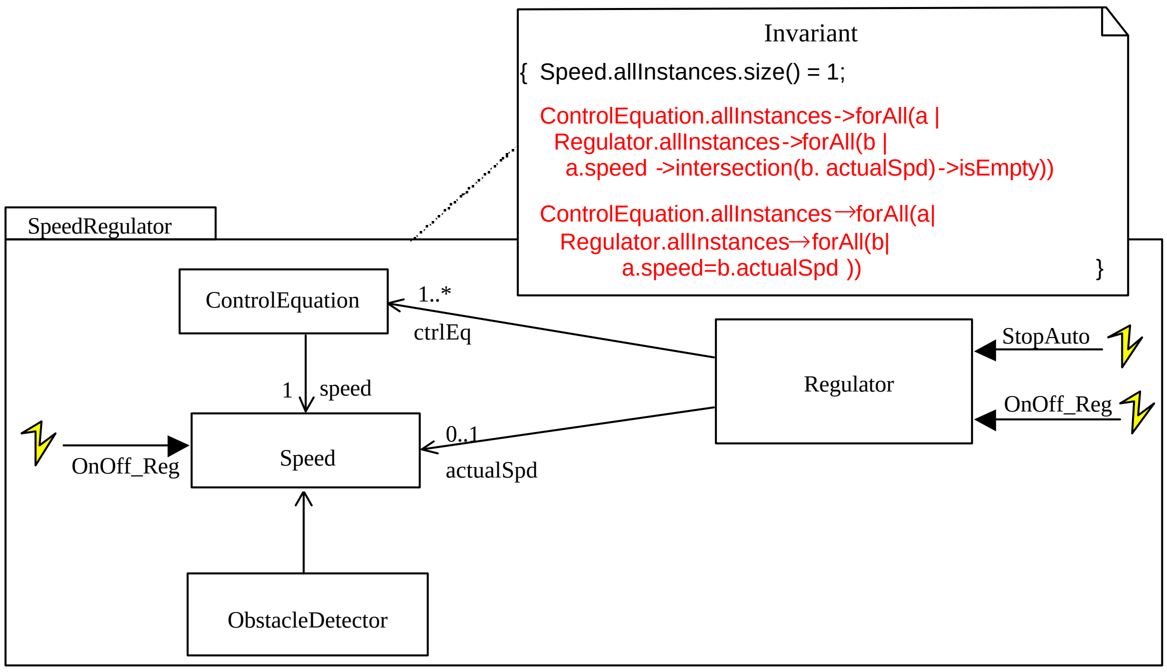

Invariant
{ Speed.allInstances.size() = 1;
ControlEquation.allInstances->forAll(a |
Regulator.allInstances->forAll(b |
a.speed ->intersection(b. actualSpd)->isEmpty))
ControlEquation.allInstances→forAll(a|
Regulator.allInstances→forAll(b|
a.speed=b.actualSpd))
}
SpeedRegulator
ControlEquation
1..*
ctrlEq
Regulator
StopAuto
OnOff_Reg
1
speed
Speed
0..1
actualSpd
OnOff_Reg
ObstacleDetector

Figure 37: Specification of Interrelationship Constraints using OCL Invariants.

Finally, the following modelling rule is aimed at improving the readability of class diagrams:

Modelling rule 10 : In a class diagram, if two classes (or more) have relationships with a third, and there is a relationship between the associations linking the user classes with the used class, the names of association ends pointing to that class will be identical.

This rule enables specification of relationships between associations but not of the accompanying constraints, which must be added as described above.

This is the modelling approach applied in subsequent chapters. It also entails renaming the *spd* role (subsequently known as *actualSpd*) to underscore the fact that a constraint exists between the two associations. For purposes of simplification, cardinality constraints will not be systematically repeated on the diagrams shown in subsequent chapters.

Remark: Invariant specifications can be viewed in either of two ways: from a "structural" or an "implementation" standpoint. In the "structural" view, they serve as constraints to be verified at all times by the application. Associated logic expressions must therefore be tested during execution of the application, an error being generated on failure to do so. In the "implementation" approach, they serve to facilitate implementation, by (as shown here) specifying the number of instances included in a class and assigning the corresponding roles. This second view is all the more advantageous as model structure then ensures that the application will comply with these constraints.

2.2.2.3 Step – "describe services"

The last phase in structurally describing an application consists of describing one-by-one the content of each of its classes, that is, the services (operations) performed by that class and the basic data (attributes) contained therein. This is precisely the level at which concurrency constraints are stated.

For attributes, description includes name and kind (the latter should be: boolean, integer, character string, floating value, table or aggregation – *structure* – of these basic types).

In *UML*, the description of an operation contains its name, the type of return value, if any, the list of its parameters (specifying, for each: name, type and passing modes – "*in*", "*out*" *or* "*inout*") and its impact on object state, which includes two possibilities:

– the operation does not change, even temporarily, the state of the object, i.e. it has no impact, at any time during execution, on the values of object attributes, those of object roles (references/pointers to related objects), or those of instances referenced by object roles (this property must be recursive). In such cases, the status of the current object[13] is an implicit parameter of the method operating in "in"[14] mode only;

– the operation can change the value of an object attribute or role or of an instance referenced by the object roles (albeit temporarily). In this case, the status of the current object is considered to be an implicit parameter of the method operating in "inout".

Remark: In *UML*, the impact of executing an operation is modeled via the meta-attribute *isQuery* of the *Feature* metaclass. This is a boolean attribute, which, when false, indicates that execution of the property leaves the state of the object unchanged; and, when true, that side effects may occur.

The impact of an operation on the state of an object likewise depends on the behaviour model of the application and, more specifically on the control automatons of the class whose object is instantiated. Any operation defined in an object behaviour automaton as capable of changing object status, must be considered as also impacting an implicit attribute of that object (i.e. its state) and thus should be declared as an "inout" parameter. It sometimes happens that an operation initially believed to leave the object state unchanged is redefined following specification of class control automatons as changing it.

Furthermore, detailed specification of class content requires definition of the accessibility statuses of its various members from objects other than the current instance. Three possible accessibility classifications – "public", "protected" and "private" – are thus assigned to class members:

Take, for example, the *Speed* class, whose specification contains the following attribute:

– value: This attribute serves to store the last acquired speed value. It is the "integer" type with, by default, a "private"-type accessibility (symbolized by an "–" placed in front of the attribute name).

– … and the following operations:

– starting speed acquisition: name *startAcquisition*, no output parameters or return value. This operation consists of applying the various parameters that enable data acquisition. It is wise to consider it as modifying the object or its roles. It is thus assigned "inout" status. Since it must be accessible for activation from outside the object, it is classified as a "public" method (symbolized by a " + " preceding the name of the method).

[13] The term "current object" designates the instance to which the method being specified will be applied when said method is called. It is the object identified by the keyword "this" in C++ and "self" in SmallTalk.
[14] Note that, in most languages, this status can only be verified if it is also assumed that the current instance was declared to be a constant. Compliance with this specification at the implementation stage requires in-depth analysis by the developer under his full responsibility.

– stopping speed acquisition: name *stopAcquisition*, no output parameters, no return value. This method may not induce changes in the values of object attributes or roles. While "in" status could be envisaged, as will be indicated in the control automaton of the class defined for the behaviour model, this method results in a change in object state and must therefore be considered as modifying an implicit object attribute (its state). It is thus assigned "inout " status and, as it requires activation from outside the object, is also classified "public".

– aquiring speed value at the Speedometer interface: name *acquireSpeed*, no parameters, no return value. This operation changes the value of the value attribute and is thus assigned "inout" status. There are two choices for accessibility:

– either considering this operation as always activated by the object's *startAcquisition* operation or by signals or by automatic transitions described in the object triggering behaviour diagram (see section 0). In this case, the operation must be declared with "*protected*" status (symbolized by an "#" preceding the name of the method).

– or considering it as also accessible for activation from outside the object, via any other object using the *Speed* class. Its status must then be "*public*".

For the moment, the first solution has been selected, but promotion to a higher accessibility category may be necessary later.

– providing the acquired vehicle speed value (value attribute): name *giveSpeed*, no input or output parameters, but an "integer"-type return value[15]. This method does not cause change in object attribute or role values. It is thus assigned "in" status and must be accessible from the outside for activation by other objects, i.e. "public".

A closeup of the *Speed* class structural diagram shows the following (Figure 38) :

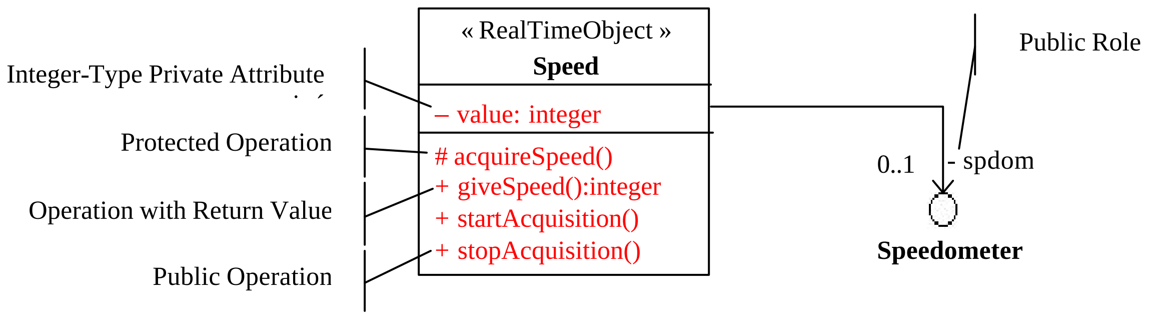


Figure 38: Detailed Structural Model of the Speed *Class*

Roles are structural characteristics of a class whose accessibility is the same as the degree of visibility assigned to a relationship. Role *Spdom* is a "*protected*" structural characteristic (references) of the *Speed* class.

Conventional practice calls for attributes to be used (for reading or changing their values) only where access is enabled in the form of specific read and/or write operations by the relevant class. For this reason, it has been decided to specify the value attribute as "private" (accessible inside the class only),

15 An equivalent solution would be to specify the method with an output parameter, but no return value. However, there is a slight difference in the impact of these two approaches on the resulting communication mode (see subsection **2.2.2.3**), which is examined at a later stage in this document.

then to always provide public access in either read mode, or, where the attribute value may be changed by another user object, in write mode. This enables the *Real Time Object* to perform concurrency control for attribute access when these operations are invoked.

2.2.2.4 Step – "specify communications"

In *UML*, messages may be sent in either of two forms – as operation calls or as signals:

- an operation call explicitly activates a point-to-point communication between a sender and a receiver. It therefore requires a structural link oriented from the sender to the receiver;
- if the message is conveyed by a signal, the activated communication is asynchronous. To generate a signal, an object (the sender) must perform a specific action, a ***SendAction***-type action.

Operation-based communication

Generally speaking, object oriented application objects communicate by exchanging operation-based messages. Moreover, communication paradigm is based on the *asynchronous communication* principle. It means that once the message is sent, the sender immediately continues execution, without waiting for sent message treatment (*Figure 39*).

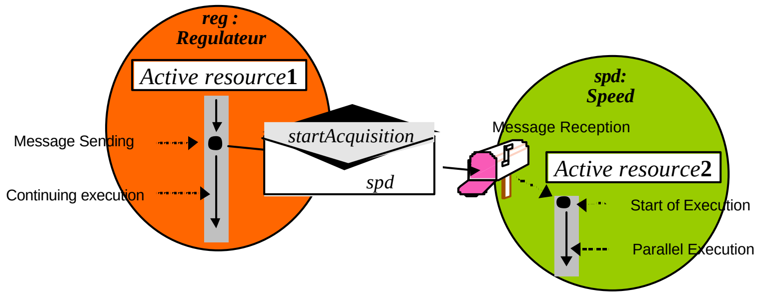


Figure 39: Asynchronous Communication.

However, this type of communication is only possible where called services have neither return values nor output parameters, thus limiting communication to unilateral exchanges. To lessen this constraint, the *Real Time Object* paradigm proposes an extension to this scheme, which has been baptized *asynchronous communication with reply* or *delayed synchronous communication*. In this approach, while the basic mechanism is the same to enable maximum parallelism, it is supplemented with synchronization points between the caller's thread of execution and the thread handling the sent message. This enables the sender to receive data return from the receiver. The synchronization mechanism is realized by a specific system object: a reply box shared by sender and receiver. An example of its operation is given in Figure 40, for a *giveSpeed* message (with output parameter) sent by a *Regulator* instance to a *Speed*-type object. Called operations may have several output parameters and a return value. A reply box is then provided for each output parameter and, as needed, for the return value.

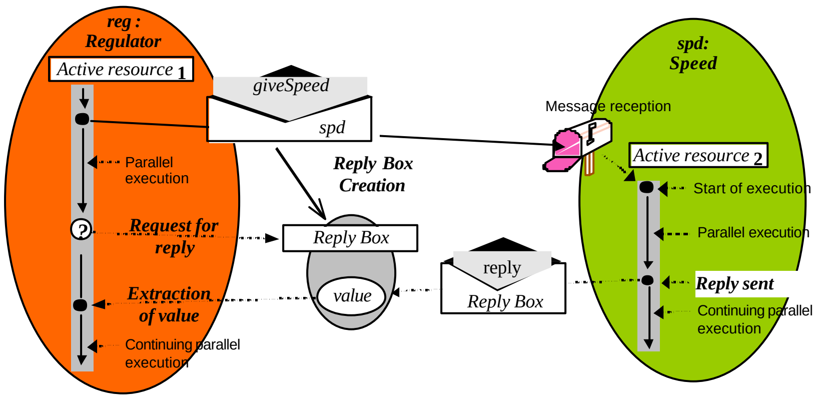


Figure 40: Asynchronous Communication with Reply.

Note that, when the called operation has a return value, this value is not dispatched until the operation is completed and the caller's thread is interrupted to await this return, immediately after the message is sent. This thread remains on standby throughout execution of the called operation. Message sending with return values thus ensures strong synchronization of sender and receiver. This type of communication is illustrated by Figure 18, where, instead of a return value, the *giveSpeed* operation provides a speed value via the appropriate output parameter.

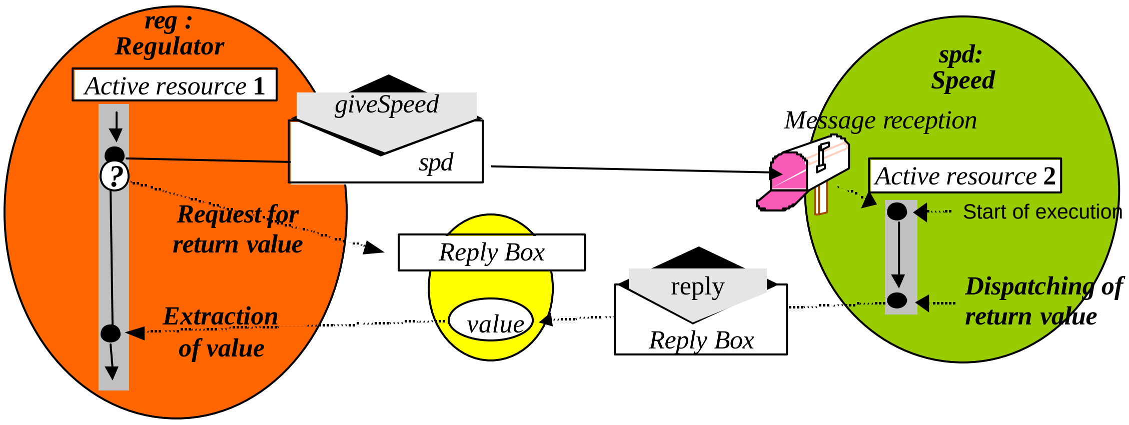


Figure 41: Strong Synchronous Communication.

Except for the synchronizations imposed by information exchanges between the message sender and the thread of execution in charge of treating it, It is possible to specify in sending messages (i.e. within the scope of the application's instantiation and behaviour models) purely logical synchronization constraints. There are two alternatives: strong synchronization, which obliges the sender to wait for the end of message treatment before continuing execution; and loose synchronization, in which the sender need await only the start of message treatment (to ensure that the receiving object was in a state compatible with such treatment) before it continues execution. In the model, this is realized via a *synchronization* tag associated with a *loose* or *strong* value.

In UML, class specification in the structural model does not require addition of data to the reference model to specify communications protocols. Indeed, the latter are implicit in the prototype for *Real Time Object* operations (refer to section 'Declaring Concurrency Constraints for Operations" on p. 50). The following modelling rule defines this point.

Modelling rule 11 : The mode of communication generated by an operation call is implicitly determined by the called operation signature, according to whichever of the following rules applies:

- by default, an operation having no return value or output parameter ("out" or "inout") is invoked in asynchronous communication mode;
- by default, an operation with output parameters ("out" or "inout") but no return value will be invoked in delayed synchronous mode (asynchronous communication with reply), on condition that its output parameters are needed by the caller: otherwise, case 1 will apply;
- an operation having a return value will be invoked in strong synchronous mode, on condition that its return value is needed by the caller; otherwise, cases 1 and 2 apply.

The table below is a recap of the various communication schemes enabled in the approach, together with typical specifications for the operations corresponding to each.

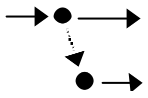
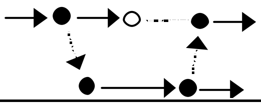
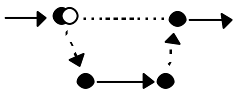
Case	***Return Value***	***Output Parameter(s)***	***Call Mode***	***Execution Diagram***	***Example of UML Operation Prototype***
1.	NO	NO	*asynchronous*		*startAcquisition ()*
2.	NO	YES	*asynchronous with reply*		*giveSpeed (> sp)*
3.	YES	YES/NO	*strong synchronous*		*giveSpeed () : integer*

Table 6: Communication Schemes and Operation Prototypes.

Signal-based communication

In *UML*, messages may be seen in either of two forms – as operation-based but also as signal-based:

– as seen in the previous section, an operation call explicitly activates a point-to-point communication between a sender and a receiver. It therefore requires a structural link oriented from the sender to the receiver;

– in the case of a message is conveyed by a signal, the activated communication is asynchronous and may also present different characteristics than operation-based message. To generate a signal, an object (the sender) must perform a specific kind of action defined as a ***SendAction***-type action.

Contrary to other UML-based approaches using this signal-based concept, in the AIT-WOODDES method, signal-based communication relies on the principle that the signal receiver is unknown to its sender and the sender is likewise unknown to the receiver. Sender and receiver do not, in fact, normally need to know each other (i.e. no structural or operational link is required between sender and receiver). The section 4 describes a specific design pattern enabling models using such signal-type communication mode to be executable.

Moreover, the behaviour of an object following signal reception is specified in a state machine. If the receiver object state so permits, signal reception can therefore cause firing of a state machine transition, resulting in the execution of an action sequence associated with this transition.

Modelling rule 12 : Signal-based communication may be used for broadcast-type asynchronous communication. It means that in this mode, the object sending the signal does not know the signal targets and the receiver object or objects do not know the sender. Furthermore, this communication mode allow a same message to transfer to several targets in same time.

Practically speaking, the signal-type communication mode is useful in three types of situation:

- to permit communication between objects for which the designer does not wish to provide structural dependency links;
- where an object O_1 needs to react to a particular state change in another object O_2, but without confining all O_1 activity to O_2 state surveillance (especially if the anticipated O_2 state change is of a sporadic nature) ;
- where several objects need to be able to react, albeit differently, to a same event.

Such situations are particularly pertinent to modelling those parts of a system where reactivity is vital.

According to the *UML* approach, all signals are typed by the class that characterizes their properties. The class specifying the signal is assigned a *« Signal »* stereotype and may own attributes that serve to define signal parameters.

It is possible for an object to send a signal without assigning values to all of its attributes. But an object receiving a signal may be led to use any one of that signal's parameters, regardless of its sender. Since the receiver has no knowledge of the sender nor, therefore, of the manner in which the signal was sent, assignment of values to all signal parameters is a necessity. The following modelling rule has been devised to meet this requirement:

Modelling rule 13 : All attributes specifying the parameters of a signal must have a default value.

In this way, if an object sends a signal without specifying values for all parameters, the receiver objects will nevertheless receive a value (if only by default) for each..

In *UML*, a signal has no operations and it is involved in no association relationships other than inheritance (from another, parent signal). This means that each signal class is itself part of a more or less generic signal hierarchy and that its characteristics are refined in the inheritance process.

In the example studied here, one could introduce both following types of signals in the model:

- the regulation system stop/start request, which is attached to the OnOff_Reg signal;
- the automobile engine stop request, which is attached to the StopCar signal.

These two requests can be interpreted in two ways:

– either by a set of operation calls from *EngineStarter* and *RegulatorOnOffButton* to the relevant objects, on *StopCar* and *OnOff_Reg* respectively. In such cases, these two classes must be able to identify and access the objects concerned. If the application and the relevant objects were to be modified, message sending would likewise need to be redefined.

– or by a signal internal to the application and automatically broadcast to all relevant objects without need for *EngineStarter* and *RegulatorOnOffButton* classes to make particular specifications.

The second solution affords a more modular design, requiring only that affected classes be able to receive the signal. The engine starter and regulator on/off button interfaces then no longer need to know the list of relevant objects or how to access it. This is the solution selected for the application described here.

The complete set of these specifications provides a structural model for the following signals (*Figure 42*) :

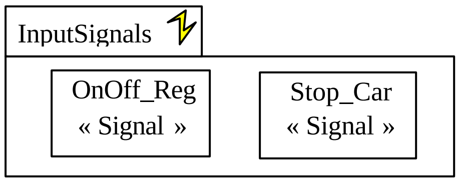


Figure 42: Signal Definition.

Signal Assignment to Classes for Sending and Receiving

The second step consists of modelling the signal "receptiveness" of various application classes and identifying potential sources of signals.

The first rule, described above as the Modelling rule 14 :and relative to this point is that any use, either in reception or in sending, of a signal has to declare before in the signal package of the application.

Modelling rule 14 : A class may only receive a given type of signal or specify that it can send such a signal instance if that type of signal was previously defined in the application's signal package.

As already seen above, in the speed regulator system, the engine starter informs the regulator that the engine has been halted and requests that speed regulation likewise be stopped. To do so, and to ensure engine starter independence from the regulation system, a signal-type message is used. An association relationship from the *Regulator* to the *EngineStarter* class would of course also preserve starter independence from the speed control function. In the second case, however, regulator instances would need to periodically "scan" the state of the starter to see whether or not the vehicle is running. Since the time constraint associated with system response to an engine halt is 100 milliseconds, the resulting scan frequency would need to be at least 10 Hz. Such monitoring of a purely asynchronous phenomenon would unnecessarily burden the system. It would be more economical to specify that the *EngineStarter* class is capable of generating *StopAuto*-type signals (*Figure 43*) and that the *Regulator* class is receptive to them (*Figure 44*). In the same way, and for the same reasons, the *RegulatorOnOffButton* class could communicate with the system via an *OnOff_Reg* signal to which the *Regulator* class is also receptive.

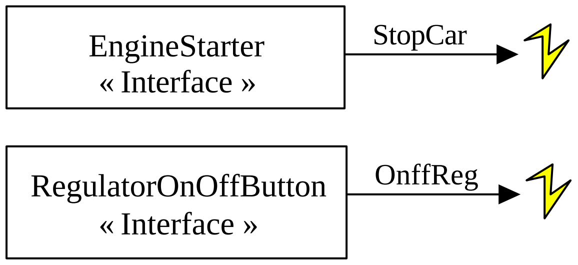


Figure 43: Signal Sending Specification.

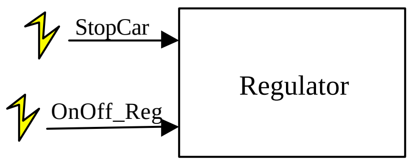


Figure 44: Signal Receiving Specification.

Specification of all signal sending and receiving operations supplements the global structural diagram for the speed regulator as follows (*Figure 45*):

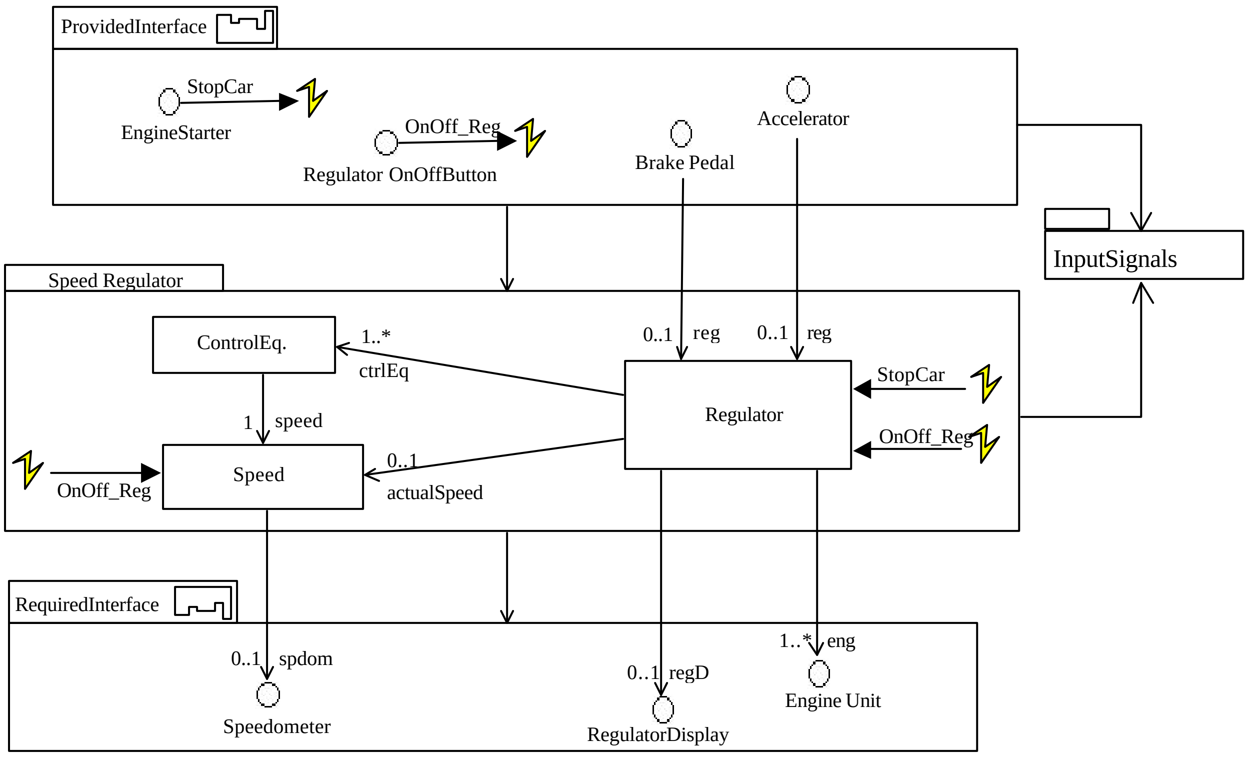


Figure 45: Signal—Class Assignments.

The specific responses of classes to signal reception are described in the chapter on the behaviour model, by defining class triggering automatons. Class signal outputs are likewise specified in the behaviour model (triggering automatons and behaviour diagrams for methods or scenarios).

2.2.2.5 Step – "declare active resources"

In the approach, concurrency is handled by specification of Real Time Objects [13] whose operation is similar to that of the Active Objects defined in concurrent programming languages such as [14], [15] and [16]. A real time object is made up of a mailbox that receives requests sent to the object and a concurrency and state controller, which manages messages as a function of its own state and the concurrency constraints associated with their execution. A real time object can be seen as a real time computer with its own real time, multitasking operating system (see [3] for more detail on this language concept).

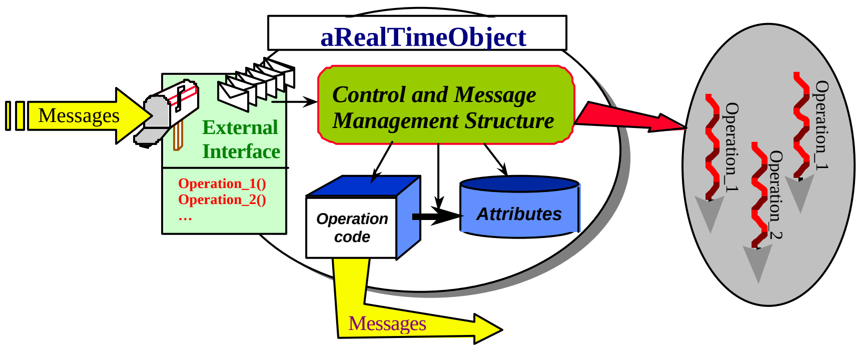


Figure 46: Outline of the Real-Time Object concept.

The real time object concept thus enables specification of two types of parallelism – intra-object and inter-object. Intra-object parallelism corresponds to a situation in which only those operations relating to a same object (in read mode only) may be executed in parallel. This view is specified in the structural model by stereotyping operations (see subsection "Declaring Concurrency Constraints for Operations" in p.50). The inter-object version calls for parallel execution of two or more objects. This view is specified by declaration of the affected objects as real time objects, i.e. by adding the *« RealTimeObject »* stereotype to the corresponding class in the structural diagram. Moreover, as an active object in UML, a real-time object class is also shown as a rectangle with a heavy border.

The following types of objects may be eligible for real time stereotyping:

– objects triggering autonomous treatments (tasks);
– objects receiving signals;
– objects receiving asynchronous operation calls;
– objects acquiring data from the environment;
– objects monitoring the (sporadic or periodic) behaviour of the system environment.

Here again, the potential need for declaring class receptivity to signals should be considered. Response semantics implicitly call for an object to react as quickly as possible to a signal, independently of the rest of the application. While this behaviour is perfectly suited to *Real Time Objects*, a passive object-type receiver offers no inherent signal treatment resources. Its response must be supported at system level and may require concurrency control to preclude conflicts with other potential users while it is treating the signal. Then again, *Real Time Object* philosophy tends to consider a passive object as a resource to be managed by real time objects rather than as an autonomous entity reacting separately from the objects that manage it. It therefore seems logical to entrust the *Real Time Objects* in charge of managing passive objects with the additional task of managing the latter's responses to signals. This alleviates the need for declaring passive object classes as "signal receptive".

Modelling rule 15 : Only those classes stereotyped as *« RealTimeObject »* are capable of managing signal receptions. If a class receives a signal but does not have real time object status, it will be considered as a modelling error.

After structural analysis of this example, two concepts seem likely candidates for *Real Time Object* status: regulator (*Regulator*), and speed (*Speed*). The *Regulator* class meets two of the abovementioned criteria, i.e. it is declared "receptive" to signals and, based on project specifications, it incorporates triggering of autonomous treatment (specifically a setpoint calculation task) while monitoring changes in controlled environment state (detection of braking, acceleration, etc.). To specify it as such, it is assigned a *« RealTimeObject »* status. The *Speed* class meets criterion 2 (receptivity to signals), but likewise satisfies the fourth criterion. This class is responsible for acquiring actual vehicle speed at a rate of 2 Hz, independently of any other action in the system. It is therefore also given real time object status. By contrast, the *ControlEquation* class does not meet any of these criteria and thus remains a passive object.

Selection of classes ensuring autonomous treatments or *tasks* in the usual multitasking applications (real time objects) then takes place in the model by assigning the *« RealTimeObject »* stereotype to the classes of interest (Figure 47).

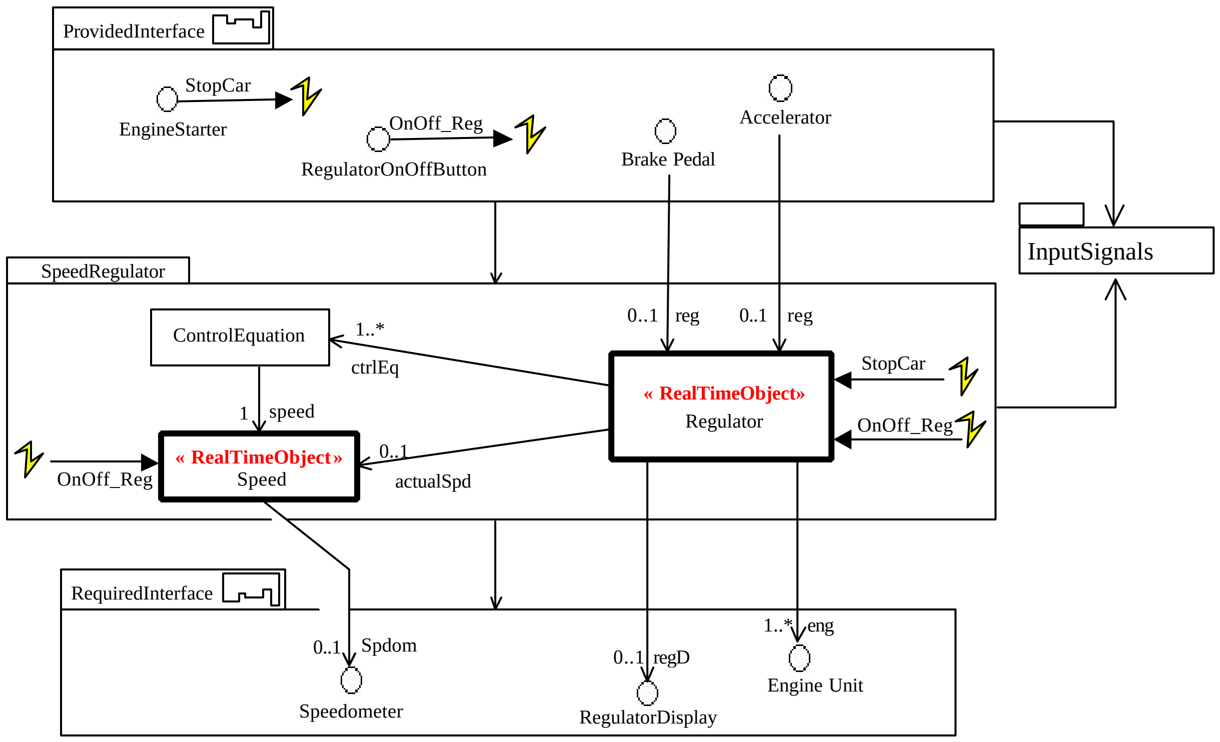


Figure 47: Specification of Real Time Objects for an Application.

It should be emphasized that the *Real Time Object* model developed in [[13] places no constraints on relationships between real time objects and other objects or on the types of communication used. Programming and execution models proposed by the method thus ensure complete real time object versatility. One significant result is that it is always possible to modify the status of a class declared as a *Real Time Object* support by removing its *«RealTimeObject»* stereotype, without causing impact to the structural model.

In this initial version of the method, the *ControlEquation* class was not selected as a potential Real Time Object support. Should this alternative become attractive at a later stage, the *«RealTimeObject»* stereotype could simply be added to the class whose realization characteristics are to be modified, along with specification of the corresponding usage constraints. These measures would not in any way jeopardize the rest of the structural model.

Remark: It should also be noted that *Real Time Object* Class declaration stereotypes only label these classes as *potential* sources of Real Time Objects. Specification of the class instances that actually become Real Time Objects takes place in the instantiation model. This enables a same class to serve, where appropriate, as both a real time object and an ordinary passive object support.

Warning: When assigned to a class in the structural model, this stereotype is not simply a redundant commentary with respect to the implementation model. To satisfy modularity and reusability objectives, the approach (like most other object-oriented methods) requires that the definition and realization of a class be kept independent of the classes that use it. A given class will not (and must not), therefore, have knowledge of its users' roles, nor of the implementation model for the application using it. Any action imposing such knowledge during specification or realization of that class would result in loss of modularity. A mere declaration of *Real Time Object* instances in the implementation model does not (and must not) enable specification and realization of the *Regulator* class as a Real-Time Object support. The application designer thus needs to add the *«RealTimeObject»* stereotype to any class he wants to be realized as a *Real Time Object* support.

Complete specification of a class declared as a potential *Real Time Object* support requires detailed specification of concurrency constraints for the various treatments potentially performed in parallel by the object, to ensure consistent use of its data.

The purpose of concurrency constraints specification is to tell the *Real Time Object* when treatments must be serialized to maintain consistency between object states and attributes, as well as between instances referenced by the object roles.

Recap of the Concurrency Issues

For manipulation of parallel systems where several active resource may simultaneously need to use a same passive resource, a strategy is required to manage concurrent access to that resource. In terms of method, this means expressing management constraints for the various resources in such a way that they can be used optimally in implementing the application. In our approach, resource management mechanisms and algorithms are imposed by the *Real Time Object* execution model [13], and the developer has no direct means of modifying them. This guarantees the most uniform and clearest possible *Real Time Object* operating semantics from one application to another.

A distinction is often made between hardware resources, such as processors, and data structures. For hardware, the concurrent usage problem is dealt with in the process of scheduling tasks as a function of real time application constraints (deadlines, priorities, etc.).

For data structures, there are typically two types of access:

– access that has no impact whatsoever on the state of the resource. This is known as *read only access*;

– access that <u>may</u> modify the state of the resource. This is known as *write access*.

A usage conflict arises for a resource when that resource is required simultaneously by two threads of execution and at least one of them is likely to use it in write access mode. In an object model context, the resource is represented and encapsulated by an object. Obviously this concerns both passive objects and *Real Time Objects*, so that the notion of *Protected Passive Object* described above can be manipulated in the model. There is said to be conflict in accessing the resource when two objects[16] (whether passive or real time) simultaneously send a treatment request (message) to the same object (resource) or when a same *Real Time Object* simultaneously sends two treatment requests to a same object (it should be remembered that the model authorizes internal parallelism in a *Real Time Object*). This situation is illustrated in *Figure 48*.

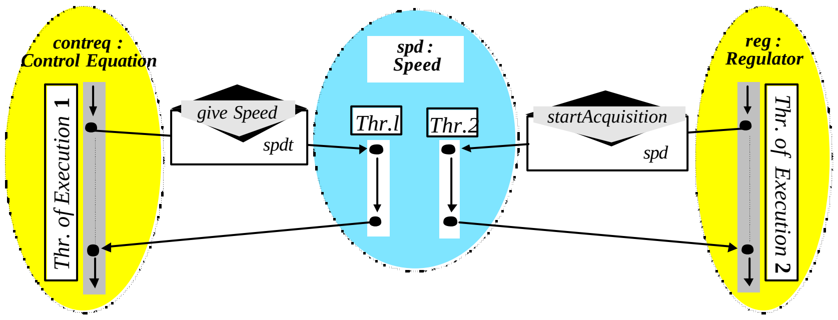


Figure 48: Parallel Access to a Same Resource.

To avoid parallel accessing of objects when one of these accesses is likely to cause changes, it is necessary to specify whether the object is accessible in *read only* or in *write* mode. Since the type of access enabled is fully dependent on the type of operation involved in using the object (in the example given here, *giveSpeed* and *startAcquisition* operations respectively), concurrency constraints has to be specified at the structural model level, on an operation by operation basis, for each affected class. This means that all executions of a given operation will be considered to have the same concurrency constraints. To guarantee object consistency, the developer must therefore account for the worst case execution scenario: a single case where an operation accesses the object in write mode is enough to require specification of write access for that object. Conversely, before concluding that read only access is enough, the developer must verify that this is true in all cases. These measures apply to both *Real Time Object* and *Protected Passive Object* support classes, i.e. those performing concurrency control on received service requests.

Declaring Concurrency Constraints for Operations

Unlike the *UML* active object concept, which is dispensed from concurrency processing by serialization treatments (as a result of the "Run-To-Completion" hypothesis of execution models for *UML* state machines, a real time object is a multitask model that requires concurrency management to execute object operations. Since *UML* semantics are not suited to this model, a new tagged value, *{concurrencyMode}*, is introduced, with the following possibilities: *read*, *write* and *parallel* (see [3]).

[16] This problem is not caused by the type of object per se. In the case of passive objects, it occurs when such objects require the resources of other, real time objects for execution.

By default, it is considered that all Real Time Object or Protected Passive Object operations are likely to modify the state of an object and must be serialized. This is equivalent to attaching the tagged value *{ concurrencyMode=write }* to all the operations of that object. It is important to notice that in this case, the execution behaviour of an real-time object is similar to this one of an active object as defined in UML, i.e. only one message at a time.

Where an operation uses object data (limited to state, attributes or roles alone) in a read only mode, it can be executed in parallel with any other object operation that does itself use the object in this same mode. The designer then attaches to it the tagged value *{ concurrencyMode=read }*.

When an operation does not make use of any object data (state, attributes or roles), it can be executed in parallel with any other operation provided by that object (including itself). The designer can then assign it a tagged value *{ concurrencyMode=parallel }* dispensing it from concurrency constraints. A call to a method with this specification results in execution of the associated operation without concurrency processing.

In some special cases, even where an operation does not fully comply with these requirements, it can also be assigned this characteristic, knowing that, in actual practice, there is no risk of concurrency problems occurring. In such cases, the viewpoint considered is that of the class taken separately and not that of an operation's specific use in the application. Otherwise, said operation could not be reused in other contexts as yet unknown to the designer.

Modelling rule 16 : An operation owns three concurrency modes specified via the tagged value *concurrencyMode* , with the following possibilities:

– *write*, the operation may use object attributes or roles in write mode;
– *read*, it may only use these attributes or roles in read mode;
– *parallel*, it uses neither roles nor attributes.

Take the previously mentioned example of *Speed* class. The concurrency constraints set for its various methods are as follows:

– *startAcquisition*, *stopAcquisition* and *acquireSpeed* operations induce a change in object state. They thus use a write access to the object and are tagged *{ concurrencyMode = write }*;
– the *giveSpeed* value only returns the value of the object's *value* attribute. It has read only access and is therefore tagged *{ concurrencyMode = read }*;
– finally, the *Speed* class may be enriched with a *convertSpeed* operation. This operation is intended only to convert a speed value in km/h, received as an argument, to a speed value in m/s, dispatched as a return value. It accesses none of the object attributes either in read only or write mode and may thus be executed at any time without involving concurrency conflicts on the targeted object. Its tagged value is *{ concurrencyMode=parallel }*.

The resulting *Speed* class specification is therefore (*Figure 49*) :

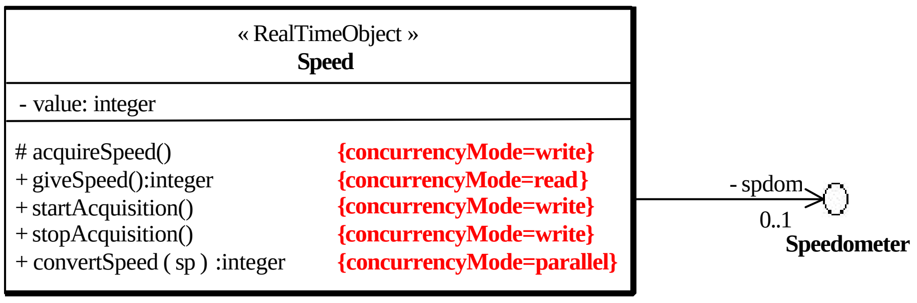

« RealTimeObject »
Speed
- value: integer
# acquireSpeed() {concurrencyMode=write}
+ giveSpeed():integer {concurrencyMode=read}
+ startAcquisition() {concurrencyMode=write}
+ stopAcquisition() {concurrencyMode=write}
+ convertSpeed (sp) :integer {concurrencyMode=parallel}
- spdom
0..1
Speedometer

Figure 49: Concurrency Constraint Specification for a Real Time Object.

Remark: There is no general algorithm for *C++* program code analysis that is capable of determining with absolute certainty whether or not a given method will perform write operations on a given object.[17] More specifically, the "***isQuery=true***" status of an *UML* (or the equivalent, i.e."***const***" for *C++*) can only guarantee that modification to the object be limited to a single situation, i.e. that of the object itself declared as a constant. This is rarely the case for a *Real Time Object*… The designer must therefore verify the relevance of concurrency specifications through precise analysis of the application code.

Deadlocks

The concurrency management protocol previously presented is the "1 writer/ N readers" type in which the managed resource is the set of data represented by the object, i.e. its state, the values of its attributes and roles and the values of instances referenced by its roles. When an operation accessing the object in write mode executes, it reserves the object (resource) exclusively for its own use. Any other operation requiring use of that resource (whether in read or write mode) is then placed on standby until completion of the writing operation. If concurrency constraints are correctly expressed, this protocol guarantees that the object will remain in a consistent state. However, this protocol may sometimes lead to situations of deadlock. A typical such case is that where the real time object sends itself a message to enable realization of another treatment requested of it. Take the example of *Speed* class operation *startAcquisition*. Its main objective is to initialise the procedure for acquiring actual speed values. However, calling *acquireSpeed*, once this initialisation has taken place, may be considered as a second initial speed acquisition request. Since both operations access the object in write mode, they are serialized. On commencement of *startAcquisition*, the operation is given access to the object in *write* mode and, consequently, is prohibited from executing any other object operation. After this initialisation, when *startAcquisition* requests execution of *acquireSpeed* to update speed value, the object sends itself a message that cannot be treated for as long as *startAcquisition* is executing. **f**, on the other hand, *startAcquisition* waits for completion of *acquireSpeed* before it terminates the procedure and frees access to the object, the situation can be said to be deadlocked.

The model for *Real Time Object* programming proposed here offers two mechanisms for preventing deadlocks:

– asynchronous message sending that frees the caller and alleviates the need for him to wait for completion of a treatment. This is the solution required for the above example. Asynchronous messaging will be discussed in subsection 1.8.

[17] This is true, in particular, where the program enables dyanmic address manipulation, using pointers, for example.

- concurrency control "freeze" capability for well defined situations where it is possible to consider treatments previously considered independent of each other, as parts of a same nesting operation (notion of transaction internal to an object). In this way, these various treatments could be executed without mutual concurrency control. This also applies in the abovementioned example, where the call by *startAcquisition* to the *acquireSpeed* operation could be seen as a standard call to an *acquireSpeed* subroutine capable of executing sequentially in the same thread of execution as that of the containing operation *startAcquisition*. Note that the latter case is only useful in managing models in which the designer provides a detailed description of application functions and, seeks, for example, to reduce sizes of critical sections induced by concurrency constraints. This solution is thus more appropriate for a detailed real time design model or a prototype analysis model.

Class Attribute Management

Class structural features (attributes or roles whose scope is the class) are characteristics shared by all the instances of a class. This is the only authorized data sharing mode between different instances. It typically enables storage of all existing instances and gives access to all of them. Because two real time object-type instances of a same class may seek to simultaneously use a same member of that class, this type of characteristic must be integrated into concurrency control but cannot be simply attached to the integrity of the instance supporting the treatment, since it concerns all existing instances.

Attachment of specific concurrency control to class members is thus necessary to enable detection by the *Real Time Object* of cases where these members are already reserved by another *Real Time Object*. From the specification standpoint, no specific declaration is required for a class possessing class members. The designer must simply remember to include them in his impact analysis of treatments performed by operations on the object concerned: if an operation modifies the value of a class member, it is considered to access this member (or, more globally, the object) in write mode.

Class operations[18] must also own the specification of their concurrency constraints. However, these operations are only designed to use class members (attributes or roles). For this reason, it is usually easier to analyze their impact.

2.2.2.6 Step – "declare shared resources"

This activity in modelling is likewise concerned with objects that are likely to be used in parallel by several real-time objects. When the objects used are themselves *Real Time Objects*, their internal concurrency control guarantees the consistency of data encapsulated from one to another. When the object is a passive one, however, it must be given internal concurrency control capability to preclude two parallel user threads from becoming entangled and creating inconsistencies in object attribute values (Figure 50). Said object must then be defined as a *Protected Passive Object*.

[18] These class methods can be called and executed independently of a specific class instance. In a way, they can be considered as global functions whose visibility is limited to the context of the encapsulating class.

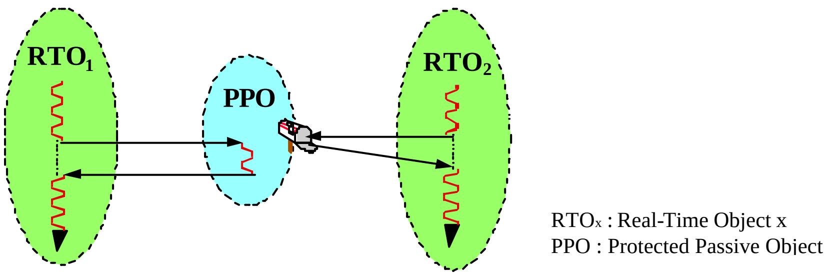


Figure 50: Use of a Protected Passive Object.

Such type of object is declared through attachment of the *« ProtectedPassiveObject »* stereotype to a class. In such cases, the model assumes that, at the time of realization, an internal concurrency control mechanism (e.g. mutual exclusion of various object method executions, 1 writer/N reader-type protocol, etc.) is attached to said object type. This mechanism ensures consistency of information encapsulated by the protected passive objects in all cases.

Remark: In the same way as for real time objects, it should be noted that Protected Passive Object stereotyping of classes only labels them as *potential* sources of protected passive objects. Specification of the class instances that actually become Protected Passive Objects takes place in the instantiation model. Here again, this enables a same class to serve, where appropriate, as both a protected passive object and an ordinary passive object support.

As was the case for real time objects, this annotation is likewise required at the class level to enable specification of classes and their realization, independently of the particular use being considered here.

Remark: It should be emphasized that association of concurrency control with a passive object, via the *« ProtectedPassiveObject »* stereotype, has significant impact on implementation (size of memory, execution time). For this reason, such control should not be systematically associated with all passive object instances in the application. Even if this choice is guided by application model analysis tools, the complexity of the problem (exponential logic, general case undecidability) means, for all intents and purposes, that the developer is himself responsible for choosing which objects are protected. The fact that an instance can only be used by a single real time object instance may lead to the conclusion that concurrency control is unnecessary. This may not be true, however, since, according to the proposed Real Time Object model [17], a same real time object can support parallel execution of more than one of its own operations, thereby itself inducing conflicts in access to any one of its roles, if that particular usage has not been fully integrated into its concurrency constraints (see subsection 1.7).

The existence of more than one access path from different real time objects to a same passive object implies risk of conflict. However, the implication is a potential one, since the structural model is a class model and, if conflict occurs, it does so between objects (class instances). In the class model example shown in Figure 51, there is risk of conflict for access to passive object *PPO*. The first solution to this problem is to use the concept described above, i.e. to add to the *PPO* class a *« ProtectedPassiveObject »* stereotype. In this way, any *PPO* instance will become capable of managing potential conflicts created by simultaneous calls from real time object class instances *RTO$_1$*, and *RTO$_2$*. The advantage of this solution is that it is valid for any implementation of the structural model. The drawback is that, even where the application resulting from implementation of this model does not generate conflict (i.e. where *RTO_1* and *RTO_2* do not use the same *PPO* instance), a *PPO* instance remains a protected passive object. This solution involves additional expense (in terms of generated code and execution time). The second solution to the problem of managing concurrent accesses to a same shared resource relies on a finer analysis of the roles played by a same passive object (the shared resource) with regard to various other objects. If the two roles *role1* and*role2* do not reference the same *PPO*, there will be no conflict in access.

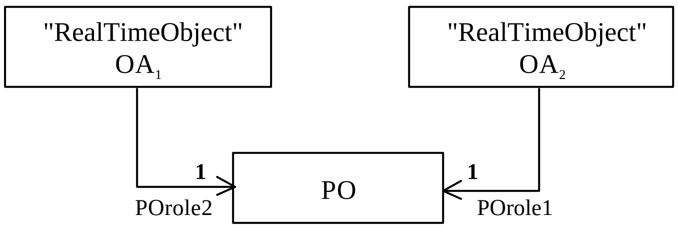


Figure 51: Use of Shared Resources.

2.2.2.7 Summary

This section described various aspects of the method for building the structural sub-model of an application. This sub-model is based on a generic architecture intended to permit clear identification and suitable structuring of the various interfaces between the system and its environment. Regaring other OO approach dedicated to embedded system development, the points highlighted in this chapter were the following:

– clarification of UML signal-based communication ;
– extension of UML active object concepts to real time (multitasking) objects;
– introduction of class specialization based on a *« ProtectedPassiveObject »* stereotype to enable modelling of shared resources;
– concurrency management for real time objects;
– impact of specifying Real Time Object class operations on inter-object communication models

2.2.3 Activity – "describe behavioural view"

To describe the behavioural aspect of an application, *UML* proposes, among others, the use of so-called "state diagrams". *UML* state machines are largely based on an object-oriented variant of ROOMchart [18] and Harel statecharts [19].

In order to improve readability and reuse of the behavioural sub-model, one has introduced two complementary views: **a protocol view** and **a reactive view** (Figure 52) [9]:

– a protocol view of the class behaviour focuses on a description of the protocol for use of object services. The transition trigger events of such an automaton are necessarily the operation call type. This view is also known as a "life cycle", since it tends to describe what an object *can* do;
– a reactive view of the class behaviour centers around a description of object class reactivity, i.e. object response to both following kinds of event: a signal receipt, a given amount of time passed or achievement of a particular state after triggering of completion transitions. This view is concerned with what an object *must* do.

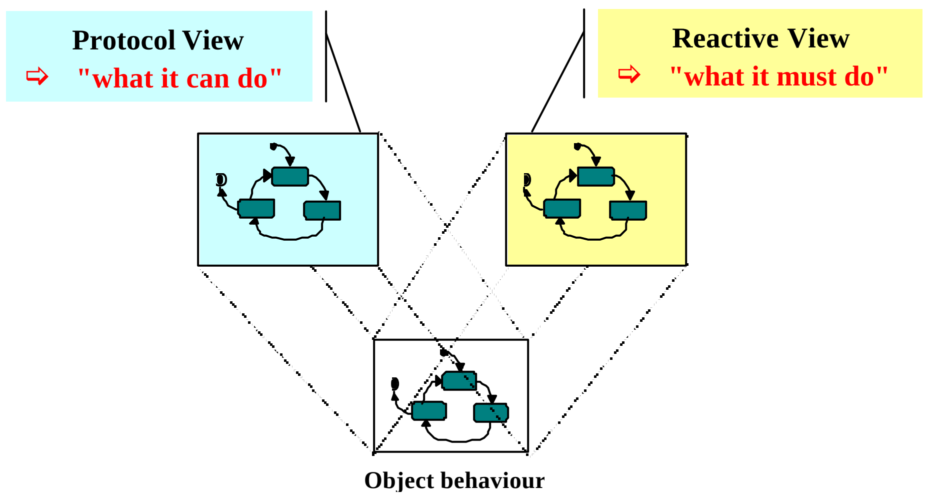


Figure 52: Two Points of View for Object Behaviour.

Both previous views contribute to model the logic behaviour of the domain classes and at this point all algorithms issues have been put aside. Indeed, in order to ease reusability, maintainability and evolutivity, the algorithmic parts are postpone in a specific model, the method behavi our model.

As delineated in Figure 53, in order to construct the behavioural model sketched just above, the method proposes the activity refers to four sub-activities:

- The **"describe protocol view"** activity aims at describing the protocol view of a class behaviour through a state machine diagram.
- The **"describe reactive view"** activity aims at describing the reactive view of a class behaviour through a state machine diagram.
- The **"describe domain algorithms"** activity aims at describing the class operation behaviour which following the current method allows the analyst to specify the application domain algorithms. This step produces its results under the form of activity diagrams.
- The **"declare real-time QoS"** activity aims at declaring within the different behavioural view the real-time QoS of the application.

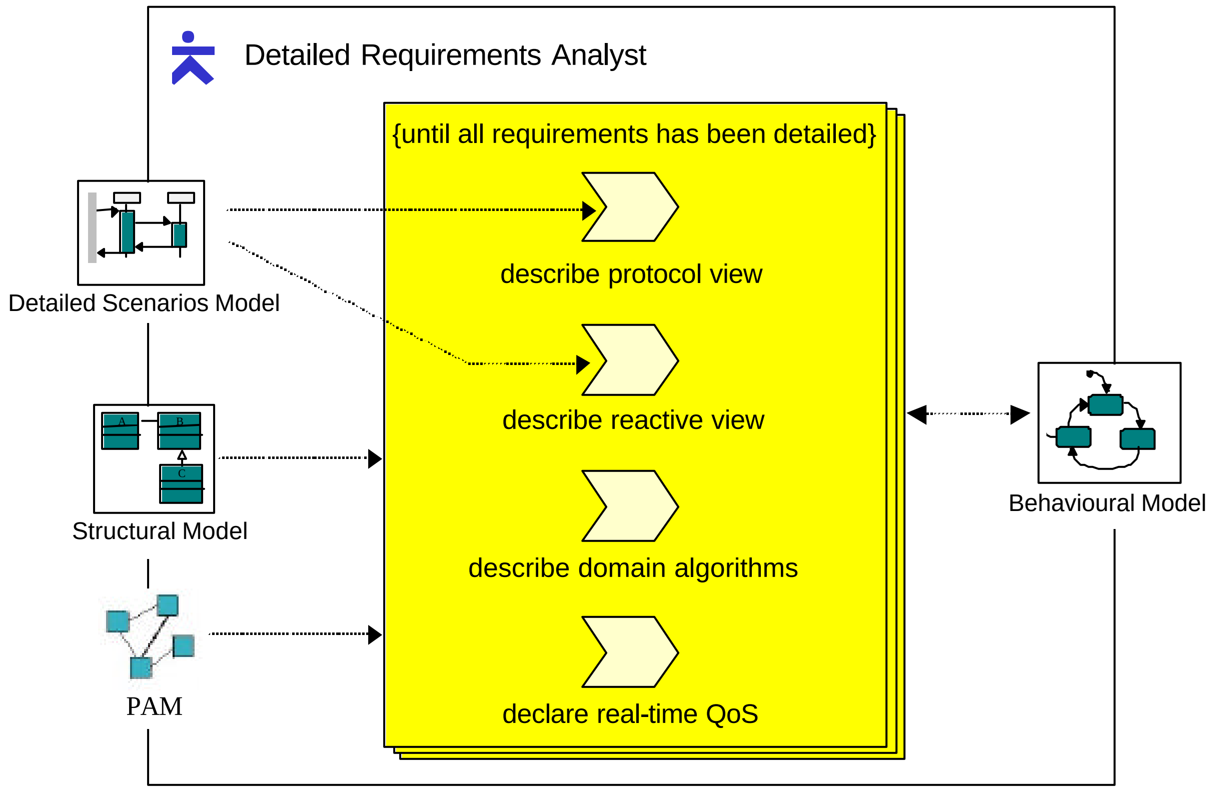


Figure 53: Activity – "build behavioural model" of the AIT-WOODDES methodology.

Before describing the different sub-activities that composed the behaviour description activity (see sections), we will remind you the state machine semantics we have adopted for the approach due to UML semantics variation points and existing ambiguities. If you need more details on this subject, please refer to the UML profile attached to this method [3].

2.2.3.1 Semantics of the state machine execution

This section aims know at remind you quickly the execution semantics of the class behaviour, with emphasis on two points[19]:

- describe choices relative to semantics variation points of UML;
- solve indeterminism issues pertaining to UML semantics ambiguities or lack of fulfilment.

In *UML*, state machines can be used both to describe the behaviour of particular entities such as classes or operations but also to model global behaviour involving several objects such as use cases or packages. In the approach we propose here, state machines are only used to describe the behaviour of object classes. Each automaton is a specification of class behaviour. The behaviour of future class instances is contained in an instance of this state machine specification. Classes whose behaviour is described include passive object and protected passive object classes as well as real time object classes. If sometimes we speak about the execution semantics of an object, it means implicitly the

[19] The first point should be always done explicitly in any approach claiming to be UML-based! And the second point also for approaches targeting real-time systems development, hard or soft constrained, whereas a real-time systems must by essence DETERMINIST!

execution semantics of the state-machine describing the behaviour of the object. So all this discussion leads us to sate the following modelling rule:

Modelling rule 17 : Real Time Object classes must own one and only one state machine describing their behaviour. Other classes (passive objects or protected passive objects) may also own at most one state machine describing their behaviour.

For class behaviour specification, the *UML* state machine selected here is a restricted view whose overall philosophy follows this one of "protocol" machine as described in [20] (see pp. 2-153). This type of state machine operates on the principle that transition firing will not trigger execution of a sequence of actions specified in the right-hand portion of the fired transition, but instead executes a method implementing the operation attached to the fired transition. Such *UML* automatons behave as though, on reception of the transition trigger event, an internal event were generated within the state machine context. This internal event in turn triggers execution of the method implementing the operation specified by the transition trigger event.

One significant restriction of state machine we use there is it has no orthogonal composite states. The substates of a composite state are thus either simple states or non-concurrent states. The orthogonal state feature of conventional *UML* machines was not preserved here, since state machine execution semantics associated with this notion are currently still very ill-defined. However, this feature will most likely be incorporated later, once the *UML* standard has evolved toward a greater degree of clarity. Indeed, internal concurrency introduced through concurrent composite states may be easily replaced thanks to concurrency mechanism of real time objects (cf. 2.2.2.5).

Modelling rule 18 : A state machine does not contains concurrent composite states. For this reason, modelling elements relating to this notion (e.g. *fork* or *join*-type pseudo-states, and the notion of *SynchState*) are likewise absent from behaviour models in this proposed method.

To permit simplification of class behaviour in an application, the notion of actions in a state, whether it be incoming or outgoing has been omitted. As has the notion of activity, which enables introduction of parallelism inside the object. Instead, this kind of concurrency is already available via the notion of Real-Time Object.

Modelling rule 19 : States of a state machine must contain neither entry actions, nor exit actions nor activity.

-

In order to position the semantics with respect to the *UML* standard, the following discussion is organized in the same way as paragraph 2.12.4 (see chapter 2 - *UML Semantics*, topic 2.12 - *State Machines*, pages 2-143 to 2-152) of *UML* standard, V 1.3, [20]. Paragraph 2.12.4 is divided into subparagraphs that deal with the different elements of the *UML* meta-model involved in specifying state machines. For simplification purposes, two types of elements have been omitted here:

- those relating to the notion of "concurrent state", since the approach presented here does not integrate this notion (see Modelling rule 2);
- all elements that do not change, i.e. are taken over as such into the approach and therefore require no further study.

UML state machine semantics are described in terms of the mechanisms of a hypothetical machine that implements a state machine specification. There are three key components of this hypothetical machine:

- an ***event queue*** that holds incoming event instances until they are dispatched;
- an ***event dispatcher*** that selects and dequeues event instances[20];
- an ***event processor*** that processes the current event according to *UML* semantics. This component is simply referred to as the "state machine".

The rest of this section tackles this three points.

The event queue and dispatcher

The real-time object paradigm is the main model element introduced by the current approach to model real-time application. The state machine semantics described here is relative to this context, i.e. we depict in this section the execution semantics of statemachine describing the behaviour of a real-time object.

The event queue of the state machine describing the behaviour of a real-time object is a mailbox [21]. In *UML*, events are dispatched and processed one at a time. The order of dequeuing is not defined, leaving to the user the possibility of modelling different priority-based dequeueing schemes.

In our case, a message posses always a required real-time feature (see sections 2.2.3.5 et 2.2.4.3) either explicitly specified by the sender or implicitly inherited by constraint propagation. Two real-time constraint specification models are possible, i.e. priority-based or deadline-based, but it is not possible to mix both policy within a same application model. In both cases, the event with the severest constraint (highest priority or shortest deadline) is selected for processing in the object mailbox.

Event processor – run-to-completion step

In *UML*, event processing semantics are based on the "run-to-completion" assumption, meaning that an event can only be dequeued and dispatched if processing of the previous current event is fully completed.

The run-to-completion assumption simplifies the transition function of a *UML* state machine since, by serializing treatments triggered by various events, it precludes conflicts between them.

The run-to-completion assumption is certainly effective in ensuring the consistency of a conventional *UML* state machine, in which transitions specify all of the required processing in the form of a given sequence of actions. However, if, as is the case here, the approach relies on the notion of "protocol state machine" defined in *UML*, this assumption is no longer adequate. This category of state machine (p. 2-153 [20]) is based on the premise that transition firing does not trigger execution of a sequence of

[20] The UML standard does not define the order in which events are dequeued. This point is a well-identified semantics variation point that any UML-based approach should have to explicitly be described before anything else!

actions specified by the fired transition, but instead executes the method implementing the operation attached to said transition. This type of *UML* automaton functions as though receipt of a transition triggering event caused generation of an internal event implementing the operation associated with the trigger event.

By the time a protocol-transition generates an internal event, it has fully completed the processing associated with receipt of its trigger event. The automaton underlying the fired protocol transition is thus considered to have performed an RTC step. It can then dequeue and process another event. At this stage, however, nothing guarantees that the method triggered by firing the previous protocol transition has been fully executed. Under such conditions, several methods may be led to execute in parallel and, therefore, to conflict with each other on accessing shared resources.

To avoid this type of situation, the following condition can be added to *UML* run-to-completion semantics:

"Where the previous current event transition is the protocol-type, its processing is only deemed completed once the method associated with the processing underway has fully executed."

In this case, the protocol transition is said to have performed an "RTC step" once the method triggered by firing it has been fully executed. This solution alleviates possible concurrency conflict problems. Adding the new condition enables the same behaviour as for the standard "run-to-completion" assumption, i.e. serializing of intra-object treatments.

The real-time objects behaviour semantics have been somewhat modified to allow for more parallelism within the application. As already demonstrated above, simply applying the standard "run-to-completion" assumption to protocol-type UML state machines may lead to situations where several methods execute in parallel. The real-time object approach thus consists of maintaining this possibility for as long as no problems occur, i.e. the methods executing in parallel are not concurrent. As described in section 2.2.2.5, in specifying the operations that define a class interface, the user adds a property specific to each operation relating to its concurrency constraint. An operation may thus be qualified as a write, read or parallel operation. The execution model proned here is namely based on the "1 writer/N readers" protocol. The challenge here, therefore, is to adapt the initial "run-to-completion" assumption defined in *UML* to protocol-type state machines so that read operations, which do not generate concurrency problems, can take place in parallel. Execution of write operations must then be serialized to avoid conflicts in accessing shared resources.

As already seen in previous paragraphs, the class behaviour automaton comprises two types of transitions: protocol- and triggering-transitions. In both cases, transition firing has the same impact on the behaviour of the described object, i.e., it triggers execution of an associated-method (which is implicitly specified, via the name of the triggering event, for protocol transitions and explicitly specified, through a *CallAction* for triggering-transitions). In the rest of this document, the term ***reader-transition*** will refer to a transition whose associated-method concurrency mode is the read type. In the same way, the terms ***writer-transition*** and ***parallel-transition*** will designate transitions whose associated-method concurrency modes are write and parallel respectively.

The aim is thus to construct a mechanism that enables or inhibits firing of a transition as a function of the type of associated-method concurrency and the execution context (i.e. types of methods being executed). Table 1 shows specification of this protocol by execution context (shown horizontally) and type of transition to be fired (shown vertically). The word "yes" appearing in a cell indicates that firing of a transition is possible in whatever the given context, with "no" meaning the opposite.

Execution context / Type of transition to be fired	1 writer method	N reader methods (N≥0)	N parallel methods (N≥0)	No method executing
Parallel-transition	Yes	Yes	Yes	Yes
Reader-transition	No	Yes	Yes	Yes
Writer-transition	No	No	Yes	Yes

Table 7: Transition Firing Situations as a Function of Associated-Operation Concurrency Mode and State Machine Execution Context.

Conflicting transitions

In *UML*, it is possible for more than one transition to fire at the same time. When this happens, such transitions may be in conflict with each other. Take the case of two transitions originating from the same state, triggered by the same event, but with different guards: If both guard conditions are true, there is a conflict between the two outgoing transitions. Only one of them must be fired. **But which one and what criteria can be used to make the choice?**

UML semantics provide no answer to this question!

In *ACCORD/UML*, if such a situation occurs, the model is ill-formed. In real time system development, one important property of the developed application is its determinism. And conflicting transitions give rise to the type of non-deterministic situation this approach seeks to eliminate.

In *UML*, the notion of conflicting transition is defined as follows:

"Two transitions are said to conflict if they both exit the same state or, more precisely, if the intersection of the set of states they exit[21] is non-empty."

UML semantics proposes a transition priority rule based on a state containment hierarchy. However, this rule does not provide solutions to all the possible conflicts arising between transitions. The situation described in Figure 54, for example, remains non-deterministic in terms of *UML* semantics. Transitions t_1 and t_2 are in fact triggered by the same type of event and originate from the same state.

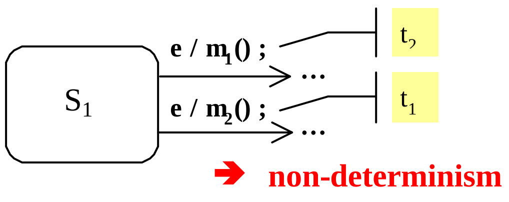


Figure 54: Non-Deterministic Situation in a UML State Machine.

The systems aimed by AIT-WOODDES are real time ones, where non-deterministic situations are unwelcome. The approach has also to integrate the priority rule defined in *UML* (see above) and add a

[21] If the source state of a transition is a nested state, i.e., part of a state containment hierarchy, a fired transition will in fact exit not only the nested source state, but also, transitively, all of the latter's parent states.

modelling rule enabling elimination of any such remaining contentious situations. In the situation described in Figure 54, with the above execution semantics, the situation becomes deterministic and can be statically analysed. Several cases are possible depending on the type of concurrency constraints (Write, Read or Parallel) associated with methods m_1 and m_2. All of these potential cases are summarized in Table 8.

m_1 / m_2	**W**	**R**	**P**
W	MEN	MEN	PE
R	MEN	PE	PE
P	PE	PE	PE

Key:

W: writer method

R: reader method

P: parallel method

MEN: Mutual Exclusion Necessary

PE: Parallel Execution

Table 8: Rules Governing Conflicts in Transitions Exiting a State.

This table demonstrates that, if one of methods m_1 or m_2 is a writer method, the two transitions t_1 and t_2 exiting the state described in Figure 54 must be mutually exclusive. In all other cases, the policy of "1 writer/N readers" enables concurrent execution of both methods. A generalization of this case then provides the following modelling rule:

Modelling rule 20 : The guard of a writer-transition exiting state ***S*** and triggered by a ***w*** type event must be mutually exclusive with respect to all guards of the set of transitions exiting ***S*** and also triggered by ***w***. Otherwise, the model is considered to be ill-formed.

Example of execution

The following paragraphs provide an example of the multitasking behaviour of a real time object, based on all of the above described rules. This example is based on an object ***O*** with three operations (***op1_w***, ***op2_r*** et ***op3_p***), whose concurrency modes are write, read and parallel respectively.

In the first sequence illustrated in Figure 55, ***O*** receives the successive messages m_1***_w***, m_2***_w***, m_3***_r*** and m_4***_r***. Message m_1***_w***[22] is associated with a writer method that cannot execute, since no treatment is currently underway. Message m_2***_w*** must be serialized, since a writer method, ***op1_w***, is already executing. Message m_2***_w*** is accounted for as soon as the previous execution of ***op1_w*** is completed. Message m_3***_l*** arrives during execution of the operation associated with m_2***_w***. The newly arrived message is associated with a read operation, ***op2_r***. Since a write operation, ***op1_w***, is then underway, ***op2_r*** execution must be serialized. Following execution of ***op2_r***, m_4***_r***, a message associéated with ***op2_r***, is received. The associated operation is the read type and there is only one other operation – also a read – underway. In this context, ***op2_r*** can thus be carried out in parallel.

[22] m_X***_w*** is the name of the message triggering operation ***opX***. The suffix "***_w***" indicates that the triggered method is the write type. Suffixes "***_r***" and "***_p***" indicate that the triggered method is the read or the parallel type respectively.

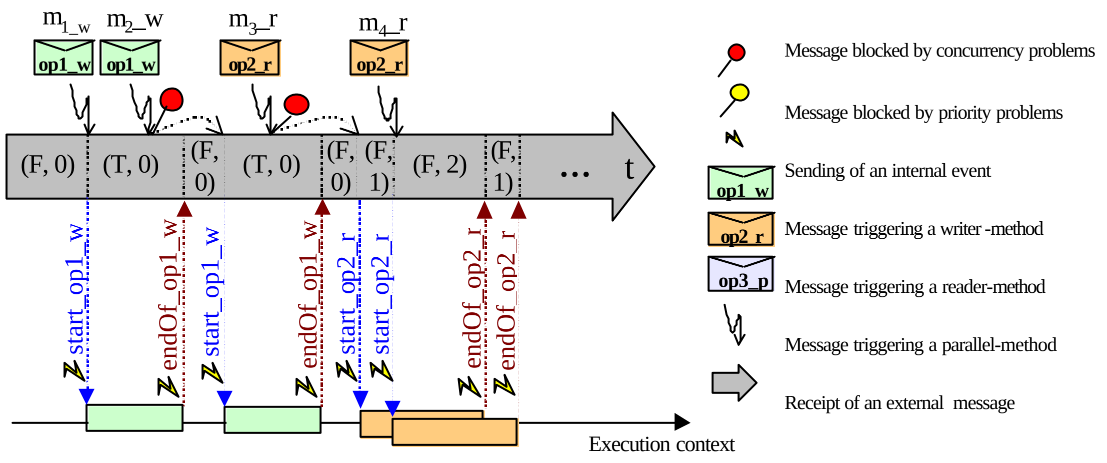

m1_w m2_w m3_r m4_r
op1_w op1_w op2_r op2_r
(F, 0) (T, 0) (F, 0) (T, 0) (F, 0) (F, 1) (F, 2) (F, 1) … t
start_op1_w
endOf_op1_w
start_op1_w
endOf_op1_w
start_op2_r
start_op2_r
endOf_op2_r
endOf_op2_r
Execution context
Message blocked by concurrency problems
Message blocked by priority problems
Sending of an internal event
Message triggering a writer -method
Message triggering a reader-method
Message triggering a parallel-method
Receipt of an external message

Figure 55: Illustration of the Implemented RTC assumption (1/3).

In the subsequent sequence, after ***m₄_p***, ***O*** receives messages ***m₅_w***, ***m₆_p***, ***m₇_r***, ***m₈_w*** and ***m₉_r*** successively. The write-message ***m₅_w*** is received while no processing is underway. For this reason, associated-method ***op1_w*** can begin executing immediately. Message ***m₆_p*** arrives during this execution and triggers operation, ***op3_p***, whose concurrency mode is parallel. The method implementing this operation can thus execute regardless of its execution context. Later, following completion of the writer-method previously underway, the object receives read-message ***m₇_r***. Execution of associated-method ***op2_r*** is immediately possible, since the only other method executing is a parallel one. During execution of the method triggered by handling of ***m₇_r***, write-message ***m₈_w*** is received, but is blocked for reasons of concurrency (i.e. a read message is being processed). While the same execution is still underway, read-message ***m₉_r*** follows on ***m₈_w***. Message ***m₉_r*** has a time constraint that gives it priority over ***m₈_w***. Since ***m₉_r*** is associated with a reader method and the method executing is also the read type, processing associated with ***m₉_r*** can take place parallel to the execution underway. Message ***m₈_w*** is processed sequentially, following completion of the two reader methods.

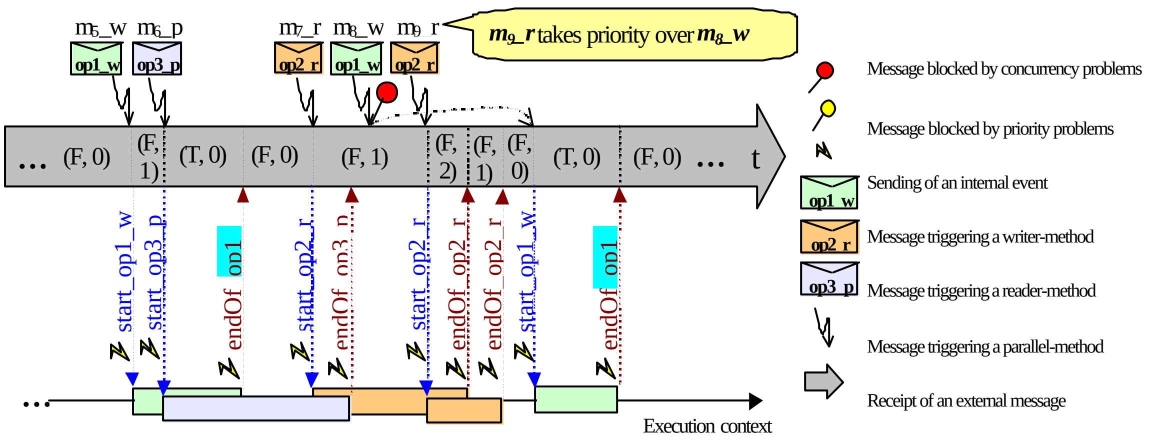

m5_w m6_p m7_r m8_w m9_r
m9_r takes priority over m8_w
op1_w op3_p op2_r op1_w op2_r
… (F, 0) (F, 1) (T, 0) (F, 0) (F, 1) (F, 2) (F, 1) (F, 0) (T, 0) (F, 0) … t
start_op1_w
start_op3_p
endOf_op1
start_op2_r
endOf_op3_p
start_op2_r
endOf_op2_r
endOf_op2_r
start_op1_w
endOf_op1
Execution context
Message blocked by concurrency problems
Message blocked by priority problems
Sending of an internal event
Message triggering a writer-method
Message triggering a reader-method
Message triggering a parallel-method
Receipt of an external message

Figure 56: Illustration of the Implemented RTC Assumption (2/3).

The following sequence implements the three messages ***m₁₀_r***, ***m₁₁_w*** and ***m₁₂_r***. This configuration resembles the ***m₇_r***, ***m₈_w*** et ***m₉_r*** example given earlier, except that, this time, the second read-message, ***m₁₂_r***, has a time constraint with lower priority than write-message ***m₁₁_w***. Message ***m₁₂_r*** is therefore blocked until ***m₁₁_w*** has been taken into account, since the latter has a severer time constraint. Since the operation associated with this method is the write-type, processing of the associated-method is serialized and starts as soon as the method underway is fully executed. Execution of method ***op2_r*** associated with ***m$_{12}$_r*** must likewise be serialized, since a writer method is already underway, and must wait to be executed for the associated method triggered by ***m$_{11}$_w*** to be completed.

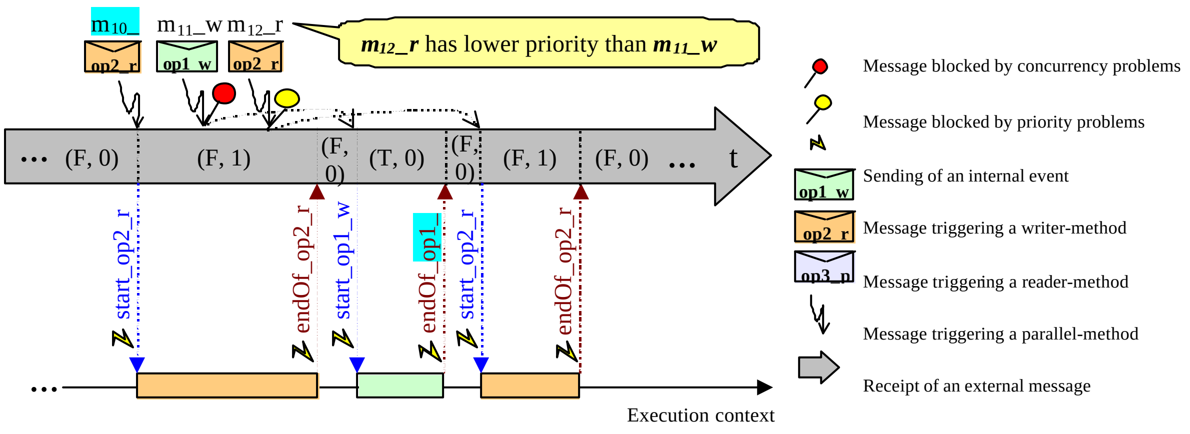

m10_
op2_r
m11_w
op1_w
m12_r
op2_r
m12_r has lower priority than m11_w
… (F, 0)
(F, 1)
(F, 0)
(T, 0)
(F, 0)
(F, 1)
(F, 0) …
t
start_op2_r
endOf_op2_r
start_op1_w
endOf_op1_
start_op2_r
endOf_op2_r
Execution context
Message blocked by concurrency problems
Message blocked by priority problems
Sending of an internal event
Message triggering a writer-method
Message triggering a reader-method
Message triggering a parallel-method
Receipt of an external message

Figure 57: Illustration of Implemented RTC Assumption (3/3).

2.2.3.2 Activity – "describe protocol view"

The protocol view corresponds to a specific use of *UML* state machines, i.e. specification of call protocols for a class, also known as the "life cycle" of that class. Protocol automatons define the context and the order in which class operations may be called. In a way, they act as the "operating modes" of the classes whose behaviour they describe. Operation behaviour is then contained in the methods specification describing the implementation of operations, rather than in basic action sequences dispersed over the various transitions of a state machine. Protocol state machine transitions are a more restricted version of standard *UML* transitions. They have been given the name ***protocol-transition*** and comprise only a trigger event limited to operation calls (*CallEvent*) as well as a possible guard condition. The right hand side of the transition specification is empty, since the action sequence is implicit, i.e. contained in the method specification associated with the operation invoked by the received event. In subsequent paragraphs, the method executed after firing of a protocol-type transition is referred to as ***associated-method***. This term designates a method that implements the operation associated with a protocol-type transition trigger event.

Modelling rule 21 : The protocol view of behaviour automaton contain only protocol-type transitions whose syntax is as follows:

event-name '(' comma-separated-parameter-list')' '[' guard-condition ']'

where **event-name** is the name of an operation belonging to the interface of the class defining the execution context of the protocol automaton containing the transition.

Note that invoking the associated-method does not trigger a new event in the application, but instead serves as an internal event of sorts, within the context to which the state machine is attached.

As describe in Figure 58, this activity, performed by the detailed requirements analyst, may be divided into both steps:

- The **"identify states and transitions"** step aims at building the protocol view of the class' behaviours. The analyst identifies characteristics state of a class and links them via transitions.
- The **"process unexpected messages"** step focuses on the specification of the behaviour following the receipt of messages that are unexpected in a given state of the object.

The output of this step is the work product called **Protocol View**. This latter is a state machine diagram describing the protocol view of the global behaviour of class.

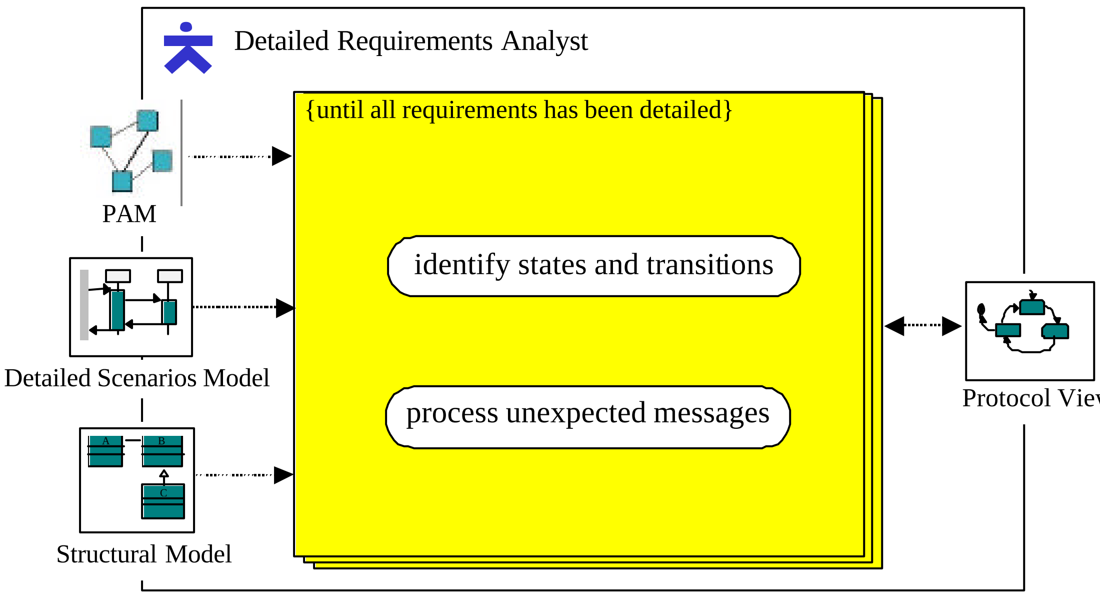

Detailed Requirements Analyst
{until all requirements has been detailed}
identify states and transitions
process unexpected messages
PAM
Detailed Scenarios Model
Structural Model
Protocol View

Figure 58: Activity – the "describe protocol view" Activity of the AIT-WOODDES Methodology.

Step – "identify states and transitions"

In *UML*, if multiple outgoing transitions emanate from a junction or a choice point, only one of the outgoing transitions, whose guard is true, is taken. Where multiple transitions have guards that are true, a transition from this set is chosen and fired by an algorithm that is not specified in *UML*. To supplement this variation point, one applies a modelling rule with enough scope to preclude this type of situation. By applying Modelling rule 22 :, a pseudo-state of the junction or choice type may only has one outgoing transition whose guard is true. Therefore, non-determinist issues can arise from such modelling point that is very important in the real time domain.

Modelling rule 22 : If a ***junction*** or ***choice***-type pseudo-state has multiple outgoing transitions, their respective guards must be mutually exclusive, so that only one transition can fire at a time. In the opposite case, the model will be considered to be ill-formed.

The following paragraphs provides simple examples of specifications for state machine protocol views illustrating the different steps of the "describe protocol view" activity. In the same way as for previous chapters, *Speed* and *Regulator* classes serve as illustrations, in this case of behaviour modelling.

Analysis of application specifications results in attachment of two states to the regulator *Speed* class:

– a state known as "***InService***", in which the object is operational and can perform acquisition operations and updates of its attributes;

– and a "***Standby***" state, in which the object is awaiting a request to start acquisition operations.

On this basis, the analysis is then refined by specifying which operations can be called in the different states and which transitions are triggered by calling them.

The first issue of interest is transition from one state to another. According to *UML* rules for protocol automatons, transition between two states can only be achieved by receiving an operation call. In the example given here, transition from *InService* to *Standby* is achieved by receiving a call to the *Speed* class operation *startAcquisition*. The opposite transition is associated with the *stopAcquisition* operation.

An object owns two special families of operations – create operations and destroy operations. The first are called to instantiate the objects of a class and the second to destroy target instances.

The responsibility of a state machine attached to a class is to describe the overall behaviour of the instances of that class, i.e. its instance creation and destruction stages. To do so, *UML* introduces two types of particular states: initial states and final states.

final states: In *UML*, a final state is a special kind of state signifying that the enclosing composite state is "completed". If the enclosing state is the state machine top state, this means that the entire state machine has completed. *UML* semantics do not explain the significance of state machine completion or the impact of this "completed" state on the context attached to a state machine. It is now recognized that *UML* class instantiation and destruction semantics are globally unclear with respect to the notion of behaviour specification via a state machine. The notion of state machine instance does not in fact exist in *UML*.

To remedy this lack of clarity, the present method takes the following position: each time a final state is reached, the entire state machine is considered to be "completed". If a composite state is completed, the enclosing state can also be considered to have reached a final state, i.e. it is also "completed". The same is true of states that are transitively owned through the hierarchy rooted in the top state, which encloses all state machine states. Moreover, if a state machine has completed and its context is a class, then the state machine instance and the object whose behaviour it describes are destroyed. This leads to the following modelling rule:

Modelling rule 23 : An event triggering protocol-transitions whose target state is a final state must be associated with a destroy-type operation. This same destroy operation must belong to the interface of the statemachine context class.

For the *Speed* class, final state can be reached from both *Standby* and *InService* states. The model thus contains several final transitions, and each of these is associated with a *Speed* class destroy operation by default. The result is a preliminary version of a protocol automaton describing the life cycle of *Speed* class objects (Figure 60):

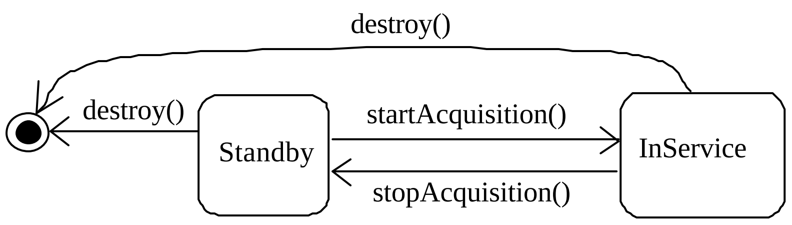


Figure 59: Specific Case of Final Transitions and Destroy Operations.

In *UML*, initial transitions (meaning all transitions originating from an initial state) do not usually have trigger events. They are thus executed automatically, when a transition entering a composite state is fired. However, the initial transition from state machine top state is an exception to this rule*: it is the only initial transition that can own a trigger event (which must be the* ***CreateEvent*** *type)*. This special kind of event is linked with a special operation, a class constructor operation whose execution generates a new class instance. *UML* semantics also require that an initial state have one and only one outgoing transition, as opposed to the different constructor operations available to a class. There is thus a contradiction between the multiple instantiation capabilities (different create operations) of a class and the need to specify its behaviour with a top state having no more than a single initial transition, even where that transition is required for class instantiation modelling. To overcome this inconsistency in *UML* semantics, one authorizes here the initial state of a state machine top level to have several outgoing transitions triggered by a *CreateEvent*-type event. This is treated as a divergence from *UML* semantics, since the need for several create operations per class seems inevitable. We will try to overcome this issue in UML2.0.

Project specifications call for the *Speed* class to have a single create operation associated with the initial transition resulting in the *Standby* state. A supplemented version of the protocol view of *Speed* class behaviour then shows (Figure 60):

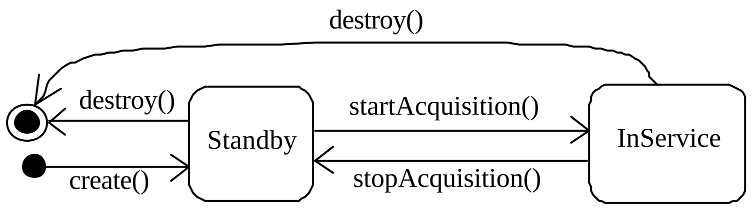


Figure 60: Initial Transition and Speed *Protocol View.*

The problem raised above can be illustrated by adding to the *Speed* class an additional create operation. Take, for example, the operation *create(spd: integer)*, whose execution results in creation of a *Speed* class instance directly in the *InService* state. The model of the *Speed* class behaviour protocol view then becomes:

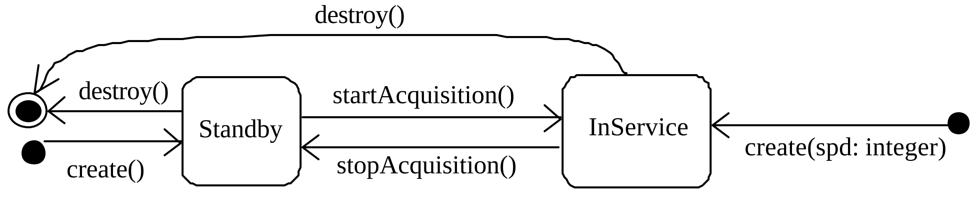


Figure 61: Definitive Protocol View of the Speed Class Behaviour Automaton.

The next step is to specify, on a state by state basis, which operations may be called. In the *InService* state, operations that are both legitimate and consistent with the standard are: *acquireSpeed* and *giveSpeed* (Figure 62).

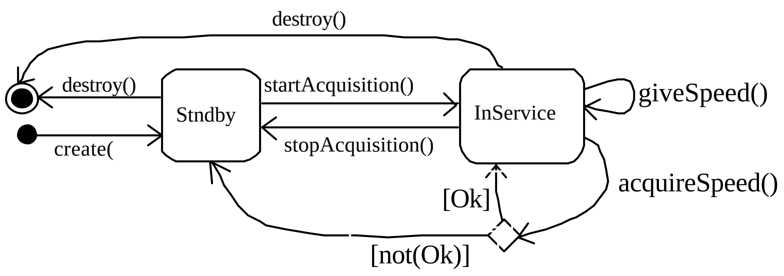


Figure 62: **Speed** *Class Protocol Automaton*

Finally, note that a transition triggered by receipt of a call to *acquireSpeed* has two possible paths in the protocol automaton: transition from the *InService* state to itself; and transition between the *InService* and *Standby* states. Such transitions are composite ones, each made up of three simple transitions connected by a "*choice*"-type pseudo-state. Since guard conditions for the two transitions originating from "*choice*" transitions are exclusive, the choice point is deterministic. Choice is made by dynamic evaluation on executing the associated method for *acquireSpeed*, using the boolean expression *Ok*. If this boolean evaluates true at the end of the first simple transition ("*choice*"), the target state of the transition triggered by *acquireSpeed* is *InService*. If the expression *Ok* evaluates false, the object returns to *Standby* state at the end of the transition. This second path corresponds to cases where the speed acquisition operation detects a communication problem with the speedometer. It then places the object on *Standby* (by performing any necessary treatments), since resumption of speed acquisition may, for example, require specific sensor reconfiguration.

Step – "process unexpected messages"

The preceding paragraphs explained how object behaviour is specified for an operating mode considered to be "normal". The different state machine views (protocol and triggering) used to specify the overall behaviour of an object describe its potential behaviour as a function of state. Now, it may happen that, while an application is running, one of its objects receives a message that it cannot handle in whatever its current state. This is what as known as an "unexpected message".

In such situations, *UML* calls by default for the "unexpected" message to be ignored and lost. It is, however, possible to "defer" the message, i.e. to save it until the next state reached by the object. When the object reaches this subsequent state, the saved message is either again deferred in the context of the new state, immediately handled, or, if it is neither deferred or handled, ultimately lost.

In a real time context, it may be wise to modulate this behaviour. An object may, for example, be temporarily incapable of handling a message at a given time, but later recover the required capability. The following rule is thus applied by default: a message that cannot be immediately handled is saved and placed on standby until the object is capable of processing it. In the event of malfunction or errors in application design, such a strategy could, however, keep messages on standby for indefinite periods of time.

To model this type of behaviour the approach we present here therefore proposes additional mechanisms for refining the description of object response to unexpected messages. These mechanisms can only be implemented for real time or protected passive objects and apply both to operation call and signal-type messages.

By default, one prefers to consider that messages not immediately processed in the current object state are saved and put back into the object mailbox. When compared with standard *UML* semantics, this amounts to saying that all events that can be received by a state machine are deferred for any state in which they do not trigger transitions. Formally speaking, with respect to *UML*, this means that each top state of each behaviour automaton defers all of the events processable by the object whose behaviour it describes. (Indeed, an event deferred by a given state is implicitly considered to also be deferred by all the direct or indirect sub-states of that state). Therefore, since any one state is necessarily a sub-state of the top state, if the processed event does not trigger a transition, regardless of the state of the automaton, it is deferred and not lost as was the case by default in standard *UML* semantics.

Modelling rule 24 : By default, an event that cannot be handled in a given behaviour automaton state is saved for subsequent processing.

In addition, to enhance the expressive power of models developed with our approach and to specify particular situations resulting from receipt of unexpected messages, the model selected offers three special mechanisms for refining the description of a class behaviour automaton:

1. immediate rejection of a message with generation of an exception: Case where the instances of a class cannot receive a message because they are not, at the time of receipt, in a state compatible with its treatment. This situation is interpreted by the application as an instance usage error and results in generation of an exception by the receiving object. The type of exception generated can be specified for each state attaching to a state the tagged value *RejectedException* typed *userExceptionClass*. The *userExceptionClass* may be either a class predefined in the underlying development framework, e.g. *RejectedException* in *ACCORD/UML*, or a child of this class. It contains the name of the raised exception, which must be of a type that can "inherit" from the *IncompatibleState* type proposed by AIT-WOODDES framework. The default exception generated by the object will also be *IncompatibleState*. If the user has provided for catching an exception, she/he can then perform the relevant replacement operations. If not, the exception is routed back to the application's real time object execution support layer, which detects a serious application error and dispatches it to all of the real time objects to trigger a complete application halt.

2. immediate message discard: Case where the message is destroyed immediately on receipt if the object is not in a state compatible with message processing[23]. Where the message calls for execution of an operation with a return value or output parameters, the designer must define replacement values for each such value and parameter. It is also possible here to specify generation of an exception, using the tagged value *IgnoredException*. The generated exception must then be a type that can inherit from *IgnoredMessage* class defined in the AIT-WOODDES framework. An instance of the *IgnoredMessage* class is generated by default. If the user has provided for catching the exception, she/he can then use it, while performing the necessary replacement operations. If not, the exception is routed back to the real time object support, which takes it into account in processing output parameters or the return value. In this case, the exception is dispatched only to the object concerned, and the message sender ignored.

[23] This is the behavior by default in UML

3. placing message on standby: The third case is also the default operating mode. The unexpected message is placed on standby until the object is able to handle it, provided that, in the meantime, its deadline has not expired. If the message has no predefined processing deadline, it may remain "on hold" indefinitely (which is why a deadline should always at least implicitly be attached to each message treatment by a real time object).

To specify the appropriate response to receipt of an unexpected message, the designer has available two tagged values, ***{rejectedEvent}*** and ***{ignoredEvent}***. These values contain a list of types of events rejected by the object and a list of events ignored by it in a given state. Each of these specifications is only valid for the state to which it is attached.

Modelling rule 25 : A tagged value ***{rejectedEvent = (comma-separated-list-of-event-types)}*** attached to a statet ***S*** specifies that, in this state, any event whose type corresponds to one of those on the ***comma-separated-event-list*** is rejected.

The tagged value ***{ignoredEvent = (comma-separated-list-of-event-types)}*** attached to a state ***S*** specifies that, in state ***S***, any event whose type corresponds to one of those on the ***comma-separated-event-list*** is ignored.

Note that another exception management solution is that offered by the loose synchronous messaging mode. In such cases, the sender execution thread is suspended until message treatment actually begins. This thread thus enables receiving and handling of exceptions generated for said message.

In the example used here, the *Speed* class control automaton can be refined by specifying reactors to unexpected messages. If a *Speed* type object receives a message requesting a regulation halt, it is placed in *Standby* state and then preferably ignores the message. The *Standby* state has a tagged value, *IgnoredEvent* , containing the names of the concerned events, in this case *stopAcquisition*. By contrast, if, in the same *Standby* state, an object receives a message requesting acquisition (*acquireSpeed* operation call) or even a request for actual vehicle speed (*giveSpeed* operation call), then the object must reject it (and generate an exception). This results in a new *Speed* class protocol automaton, as shown in Figure 63.

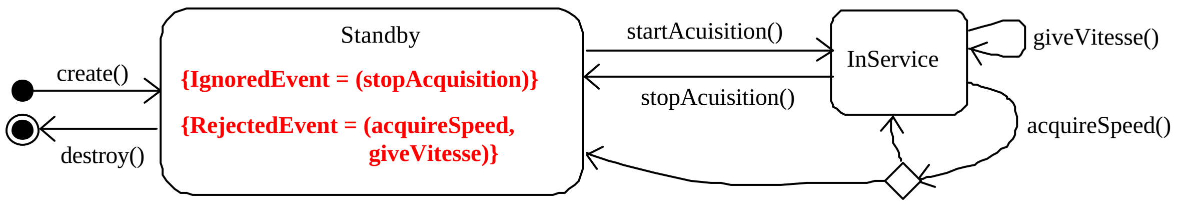


Figure 63: Specification of the Speed Class Response to Unexpected Messages.

In general, object-oriented approaches only specify preconditions attached to a transition where they are needed to reference data accessible within the context of the object. As already stated above, this entails several distinct types of data:

– arguments supplied to the operation;

– object roles and attributes;

– object state, as specified by the control automaton.

Of particular interest are the types of data likely to generate non-compliance with a precondition:

1. If referenced data relate only to argument values, there is always method rejection, followed by generation of an exception and an application abort, unless the exception is caught in the execution thread of the message sender.

2. Where an object state is involved, the same conditions apply as above and, where the object is a real time object, any of the abovementioned mechanisms (rejection, discarding, standby) may be activated.

3. Where the data of interest are object roles and attributes, and a Real Time Object is involved, one uses the same type of behaviour specifications as those defined for object state constraints: rejection, discarding or standby. If the object is a passive one, its behaviour is defined by default, i.e. there is generation of an exception, which, if not reversed, aborts the application.

Obviously, it is not always possible to determine whether or not a given expression refers only to object roles, attributes or state (if it uses pointers, for example). Then again, a same precondition may make use of different types of data (e.g. method parameters and object attributes) and, in such cases, the type of parameter responsible for violating it is often difficult to identify. The programmer must therefore explicitly specify, using one of the preceding specifications, which real time object behaviour is appropriate for each method precondition. By default, the behaviour elected will be the same as for passive objects.

2.2.3.3 Activity – "describe reactive view"

Real time objects specifically correspond to the "reactive" elements of an application. Such elements are capable of responding to received signals, reacting when some condition is reached, even initiating automatic treatments on achieving a particular condition. Triggering state diagrams introduced into the behavioural models provide a restricted view of overall object behaviour. They are an abstraction, focussing on the reactivity of a class, by concentrating on four special aspects of its reactive behaviour:

1. so-called "automatic" object behaviour relating to a ***CompletionEvent*** (behaviour triggered after the object enters a specific state);
2. behaviour occurring in response to a ***ChangeEvent***, i.e. a boolean expression that becomes true;
3. behaviour responding to a ***TimeEvent,*** i.e. expiration of a specific deadline;
4. response to signals to which the object class was declared receptive in the structural model (**Figure 45**).

This view concerns only real time objects. Specification of object behaviour relating to real time status is thus separated from the so-called "protocol" aspect of its behaviour, which may be independent of real time status. Thus, if an object emanating from a real time object class is instantiated as a standard object, it preserves the protocol content of its behaviour specification. By separating these two aspects, it is possible to again build objects with greater portability, which are reusable in a different context, for example with object approaches not necessarily dedicated to the real time domain.

Triggering automaton semantics are based on that of *UML* protocol automatons. This leads to the restriction of *UML* state machines, and like the addition of protocol-transitions to protocol automatons, a new specification of standard *UML* transitions is also introduced here.

The main restrictions about the triggering view of a state machine are related to the abovementioned new type of transition, known as a ***triggering-transition***. Its semantics is also defined as follows:

– The lefthand portion of a triggering-transition specification contains trigger events such as ***SignalEvent***, ***ChangeEvent***, ***TimeEvent***, or ***CompletionEvent*** (completion transition);

– The righthand portion of the transition specification, which contains the list of actions to be executed, is restricted to a single, operation call-type action. Performance of this action generates an operation call internal to the object, which results in execution of the method implementing the invoked operation. Unlike protocol automatons, where action executed by firing the transition is implicitly specified, in triggering automatons, the operation call action is explicitly specified in the righthand portion of the transition.

Modelling rule 26 : Triggering views of behaviour automatons contain only ***triggering-transition***-type transitions whose syntax is as follows:

'event-name '(' comma-separated-parameter-list ')' '[' guard ']' / …

… invoked-operation-name '('comma-separated-parameter-list ')'

where:

- *event-name* is either the name of a signal declared in the structural model of the class defining the triggering automaton context (*SignalEvent*), or the name of an event of the *ChangeEvent* or *TimeEvent* kind declared in the automaton. If there is no event-name in the specification, the transition is a completion-transition and the event is an internal one of the *CompletionEvent* kind;
- *invoked-operation-name* is necessarily the name of an operation of the class defining the execution context of the protocol automaton.

To model the reactive view of the class behaviour, the detailed requirements analyst operates the following steps (Figure 64):

- The **"identify reactive transitions"** step aims at building the reactive view of the class' behaviours. The analyst identifies reactive behaviours of a class and add matching reactive-transitions between states already identified in the protocol view.
- The **"process unexpected messages"** step focuses on the specification of the behaviour following the receipt of messages that are unexpected in a given state of the object.
- The **"declare automatic treatment"** step aims at declaring automatic behaviours of classes through introduction of specific transition in the reactive view.

The output of this step is the work product called **Reactive View**. This latter is a state machine diagram describing the reactive view as defined previously of the global behaviour of class.

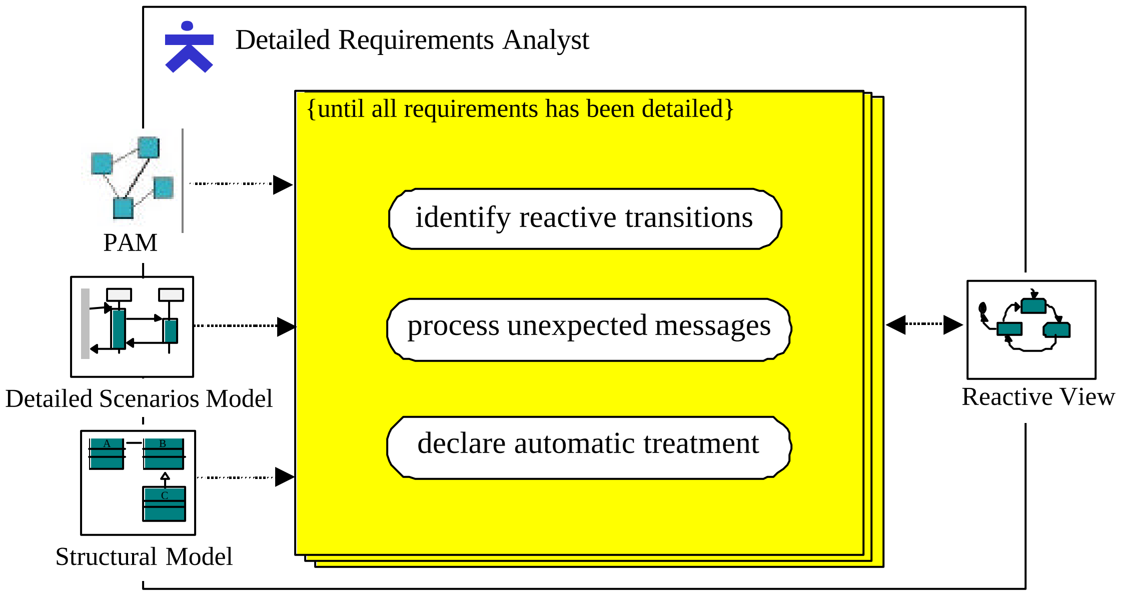

Detailed Requirements Analyst
{until all requirements has been detailed}
identify reactive transitions
process unexpected messages
declare automatic treatment
PAM
Detailed Scenarios Model
Structural Model
Reactive View

Figure 64: Activity – the "describe reactive view" Activity of the AIT-WOODDES Methodology.

Step – "identify reactive transitions"

The *Regulator* class of the application example is used hereinafter to illustrate modelling of an automaton triggering view. Figure 65 shows an excerpt from the protocol view of its behaviour automaton.

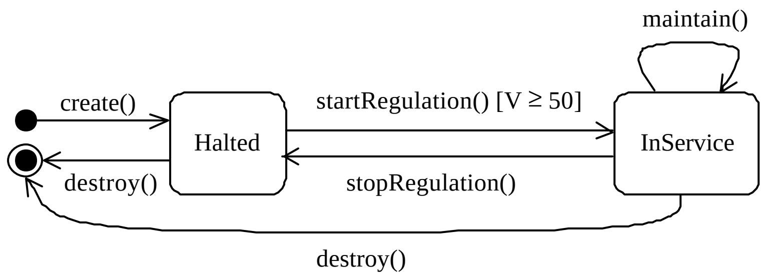


Figure 65: Excerpt from the Protocol View of the Regulator Class Behaviour Automaton.

The next paragraphs in this presentation supplement the description of *Regulator* class behaviour by specifying its triggering view. As seen in a previous chapter, this class was declared receptive to two types of signals: *OnOff_Reg* and *StopCar*. Project specifications call for *OnOff_Reg* to trigger a complete halt in regulation if the system is in service and, where the opposite is true, to start it up again. Where the regulator is not operating (*Regulator* instance in *Halted* state), this signal is attached to the *startRegulation* method. If the regulator is already in service (*Regulator* instance in *InService* state), the signal is attached to the *stopRegulation* signal. The *StopCar* signal must result in system shutdown and, more specifically, in destruction of the object once speed acquisition is halted. It is thus natural to attach this signal to *destroy* method execution, when the object is in *InService* state. This results in the triggering automaton below (Figure 66):

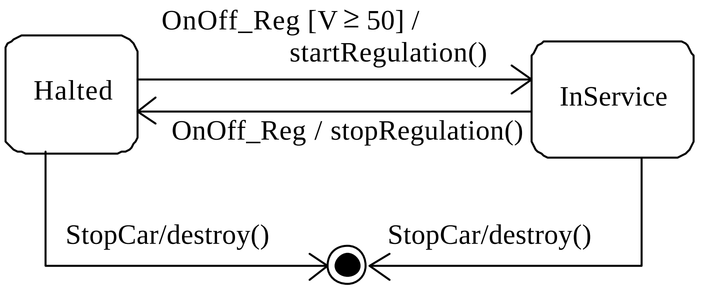

OnOff_Reg [V ≥ 50] / startRegulation()
Halted
InService
OnOff_Reg / stopRegulation()
StopCar/destroy()
StopCar/destroy()

Figure 66: Regulator *Class Reaction to OnOff_Reg and StopCar Signals.*

Remarks: In a multitasking context, prudence should be exercised in matters relating to object destruction. This is because, once a real time object is destroyed, it can no longer receive messages; and attempts to send messages to an non-existent object may cause serious application error. In the case studied here, the result would be a sudden stop in the application, without assurance that all operations normally preceding such an abort have been duly carried out. Object destruction must therefore be consistently and carefully synchronized over the entire application. This is typically specified in a special application stop scenario (see section 2.2.4 on the interaction model building).

The reaction of a given object must be one of the types of behaviour authorized for that object (no object can do anything it has not been given the right to do), as specified in the protocol view of its behaviour. Triggering-transitions associated with signal reception must therefore always be "legal" with respect to the object control automaton. In practice, this means that a triggering automaton is always included in the global control automaton of the class. Consider, for example, a transition t_1 for the *Regulator* class control automaton and transition t_2 of the triggering automaton for the same *Regulator* class. Transition t_1 is triggered by receipt of the call operation *startRegulation* and has as its source state *Halted* and its target state *InService*. Transition t_2 is triggered by reception of *OnOff_Reg* signal instances triggering execution of the action that calls the *startRegulation* operation; and this transition has *Halted* and *InService* as source and target states respectively. Transition t_2 of the *Regulator* class triggering automaton is "legal" since the *Regulator* class protocol automaton already owns transition t_1.

Modelling rule 27 : If a triggering-transition t_{dec} of the triggering view of a behaviour automaton for a class "*C*" is taken to have the following: source state S_1 and target state S_2, triggered by an event *Evt*, of the *SignalEvent*, *ChangeEvent* or *CompletionEvent* type and resulting in execution of an action attached to the *calledOpe* operation, then t_{dec} is legal only where the protocol view of the behaviour automaton of *C* has a protocol-transition t_{pro} with the following characteristics: source state S_1 and target state S_2, triggered by an event of the *CallEvent* type associated with *calledOpe*.

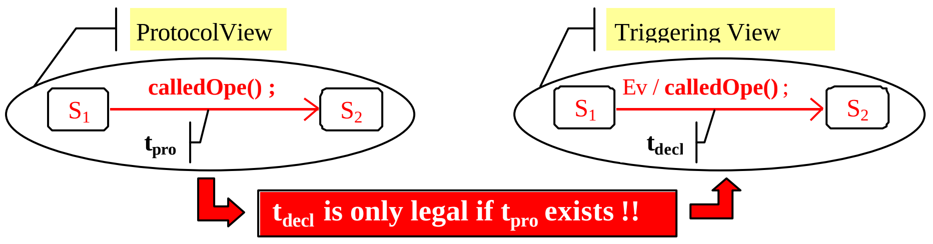


Figure 67: Definition of a "Legal" Triggering-Transition.

One important, direct consequence of this modelling rule is that all of the possible states of an object class are specified in the protocol automaton for that class. It is thus impossible, in a class triggering automaton, to have a state that was not previously introduced into the protocol automaton for that class.

Modelling rule 28 : The complete set of states defined in the triggering automaton of a class is included in the set of states defined in the protocol automaton for the same class.

Remark: Initial transitions in a *UML* state diagram can only have operation call-type triggers. The top state of a triggering state diagram thus never owns an initial state.

Signal Event and Triggered Operation Parameters

Where the receiving object state so allows, signal reception triggers firing of a transition specified in the triggering view of its behaviour automaton. Transition firing causes execution of the *CallAction*–type action specified in the righthand portion of the transition. This in turn generates an internal call to the associated operation, which results in execution of the method implementing the invoked operation.

Since communication by signal is asynchronous, the invoked operation must not have output parameters or a return value. If it did, this would be a modelling error, since signal reception is associated with receipt of an asynchronous message whose sender is unknown to the receiver (see section ‘Step – “process unexpected message” on p.68).

Modelling rule 29 : If the trigger event of a triggering-transition is the *SignalEvent* type, the operation specified in its righthand portion may have neither output parameters nor a return value.

On the other hand, operations specified on the triggering view transitions of a triggering automaton may very well have input parameters in all cases. If an operation triggered by signal reception has input parameters, there are two possibilities: either the signal reception event triggering the transition has parameters or it does not.

For a **signal carrying no information,** there are in turn three possibilities:

The triggered operation, specified on the fired transition, does not have parameters either. In this case, there is no particular problem.

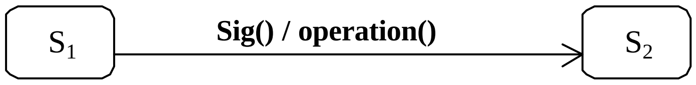


Figure 68: Reception of a Non-parameterized Signal Triggering a Non-parameterized Operation

Each of the input parameters of the invoked operation has a default value. In such cases, default values are used to execute the method implementing the operation specified on the fired transition.

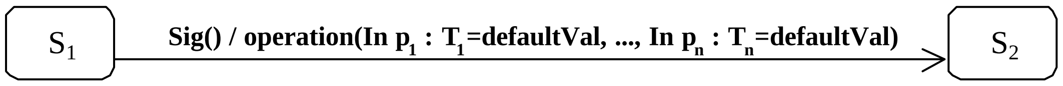

S_1
Sig() / operation(In p_1 : T_1=defaultVal, ..., In p_n : T_n=defaultVal)
S_2

Figure 69: Reception of a Non-parameterized Signal Triggering a Parameterized Operation

A calculation procedure exists for the values of each parameter of the triggered operation, for each transition specified in the triggering automatons. In such cases, the designer must associate with the transition a segment of code specifying this initialization. Said specification may be realized using a tagged transition value { initParam }. This tagged value is the Expression type and contains a string of characters evaluated in the selected implementation language; it likewise enables assignment of values to the parameters of the triggered operation. In cases where both default values and a dedicated calculation procedure specification are available, the values obtained with the calculation procedure will determine those of the parameters used by the operation's associated-method.

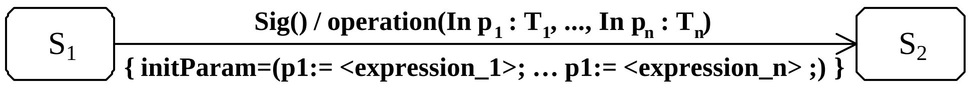

S_1
Sig() / operation(In p_1 : T_1, ..., In p_n : T_n)
{ initParam=(p1:= <expression_1>; … p1:= <expression_n> ;) }
S_2

Figure 70: Reception of a Non-parameterized Signal Triggering an Operation with Parameters Initialized using an Additional, Explicit Expression

In the preceding situation, as the received signal did not have parameters, the event generated by its receipt did not have parameters either. However, unlike the "pure" signals described in Harel-87 statecharts, ***UML* signals are perfectly capable of carrying information in the form of parameters attached to signal output.** This means that a signal-type event can have parameters and it is thus necessary to specify the mechanism enabling passage from the parameters of an event that triggers firing of a triggering-transition (specified in the lefthand portion of the transition) and those of the operation call action executed as a result of transition firing (specified in the righthand portion of the transition). In the case examined here, there are two possibilities:

The signature of the action described on the righthand side of the fired transition is specified in a manner consistent with the signature of the trigger signal event. This means that, if the signal event triggering the transition is considered to own an ordered list of n parameters $(pe_i)_{i\in[1..n]}$, the signature of the operation triggered by its receipt must begin with a list of input parameters $(po_i)_{i\in[1..n]}$ of the same type. In addition, for any i from 1 to n, the parameter po_i of the operation has the same type as received signal event parameter pe_i. In this specific case, note that the parameters contained in the operation signature must number at least n (the number of trigger signal event parameters). In cases where the action specified on the transition righthand side has more than n parameters, all other parameters (whose subscript value is necessarily higher than n) must have a default value.

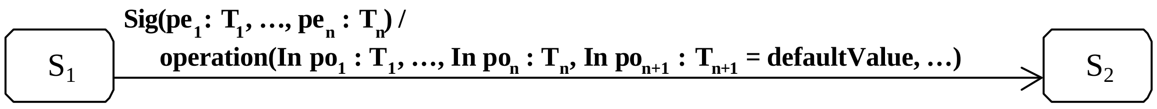

S_1
Sig(pe_1: T_1, …, pe_n : T_n) /
operation(In po_1 : T_1, …, In po_n : T_n, In po_{n+1} : T_{n+1} = defaultValue, …)
S_2

Figure 71: Reception of a Parameterized Signal Triggering a Parameterized Operation

A procedure exists to specify use of data carried by the signal reception event, i.e. the passage from event parameter values to the input parameters of the triggered method. This specification is achieved via the tagged value {useSigData}, which contains the necessary segment of code. Said tagged value is the Expression-type and contains a string of characters which, evaluated in the selected implementation language, enables assignment of values to the parameters of the triggered operation as a function of parameter values provided by the signal event.

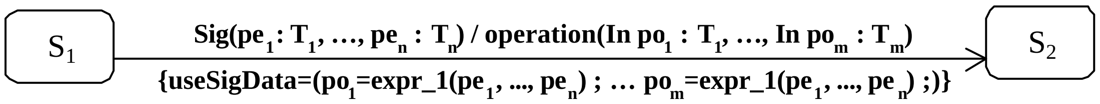

S_1
Sig(pe_1: T_1, …, pe_n : T_n) / operation(In po_1 : T_1, …, In po_m : T_m)
{useSigData=(po_1=expr_1(pe_1, ..., pe_n) ; … po_m=expr_1(pe_1, ..., pe_n) ;)}
S_2

Figure 72: Reception of a Parameterized Signal Triggering a Parameterized Operation and Explicit Signal Parameter Use Expressions

Step – "process unexpected messages"

This step is identical as this one of the "describe protocol view" activity (see section Step – "process unexpected messages" p. 68).

Step – "declare automatic treatment"

To afford maximum object autonomy, the triggering automatons may also serve to specify automatic treatments performed by the object when it reaches a particular state. To do so, it is possible to specify, in a *UML* state machine, a set of transitions known as ***completion-transitions***. This type of transition has no explicit trigger, although it may have a guard defined. Completion transitions are triggered by a special internal event known as a "completion event". Such an event is implicitly generated by the state machine each time it reaches a stable state, i.e. when all transition and entry actions and activities in the currently active state are completed. A completion event is processed as soon as it is sent. If the current state of the object has an outgoing completion transition, firing of this transition takes priority over that of any other transitions emanating from that state.

In the this approach, states have neither entry actions nor activity. This is why state automatons describing class behaviour generate a completion event each time a transition finishes executing and reaches a state. Thus, when a state has a completion transition-type outgoing transition, the operation associated with such a transition is executed as soon as the state is reached and takes priority over all other transitions emanating from that state. One of the results is that such a state cannot execute anything other than a completion transition.

Moreover, if a state has several outgoing completion transitions, because they are all triggered by the same type of event (completion event), all of them can potentially be activated and must therefore be mutually exclusive. This mutual exclusion conditions must be ensured by the respective guard of each outgoing completion transition of the state. Any other solution would be a modelling error.

Modelling rule 30 : If a state has several outgoing completion transitions, all of those associated with a write operation (i.e. with a *concurrencyMode* attribute positioned to *write*) must have guards whose respective conditions are defined such that the transitions are mutually exclusive, i.e. that only one of them can be fired at a time.

In the speed regulator example, it is advantages to enable automatic triggering of *Speed* class acquisition operations as soon as the object enters *InService* state. To do so, a completion transition is used and the following triggering view is specified (*Figure 73*):

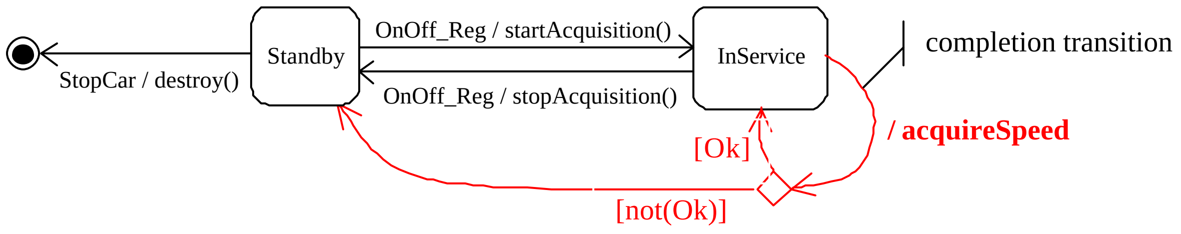


Figure 73: Automatic Triggering of a Transition on Entry into a State.

In the present case, a *Speed*-type object executes the *acquireSpeed* operation as soon as it enters the *InService* state. If acquisition takes place correctly, the transition restores the object to the same state, on completion of the method associated with the triggered operation. What is specified here is the cyclic, autonomous operation of the object, since, once it reaches *InService* state, a completion event is implicitly sent by the control automaton, again triggering the completion transition and so on…

If, however, acquisition does not take place correctly, the transition places the object on *Standby*, which halts cyclic speed acquisition.

Again using the speed regulator as a basis, the situation where an instance of the *Speed* class is in *InService* state and receives an *OnOff_Reg* signal, provides a good example. In this specific case, it would be advantageous for *OnOff_Reg* signal receipt to be taken into account to trigger firing of the *InService* transition to *Standby* and thus stop speed acquisition. But no such condition can exist, since, if speed acquisition always takes place correctly, and because a completion event takes priority over any other received event, the outgoing completion transition from *InService* state is inevitably triggered again each time it completes. It may then be asked how the endless loop triggered by this situation can be broken, i.e., how the internal completion event, which overrides all others, be cancelled out.

The answer to this question is to add to the completion transition a guard enabling choice of the behaviour adopted as a function of object mailbox content. The approach calls for use, in the completion transition guard, of a specific operation, ***isEvent***. (<***Eventname***>). This operation uses as input the name of an event, which then becomes true if the event is present in the mailbox, or false if it is not. Specification of a guard of the *[Ø isEvent(OnOff_Reg)]* type on the previous completion transition ensures that the completion transition is only fired if the object has not received an *OnOff_Reg* signal requesting halt in speed regulation. This is because *isEvent(OnOff_Reg)* becomes true if an *OnOff_Reg*-type signal is present in the *Regulator* object queue. In such cases, the condition *Ø isEvent(OnOff_Reg)* is false and the guard is thus invalid. The completion event is then lost without having triggered the completion transition and the loop is broken. The regulator object can subsequently take into account the events waiting in the queue. The major advantage of this technique is to enable direct clarification, in the class behaviour model, of transition priorities.

To ensure that the speed regulator behaves as expected, a *[Ø (isEvent(OnOff_Reg)) Ù Ø (isEvent(StopCar))]* guard is added to the completion transition executing the *acquireSpeed* operation. This additional guard ensures that the completion transition will only be fired automatically if the object of interest has not received *OnOff_Reg* or *StopCar* signals. Receipt of this type of event does, however, break the loop created by the completion transition. A transition exiting the *InService* state, triggered by *OnOff_Reg* or *StopCar*, may then be fired (Figure 74).

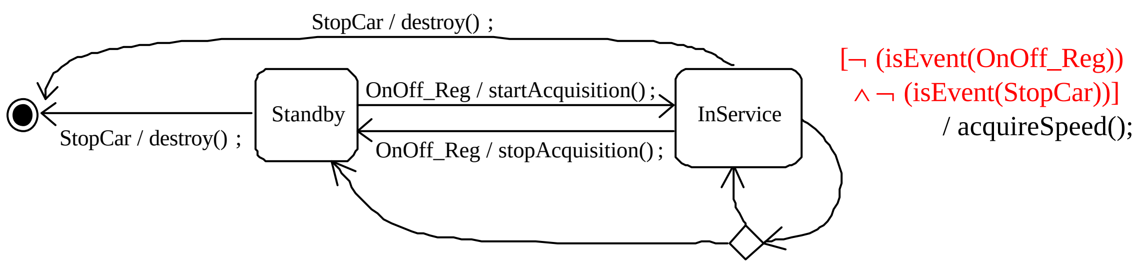


Figure 74: Settling Triggering Conflicts by Adding Preconditions.

2.2.3.4 Activity – "describe domain algorithms"

The subchapter 2.2.3.1 above described how to model the control logic of a class using a state machine with two viewpoints – triggering and protocol. Each of these two automatons specifies the behaviour of an object in response to an operation call or a signal, following a change materialized by a boolean expression that becomes true or after entering a particular state. In each of these cases, the impact of the event is the same, regardless of its type. The receivi ng object executes an operation call-type action. Execution of this action calls the operation associated with it (always a receiving object operation) and thus executes the algorithm specified by its associated-method. To further refine the application model, it is then necessary to specify its algorithmic components, i.e. class methods.

This stage in modelling is chiefly concerned with the external specification of methods, i.e. with method synchronisation points. These points are marked by sending messages and, where appropriate, awaiting replies to them (case of strong synchronous and delayed synchronous messages). Up until now, it was enough to specify that a class was capable of receiving messages in the form of signals or operation calls, without defining aspects of the message sending context. one now proposes to provide such definitions in the methods behaviour automaton. Like the Object Flow diagrams of the CLASS-RELATION [9], methods behaviour diagrams describe the sequence of elementary actions used by the method, together with the objects manipulated and the messages sent during method execution.

2.2.3.5 Activity – "declare real-time QoS"

One of the bases for *Real Time Object* modelling is that real-time constraints are generally attached either to messages sent or received by the *Real Time Objects* of an application, or to clocking of cyclic treatments initiated automatically by these objects.

Remark: In this subchapter, "time constraint" refers essentially to deadline, periods, etc. However, for systems allowing static computation of various application task priorities (e.g. with RMA-type techniques), it is possible to use a development environment version based on the notion of priority. In such cases, time constraints are expressed using the same *RTF* (Real-Time Feature) tagged value, but then have a single parameter known as *Priority*. A time constraint thus takes the following form:

{ RTF = (Priority, priority-value) }

where *priority-value* is a priority value of the underlying real time operating system.

To specify the real-time constraints in a model, one have introduced a specific tagged value *RTF*. This value is attached to model elements in order to model a real-time constraint. This constraint can be attached to the action that generates a message output. It is then applied to treatment of the corresponding event by the object receiving the message. The *Deadline* value is defined with respect to an absolute reference date, *RefDate*, and is given as a parameter of the *RTF* value by specifying the reference unit of the time characteristic in the following form:

{RTF = (RefDate, Deadline(value, unit))}

where *unit* may have one of the predefined values "sec" or "mSec" (or another value defined by the user and convertible to one of the two reference units[24]) and *RefDate* is a TimeVal-type value introduced by the framework to express the concept of absolute date.

Once it has been specified that a treatment must be completed before a given date, the user may wish to also specify that said treatment may only begin on a certain date. To do so, she/he uses the concept of latency, as expressed by the *ReadyTime* parameter of the tagged value *RTF*. This parameter is also expressed in terms of delay with respect to an absolute reference date, the same date as is used to define deadline. The time constraint attached to the model element to be specified then becomes:

{RTF = (RefDate, Deadline(value2, unit2), ReadyTime(value1, unit1))}

As regards the real-time constraints attached to sent messages, such constraints are specifically concerned with treatments triggered by received messages. In *UML*, messages may be sent in either of two different forms: signals (*CallAction*-type action) or operation calls (*SendAction*-type action).

For signals, the one enables real-time constraint specification at two different points in the model:

- signal output;
- signal reception.

Real-Time Constraints and Signal

Sent signal specification involves both the structural class model, in which are specified, among others, the types of signals that can be sent by each class and the model specifying the behaviour of an

[24] The time unit used by default in the *ACCORD/UML* development environment is the second (" **sec**").

operation. In the first case, if a time constraint is attached to a type of signal that a given class is capable of sending, this constraint will have the same value for each instance of that class and each time the predefined type of signal is sent. Under these conditions, it is not necessary to specify the *DateRef* parameter for constraint *RTF*. The reference date used as a basis is, in fact, the date on which the signal is sent.

As an example, the diagram in Figure 43 specifies that the *EngineStarter* class is capable of sending *StopCar* signals. As confirmed by the speed regulator project specifications (or its dictionary), when the car engine stops, the system must also be able to stop, in less than *100 ms*. Modelling of this time characteristic in the class structural diagram modifies the previous specification as shown in Figure 75 for the *EngineStarter* class:

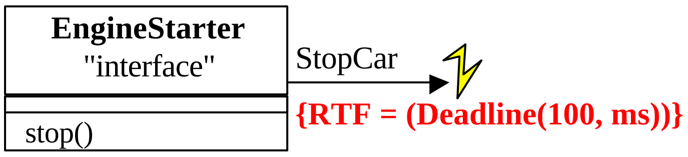


Figure 75: Time Constraint in the Structural Specification of a Type of Sent Signal

On signal receipt point of view there are two levels at which real-time constraints can be specified: either at structural level, i.e. on declaration of the received signal type (**Figure 44**); or at behaviour level, i.e. on triggering-transitions activated by a signal-type event.

In the first case, the *DateRef* parameter of the *RTF* constraint is not specified. This is because the reference date implicitly attached to the message is the date on which the signal was sent. The relative deadline expressed by the *Deadline* parameter must then be determined according to signal output date.

For example, the *Regulator* class is receptive to *StopCar* and *OnOff_Reg*-type signals. To specify that the *Regulator* class must respond to receipt of an *OnOff_Reg* signal instance within 100 ms, a time constraint is simply added to the declaration of receptivity to the *OnOff_Reg* signal and the structural specification of the *Regulator* class is then supplemented as follows (Figure 76):

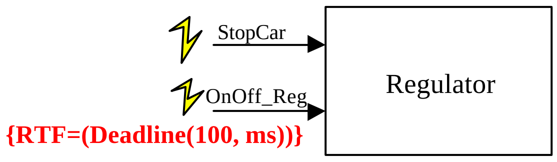


Figure 76: Real-Time Constraint in the Structural Specification of Signal Receipt.

This model has a significant drawback in that it penalizes the reusability of the object specified. Whatever the *OnOff_Reg* sender, the real time constraint attached by *Regulator* instances to the treatments triggered by signal receipt will always be the same: 100 ms. However, for this particular communication mode, it may be useful to specify a signal reception constraint. Signal-type communication is in fact a means for transmitting data. The same signal instance can thus be perceived by several objects and the treatment induced by signal receipt is specific to each receiving object. In such cases, it may be advantageous for the user to specify a different time constraint for each treatment, i.e. to specify it at signal receipt rather than signal output.

Where a triggering-transition is assigned a time constraint, then, regardless of the signal sender, this time constraint is attached to whatever the triggered treatment. Therefore, in the example given in Figure 38, if the object whose behaviour is described receives a signal while it is in *Standby* state, it triggers the *startRegulation* treatment with a relative deadline of 100 milliseconds. Since the absolute date serving as a reference is not specified here, the absolute reference date used to compute the absolute deadline for the triggered treatment will be that on which the signal is sent. In the same way, if the receiving object is in *InService* state, reception of the same signal, *OnOff_Reg*, triggers *stopRegulation* treatment with a relative constraint of 100 milliseconds in respect to the date on which the received *OnOff_Reg* signal instance was sent.

Specification of a real-time constraint on the triggering-transitions of an object's behaviour does not in any way hinder its being reused in a different context from that of this approach, i.e. real time objects. In this case, the behaviour aspect specified by the triggering view is not, in fact, preserved.

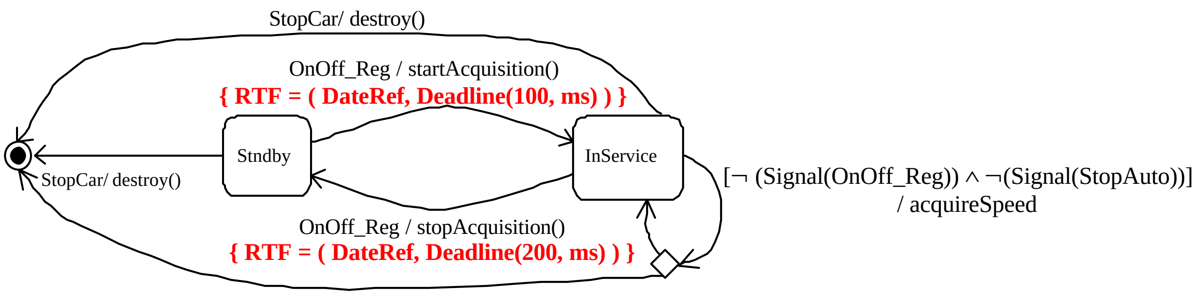

StopCar/ destroy()
OnOff_Reg / startAcquisition()
{ RTF = (DateRef, Deadline(100, ms)) }
Stndby
InService
StopCar/ destroy()
[¬ (Signal(OnOff_Reg)) ∧ ¬(Signal(StopAuto))]
/ acquireSpeed
OnOff_Reg / stopAcquisition()
{ RTF = (DateRef, Deadline(200, ms)) }

Figure 77: Declaration of a Time Constraint Placed on a Triggering-Transition.

Remark: The position of a time constraint with respect to a signal (at output or reception) has definite impact. Specifying the constraint at signal output means that, regardless of the sender, the object will respond to that constraint. In the event of conflict, i.e. if a signal has two time constraints, one at output and one at reception, the constraint specified at output takes priority. In this way, the client is kept firmly "in the driver's seat".

Time Constraints and Cyclic Treatments

The second type of constraint relates to the cyclic performance of a treatment. As already shown in a previous section (see section called Automatic Treatment Specification), a user may specify cyclic treatments in the triggering automaton of a real time object. By attaching real-time constraints to the cyclic treatments of an object, the designer can adjust the frequency of these treatments to make them periodic. In the example taken here, two treatments are initiated automatically and performed cyclically:

– acquisition of data from the sensor simulator via the Speed class *acquireSpeed* operation;

– the control loop realized by the *Regulator* class.

Data acquisition from the sensor simulator takes place via the *Speed* class *acquireSpeed* operation. This cyclic treatment is specified in the class triggering automaton. To clock the cycle, the user can add to this model a tagged value {*RTF*}, that he then places on the completion transition associated with his cyclic treatment ("completion transition"). To define the periodicity of the treatment, the tagged value has a "*Period*"-type parameter that determines the value of the period and the unit of reference. The date of achievement of whatever the state enabling cyclic triggering of the treatment defines the date on which the first cycle begins. The starting dates of subsequent cycles will be computed by shifting forward each by the equivalent of the value of one period with respect to that of the immediately preceding cycle. By default, the deadline for occurrence of a treatment is the starting date for the next occurrence. This is referred to as "periodic treatment to deadline". Specification of the speed acquisition cycle at 2 Hertz is thus expressed as follows (*Figure 78*) :

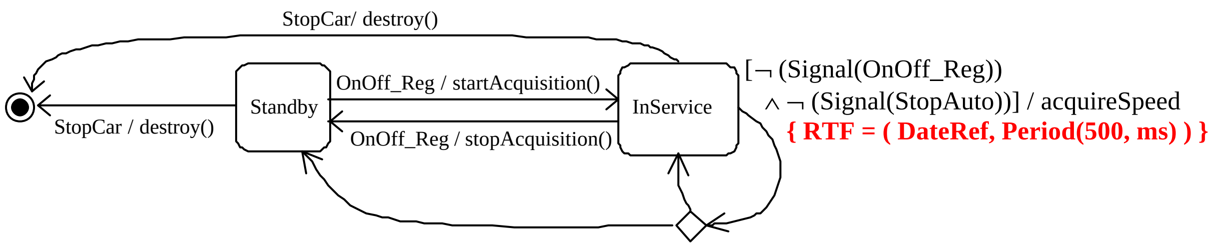

StopCar/ destroy()
OnOff_Reg / startAcquisition()
Standby
InService
StopCar / destroy()
OnOff_Reg / stopAcquisition()
[¬ (Signal(OnOff_Reg))
∧ ¬ (Signal(StopAuto))] / acquireSpeed
{ RTF = (DateRef, Period(500, ms)) }

Figure 78: Declaration of a Periodicity Constraint for Cyclic Treatment.

Remark: Use of a numerical real-time constraint value to specify a period limits the reusability and modularity of the specified class. This is because, under such conditions, all instances of the *Speed* class exhibit the same behaviour. In *InService* state, all *Speed* instances perform the *acquireSpeed()* operation periodically, at a frequency of 10 Hertz. To alleviate this drawback, it is possible to use an intermediate attribute of the *Period* type. If this is a class attribute, then the time constraint will be the same for all instances. If not, the constraint may differ from one instance to another. For the *Speed* class, an acquisition treatment period *acquisitionPeriod* can be specified (Figure 79) and this attribute added to the structural definition of the class (*Period*-type with controlled access).

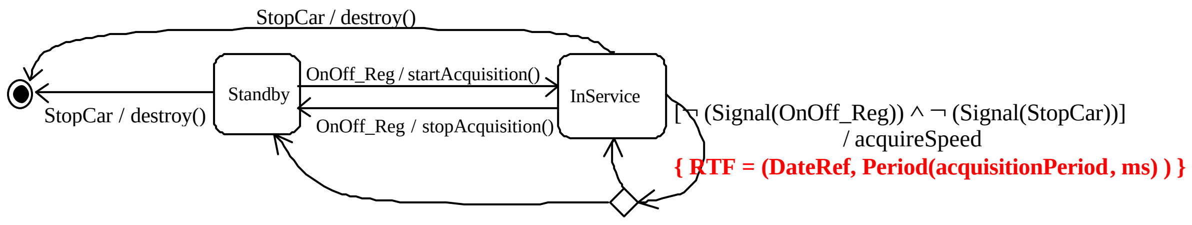

StopCar / destroy()
OnOff_Reg / startAcquisition()
Standby
InService
StopCar / destroy()
OnOff_Reg / stopAcquisition()
[¬ (Signal(OnOff_Reg)) ∧ ¬ (Signal(StopCar))]
/ acquireSpeed
{ RTF = (DateRef, Period(acquisitionPeriod, ms)) }

Figure 79: Time Constraints and Attributes.

Finer timing distinctions can be made by setting a deadline for each occurrence that differs from the starting date of the following occurrence. This can be done by simply including a second parameter in the tagged value that determines duration, to specify the desired interval from the start of cycle one to cycle one deadline. For example, if the user wants acquisition to take place less than 50 milliseconds from the triggering of each cycle, the time constraint becomes: *{ RTF = (Deadline(50, ms), Period(500, ms)) }*. Finally, to specify an initial delay in the start of cycling treatment (valid for each occurrence), a third parameter known as "*Readytime*" is added to the tagged value (Figure 80):

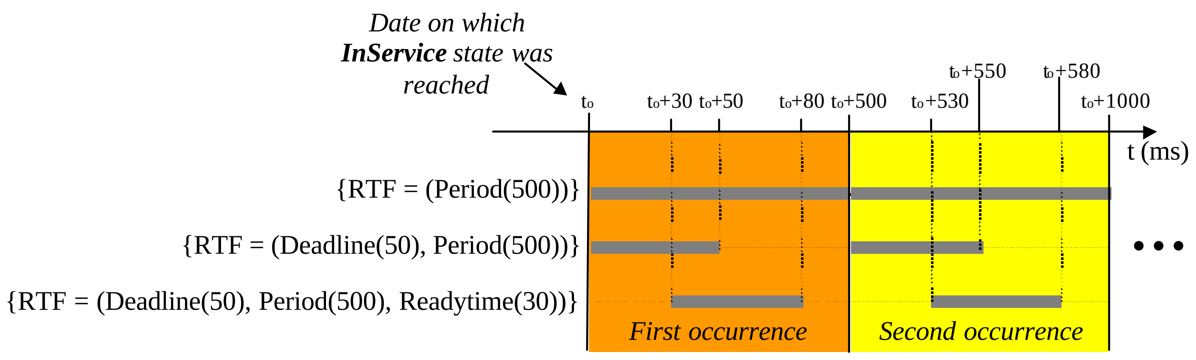


Figure 80: Specification of Deadlines and Initial Time-Delays.

Note that the preliminary conditions attached to a cyclic transition implicitly set a periodic treatment stop criterion. In the previous specification (*Figure 78*), the treatment is not re-initiated if an *OnOff_Reg* or *StopCar* signal is received by the object. Three other types of stop criteria can be attached to a cyclic treatment:

1. *a maximum number of occurrences.* For example, to limit cyclic treatment to a maximum of 1000 occurrences, the constraint is expressed in the following manner: *{ RTF = (DateRef, Period(500, ms), OccNb(1000)) }*

2. *an overall deadline.* For example, to limit cyclic treatment to a maximum of 10 seconds, the constraint is expressed in the following manner: *{ RTF = (DateRef, Period(500, ms), GlobDl(10)) }*

3. *a special hook function.* The approach enables interruption in the period of a cyclic treatment in dynamic mode. This means that, at the end of each occurrence in a periodic treatment, the user can specify a call to a specific user operation. This operation must have a boolean-type return value. If the result is true, periodic treatment continues, if not, it stops.

The specification of this method involves an additional *RTF* tagged value parameter, *continue(<userMethod>)*. The time constraint associated with the *acquireSpeed* service of the *Speed* class triggering automaton thus becomes: *{RTF=(DateRef, Deadline(100, ms), Period(500, ms), continue(afterAcquisition) }* where *afterAcquisition* is a *Speed* class operation with no parameter and a boolean return value.

All of the above then leads to the following modelling rule:

Modelling rule 31 : Time constraint modelling consists of adding a special tagged value, *RTF* (Real-Time Feature), in a *UML* constraint attached to an element of the application model. This tagged value can be attached to *Action*-type model elements, either explicitly by the user or implicitly by the development environment, for operation call or signal sending actions (*CallAction*, *SignalAction*). The tagged value may have several time parameters:

{RTF = (DateRef, Deadline(xxx, ms), Readytime(xxx, ms),
Period(xxx, ms), NbPeriod(xxx), continue(<useOperation>)}

where :

– *Deadline* is a parameter specifying the deadline for the request;
– *Readytime* specifies the starting date for the request;
– *Period* assigns a periodic constraint to a request;
– *NbPeriod* specifies the maximum number of periods for a periodic request;
– *continue(<userOperation>)* contains the specification of an operation, *<userOperation>*, which is invoked at the end of each treatment where so specified. By default, this parameter is NULL. Its prototype takes the boolean form *<userOperation>()*.

Remark: The parameters Deadline and *Readytime* are relative dates based on an absolute date, which is also contained in the message received by the called object. By default, this date corresponds to the date on which the message was created.

In some cases where the real time characteristic assigned to a treatment is a deadline, it may happen that a treatment is not fully completed when the time limit expires. To deal with this type of situation, one enables specification of a particular user treatment for timeouts. This is not achieve automatic treatment, such as task deletion, which would be difficult to obtain from outside the task context, while execution is underway. It is impossible, for example to know what resources are being used by that task and must therefore be freed or which attributes have been modified by it and need to be restored as they were prior to method execution. For this reason, one entrusts the user with the responsibility for specifying his own treatments in the event a deadline is overrun. To specify a special "timeout" treatment for a service, the user must add to the tagged value *{RTF}* previously attached to that service an ***abort(<userOperation>)*** parameter. The user method ***<userOperation>***, which calculates the abort criterion, has a boolean-type return parameter. If this parameter evaluates true, then treatment of the service underway must be aborted; otherwise it continues. Where the user does not provide timeout specifications for a request, the timeout will be treated by default.

Specification of Execution Times for Methods

For class methods, one introduces an additional tagged value enabling specification of an important time characteristic, $\textbf{WCET}_{\textbf{incr}}$, which serves to increase the WCET (**W**orst **C**ase **E**xecution **T**ime) [22] [23]. To specify this new concept, one proposes a tagged value ***{ WCET = param }***, in which ***param*** is the value of maximum execution time. The tagged value is added to methods specifications in the structural diagram for application classes (Figure 81). This time constraint is in fact a property of methods on which users cannot act. It is therefore attached to the object class that supports the operation. Specification of this time characteristic is useful for the validation of application scheduling properties.

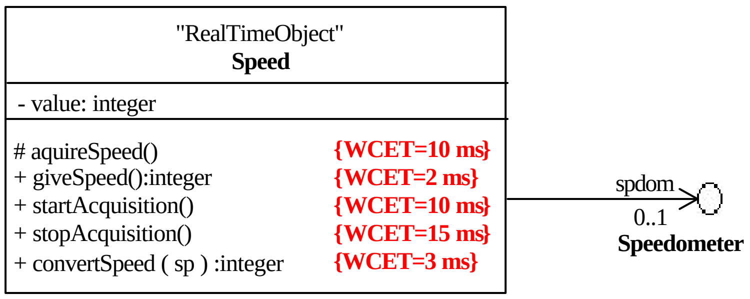

"RealTimeObject"
Speed
- value: integer
# aquireSpeed() {WCET=10 ms}
+ giveSpeed():integer {WCET=2 ms}
+ startAcquisition() {WCET=10 ms}
+ stopAcquisition() {WCET=15 ms}
+ convertSpeed (sp) :integer {WCET=3 ms}
spdom
0..1
Speedometer

Figure 81: WCET Specification for a Method.

2.2.3.6 Consistency rules

Previously, one defined a set of rules for maintaining consistency between the parameters and events received by an object and the parameters of operations triggered by such receipt. The following pages are devoted to the modelling rules required to ensure the consistency of transition specifications in behaviour automatons with various aspects of the structural model.

Protocol-transitions have the following syntax: **<triggerEvt> [‘guard’]**.

The trigger event required here is the *CallEvent* type and the operation attached to this event must belong to the interface of the class defining the protocol view context. If the specified transition causes a change in state, then the operation attached to the call cannot be specified as a read-type or parallel one.

Modelling rule 32 : Protocol-transitions causing a change in state must not be associated with an operation declared to be the read[25] or parallel[26] type in the structural specification of the class defining the automaton context.

According to subsection 0, the syntax of a triggering-transition is: **<triggerEvt> [‘guard’] / calledOpe(‘params’)**.

The lefthand portion of a triggering transition specification contains a single trigger event and the righthand portion a single action. The triggering view of an automaton allows for trigger events of the *ChangeEvent*, *TimeEvent*, *CompletionEvent* or *SignalEvent* kind. In the case of a *SignalEvent*, the type of signal linked to the triggering-transition trigger event must be included in the application's structural model as one of the signal types that the class is capable of receiving. The class must therefore specify a reception designating this type of signal.

Finally, if the source state of a triggering-transition differs from its target state, then the operation attached to the operation call action cannot be specified as "read" or "parallel".

Modelling rule 33 : If the trigger event of a triggering-transition is the *SignalEvent* type, then the class defining the automaton context of this view must include in its interface definition a reception[27] designating such a signal. In addition, the action specified on the righthand side of the transition must be the (*CallAction*) type and the operation associated with this action must be an operation of the class that defines the automaton context. Finally, if the transition causes a change in object state, the operation called during triggering-transition firing must not be declared as a read or parallel type.

A signal with parameters may have in its specification (section 2.2.2.4) a list of attributes defining the parameters it is capable of carrying. In cases where the signal associated with a signal event has attributes in its specification, the signal receiver has the right to use it. However, a same signal may be

---

[25] A read-type operation is one which has the tagged value *{concurrencyMode=reader}*.

[26] A parallel operation has the tagged value *{concurrencyMode=parallel}*.

[27] A reception is the specification, in the structural model of a class, that said class is capable of handling a given type of signal.

sent in different ways by different instances of different classes; and the only data available to the signal receiving instance is the signal specification. A receiving instance cannot, therefore, know if a signal was sent with or without parameters, even where its specification contains attributes. To solve this problem, it was decided to require all signal attributes to have a default value (Modelling rule 13 :). To comply with this rule, a received signal type event with parameters must always be accompanied by a list of parameters corresponding to the list of attributes specified for the received signal. This implies that the event generated by signal receipt will have as many parameters as the signal has attributes and the type of the i^th^ parameter will be identical to that of the i^th^ attribute of the classifier specifying the type of signal received, for an i value ranging from 1 to the number of signal parameters (Figure 82).

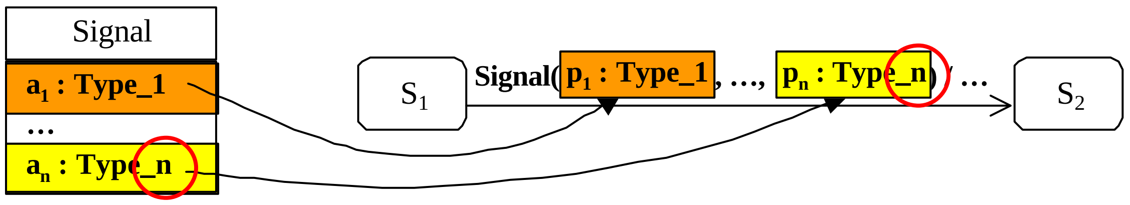


Figure 82: Relationship between a Signal Event and the Associated Signal Specification

Modelling rule 34 : The signature of a signal receipt type event specified on the left-hand side of a triggering-transition must be consistent with the classifier specification defining the signal associated with the signal event received. Therefore, considering that classifier attributes specifying the signal are contained in an ordered list $(a_i)_{i\in[1..n]}$ and that the parameter list for the event generated by received of this signal is $(p_j)_{j\in[1..m]}$, then:

– n=m ;
– $\forall$ i∈[1..n], a_i and p_i are of a same type and cardinality.

2.2.3.7 Summary

This chapter described various aspects of the method for building a behaviour submodel. In doing so, it provided answers to two important questions:

– How can *UML* state diagrams be used to describe the behaviour of real time objects without jeopardizing the object-oriented viewpoint inherent in this approach?
– How are real-time constraints included in the behaviour model?

State diagrams are used at two different levels of granularity to build the behaviour model of an application (Figure 83) :

– Class behaviour is described by a restricted and specialized form of state diagram. This specialization is essentially founded on the same principle as a *UML* protocol-type state machine.
– Operations behaviour (TBD).

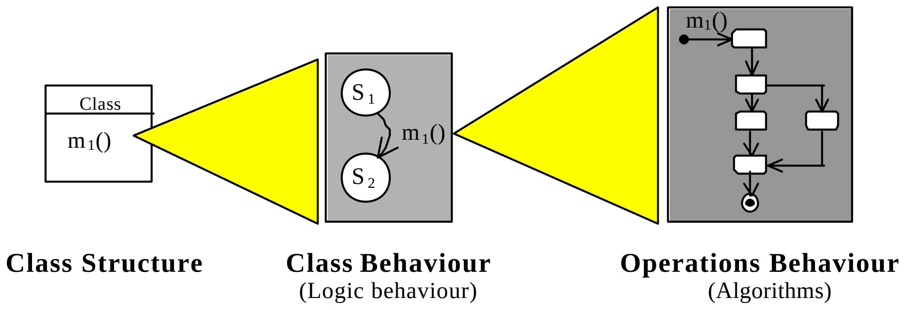


Figure 83: Use of UML *State Diagrams at Different Levels of Granularity.*

The section was therefore concerned with how to model the classes behaviour of an application. This is achieved by breakdown of the class behaviour into two automaton views, the protocol view and the triggering view. These views are special projections (or abstraction) of a same state machine describing the overall behaviour of a class:

- the protocol view contains a description of class response to operation call-type operations. It is also known as the object "life cycle";
- the triggering view focuses on the reactivity of class behaviour. It describes class response to signal sending-type messages, but also to automatic class behaviours or those deriving from a given boolean condition.

A set of rules was also provided for constructing these two views of object behaviour and for maintaining consistency between them as well as between the behaviour and the structural models.

The chapter likewise compares execution semantics for real time object-based behaviour models with those of standard *UML* models. More specifically, it explains how the basic "run-to-completion" assumption made for *UML* state machine semantics is supplemented, to enable correct, accurate use of *UML* protocol state machines in a real time object context.

Finally, the preceding pages also set forth modelling rules for specification of class operations and time constraints in the behaviour model.

The behaviour of a real time object class can be considered as a state machine having several orthogonal regions: A main region describing the global behaviour of the class can be observed from two different, complementary viewpoints and others specifying class operations behaviour. The result of this specification for the *Regulator* class of the example used in this document is given in Figure 84.

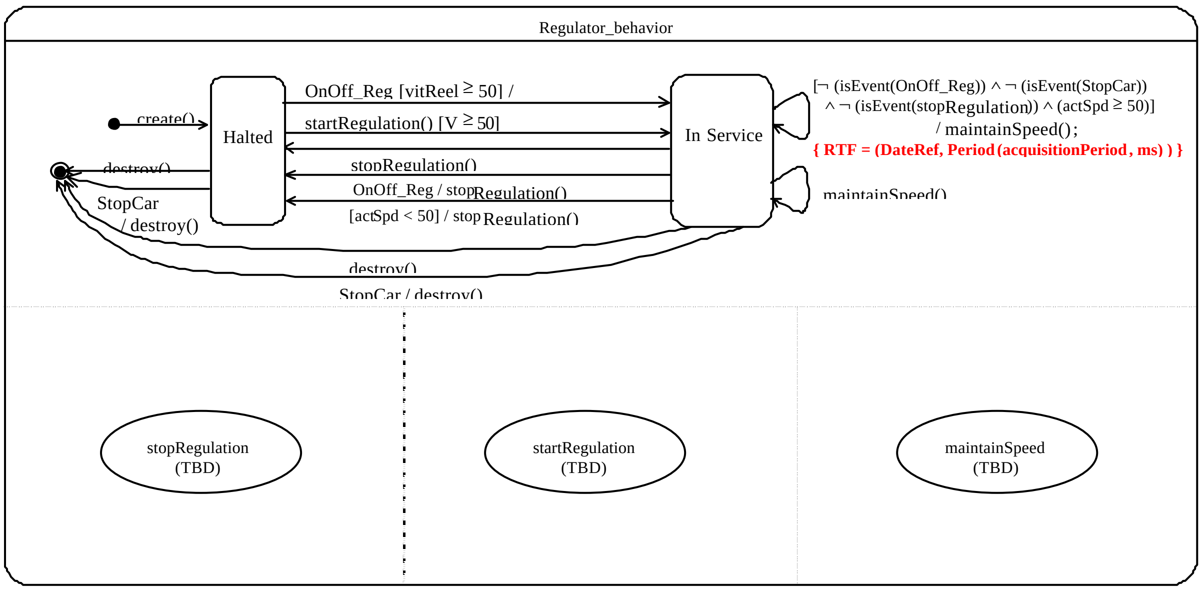

Regulator_behavior
create()
Halted
OnOff_Reg [vitReel ≥ 50] /
startRegulation() [V ≥ 50]
In Service
stopRegulation()
OnOff_Reg / stopRegulation()
[actSpd < 50] / stopRegulation()
destroy()
StopCar / destroy()
destroy()
StopCar / destroy()
[¬ (isEvent(OnOff_Reg)) ∧ ¬ (isEvent(StopCar)) ∧ ¬ (isEvent(stopRegulation)) ∧ (actSpd ≥ 50)] / maintainSpeed();
{ RTF = (DateRef, Period(acquisitionPeriod, ms)) }
maintainSpeed()
stopRegulation (TBD)
startRegulation (TBD)
maintainSpeed (TBD)

Figure 84: Complete Specification of Regulator Class Global Behaviour.

If *Regulator* class behaviour had been modelled using a standard object-oriented approach, the state machine specification would have been approximately as shown in Figure 85.

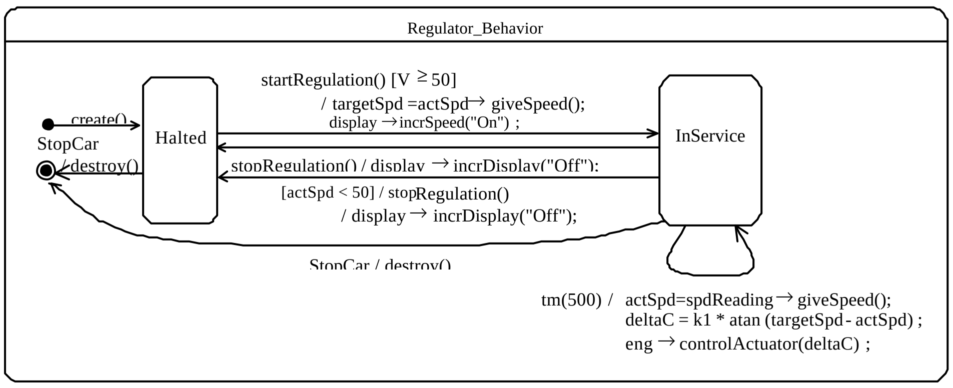

Regulator_Behavior
create()
Halted
startRegulation() [V ≥ 50] / targetSpd =actSpd→ giveSpeed(); display →incrSpeed("On") ;
InService
StopCar / destroy()
stopRegulation() / display → incrDisplay("Off");
[actSpd < 50] / stopRegulation() / display → incrDisplay("Off");
StopCar / destroy()
tm(500) / actSpd=spdReading→ giveSpeed(); deltaC = k1 * atan (targetSpd - actSpd) ; eng → controlActuator(deltaC) ;

Figure 85: Specification of Regulator Behaviour using a Standard Object-Oriented Approach.

Clearly, the main drawback of approaches that specify all object behaviour by means of a same, single model is a destructured object paradigm. In the state machine depicted in Figure 85, both *Regulator* control logic and its algorithmic specification are given in the same state diagram. Under these conditions, it is very difficult, for example, to take advantage of inheritance characteristics deriving from the object paradigm. In such cases, implementation of *Regulator* class operations (*startRegulation*, *stopRegulation*, etc.) is also often coded over several transitions. This characteristic then makes it difficult to reuse inherited operations, may generate excess activity and encourages users to construct highly complex state diagrams that quickly become hard to maintain.

The approach presented here to modelling class behaviour does not have these drawbacks for the object paradigm. In the first place, control logic and specification of algorithms are separate. It is thus easier, both to reuse classes and to ensure application maintenance. Secondly, because an operation is not implemented over the multiple transitions of a behaviour automaton, it is also easier to reuse inherited transitions in child classes. If, for example, the *Regulator* class is specialized to modify the speed control equation, since said equation is contained in the operation *maintainSpeed*, this operation can be simply overwritten to redefine its behaviour specification.

The same approach enables a client to reuse existing code outside this context, i.e. without referencing real time objects. A class that inherits operations from a class in another code can thus easily use these operations in a description of its own control logic. The difference between an inherited operation and one proper to the child class is that the implementation model for an inherited operation does not specify behaviour as an orthogonal region in the overall description of child class behaviour.

2.2.4 Activity – "describe interactions view"

In the various object-oriented approaches [5], [6], [24], sequence diagrams are generally used for the purpose of illustrating specific problems:

– Their first role is often to facilitate communication among the various partners in a project. All of these "players" (from the system developer to its user) are familiar with such diagrams, whose intuitive-type representation capability makes them a powerful communications tool. Sequence diagrams can thus alleviate modelling errors that may otherwise arise from misunderstandings among project participants, even at the predevelopment stage.
– In the course of application development, they also serve to model system interactions. As such, they are a practical, effective means for identifying object concepts (e.g. new classes or operations). Their level of granularity (or of detail) varies according to the phase in which they are implemented.
– Finally, once detailed system design has been determined, sequence diagrams can be used to illustrate significant aspects of application behaviour, thus providing a basis for subsequent testing and validation.

In all cases, these diagrams tend to act as informal specifications [25]; and it is not uncommon, at advanced stages in development, for information given in diagrams at a high level in this process (e.g. between preliminary and detailed analysis), to disappear. One therefore sees them as more formal, albeit incomplete, specifications, that define the behaviour required by the application. This means that a sequence diagram specified at the preliminary analysis stage will be reviewed, corrected and updated as part of an ongoing process of model refinement, until the system implementation model is finalized. The elements of a sequence diagram serve in fact as a basis of reference for elements identified in other facets of the modelling process. More specifically, they make use of object and message concepts that are closely linked to other important notions – i.e. class, operation or signal. This is why each of the elements identified in them must be consistent and in harmony with the specifications of other modelling diagrams (state and class diagrams).

The following paragraphs describes the different step that contribute to the "describe interactions view" activity. Indeed, as depicted in , this latter is split in three steps (Figure 86):

- The **"build scenario basis"** step aims at building the model basis for the interactions model. This one is then detailed/refined in the following step.
- The **"refine scenario"** step aims at detailing the application scenario. The output of this step is the work product **Detailed Scenarios Model**.
- The **"declare real-time property"** step consists in declaring real-time QoS on messages exchanged in sequence diagram to perform a given function.

- The **"organize scenarios"** step aims at giving an overview of the dependencies existing between the different scenarios identified in the model. For that purpose the analyst builds activity diagrams.

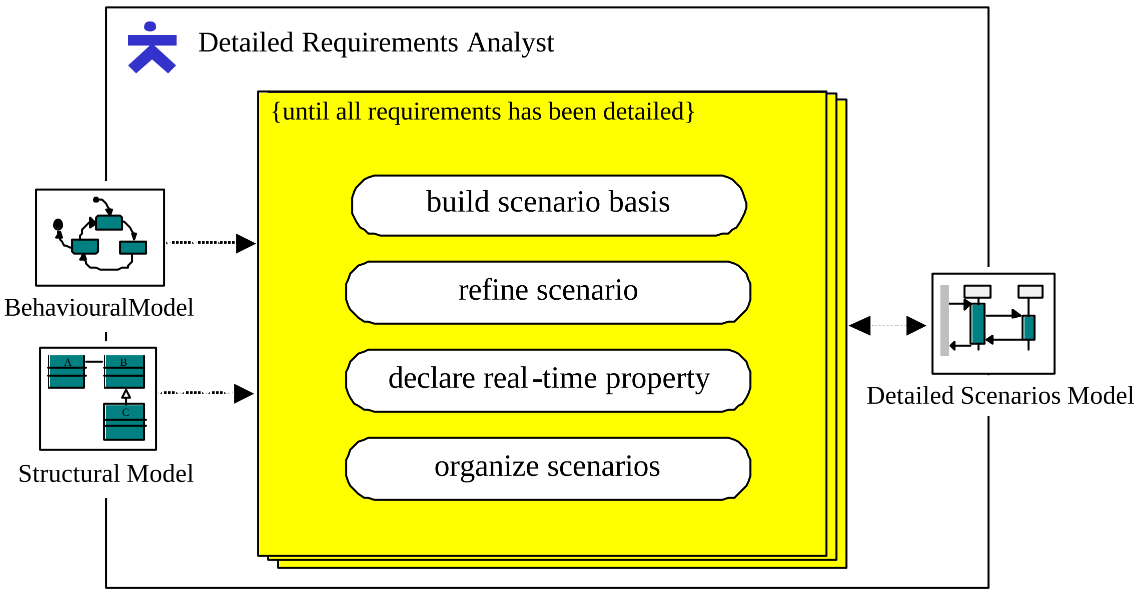


Figure 86: Activity – the "describe interactions view" Activity of the AIT-WOODDES Methodology.

The "Detailed Scenarios Model" work product is described through a set of two kind of diagrams. First of all, it is a set of sequence diagrams that depict a message sequence exchanged between instances of domain application classes in order to perform a given function (modelled by an interaction in the model). But the detailed scenarios model may also be composed by a set of specific activity diagrams construct during the "organize scenario" step.

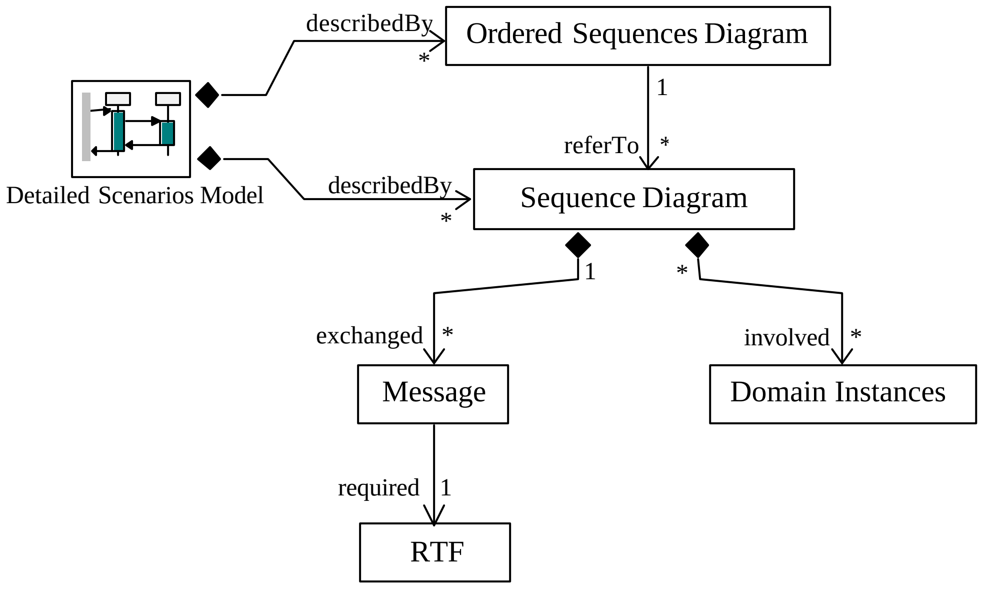


Figure 87: The "Detailed Scenarios Model" Work Product.

2.2.4.1 Step – "build scenario basis"

The purpose of this development step is to build the basis sequence diagrams that will then be detailed and refined iteratively in the step "refine scenario" until achievement of the "build detailed analysis model" activity.

The main point is that the interactions model …

2.2.4.2 Step – "refine scenario"

In order to well-suite the method, some sequence adaptation has been proposed. These one are presented in the following paragraphs, using examples of diagrams excerpted from speed regulator models. These illustrations serve as a pretext for closer study of sequence diagram refinement and the potential impact of the modifying them, on other (structural and behaviour) models.

Message specification in a *UML* sequence diagram must always explicitly include its sender and its receiver.

The proposed method calls for amending this rule in two types of situations:

– where specification of a particular message sequence is required in response to operation call-type events;

– for sequence diagram modelling of sending and receiving specialized signal-type communications.

A sequence diagram models the message sequences exchanged for the purpose of performing a given task. This means that, depending on the level of granularity or detail and the scope of the project, such a diagram may rapidly become too large and complex and thus involve large numbers of objects and/or messages. To limit such complexity and enhance readability, one enables factorisation and abstraction of certain parts of a sequence diagram, by allowing an object to receive an operation call without specification of the sender. In this case, only the event-aspect of the message is modelled, i.e. the interaction triggered by the call to a specific object.

Modelling rule 35 : In a sequence diagram, it is possible to refer to a more detailed description of a specific interaction message in another sequence diagram. To do so:

- the name of the message of interest is preceded by an asterisk written as a superscript;
- the message is linked to the new interaction (represented by the additional diagram) through an abstraction relationship[28] with a "***refine***" stereotype, oriented from the message to the interaction;
- the sequence diagram associated with the message refinement interaction, is assigned the same name as the message of interest, together with a postfix in the form of a number, if more than one case is possible. Additionally, where several refinements are available, the various abstraction relationships between the message and the interactions must be mutually exclusive.

[28] An abstraction (***Abstraction*** meta class) is a dependency relationship that relates a set of elements (at least two) representing the same concept at different levels of abstraction or from different viewpoints. The "***refine***" stereotype signfies that the target element or set of elements in the relationship is a refinement of the source element; or, conversely, that the source element is an abstraction of one or more target elements.

The following pages provide illustration of two situations to which this modelling rule applies. The first involves factorising a part of a sequence that repeats itself several times in a same diagram; and the second example demonstrates the usefulness of this rule for enriching part of the diagram without overloading it.

Figure 88 shows a sequence diagram for an interaction sequence with two identical parts.

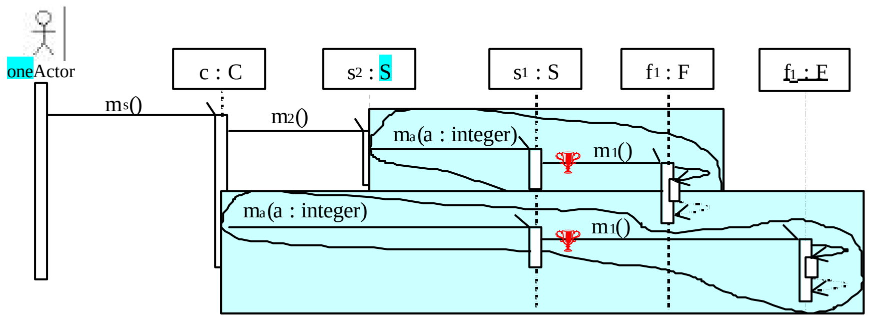


Figure 88: Simplification of a Sequence Diagram 1/3.

It is thus possible to factorise these two parts of sequence by cutting the sequence diagram in Figure 88 along the dotted lines marked with an icon in the form of a scissors (this results in the simpler sequence diagram shown in Figure 89).

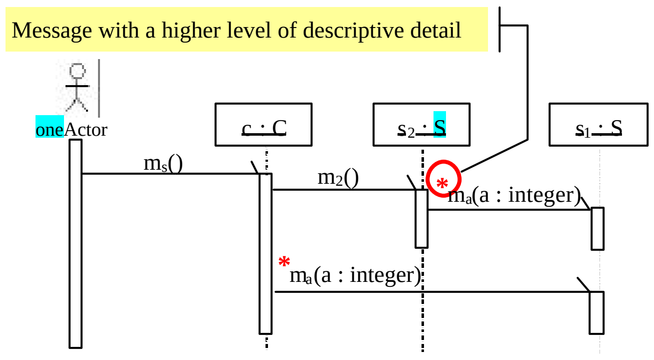


Figure 89: Simplification of a Sequence Diagram 2/3.

The next step is to add another sequence diagram (Figure 90) that details the effect of message *m_a()*. Message *m_a()* can also be considered as an abstraction of the interaction described by the sequence diagram in Figure 90.

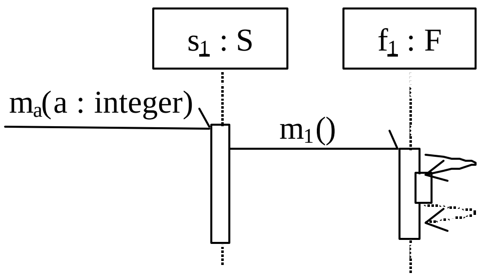


Figure 90: Simplification of a Sequence Diagram 3/3.

The notation used for *UML* sequence diagrams has been incremented for a second type of situation: a specific type of communication by signal, i.e. broadcasting. This specialized communication mode has the following main properties:

– it enables sending a signal without mandatory knowledge of its potential receivers;

– the signal can be received without mandatory knowledge of the sender by all of the instances of each class that have been declared receptive to this type of signal (see section 2.2.2.4).

To enhance representation of signal-type communication characteristics, one provides two new sequence diagram notation features. These features relate to modelling of signal sending and receiving. Sending is depicted by an arrow whose arrowhead points from the action generating the signal to an icon symbolizing a flash of lightning. Its label is the name of the type of signal sent (see Figure 91). It may also have a list of signal initialisation parameters.

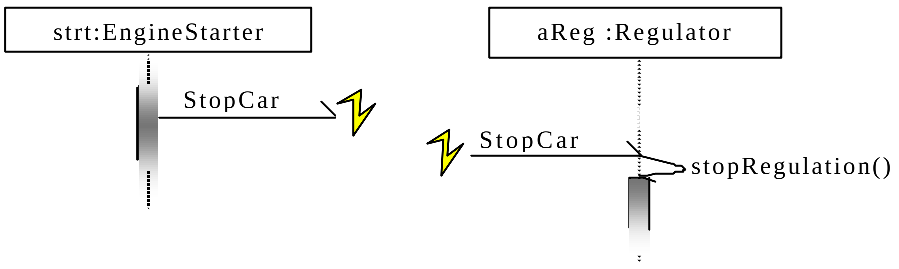


Figure 91: Depiction of Signal Sending and Receiving in a Sequence Diagram

Signal receipt is modelled by an arrow that points from the "flash" icon to the life line of the receiver object. It is also labelled with the type of received signal and a list of the parameters carried. In a sequence diagram, signal receipt may result in execution of a treatment by the explicit receiver object. In the example shown in Figure 91, receipt of the *StopCar* signal by the *aReg* instance of *Regulator* triggers execution of the *stopRegulation* operation.

2.2.4.3 step – “declare real-time property”

As already seen in previous chapters, in an *ACCORD/UML* real time object model, the notion of time is attached to the messages exchanged by objects and not to the objects themselves. In addition, messages can be divided into two main families: operation call-type messages and signal-type messages. Time constraints for a given message may then be specified in either of two ways:

1. if the message is associated with a signal, the real-time constraint is attached to either signal output or signal reception (Figure 92). In the first case, each treatment activated in the various objects receiving the signal is subject to the constraint associated with that signal; and each treatment involves the same constraint, as determined by the sender. In this mode, the client can impose his time constraints. In the second mode, where real-time constraint is a signal reception characteristic, each action triggered by the incoming signal is subject to the constraint associated with the receiving context. In this case, the constraints imposed on treatments are those set by the receivers.

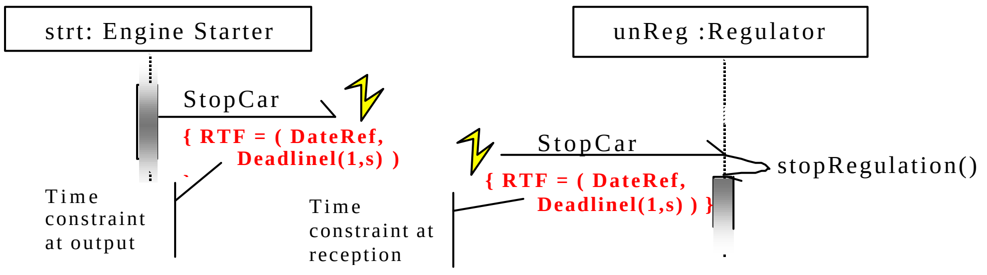

strt: Engine Starter
unReg :Regulator
StopCar
{ RTF = (DateRef, Deadlinel(1,s))
Time constraint at output
StopCar
{ RTF = (DateRef, Deadlinel(1,s)) }
Time constraint at reception
stopRegulation()

Figure 92: Time Constraints and Signals in a Sequence Diagram.

2. if the message is associated with an operation, the time constraint is again imposed by the message sender (Figure 93). To model this, the user adds a *UML* constraint containing an *RTF*-type tagged value with whatever the desired time constraint parameters.

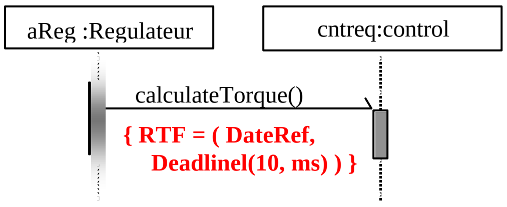


Figure 93: Time Constraint and Operation Call in a Sequence Diagram

The case of messages sent with a periodic time constraint requires special attention: under such conditions, it must be clear that the message, signal or operation call is not sent periodically by the sending object; instead, it is the treatment triggered in the receiver that takes place periodically (Figure 94).

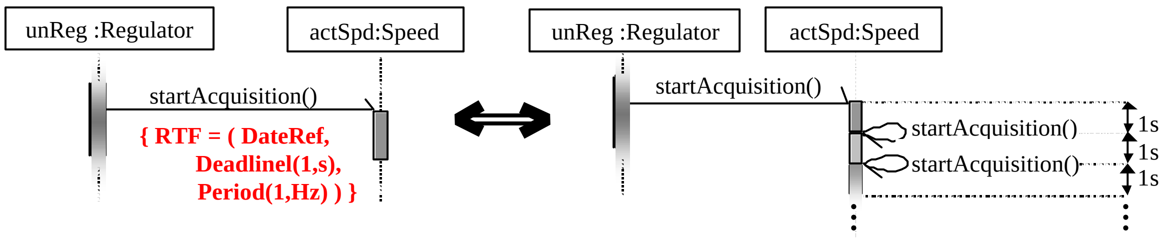


Figure 94: Message with Periodic Real-Time Constraint.

Remark: To model a time constraint in a sequence diagram, *UML* proposes message tagging and use of specific time functions (such, for example, as *receiveTime*, *sendTime*, etc.) to express message time characteristics for the constraints shown in brackets. One also allows this type of time specification, but, in order to maintain a uniform approach, introduces an equivalent, RTF-type time constraint.

2.2.4.4 step – "organize scenarios"

2.2.4.5 Consistency rules

As already seen above, the global model of an application is a consistent whole made up of three subcomponents: the structural model, the behaviour model and the interaction model. Note that the intersections of concepts manipulated in each of these subcomponents are never empty. The notion of operation, for example, is present in one form or another, in each of the submodels making up the global model. Since one of the main objectives of the *ACCORD/UML* approach is to obtain uniform, consistent models, it seems vital to provide a set of rules ensuring that these two requirements – consistency and uniformity – are met. The following paragraphs describe rules set for certain key situations where these two characteristics are important:

1. interactions between the names of objects manipulated in the diagrams and structural characteristics such as the names of association roles;
2. interactions between sequence diagram messages and operation and signal concepts as they are used in the class diagrams;
3. finally, interactions between specifications of messages received by an object in a sequence diagram and its own behaviour specification (via a state machine).

Interactions between Object Names and Association Role Names

Objects are named from the standpoint of the object using them. Their own names may thus differ from those given to the roles that (implicitly) represent the relevant structural model instances. An object name must, however, remain valid within the context of the user object (and not generate conflicts with the the latter's characteristics or the names of its attributes, roles or operations).

Modelling rule 36 : In a sequence diagram, the names of manipulated objects must not interfere with the name of an attribute or operation of the user object class (or with a name inherited from an object ancestor class).

Modelling rule 37 : In a sequence diagram, if the name of an object:

- is the name given to a user object class or to an ancestor class of the user object, then:
  - if the cardinality associated with a role is 1 or less, the used object corresponds to this role;
  - if not, for a link to exist between the object referenced by a role in the class diagram and the object used in the sequence diagram, the name given to the object in the sequence diagram must be the name of the role plus an index designating a specific object in set of objects then referenced by the class diagram role.
- is the name of a global application object[29], then:
  - the object used corresponds to this object;
  - otherwise, the object used is a local instance of the user object operation from which the sent messages originated. In the latter case, the instance used must carry the ***{transient}*** constraint.

[29] Although somewhat "out-of-phase" with object philosophy, it is possible, in *UML*, to create "global application objects", i.e. objects usable by all; thus this additional rule.

Another rule, intended to guarantee modularity and to comply with the philosophy of object-oriented methods, calls for an object using another object to always do so in compliance with the relationships specified in the application's structural model. In particular, in a sequence diagram, an instance is not authorized to send operation call-type messages to instances with which it has no relationships or, if it does, where such relationships are not oriented in the right direction.

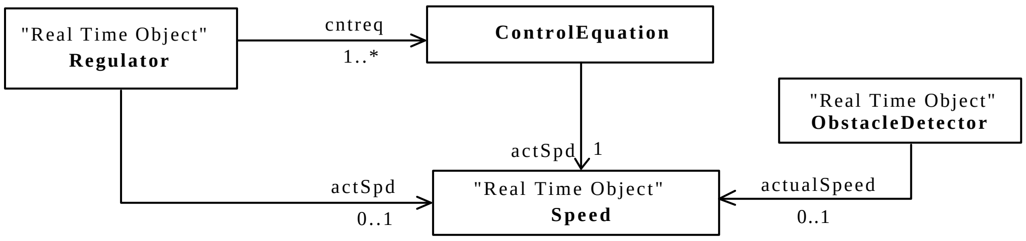


Figure 95: Example of a Structural Constraint for Sequence Diagram Expression.

With respect to the previous class diagram, the following sequence diagram (Figure 96) is valid. The *aReg* instance of *Regulator* is authorized to send a message to the *cntreq* instance of *ControlEquation*, since the *Regulator* class has an association relationship oriented toward the *ControlEquation*.

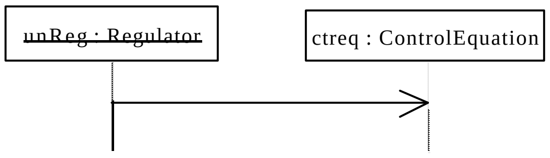


Figure 96: Valid Use of an Object by Another in a Scenario.

By contrast, the next two cases (Figure 97) are not valid for the previous structural model (Figure 95). In the lefthand diagram, the reason is the unsuitable orientation of the association, from *Regulator* to *ControlEquation* (it should have been the opposite). In the other case, lack of an association relationship between *Regulator* and *ObstacleDetector* invalidates the model.

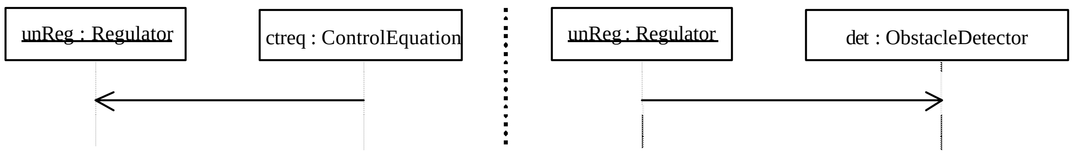


Figure 97: Two Invalid Uses of Objects in Interaction Scenarios.

Modelling rule 38 : In a sequence diagram, an object ***a*** (instance of class ***A***) can dispatch an operation call-type message ***m*** to an object ***b*** (instance of class ***B***) if:

– ***b*** is a local object whose action, performed by ***a***, generates the message. In this case, the ***{transient}***[30] constraint is attached to ***b***. This implies that all predecessors of ***m*** sent by the same action (or a parent action) must contain an instance creation-type message sent to ***A***. Instance ***b*** is implicitly destroyed when the action executed by ***a*** and generating the ***b*** creation action has completed. Moreover, class ***A*** must be connected to class ***B*** via a usage relationship denoted as

("***use***");

- ***b*** is a global application object. In this case, there is a sequence containing a global object ***b*** creation message and this sequence must have a priority constraint over the message ***m*** sending sequence;
- ***b*** is an attribute-link[31] of ***a***. In this case, ***b*** can receive only operation call-type messages originating from ***a*** ;
- ***b*** is an instance referenced by a link[32] connecting ***a*** and ***b***. In this case, classes ***A*** and ***B*** are connected by an association relationship oriented from ***A*** to ***B***, with a cardinality of at least 1. Moreover, to instantiate the association linking ***A*** to ***B*** (i.e. to create a link between ***a*** and ***b***), there must be an interaction containing a message sent by ***a***. This message must have a priority constraint over ***m***.

Interactions between Sequence Diagrams and Class Diagrams

It should first be emphasized that, to ensure global consistency with the *Real Time Object* approach, when a sequence diagram uses an asynchronous message sending mode, the receiving object must be one of the application's real time objects. Otherwise this type of communication cannot take place and makes the specification incompatible. To process an asynchronous message, at least one processing resource must be available independently from the sender. This means to be a Real Time Object. Thus the conclusion that only real time objects can receive signals. In the same way, only a real time object is capable of processing asynchronous operation calls.

Modelling rule 39 : In a sequence diagram, an object receiving an asynchronous message, whether an operation call or signal, must be a Real Time Object-type class.

A message can be said to have two viewpoints, that of the sender and that of the receiver. On the receiver side, the received message is perceived as an event, which may take the form of either a signal (*SignalEvent*) or an operation call (*CallEvent*).

A message in the form of a signal implies that the receiving object class previously declared its receptivity to this type of signal.

[30]This constraint means that the instance is created and destroyed in the course of the same action.

[31]An attribute-link, meta class ***AttributeLink***, represents a link oriented in the direction of the instance which holds the value of the object attribute.

[32]A link, meta class ***Link***, is an instance of an association between two classes. It represents a connection between two instances of the application classes linked by the association.

Modelling rule 40 : In a sequence diagram, if an object receives a message in the form of a signal, then the class of which it is an instance must have a reception associated with this type of signal.

A message in the form of an operation call implies that the object is capable of receiving this type of message. The class from which it was instantiated must therefore have previously declared itself capable of processing the message. To do so, the operation attached to the message must belong to the set of messages declared in the object class interface.

Modelling rule 41 : In a sequence diagram, an object receiving a message associated with an operation must be an instance of a class whose specification includes this operation with a visibility level compatible with the sender[33].

The diagram in Figure 14 illustrates the preceding rule.

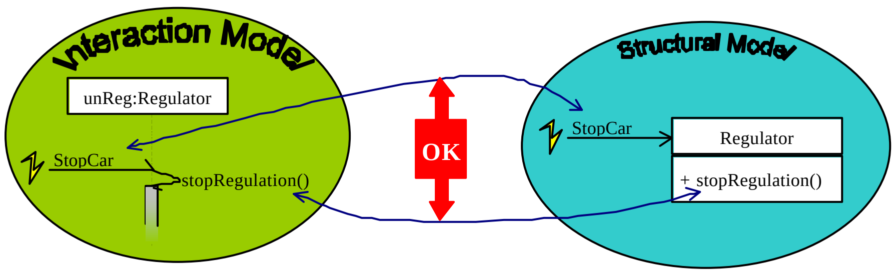


Figure 98: Consistency between Messages and Class Interfaces.

Interactions between Sequence Diagrams and Object Behaviour

As already seen above, the behaviour of an object is described by a state diagram. This diagram is made up of states and the transitions connecting them. The specification of a transition contains a transition trigger which may, among others, be an event characterized by receipt of either a signal or an operation call. While a sequence diagram essentially provides a global view of a system, it can also be considered from the standpoint of a specific object and, above all, of the messages it receives. These are perceived by the object as events. If a sequence diagram has more detailed specifications, such, for example, as states before and after a message-triggered event, the message received by an object must be consistent with the events that trigger the behaviour transitions of the class from which the object was instantiated. The following rule is intended to ensure such consistency:

[33] which may be:
public, protected or private, if the sender is an instance of the same type;
public or protected, if it is the descendant type;
otherwise public.

Modelling rule 42 : If, in a sequence diagram, the state of an object when it receives a message is specified, the state diagram describing that object's behaviour must also own the state corresponding to this condition. Moreover, this state must own at least one outgoing transition whose trigger is consistent with the message received by the object in the sequence diagram.

Moreover, if, in the sequence diagram, the treatment of a message owns specifications of the states before and after the event, then the state transition of the object processing the message must have <u>at least</u> one transition whose source state is the state prior to message receipt and whose target state is the post-message processing state. If there is also a guard condition for message receipt processing, then this condition must be compatible with one of the transitions complying with the previous modelling rule.

This rule ensures that the state diagram specifying the behaviour of a class will enable its future instances to perform their potential roles in the sequence diagrams in which they are present. The aim here is to ensure the consistency of object global behaviour as specified in its class state diagram with the specific behaviours required of it in the various sequence diagrams.

As shown in Figure 99, for *Regulator* class object *aReg*, the state of the object is specified for each treatment. This includes its state before receiving the message and that reached after message processing. To specify this characteristic, an additional notation point has been introduced into the *UML* sequence diagram. This includes the name of the pre- and post-treatment states in a textual note containing the name in brackets (*[state-name]*). As part of a current review of *UML*, the possibility of adding the notion of "state" to sequence diagram specifications and in message sequence charts (MSCs) is being examined [26]. One noteworthy proposal for extending the standard sequence diagram is Harel's "Live Sequence Chart" (LSC), [27].

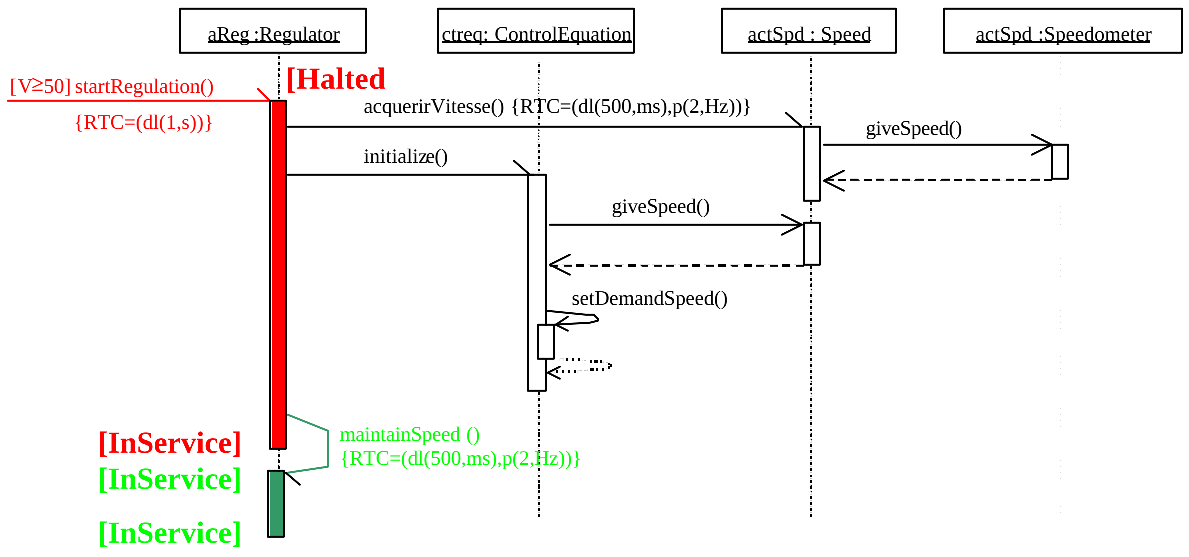


Figure 99: Sequence Diagram Describing Regulation System Startup.

In the *Halted* state, the *aReg* object starts periodic vehicle speed acquisition via the message *acquireSpeed* sent to the *carSpd* object with the following constraints: deadline ½ second and period ½ second. Then it initializes the *contreq* object and, once this has been achieved, launches the "maintain vehicle speed", also with a time constraint of ½ s deadline and ½ s period). At the end of this treatment, the object reaches *InService* state. It can thus be concluded that the *Regulator* behaviour diagram must have at least two states – *InService* and *Halted*. Additionally, the *Halted* state must have at least one outgoing transition whose trigger event is *startRegulation* and whose target state is *InService*. Because the *maintainSpeed* message is dispatched with a periodic time constraint, it is also periodically re-initiated in the *InService* state. The *InService* state must therefore have at least one outgoing transition whose trigger event is *maintainSpeed* and whose target state remains *InService*. The mention "at least one outgoing transition" refers to the fact that a same state can have several outgoing transitions with the same trigger event insofar as they have different guards enabling these transitions to be mutually exclusive. Thus, if the guard condition [V≥50] is added to treatment of the *startRegulation* message, the consistency relationship between the sequence diagram in Figure 99 and the *Regulator* class sequence diagram becomes the following: the *Regulator* class behaviour diagram must have at least the two states *InService* and *Halted*. In addition, the *Halted* state must have one outgoing transition that is triggered by receipt of the *startRegulation* event accepting the *[V³50]* condition for actual vehicle speed: (Figure 100).

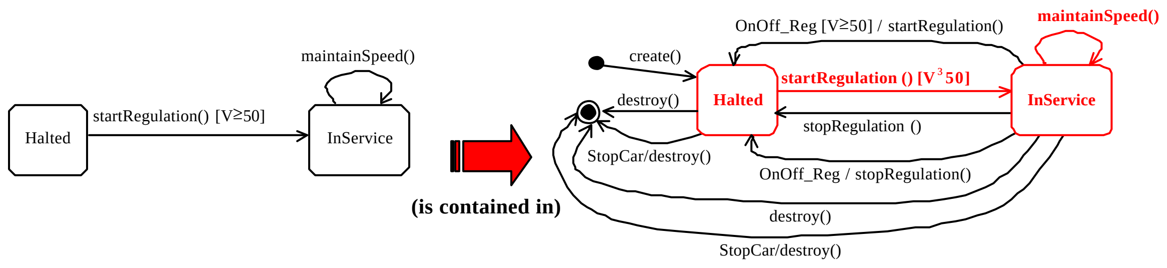


Figure 100: Sequence Diagram Constraints Transferred to the Regulator Global Behaviour Specification.

Interactions between Sequence Diagram and Operation Behaviour

It is possible to view sequence diagrams from yet a different perspective. Rather than seeing it as a set of instances cooperating with each other to perform a given task, the focus can be placed on a specific interacting object. The focus point then becomes the behaviour of a given object in realizing a particular sequence. Attention centers specifically on the sequence realized by the object following receipt of a message. This approach has two different objectives: to automatically generate part of class operation behaviour ([25]) or to verify that application behaviour enables a object to perform the roles assigned to it in the different sequence diagrams making up the interaction model.

Here, the principle is to isolate, for each object involved in the sequence diagram, each message received by that object, and to build a message sequence made up of messages sent by the object in response to trigger message receipt. The following rule can then be applied to this sequence:

Modelling rule 43 : For any sequence, if the trigger message is an operation call, specification of operation behaviour must contain a path including the message sequence initiated by message receipt in the sequence diagram. If the message is linked to a signal, message receipt is perceived as a signal-type event that triggers execution of an operation. The preceding case then likewise applies.

2.2.4.6 Summary

This chapter described various aspects of the method for building an interaction submodel. The submodel is based exclusively on *UML* sequence diagrams. Technical points highlighted in this chapter were the following:

– proposed additional sequence diagram notation for modelling of specialized signal-type communications and factorising some parts of a sequence diagram;

– real-time features description in the sequence diagrams using the tagged value ***RTF*** (Real Time Feature);

– consistency rules between interaction model sequence diagrams and structural model class diagrams;

– consistency rules between the sequence diagrams and behaviour model state diagrams.

3 PHASE – "BUILD DESIGN MODEL"

The goal of the design phase is to choose a single "optimal" solution for the system described in the analysis. [9] It identifies things such as concurrency models (which objects are active), scheduling policies, organization of software elements within deployable components, inter-processor communications, error-handling policies, etc.

3.1 Using UML for design

The following is based on two well-known processes:

1 - The Unified Software Development Process (RUP).

2 - Rapid Object-Oriented Process for Embedded Systems (ROPES).

RUP is probably the most well known process for developing o-o application using UML. It is a generic, well-organized[34] iterative process that relies heavily on UML semantics and notations. However, RUP does not address embedded\real-time issues at all.

ROPES is also an iterative process that uses UML, which is similar to RUP in many ways. However, the entire focus of this process is real-time embedded systems, which RUP misses for our point of view.

The specification of the design process is specified in the following order: artifacts, roles (workers), phases and activities.

3.1.1 Artifacts

The term artifact refers to concrete, tangible model entities that are being manipulated by role-players (or workers) during their workflows. An artifact may be an input or an output to a workflow.

Figure 101 and Figure 102 specify the design artifacts and their relations. This section defines the artifacts of the Design Model. The way to produce the artifacts and the role players that produce them are described in the subsequent two sections.

[34] By well organized we mean that it is consistently organized around roles\workers, activities and artifacts. Thus complies with OMG "Unified Process Model (UPM)" that defines the same structure for all processes.

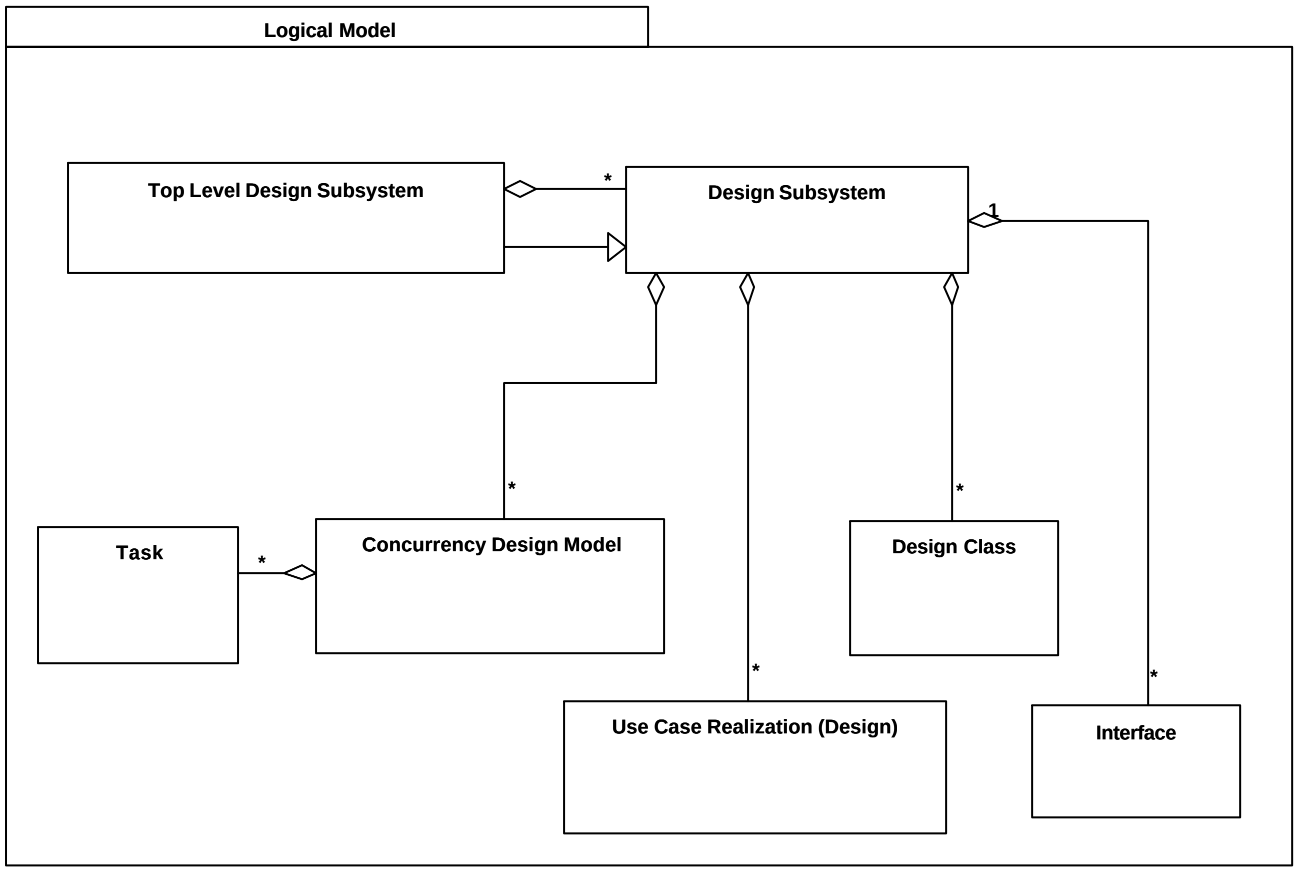


Figure 101: Design Model Artifacts – Logical Model. The Logical Model is divided to Design Subsystems. Each subsystem, realizes a design use-case using design classes, interfaces and tasks. The various tasks and the strategies to manage them are described in the Concurrency Design Model. In essence, a Concurrency Design Model may be specific to a subsystem, though usually the general strategies are described in the Top-Level Design Subsystem.

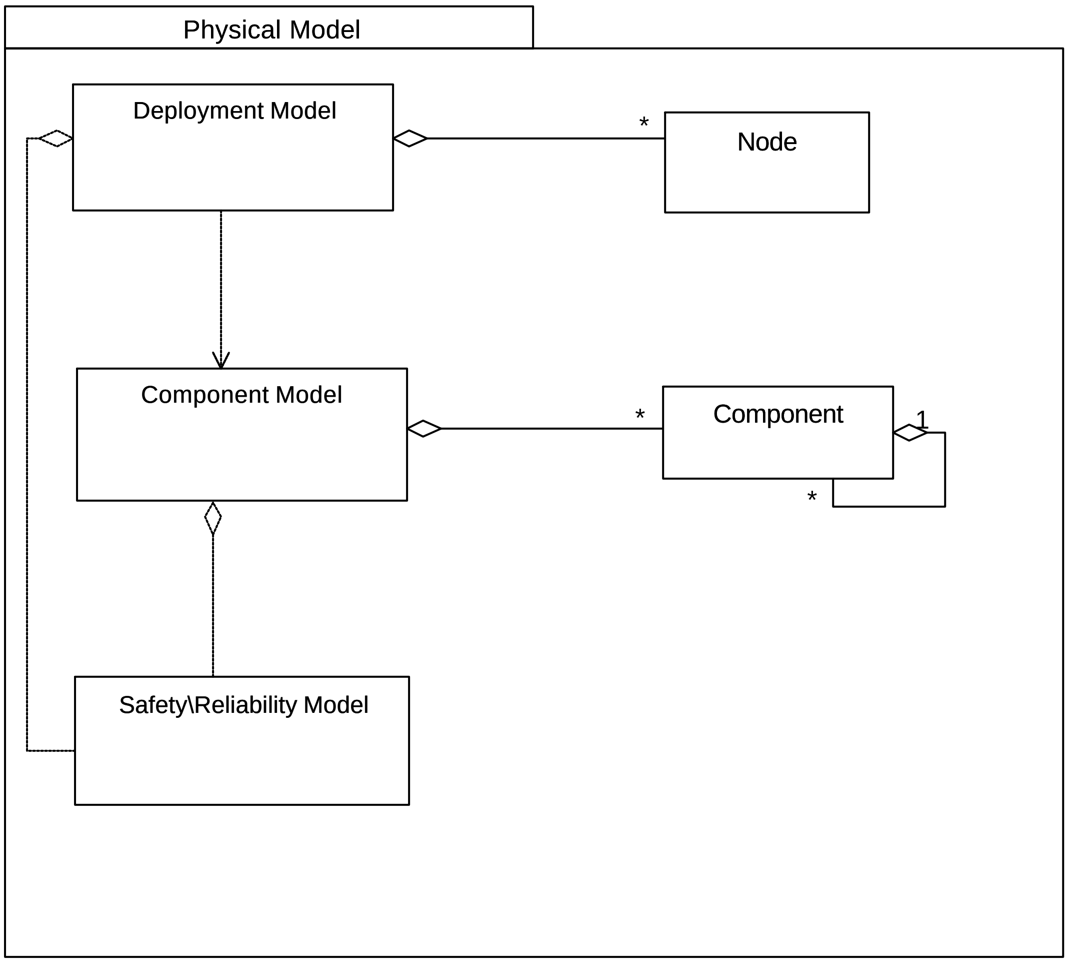


Figure 102 Design Model Artifacts - Physical Model. The Physical Model deals with the physical and the runtime aspects of the Design Model by describing the physical architecture of the system. It contains a Deployment Model that describes the way the physical software components are distributed over the Nodes. A Component Model, that describes how the physical components are arranged and a Safety/Reliability Model that aims to ensure safety and reliability at the physical architectural level.

3.1.1.1 Artifact: Design Model

The **Design Model** is an abstraction of the software solution to the problem shown in the Analysis Model. It consists of a Logical Model and a Physical Model (see below).

3.1.1.2 Artifact: Logical Model

The **Logical Model** describes the realization of use-cases, focusing on how functional and non-function requirements, as well as other non-physical constraints, impact the system under consideration.

3.1.1.3 Artifact: Design Subsystem

The **Design Subsystem** artifact enables organizational division of the Logical Model into more manageable pieces. Every Design Subsystem may be further divided into more Design Subsystems. Design Subsystems consist of artifacts such as Design Classes, Interfaces, Concurrency Model, etc. as shown in Figure 101.

A Design Subsystem is characterized by the following:

- It must be *cohesive,* i.e. its contents should be strongly related.
- It must be *loosely coupled,* i.e. the dependencies on the other subsystems should be as minimal as possible.
- It must be *self-contained,* i.e. it should represent a set design concerns that can be addressed separately and possibly concurrently by different teams.
- The top layers subsystems should have straightforward tractability into the analysis model.
- Design Subsystems may realize interfaces, thus enabling its use by other subsystems in a “black box” manner.
- It may wrap a legacy or other reusable software product and provide interfaces to it.

3.1.1.4 Artifact: Top Level Design Subsystem

Top Level Design Subsystem is the top most Design Subsystem. A Design Model contains one and only one of these artifacts. The reason it is distinguished from the other Design Subsystem since is that it encapsulates the entire logical design.

3.1.1.5 Artifact: Design Class

A **Design Class** is a seamless abstraction of a class or a similar construct in the system implementation. The abstraction is seamless in the following sense:

- The language used to specify a design class is the same as the programming language. Consequently, operations, attributes (along with their visibility), types, arguments, etc. are specified in the language syntax.
- The relationships in which a design class is involved often have straightforward meaning to the class’s implementation. For example, associations may be mapped to class attributes.

- The methods (that is, the realization of operations) have straightforward mappings to the corresponding methods in the implementation of the class. This means that methods are specified using the programming language syntax.
- A Design Class may postpone handling of subsequent implementation activities by noting them as *implementation requirements.*
- A Design Class may be given a stereotype that is seamlessly mapped to a construct in the programming language. For example, an <<exception>> in a Java class or a <<form>> in a Visual Basic application.
- A Design Class may realize and thus provide interfaces it makes sense to do so in the programming language. For example a Java class may provide one or more interfaces.
- A Design Class may be denoted as <<active>> implying that all of the objects of that class are active. The detailed semantics of this is programming language and used technology dependent.

The Design Class relations can be described using class diagrams. A statechart diagram or an activity diagram may describe the class's behavior. The role it plays in various scenarios can be described in a sequence diagram or a collaboration diagram. All above diagrams have direct impact on the Design Class artifact.

3.1.1.6 Artifact: Interface

An **Interface** describes a set of operations provided by a design class of a design subsystem. A Design Subsystem that realizes an interface must contain Design Classes or Design Subsystems (recursively) that provide this interface. A Design Class that realizes an Interface must provide methods that realize the operations of the interface or alternatively must be denoted as <<abstract>>.

Class diagrams, sequence diagrams and collaboration diagrams may be used to find and specify interfaces.

3.1.1.7 Artifact: Use Case Realization (Design)

A **Use Case Realization (Design)** is a collaboration within the Design Model that describes how a specific use case is realized, and performed, in terms of design classes and their objects (ref … UP). This artifact must have straightforward mappings to the use case realization specified in the Analysis Model and consequently to the Use Case model provided in the Analysis. As opposed to the Use Case Realization in the Analysis Model, here most non-functional requirements such as time constraints or other domain specific requirements are captured.

A Use Case Realization (Design) is described by at least one of the following:

- A textual flow-of-events description
- A Class Diagram depicting the participating design classes.
- Interaction Diagrams (Sequence Diagram and\or Collaboration Diagram) demonstrating particular flows or scenarios of the use case in terms of the interacting design objects and\or design subsystems.

Class Diagrams, Sequence Diagram, Collaboration Diagrams and Activity Diagrams may be used to describe this artifact.

3.1.1.8 Artifact: Concurrency Design Model

The **Concurrency Design Model** specifies the different independent tasks\threads in the design, how objects are packaged into these tasks and the strategies of synchronizing and managing these tasks.

The Concurrency Design Model may be composed of the following UML diagrams:

- Class and Object Diagrams that focus on the way the Active Classes and Active Objects are organized. In UML a Task is modeled by an Active Object, which controls a 'thread of execution'. Usually during construction these objects allocate an OS thread specifying all its attributes such as scheduling policy and priority. Typically, (but no mandatory) Active Classes are composite classes tightly aggregating component objects, which are accessed by the aggregating Active Class. (figure … is an example of such diagram)
- Sequence and Collaboration Diagrams where the messages are differentiated by colors or line styles, where each color or line style is associated to a different task. A more stand UML approach to achieve the same thing would be to prefix each method name with the task name. (figures … and … demonstrate such diagrams).
- Statecharts may used after Rambaugh's approach (ref Rambaugh J. et. al. Object-Oriented Modeling and Design, Englewood Cliffs, N. J.: Prentice Hall, 1991) which notes that concurrency by objects generally arises by aggregation – that is, a composite object is composed of component objects, some of which are active. In this approach, a single state of the composite object may be composed of multiple states of its active component. The procedure where the composite object accepts events and dispatches them to its aggregate active object may be modeled by a statemachine. Because every active aggregate a task it is natural to denote this as an orthogonal region with the composite object's state[35]. The use of these statecharts are most applicable where the object has TRUE states which wait for asynchronous signals and when orthogonal regions come and go. (See figure … for an example)
- Activity Diagrams may be used in a similar way to Statecharts. Here, instead of orthogonal regions, forks and joins separated by dashed 'swim lanes' are used. (see figure … for an example).

[35] It is important to note that usually separate threads do NOT implement orthogonal regions. The common implementation is that the same threads executes the entire statemachine.

3.1.1.9 Artifact: Task

A **Task** is a flow of execution within the application. In non real-time environments it is refereed to a **Thread**. For further details see Concurrency Design Model.

3.1.1.10 Artifact: Physical Model

The **Physical Model** describes with the physical runtime implementation of the system under consideration. The Physical Model consists of a Deployment Model a Component Model and a Safety/Reliability Model.

3.1.1.11 Artifact: Component Model

The **Component Model** describes the packaging of generated object artifacts that exist in runtime into libraries, executables, databases, etc.

The term *component* applies to a physical software artifact that exist in runtime. It may be an executable, a library, a database table, a DLL, etc. Like classes, components may realize interfaces so when one component is calling another it must conform to that interface. Also like classes, components may have instances so in some sense components are *types* of instances that exist at particular memory addresses, processors or disks. It is not uncommon to have several replicates of a component all over the system, for example a TCP/IP implementation may be present on every node.

Components are also binary-replaceable things (as opposed to classes). This means that a new version of a component may replace a previous version without recompiling the other components, as long as the new version meets the same interfaces.

In the Physical Model, the Component Model is specified by a set of component diagrams, which specify which components are present, of what kind (executable, DLL, COM-Server, etc.), what interfaces they implement and how they are organized. Reference 1 (M1) describes these diagrams.

3.1.1.12 Artifact: Component

A physical, replaceable part of a system that packages implementation and provides the realization of a set of interfaces. (UML Standard). For further detail see Component Model.

3.1.1.13 Artifact: Deployment Model

The **Deployment Model** specifies strategies for the distribution of runtime components onto processors and devices, as well as management of collaboration among distributed object (inter-thread and inter-process communications). This artifact is the one that is most affected by the hardware design, i.e. the electronic devices that execute the software (referred to as Nodes) and physical communication media that links them together.

The software concerns for each node (or processor units) are:

- Envisioned purpose and scope of the software executing on the processor.
- Computational power of the processing unit.

- Availability of development tools, such as compilers for the selected language, debuggers and in-circuit emulators.
- Availability of third-party components, including operating systems, container libraries, user-interfaces, etc.
- Previous experience and user expertise with the processor.

Deployment Models consist of Deployment Diagrams, which specify the nodes and their interrelations and the runtime components they contain. Also, sometimes it is useful to specify tasks, subsystems and other non-physical software that are implemented on the node, especially in systems that are not designed in a component based manner.

3.1.1.14 Artifact: Node

A classifier that represents a runtime computational resource which generally has at least memory and often processing capability. (UML standard). For further details see Deployment Model.

3.1.1.15 Artifact: Safety/Reliability Model

The **Safety/Reliability Model** aims to ensure the system will meet its reliability and safety requirements by arranging and/or adding components within the Deployment and Component Models. It specifically addresses strategies for:

a. Global error handling.
b. Safety processing.
c. Fault tolerance.

By safe we mean that the system does not creates accidents leading to injuries, loss of life and damage to property. By reliable we mean a system that performs correctly for a long periods of time.

Usually the Safety/Reliability model realizes well-known safety designs patterns to achieve its goal. For example: Watchdog Pattern, Homogeneous Redundancy Pattern, etc.

3.1.1.16 Artifact: Architecture Description

The **Architecture Description** is a document referring to all significant artifacts in the model from an architectural point of view. It provides a high level overview of the architecture.

3.1.2 Roles\Workers

A Role denotes one of several roles that may be played by an individual (or a small group of individuals) in the process.

3.1.2.1 Role: Architect

The architect is responsible for the integrity of the design, ensuring that models as a whole are correct, consistent and readable. In addition, the architect is responsible for the Logical Model, the Physical Model and the Architecture Description.

The architect is not responsible for the continuous development and maintenance of the artifacts in the models. This is the responsibility of the Use-Case and the Component Engineers.

3.1.2.2 Role: Use-Case Engineer

The use-case engineer is responsible for the integrity of one or several use-case realization (design), ensuring they fulfill the requirements made on them. The realization of the design use-case must correctly realize the analysis use-cases as well as to be consistent with the analysis use-case model.

The use-case engineer is not responsible for the design-classes, interfaces, subsystems and relationships employed in the use-case realization.

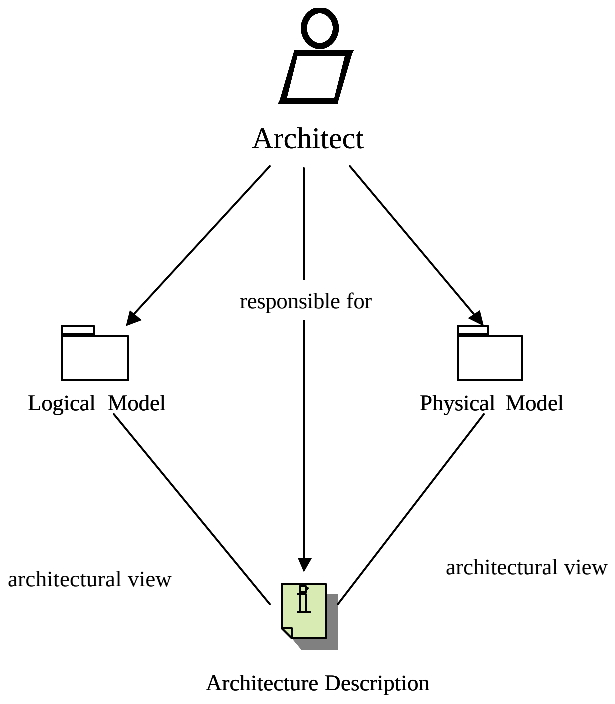


Figure 103:Architect Responsibilities.

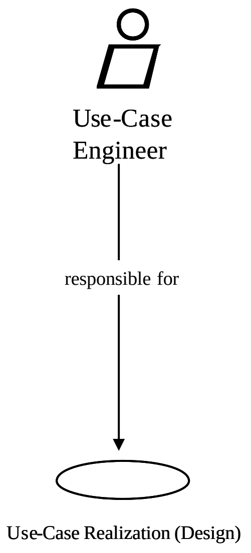


Figure 104: Use-Case Engineer Responsibilities.

3.1.2.3 Role: Component Engineer

The component engineer defines and maintains the operations, methods, attributes, relationships and implementation requirements of one or several design classes, making sure that each design class fulfills the requirements made on it from the use-case realizations in which it participates.

The component engineer may also maintain the integrity of one or more subsystems, making sure their contents are correct, their dependency on other subsystems is minimal and that they realize the interface they provide.

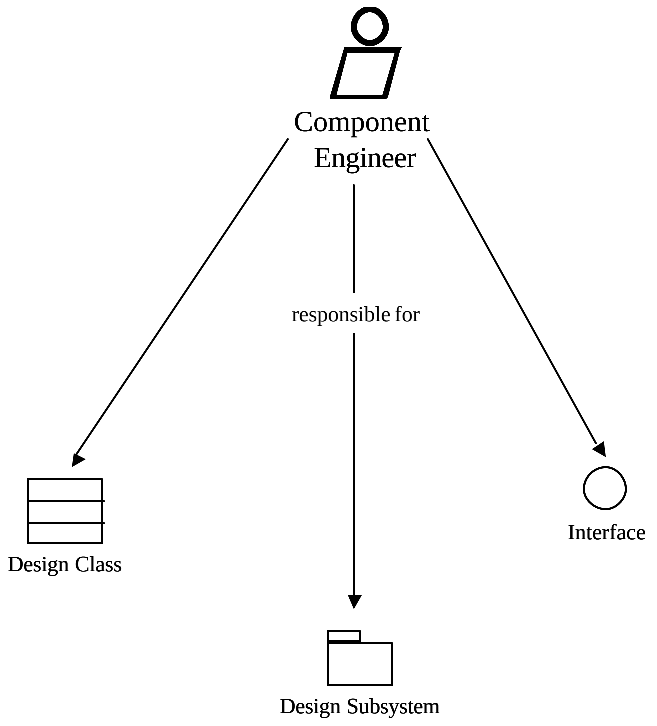


Figure 105:Component Engineer Responsibilities.

3.1.3 Phases and Activities

The design process has three phases as shown in **Erreur ! Source du renvoi introuvable.**.

The process starts with an Architectural Design performed by the architects. Then the use-case engineers perform a Mechanistic Design realizing each design use case in terms of the participating design classes, interfaces, subsystems, tasks, etc. while, the Component Engineers start to perform detail design of the design classes which serve as input back to the Mechanistic Design activity and vice versa. Finally the design of the entire subsystem is concluded.

Throughout the design, the developers identify new candidates for new classes, interfaces, components, etc. as the Design Model evolves.

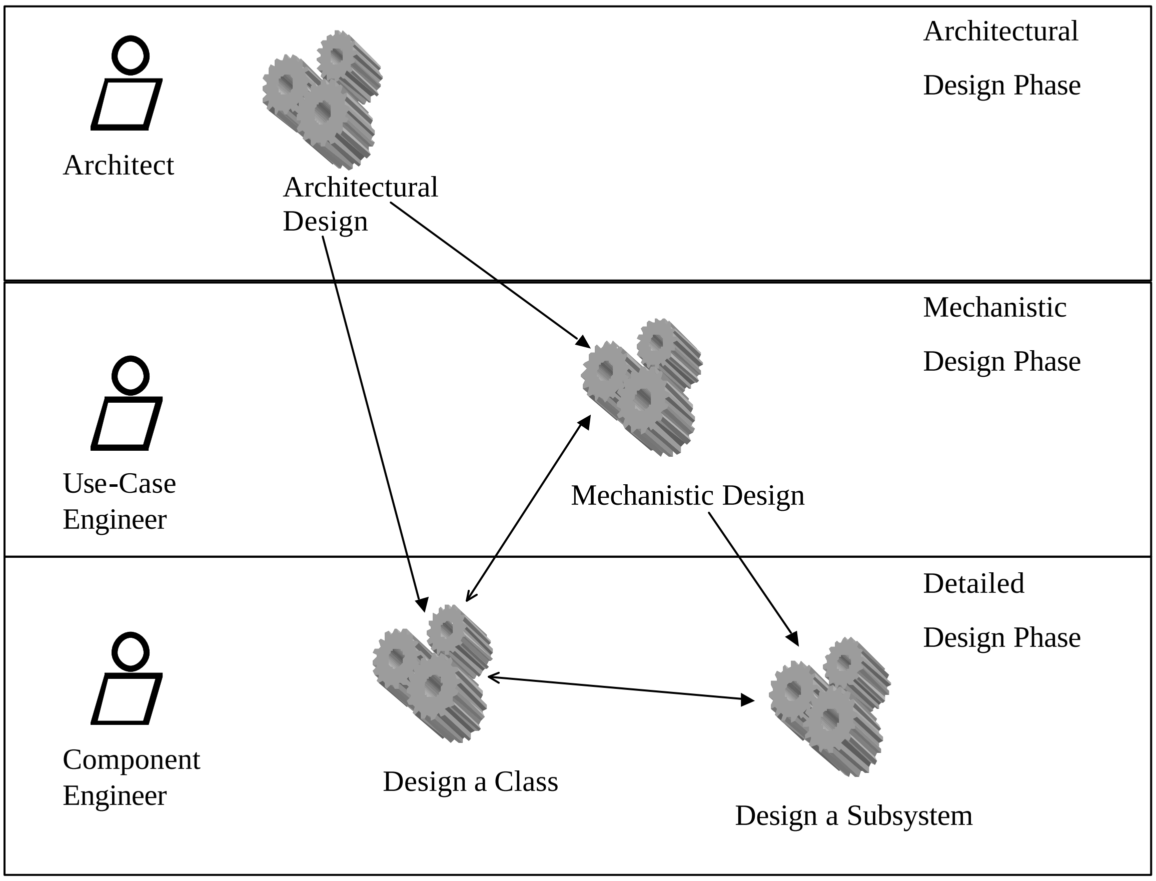


Figure 106: Design phases and activities.

3.1.3.1 Phase: Architectural Design

The purpose of architectural design is to outline the Logical and Physical Models of the design by identifying the following:

- Nodes and their network configurations.
- Design Subsystems and their interfaces.
- Architecturally significant Design Classes.
- Major Tasks \ Active Classes.
- Runtime components and their distribution.

Generic design mechanism including Safety\Reliability mechanisms, Task synchronization strategies, node communications strategies, etc.

The architect has to balance various quality factors such as efficiency, extendibility, reusability, portability, etc. throughout this activity in order to come up with the optimal architecture that will serve as a platform for a high quality design.

Figure 107 shows the input and results of this activity.

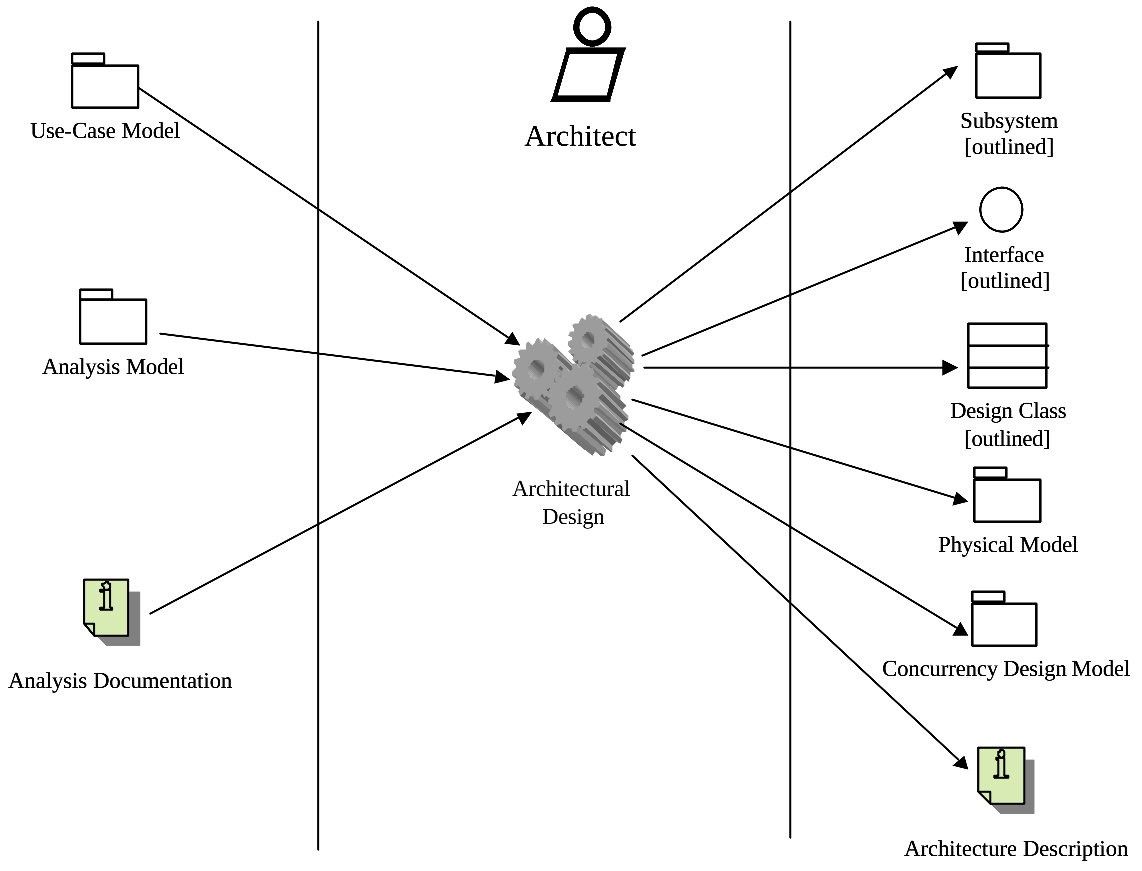


Figure 107: the "architectural design" activities of the AIT-WOODDES Methodology.

The activities of architectural design are:

1 - Identifying Nodes and Network Configurations

2 - Identifying Subsystems and their Interfaces

d. Identifying Application Subsystems
e. Identifying Middleware and System-Software Subsystems
f. Defining Subsystem Dependencies
g. Identifying Subsystem Interfaces

3 - Identifying Design Classes from Analysis Classes

4 - Identifying Tasks and Active Classes

5 - Identifying Components and their Interfaces

a. Identifying Reusable Components
b. Specifying Component Organization
c. Identifying Component Interfaces

As in all phases, the order of activities is optional, and it is not uncommon to go back to previous activities and update their artifacts as a result of a subsequent artifact.

Activity: Identifying Nodes and Network Configurations

Some systems are implemented over several nodes. In this case, the ways the various components are deployed over the network, the role of the nodes, the communication protocols, etc. have a major impact on the software architecture. For example: the decision to implement a system using a client-server architecture and choosing CORBA as the distribution technology may impact all subsequent phases of the architectural design.

Aspects of network configuration include:

- Which nodes are involved, and what are their capacities in terms of processing power and memory size?
- What types of connections are between the nodes, and what communication protocols will be used over them?
- What are the characteristics of the connections and communication protocols, such as availability, bandwidth and quality?
- Is there any need for redundant processing capacity, fail-over modes, process migration, keeping backup of data, etc. (In other words what safety patterns are to be used in the safety/reliability model across the network) ?

[Test Case Speed Regulator]

Figure 108 shows a possible deployment diagram for the speed regulator system[36]. The RegulatorCtrl is the “brain” \ “heart” of the system. It performs all feedback, control and monitor activities. The EngineAdaptorUnit and SpeedometerAdaptoUnit are hardware adaptors that mediate between specific kinds of engines and speedometers to the RegulatorCtrl. This way the RegulatorCtrl hardware\software unit can be reused over a wide range of cars. The RegulatorInterface serves as the front end of the speed regulator: using it, the user can turn on and off the feedback loop; see the regulated speed, and more. In addition, this node is also connected to the brake and accelerator pedals, as they are in effect part of the “User Interface” of the speed regulator. The brake and accelerator pedals are not shown as nodes on the diagram, since these are not considered as processing units.

The physical protocols and connections used between the nodes are specified in the diagram as labels on the communication paths. The speed regulator uses a proprietary protocol over these connections. In case of any failure in communications between the nodes, an error LED is lit on the RegulatorInterface and the feedback loop is turned off.

[36] The test case should only be related to as an example. It is not suggested that real-life speed regulators are implemented using distributed software elements or even using connections such as RS232.

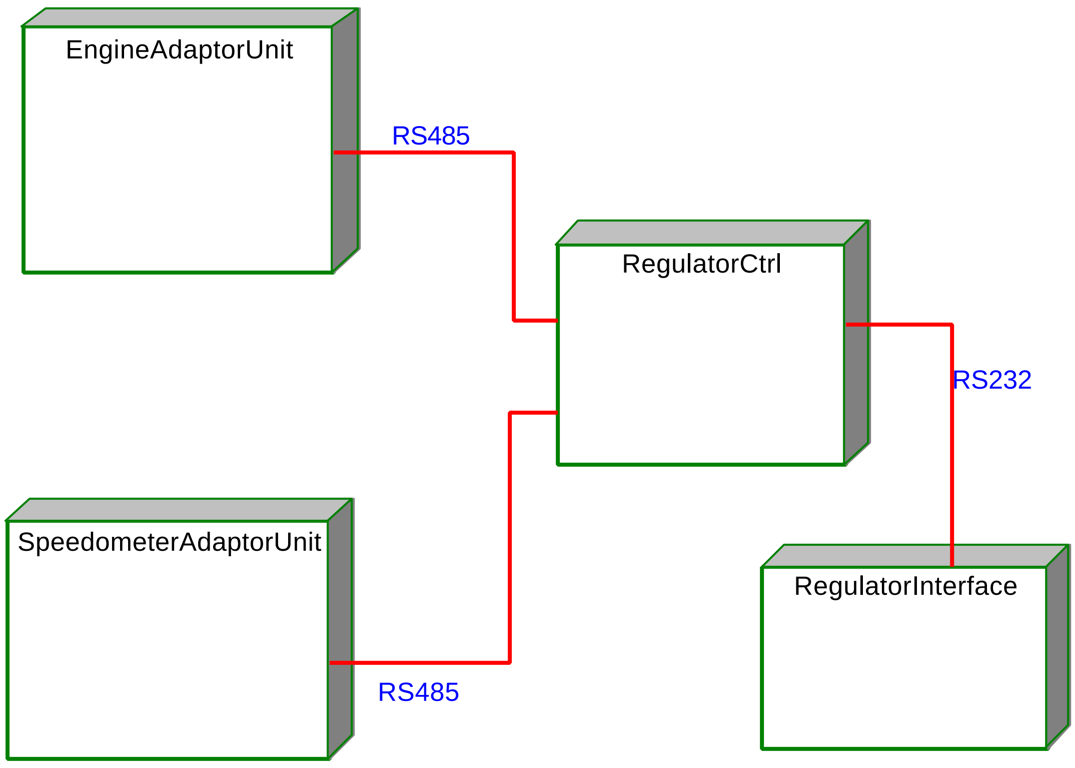


Figure 108: Deployment Diagram for the Speed Regulator system.

Activity: Identifying Subsystems and their interfaces

Subsystems can either be divided initially as a way to divide design work or found as the design model grows and needs to be decomposed. In addition, some subsystems may not be developed as part of the project, i.e. some subsystems may be reused from other projects or be subcontracted.

According to the approach presented here (based on the Unified Process) the subsystems may be arranged in four layers, as seen in Figure 109. The Application specific layer holds subsystems and packages that are not used by other packages and subsystems. The Application general layer contains application logic that is being reused by other packages of the application. The two bottom layers, middleware and system software, represent subsystems and packages that can be reused by other applications as well. These may be developed as part of the project or reused from standard or legacy software systems.

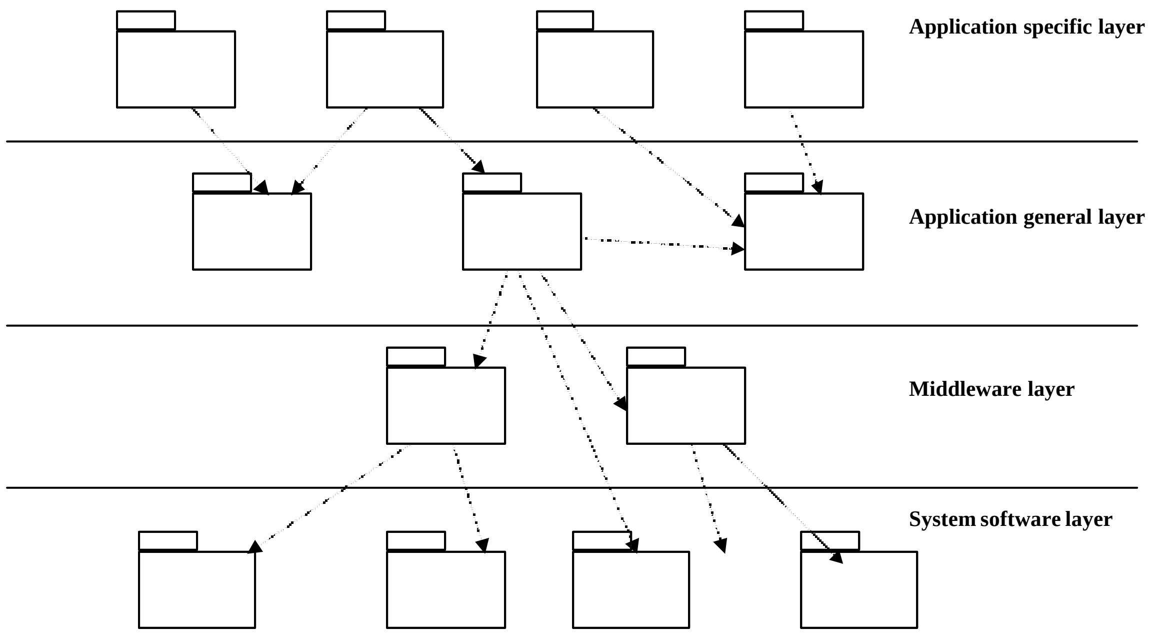


Figure 109: Subsystem layering

3.1.3.1.1.1 Identifying Applications Subsystems

In this stage architect specifies the subsystems in the application-specific and the application-general layers. The general strategy is to start from the analysis packages, identify service subsystems that do not break the structuring of the systems according to the services it provides, and arrange the packages in the two layers. Yet, this is just a general strategy, other considerations such as the organizational structure of the project, available human resources, etc. may also come into play since this division has an impact of how the work is distributed among the role players.

3.1.3.1.1.2 Identifying Middleware and System-Software Subsystems

These two layers are the foundation of the system, since all functionality rests on top of the operating system, communication software, GUI design kits, execution frameworks, etc. During this activity the architect chooses these software products and validates that they fit into the overall architecture and provide cost-effective implementation of the system.

The architect tries to limit the dependencies of the application subsystems, especially the middleware subsystems. The reason for this is that in many cases there is only little control over products developed by other vendors. The way to eliminate dependencies is to treat acquired software as a separate subsystem with explicit interfaces, try and acquire software that conforms to common well-known standards.

3.1.3.1.1.3 Identifying Subsystem Dependencies

If the contents of subsystems relate to each other then there is a dependency between these subsystems. The direction of the dependency is the same as the direction of the relationship, while a symmetric relation is translated to two dependencies (with opposite direction), which is not desirable.

In case the content of the subsystem is still unknown, the dependencies are drawn according to the dependencies between the analysis packages.

If some of the subsystem interfaces are known, then dependencies to these subsystems should relate to the interfaces.

[Test Case Speed Regulator]

Figure 110 shows a possible division of the Speed Regulator to subsystems. The regulatorUI contains the design of the user interface used by the speed regulator end users. It is the only subsystem located in the Application Specific layer since it is the only one that is not being used by the other subsystems. In the Application Generic layer we have the regulatorControl, which is responsible for the feedback loop that maintains the speed. It uses the speedometerAdaptor to sample the current speed of the car and readjusts the speed using the engineAdaptor subsystem. The regulatorControl reuses a generic feedback subsystem that has classes that implement generic PID feedback loops. Since the feedback subsystem is not related to the application at all, it resides in the Middleware layer. Two additional layers are specified in the Middleware layer: the execution framework, which contains generic event handling and dispatching, multi threading support, timings utilities, containers and more and the hwUILib that provide services such as turning on and off LEDs, displaying text on hardware devices, etc. The system layer contains the hardwareServices subsystem, which contains all the services the hardware platform provides. Any one of the above of the subsystem may be further divided into subsystem as the design evolves.

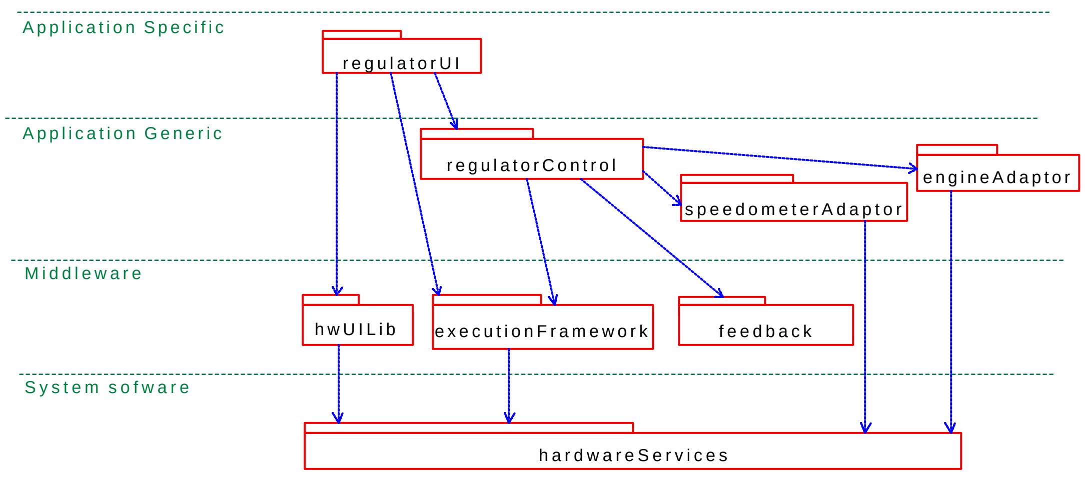


Figure 110: Speed regulator subsystems

3.1.3.1.1.4 Identifying Subsystem Interfaces

The interfaces provided by the subsystem define services that are accessible from the “outside” of the sub-system. Classes or subsystems within the subsystem implement these.

If a subsystem has a dependency pointing at it, then probably one interface or more should be defined for it providing the required services. In addition, if there is an analysis package that can be traced from the subsystem, then any analysis class that is referenced from outside the subsystem may imply an interface of this subsystem.

[Test Case Speed Regulator]

Figure 111 demonstrates how subsystem interfaces can be shown on a class diagram. The regulatorControl subsystem adjusts the speed of the car by using the engineController and the speedSampler interfaces implemented by the engineAdaptor and speedometerAdaptor respectively. In this example, the interfaces themselves are defined in the implementing subsystems, however, in many cases, the interfaces are specified in the dependent sub system as a sort of "requirement" for the implementing subsystem.

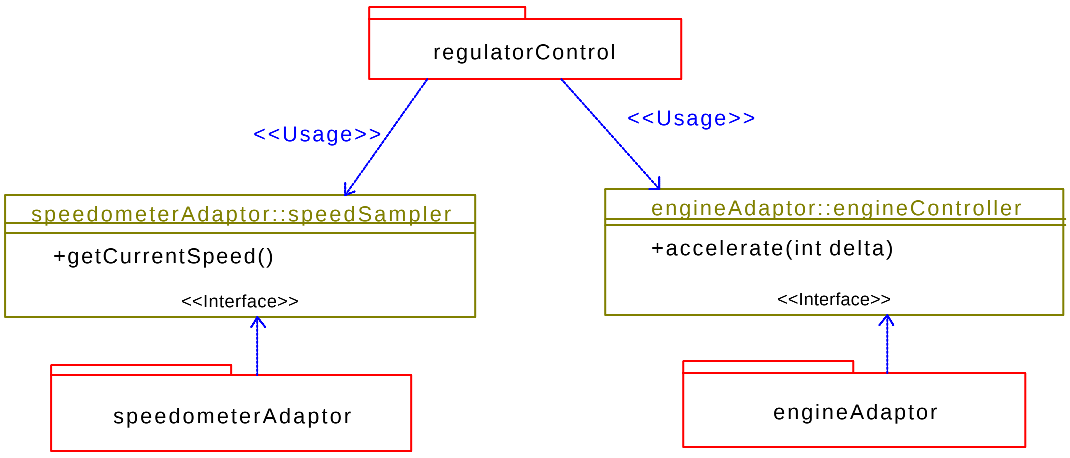


Figure 111: Adaptor interfaces used by regulatorControl subsystem.

Identifying Design Classes from Analysis Classes

Most design classes will be specified during the 'Detailed Design' activity ('Design a Class') however, it is often practical to identify architecturally significant design classes to initiate the design work during early stages of design. Yet, at this stage, one should be careful not to over specify too many classes and delve into too many details, since it may lead to unnecessary classes, i.e. classes that do not participate in any of the use-case realizations.

The most obvious design classes one can identify are the ones that correspond to the architecturally significant analysis classes. Also, the relationships between these design classes will probably correspond to the relations between the analysis classes. These relations are 'tentative relations' i.e. they me be elaborated as the design evolves.

Identifying Tasks and Active Classes

This activity is concerned with the *concurrency design model* artifact, which plays a major role in the design of real-time embedded systems because it has direct impact on system performance as well as hardware requirements[37]. Since the concurrency design model affects the entire system and also it may expose potential risks, this activity is carried out early in the design process as part of the architectural design activity.

[37] Generic processes, such as the unified process, do treat active classes in a special way and these are also identified early in the design process. However, these processes do not advocate a separate artifact such as a concurrency design model. This approach is more characteristic for processes concerned with real-time systems where scheduling and timing may affect correctness and not just performance in general.

There are numerous approaches for finding the threads and grouping objects into threads. The approach we chose to present here is based mainly on Octopus (reference .. E. Gery, R. Rinat, J. Ziegler, Octopus Concurrency Design with Rhapsody) which is a set of analysis and design methods developed by Nokia.

The input for this activity is the classes found so far in the "Identifying Design Classes from Analysis Classes" activity and the Use-Case Model: the architect needs to group the classes into task threads using the use-case model of the analysis. For this, Octopus prescribes a set of iterative stages shown in

Figure 112.

The first step is to identify *'event interactions*[38]*'*. An event interaction is a series of objects interactions designed to produce the desired response to an external event. The external event in this case can be a signal, a timeout, a call from an external software system, etc. The identification of these event interactions is derived from the design classes found so far and the analysis use-case models. A common way to model the interaction is via collaboration diagrams, but other types of diagrams such as sequence diagrams, statecharts or activity diagrams can be used as well.

After modeling the event interactions the architect 'qualifies' them: This means that the architecture decides if each message is synchronous or asynchronous for every event interaction. Usually there is no single optimal qualification and that is why octopus prescribes this activity to be iterated.

The next stage is to identify interaction groups and map them to threads. An interaction group is a group consisting of interacting objects along with their interactions. The interaction groups should be "closed under synchronous continuation": this means that whenever an interaction belongs to a group, and there is a synchronous interaction continuing it then this interaction is also in the group.

Following is a list of strategies for identifying interaction groups[39]:

- Single event groups: In simple systems it may be possible to group each response to an external or internal event to a separate interaction group. Usually this is not feasible for complex systems where many events can occur all the time or when context switch time is significant relative to the event response time.
- Event source: This strategy groups interactions that are initiated from a common source such as an actor or an external subsystem. This is many times the simplest approach, especially if the subsystems have well defined interfaces,
- Interface device (port): This strategy groups threads that encapsulate control of a specific interface such as RS232, a TCP/IP socket, etc. This is a special case of the former Event Source.
- Related information: When the same data is to be manipulated by many interacting objects, these interactions can be grouped to the same interaction group.

[38] Octopus uses the term '*Event thread'* and not '*event interaction'.* The reason for the change in terminology is to avoid confusion with threads of execution.

[39] This list is based on ROPES despite the difference of approach between Octopus and ROPES.

- Unrelated information: When two interactions seem unrelated, they are entered to different interaction groups.
- Timing characteristics: If a data arrives at a given rate, a single periodic thread can handle the data and dispatch it. In addition, a single interrupt handler can handle non-periodic events.
- Safety concerns: A common rule of thumb in safety-critical systems is to separate the safety monitoring from actuation. This implies having a separate interaction group for safety monitoring.

The last stage prescribed by Octopus is to assign priorities to each of the threads, and iterate again, reviewing the interactions and the threads identified so far.

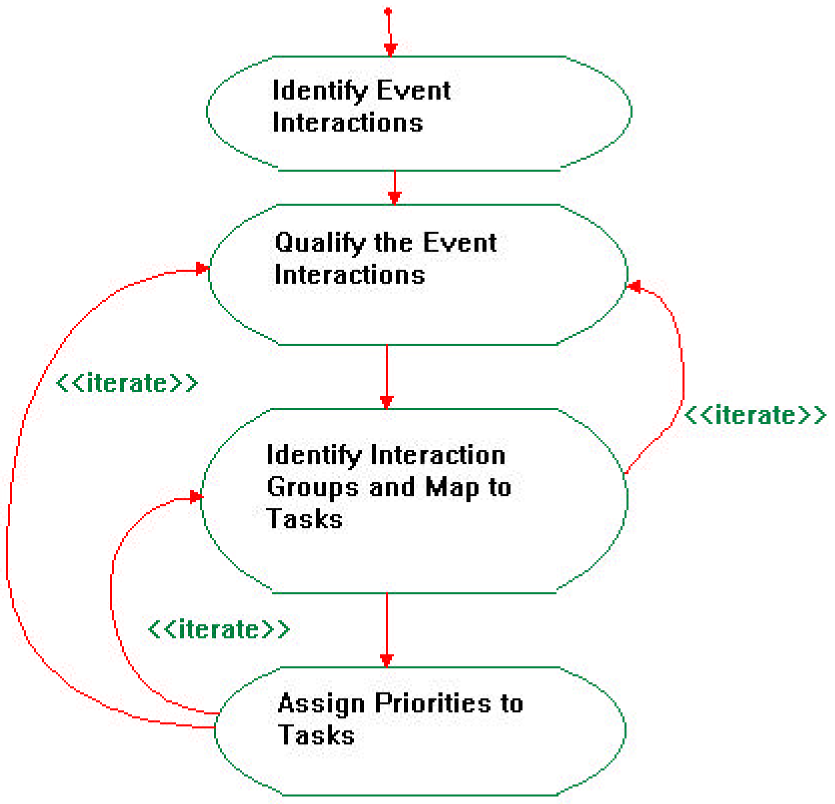

Identify Event Interactions
Qualify the Event Interactions
<<iterate>>
<<iterate>>
Identify Interaction Groups and Map to Tasks
<<iterate>>
Assign Priorities to Tasks

Figure 112: Constructing a concurrency model using Octopus

Octopus prescribes a way to group interactions into threads, but still the active classes and active objects that will control the threads are to be specified. The active object may be one the interacting objects or a newly specified object with the sole purpose of controlling the interaction. Sometimes, in simple cases, there might be a composition relation between the active object and the rest of the objects in the interaction group.

ROPES presents an alternative approach to Octopus: first the active classes, which represent the threads of execution are specified using various strategies (similar to the ones described above), and then objects of the design classes, found so far, populate each thread. Sometimes, this approach seems more straightforward than the one shown above, especially when these active classes can be mapped directly to an analysis or a design class or when very few design classes are available. Yet, it is the opinion of the author that the risks of missing threads or over specifying threads in the Octopus approach is reduced by relating to groups of interactions that are “closed under synchronous continuation” prior to the task definition. As usual a combination of approaches can be combined according to the specific case. ROPES approach fits more to the architectural design phase, while using Octopus usually requires going back and forth between mechanistic design and architectural design.

Activity: Identifying Components and their Interfaces

While Activity: Identifying Nodes and Network Configurations (section 0) was concerned with the Deployment and Safety\Reliability Model, the other activities so far revolved around the Logical Model. Here we revisit the Physical Model and complete it by specifying how the logical artifacts will be implemented as software components. The main UML diagrams that are used in this activity are Component and Deployment diagrams.

3.1.3.1.1.5 Identifying Reusable Components

Sometimes it is possible to identify a way to implement some of the logical artifacts in the form of resuable components or alternatively decide that some of the logical artifacts will instantiate third party resuable components at runtime. By resuable we mean that several instances of the same component may exist at runtime in the system possibly with different configurations. For example : suppose we decide to have a TCP/IP connection socket as a resuable component that may be configured at runtime to connect to a certain IP address with certain buffer sizes. This component can be given as a binary file to any application developer that needs such functionality without exposing any of its implementation details. The application developer can either use a specialized IDE or component technology such as COM to use this component or configure and instantiate it programmtically.

In many cases the decision to use such components (some refer to this approach as « component based design ») will be determined at early stages of the design when the underlying technologies are selected. For example the choice of CORBA, EJB or DCOM at an early stage will usually to such designs. Nevertheless, some may relate to conventional subsystems with interfaces bundled as a shared library, DLL, JAR file or any other format that can be linked dynamically as a resuable component.

After identifying such components in the system the architect may decide to purchase\subcontract these components from third party vendors. Using reusable components may also lead to better quality : especially reusability and extendibility but also reliability and other factors. On the other hand, component based approach may in some cases have an overhead which will affect efficiency, for example using CORBA or COM.

3.1.3.1.1.6 Specifying Component Organization

All logical elements are eventually implemented as components, reusable or regular static libraries, executables, command scripts, etc. In this activity the architect decides how the various subsystem will be bundled, their composition (a component may contain other components) and the way they would be deployed. Sometimes it turns out that a subsystem needs to be further divided into subsystems in order to allow its implementation in several components across several nodes.

Two conflicting considerations have to be taken into account: on the one hand components should be as self contained as possible in order to allow easier installation and deployment, on the other hand components should not contain too much so better reusability and extendibility will be gained (modular design).

3.1.3.1.1.7 Identifying Component Interfaces

In this activity the interfaces provided by the components are specified along with the dependencies between the components. The purpose of this is to reduce the risk caused by a change in one component to the other components by formalizing the relationships between the various components.

Every interface assigned to a component defines a sort of contract: what services are provided by the component and in what form. This way components engineers can be more aware of the impact their changes may cause.

A component may realize more than a single interface where the division is usually carried out according to the types of the services provided. By definition a composite component exposes all the interfaces assigned to its sub components.

Despite the advantage of specifying which interface is used by each dependent component, in some cases it is unrealistic to do so. For example, specifying such an interface for a procedural mathematical library would achieve nothing but redundant duplicate definitions of all of the public functions. In such cases dependencies without any specification of interfaces are used to signify the relations between the components.

At an early stage of the architectural design the architect specifies the major services defined by the interfaces as well as defining the interfaces provided by each component. As the system evolves (in subsequent iterations) more and more services are added to the interfaces, however changing an operation defined by one of these interface should be avoided as much as possible.

3.1.3.2 Phase\Activity: Mechanistic Design

Mechanistic Design is concerned with specifying the details of inter-object collaboration to achieve the required functional requirements. These collaborations may be generalized into design patterns mechanisms and applied in different situations.

Mechanistic Design takes use case realization scenarios from the Analysis Model and adds design level artifacts to facilitate and direct their implementation. Such artifact may be taken from the architectural design result but also from the Detailed Design, such as the container used for association, memory design patterns such as smart pointers to avoid memory leakage, etc.

Many of the collaborating design objects reappear in the various scenarios because they can be reused to solve common problems.

As mentioned above, Mechanistic Designs revolve around the realization of use-cases at the design level. The purposes of this design is:

- Identify the design classes and/or subsystems whose instances are needed to perform the use-case's flow of events.
- Distribute the behavior of the use-case to interacting design objects and/or to participating subsystems.

- Define requirements on the operations of design classes and/or subsystems and their interfaces.
- Capture implementation requirements for the use-case such as various non-functional requirements that are to be addressed during implementation.

The activities of mechanistic design are:

1 - Identifying the Participating Design Classes

2 - Describing Design Object Interactions.

3 - Identifying the Participating Subsystems and Interfaces.

4 - Describing Subsystems Interactions.

5 - Capturing Implementation Requirements.

6 -

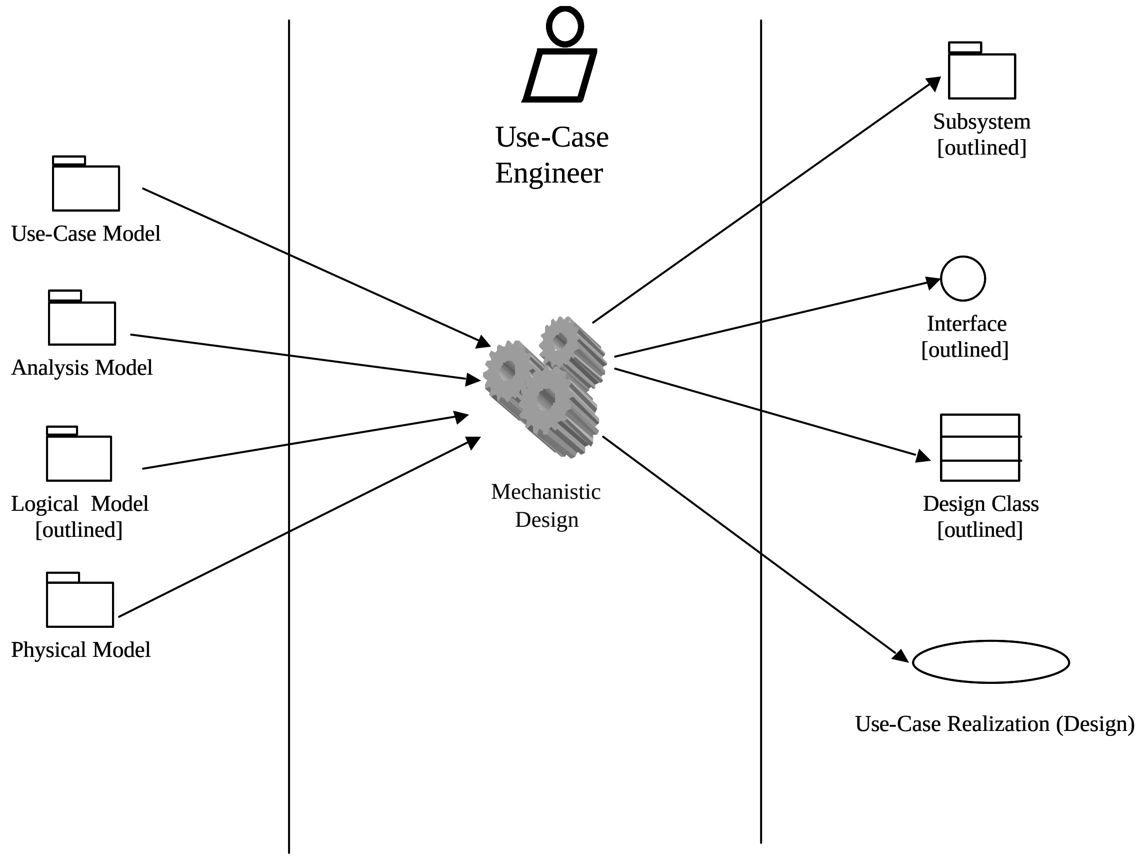


Figure 113: Mechanistic Design.

Activity: Identifying the Participating Design Classes

In this activity the Use Case Engineer identifies which of the design classes found during the architectural design or during the detailed design take part in the use case realization at hand (the use cases are found in the use case model constructed in the analysis phase).

A design class may take part in a use case realization if it can be traced to an analysis class that participates in the use case realization – analysis or if it is designed to achieve a non-functional requirement specified in the analysis.

If the use case engineer concludes that some design classes are missing, the architect or the relevant component engineer are assigned with task to define the required classes.

Activity: Describing Design Object Interactions

In this activity the Use Case Engineer draws sequence diagrams and collaboration diagrams (called interaction diagrams in general) to describe the interactions between the design objects and between the design objects and the actors that are realizing the use case.

Usually this activity is initiated by studying the analysis use case scenarios to get an outline of the design scenarios. Then the analysis classes are replaced with the design classes and parts of the analysis scenario are elaborated sometimes breaking a single sequence diagram to several diagrams.

In addition the use case engineer should relate to scenarios where malfunctions occur: for example a connection to a node is lost, an erroneous input is given, errors are reported from the system or hardware level, etc.

All and all, the purpose of this activity is to reduce the risk that things were missed and that the use case can be realized by the specified components. Specifying too many scenarios is not only unnecessary but may sometimes lead to obscurity. If the UML tool used offers animation, it may be desirable to let the animation produce the detailed scenarios and then review them or compare them to previous animated scenarios or manually specified scenarios.

Activity: Identifying the Participating Subsystems and Interfaces

Sometimes it is desirable to relate to subsystems or interfaces in the collaboration than to specific classes. This way the collaboration is independent of the internal structure of the subsystem, and in addition a more abstract description is specified leading to easier understanding of the flow.

Similar to the case of participating design classes, the participating subsystems\interfaces can either be traced to analysis classes that participate in the analysis use case realizations or represent design classes that are to be used for a non-functional requirement and contained in the design subsystem.

Activity: Describing Subsystem Interactions

Similar to describing the collaborating design classes the use case engineer may use sequence diagrams to describe the interactions with the subsystem. In this case, the subsystem itself cannot be added directly to the sequence diagram, instead either its interfaces are specified or the design class that serves as the core or façade of the subsystem (see Design a Subsystem activity in section 0). If neither of the above is available, the subsystem is probably no cohesive enough and this may suggest that restructuring of the subsystem by the architect is required.

Activity: Capturing Implementation Requirements

In this activity non-functional requirements that concern the implementation are specified. For example, specification of response times from objects or the amount of requests that can be handled concurrently by a single object.

3.1.3.3 Phase: Detailed Design

The detailed design phase is the lowest level of design. It is concerned with the definition of the internal structures and behavior of individual classes in the various subsystems.

The Detailed Design Phase consists of two activities: “Design a Class” and “Design a Subsystem”.

Activity: Design a Class

The purpose of designing a class is to create a Design Class that fulfills its role in use case realizations and the non-functional requirements that apply to it. This includes specifying the following aspects of the class:

- Its operations and event reception.
- The relationships it participates in (association, aggregation, etc.)
- Its method with the algorithms implemented (realizing the operations)
- Its imposed states.
- Its attributes.
- Exceptions handled or thrown by its operations.
- Its dependencies to and from generic design mechanism.
- Requirements relevant to its implementation.
- The correct realization of any interface it requires to provide.

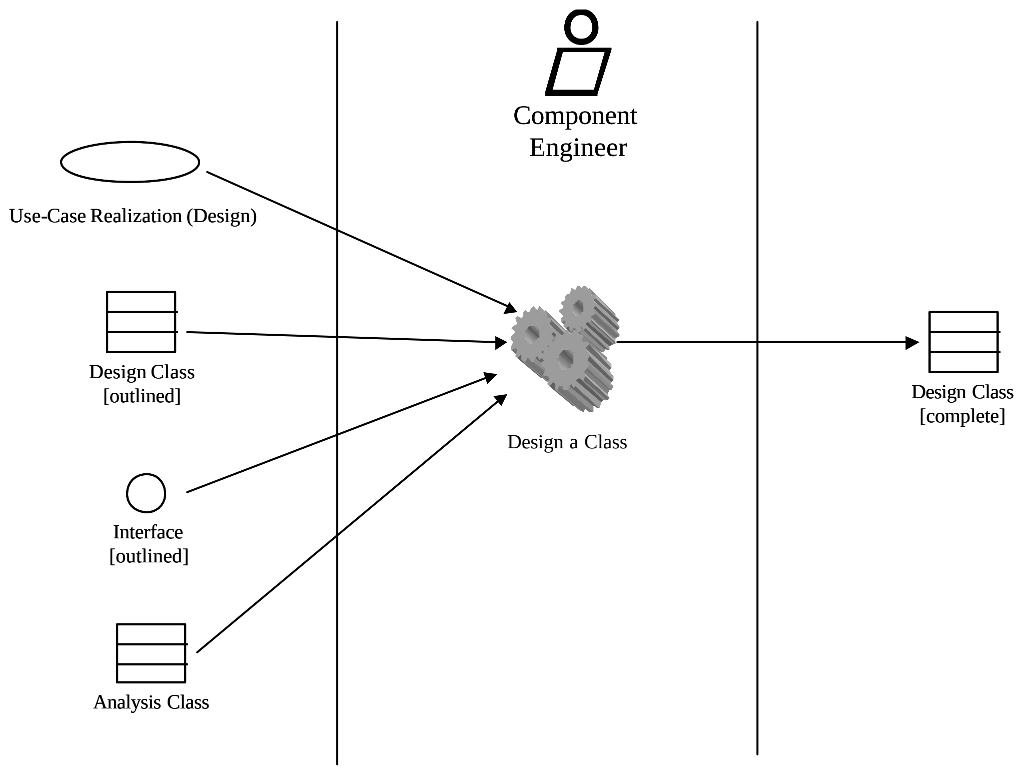


Figure 114: The "Design a Class" Activirty of the AIT-WOODDES Methodology.

Activity: Design a Subsystem

The purposes of the Design a Subsystem activity is to:

- Ensure minimal dependency to other subsystems.
- Ensure that the subsystem provide the right interface.
- Ensure the subsystem fulfils its purposes in that it offers a correct realization of the operation as defined by the interface it provides.

Note: usually a subsystem provides an interface by having one of its classes provide this interface and allowing access to its instances.

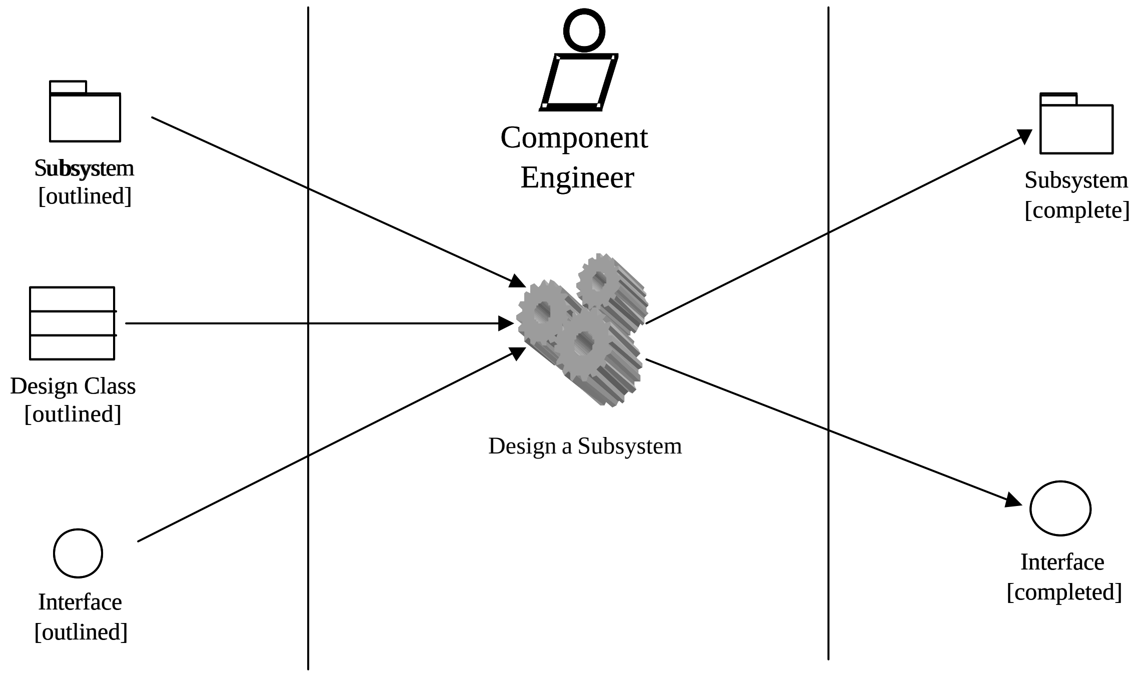


Figure 115: The “design a subsystem” Activity of the AIT-WOODDES Methodology.

4 PHASE – "BUILD IMPLEMENTATION MODEL"

Purpose of this modelling and expected results (i.e. types of diagrams the step has to provide, which level of detailed to reach, possibly some contractual and/or formatted document to supply at the end of the step for the next step, …)

Object-oriented development with a UML-based method results in graphical/textual models that have then to be mapped correctly and efficiently to programming code from which an executable implementation will be generated.

However, in the well-known UML-oriented methods, there is not much emphasis given to the transition from model to code. In the reference books such as ([Rumbaugh, 91 #486], [Desfray, 94 #474], [Booch, 86 #102], …), there is only one chapter that discusses implementation issues, and only from a general point of view. A few technical papers are available ([Rumbaugh, 1996 #152]), that give more insight e.g. on how associations can be implemented. A similar remark can be made with books on object-oriented programming such as ([Meyer, 92 #525], [Stroustrup, 2000 #524]) that generally have only one chapter to explain how features of the language relate to modelling concepts as found in the UML. There are new meta-modelling approaches introducing a method of "design by translation", that take into account non-functional requirements and thus help to bridge the gap to implementation.

In WOODDES, the entry for the implementation phase is given by the previous phase (Design) but also by the use cases established earlier, that are used to derive an architecture. The architecture is an important input for the implementation. UML provides two main diagrams to describe the structure of the implementation: component and deployment.

The overall goal of the implementation diagrams is to allow them to be part of the "build" process, i.e., they should be used both as input to the "make" process and as input to the targeting process. For this purpose, component diagrams detail how to put together the functional parts of the system (the type view), and the deployment diagrams detail how the system should be physically configured and deployed (the instance view). Some improvement of these diagrams could also render implementation easier, for example by specialising them towards specific application domains, where components exhibit some common behaviour through the use of standard interfaces.

4.1 Overview of the AIT-WOODDES approach

TBC

4.2 Diagram to use for implementation

As just said above, implementation diagrams are made of component and deployment diagrams. They are designed for the purpose of capturing the details of an object-oriented design. Components can be used to describe the individual modules of software capable of independent execution in a distributed environment (e.g. CORBA). Deployment diagrams can then be used to show how these modules are deployed upon a physical network of resources (processors, memory, …). Additional constraints can be attached to those diagrams to make them more specialised e.g. by means of some stereotypes.

4.2.1 The Component Diagram

Components and their interfaces provide the way to define a system's architecture, by offering encapsulation and information hiding; so they prevent any access to their inside except through their public interface. A component is a software artefact that exists at run-time for example an executable program or a library.

Example1: (possibly from the car speed regulator) to be completed

In the UML, a component architecture can be partly derived from the use cases: the functionality of a use case is encapsulated in some components with appropriate interfaces. Components can also be elaborated in a hierarchical structure, with for example components that are built on top of other components. These can be components that are part of the system "infrastructure", or external components that interface with. A support tool should allow to easily import such external components (e.g. COM components on Windows) with a type library into the model.

The interfaces of a component are elaborated by grouping the operations, but avoiding long list of functions. It should be noted that an object-oriented interface (e.g. obtained by creating some artefact at the interface level) is much easier to understand compared to a pure functional interface. The operations of the interface typically evolve and grow as the component matures. In order to assess the interfaces, one can derive sequence diagrams that materialise some use cases, and use the interfaces available in the component. This way allows to verify that the interfaces required for these scenarios are provided; on the contrary one has to create new operations in the interfaces.

For the purpose of implementation in WOODDES, the component diagram can be used to depict the constituents of a system or a component, and should be taken as input by an automated process for making the system. For example, one may have a component that shows the end result of the "make" process on UNIX.

As such, a component should contain information about which parts make up the system, how they should be compiled, what libraries need to be linked in, whether the system should run on UNIX or NT, whether multithreading is required, etc... To some extent, further specialisation of the component diagram may be needed to handle these aspects. An important aspect is to provide information about the threading policies of a component.

Example2: (possibly from the car speed regulator) to be completed

A component-based architecture fits with an assembling development technique. That is, much of an application can reuse already developed components, so that the remaining development work can be limited to pickup the desired components and programming only the code to glue the components together and drive them. In the case of large-scale applications with many components, the UML package concept can be used to group components together so that they can be organised.

However if the UML contains basic constructs for component-based development, some aspects of modelling components are not yet covered: It is difficult to specify components that are fully plug-substitutable. The notion of interface may be too weak to express all kinds of interactions between components; for example, in addition to simple message calls and receptions, it could include the definition of more complex transactions and in/out communications. By means of such mechanisms, a component could specify constraints (e.g. which operations and signals it will require from its connected components) on its environment in addition to the services it offers to other components. It is also recommended that the connections between components are specified independently of the specification of the components.

Additional aspects of components that should be considered for the implementation phase are:

i. *Components and interfaces*: It is necessary to take into account how components relate to different component-based architectures, such as CORBA components. In this context, it will also be necessary to take a look at how UML interfaces relate to for example CORBA interfaces, and how UML interfaces and associations relate to each other.

ii. *Components and classes*: In many cases it makes sense to think of structured classes as logical components, which may be hierarchically structured. These are components that are internally made up of other components, all of which have their own set of interfaces. This relationship has to be elaborated in more detail.

4.2.2 The Deployment Diagram

The basic purpose of an object-oriented program is to create a set of objects which will interact at run time to satisfy user's needs. The UML provides the basic mechanisms needed to draft this aspect of system's implementation by means of an object diagram. However, this diagram is used at the analysis phase, and generally represents a typical set of named instances and related links that only provides an illustration of the kind of object structures needed by the application. There is no exact specification of how and when the object structure will exist at run time.

The deployment diagram, on the other hand, goes further down, and can depict the physical organisation of the system in a run-time environment, such as how it should be configured in a distributed network. It must be noted that when building a distributed implementation, the deployment generally has an impact on the definition of components. For example, components cannot cross deployment nodes; but for a largely distributed system, this may be the case that a component crosses a boundary, and therefore it must be split in two or more sub-components. As a conclusion when building a distributed system, it may be wise to elaborate the deployment model earlier in the project life cycle.

Example3: (possibly from the car speed regulator) to be completed

The instance view of the deployment diagram is used as input to the targeting process, and should primarily show what the system will look like when it starts executing. The location of components is generally done statically for real-time embedded systems. The main reason for this is that much of the embedded software is tightly bound to hardware devices and interfaces, and so a dynamic location of components would be very difficult to realise.

Another use of UML deployment diagrams in system implementation is to specify the physical relationship among hardware and software components in the real-time infrastructure. This phase is of high importance when dealing with hybrid systems that are partitioned into hardware and software, each part being implemented in a dedicated object-oriented language e.g. Objective VHDL and C++ respectively. The generation of both codes requires then to generate the interfaces between the two sides, based on the information contained in the deployment diagrams.

In the context of real-time hybrid systems it should be suitable to have a framework to clearly distinguish between different architectural aspects of object-oriented systems. Such a framework would keep application objects, system infrastructure, hardware resources and network topology in separate UML diagrams. As a result, it would be possible to experiment with different design alternatives in some part of the system without having to modify other parts.

4.3 Model consistency rules

There are two kinds of consistency rules that must be investigated:

i. Rules that are already included in the UML notation, for which there exist checkers. For example there exists a checker to verify that class diagrams (CD) and state machine diagrams (SMD) are consistent; the same for consistency between SMD and sequence diagrams (SQD).

ii. Rules that can be defined/customised specifically for the purpose of WOODDES. For example depending on the extensions the project will bring to some UML diagrams, one may define additional consistency rules. A typical case for WOODDES will be the use of specific stereotypes that will have to be checked.

Regarding components, following rules should be verified:

- ♦ Component interfaces should avoid long lists of operations and functions.
- ♦ Components are binary-replaceable things. This means that if a revision of a component is created, it should be able to replace a previous revision of the same component without recompiling other components.
- ♦ Components are important for systems where updating, maintenance, distribution and reuse are key aspects. These are typical of real-time and embedded systems. If on the contrary, the developed system is not concerned with that, then a single monolithic component should suffice.

Regarding deployment, following rules should be verified:

- ♦ Each component must belong to one of the nodes of the deployment model, possibly through a package within a node.
- ♦ A deployment model must tell where each component executes, and how each component interacts between the processors and other electronic devices such as physical communication ports.
- ♦ Stereotyping is much recommended with the nodes, particularly by means of bitmaps to resemble the purpose of the nodes.

4.4 Deployment and targeting

4.4.1 Preliminary Assessment

Before going to implementation of a real-time object-oriented system, it is necessary to evaluate the design models made in UML in order to have an early feedback of both the functional and non-functional behaviours of the system. This step, done at the prototyping stage, should considerably reduce the iteration loop between the design and implementation of real-time systems. It is highly suitable to detect any infringement of the temporal requirements at the design level by simulation (execution of the design model) because of the smaller cost induced by a modification.

Real-time systems, in particular hard real-time ones, often execute several threads in parallel. Such threads can be defined in tasks [Douglass, 99 #70] and all the system tasks must be identified e.g. in a system task diagram. At implementation, it is necessary to map the system specification onto tasks and to allocate these tasks to processors. At this point, a scheduling analysis is needed to ensure that all tasks (processes) will meet their deadlines, and to verify the temporal requirements. There exist methods for scheduling analysis such as e.g. the Rate Monotonic Scheduling that can be applied for this purpose. However this aspect of the implementation is not fully covered by the WOODDES methodology and is left for further investigation.

4.4.2 Tool Support

Developed tools should integrate new features providing the necessary help for industrial software development. In particular, they should give full support for application deployment using visual models.

Deployment Editor:

The transformation of logical UML models into efficient implementations is a key-activity for UML-based application development. The purpose of the Deployment Editor and Targeting Expert is precisely to provide support in this activity. Deployment diagrams describe the physical architecture of the application and the relation between the logical entities like UML classes and this physical architecture. This is essentially a model based on the following main concepts:

- Nodes, i.e. the hardware platforms on which the application will run;
- Components, i.e. the executable files that make up the application;
- Logical entities that run in the threads, i.e. component property set either to *light* (only one thread for the entire system), or *tight* (one thread per instance) or *instance-set* (one thread per instance set).

An example of a deployment diagram is shown in Figure 116. The aggregation structure illustrates the relations between the different entities, which components can execute on which node type and which entities can run in each component. The UML stereotypes used in this diagram are derived from the UML profile elaborated for SDL (Telelogic).

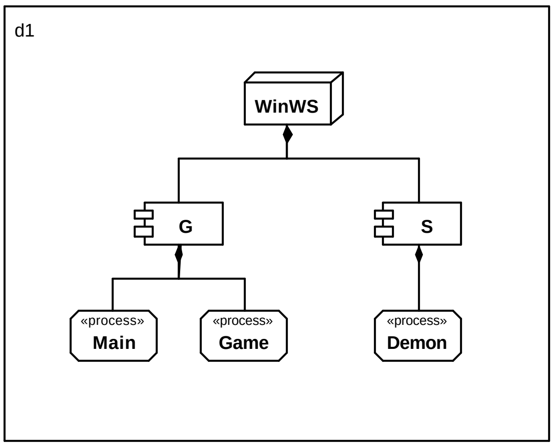


Figure 116: A Typical Component Diagram.

The UML model and the deployment diagrams cover two of the most important aspects of an application, the logical behaviour and the physical structure.

Targeting Expert:

The purpose of the Targeting Expert tool is to further define the details of each component in the physical structure. It should guide the user through implementation choices that are available for each component and make it possible to fine-tune the execution of the generated applications and to define the details of the build process. It can also be the tool for building the application, that could execute both in interactive mode, mainly for setting up the generation and build properties, and in batch mode, to build the final application.

Typical properties that are described using Targeting Expert are for example:

- Details of the code generator to be used;
- Details of the compiler that is used, e.g. optimisation issues like whether the compiler most efficiently handles characters or integers, the environment variable settings required;
- Definition of which communication links to use within the application, which encoders/decoders to use;
- Description of what tool has to be used to download the generated application onto the target platform.

These properties can be set either for a complete application, for all components running on specific nodes or for specific components only.

The Deployment editor and the Targeting expert could be part of a more general deployment environment, including also legacy code for reuse and an abstract interface to the RTOS.

5 PHASE – “BUILD PROTOTYPE MODEL”

Purpose of this modelling and waited results (i.e. types of diagrams the step has to provide, which level of detailed to reach, possibly some contractual and/or formatted document to supply at the end of the step for the next step, …)

5.1 Diagram to use

Identify the diagram to use and for each the main notations (in particular, in relation with diagram and notation proposed for AIT-WOODDES profile in deliverable M1, with explicit list and motivation on extensions, specialisation)

5.2 Model consistency rules

Definition of the main consistency rules to be defined among the diagrams to allow the global model to be unambiguous and complete

6 PHASE – "VALIDATE MODELS"

In this section we describe the validation concerns of the AIT-WOODDES methodology. The methods covered here are based on the AIT-WOODDES process definition, described in the deliverable M3, which describes in detail the process in which the methods are applied.

In the AIT-WOODDES process, seven activities have been identified: Pre-Analysis, Analysis, Design, Detailed Design, Implementation, Integration, Final Validation. In addition, a validation activity has been identified, which will be performed in parallel throughout the process to support the other activities. The purpose of the validation performed in each activity is mainly to check that the output produced is consistent with the requirements specified for the product under development and that the output of an activity is consistent with the output of the preceding activity in the process.

The validation methods described in this section will focus on the methods needed to perform the validation activity. However, before describing the actual validation methods, we summarise the validation needs of the process described in the deliverable M3.

6.1 Validation Methods of the Development Process

The validation methods identified in the development process are described in terms of use cases. In the following we give and overview of the need for validation methods that are identified in the use cases:

Pre-Analysis

Here the consumer requirements on the product to be developed are documented. An acceptance test of the consumer is also specified.

Validation methods are needed to specify and validate the consumer requirements, and to verify that the specified requirements are complete and consistent. Methods for test case generation will be needed to specify the acceptance tests.

Analysis

In this activity the requirements produced in the pre-analysis step are formalised. A product requirements specification for regression testing of the product requirements is also produced.

In order to check that the produced requirements correspond to the user requirements derived in the pre-analysis step, validation methods such as formal verification will be used. In addition, simulation can be used to support the verification.

Design

Here the details of the design model are worked out. The system is divided into several modules and the collaboration and interfaces between the modules are specified. From the validation point of view this means that the various modules must be analysed to ensure that the modules can be composed together and integrate to fulfil the requirements specification of the developed product.

The required validation methods will be simulation, formal verification, and integration test specification.

Detailed Design

In the detailed design stage of the process, the detailed behaviour and the control algorithms of the modules of the design are decided. Thus, the exact behaviour of every module must be formally specified and analysed by means of validation.

The validation methods applied here are model simulation, formal verification and test case generation.

Implementation

Here the behavioural specification of each model is transformed into a program code unit of the actual target platform. Unit level testing is also conducted.

The validation concerns here are unit level testing to check that each module performs according to the module test specification.

Integration

The units resulting from the implementation are put together to ensure that the integrated system performs according to the formal requirement specification of the whole system.

The main validation method here is integration testing to check that the system meets its requirement specification.

Final Validation

The integrated system is presented to the customer to get acceptance tested to ensure that the customer's requirements are fulfilled. At this point everything is supposed to work.

Acceptance testing will be the main validation method used for final validation.

6.2 Validation Methods for UML Models

In the following we describe the validation and verification methods used in the development process to validate UML models, i.e. the analysis and design models. The presented techniques can be applied manually or with tool support by (at least) one of the tools in the consortium[40]. The methods are divided into four parts: Sanity Checks, Simulation, Verification, and Testing. In addition we list some validation methods for implementations that may also be applied to UML models.

6.2.1 Sanity Checks

Sanity checks are used to validate assumptions that went into the development processes and are possibly violated at later stages. The sanity properties contribute only indirectly to the correctness of the model.

[40] Some of the presented validation and verification techniques are not yet supported by the tools, but are parts of current or planed developments and will become available in the near future.

Syntax and Type Checks

The UML modeling language comes with a set of requirements on the various modeling items. Checking these is necessary to ensure the consistency (which is often not obvious for large models).

Syntax and type checking should be done on the analysis and design models.

Consistency Checks

Consistency checks ensure that statecharts and activity diagrams are well-formed, as well as checks for detecting conflicts such as stereotypes of class with is in conflict with its specification or an abstract operation assigned by the body.

Consistency checks are applied to analysis and design models.

Conformance Checks

The conformance checks aim at guaranteeing a correspondence between the models in the different states of the development process. The conformance checks should ideally be done (algorithmically) to establish a refinement relation between two models.

The conformance check should be applied whenever a model is refined and made more detailed. In particular, the design model should be conformance checked with respect to the analysis model, and similarly the implementation model with respect to the design model.

6.2.2 Simulation

Simulation is the process of virtually executing or imitating of a model by means of software. This is useful for debugging the model thereby increasing the developer's confidence. Simulation is also useful for visualizing error scenarios found by other validation and verification methods.

User-Guided Simulation

In user-guided simulation the user interacts with a simulation tool (i.e. a simulator) to validate the model behavior against the intended system functionality with the purpose to find errors in the model and to gain increased confidence in the model.

User-guided simulation is suitable for the analysis model and the design model.

Random Simulation

In (intensive) random simulation, a simulator fires transitions automatically; when several transitions can be fired, the simulator fires one of them "at random". A typical solution for this random is to use a special number called seed, to calculate a series of random choices.

Usually, the seed is a parameter in the simulator that can be modified. But alternatively for a given specification and a given seed value, the simulator will always execute the same scenario.

Main advantage of random simulation is that it can handle very large models, but it is not sure that all the behaviours of the specification have been covered (a coverage view of the specification can be given to assess the validity of results).

In random simulation can be done on the analysis model and the design model.

Trace-Guided Simulation

Here the simulator run is determined by a usually machine-generated trace. The method is mainly used to explore a trace of particular interest, e.g. an error scenario found using a verification tool.

Trace-guided simulation is applied to the analysis model and the design model.

Trace-Guided Simulation using Simulated Time

(Merge this with section "Trace-Guided Simulation") An ordinary trace-guided simulation is conducted but the system outputs a time stamped trace using user's time estimations instead of the actual execution times of the model. The simulated time is used for isolating environmental influences such as code instrumentation overhead, operating system impacts, etc.

Trace-guided simulation using simulated time is applicable to the analysis model and to the design model.

6.2.3 Formal Verification

The aim of formal verification is to provide techniques and tools as design aids to provide reliable control systems. During verification of a model, special purpose algorithms are applied to formally establish properties expressed as logical formulae, sequence charts, or by observers in the model. The verification will prove or disprove the given properties. The properties are derived from the requirement specification of the modelled system or from previous stages of the development process (as described in Figure 1). In the following we describe the considered verification techniques.

Functional Verification

Functional verification is used to check the functional behaviour of a model. Properties to describe the functional behaviour can be classified as

- Safety properties
- Liveness properties.

Safety properties (or invariants) are of particular interest as they can be used to establish a model, in all reachable states, satisfies a specified condition. Dually can safety properties be used to express that a condition is never met in the model. This is of particular interest as it can be used to show that the model never behaves in an undesired or unsafe manner. A variety of specialized and efficient algorithms for checking safety properties exist, that can verify safety properties without human interaction.

Liveness properties are used to express that a desired system state is guaranteed to eventually be reached, or that a situation will be reached infinitely often (permitted an infinite run).

Properties derived from the requirement specification can be checked on the analysis model and on the design model.

From a user point of view formal verification can be used for two key use cases:

- model debugging,
- model verification as certification.

Model Checking as a Debugging Tool

In the debugging approach the user can define "breakpoints", which he/she would like to observe. The verification engine is used to check whether these breakpoints are reachable or not. In the case that a breakpoint can be reached, the verification engine will offer a scenario describing a system run leading to the breakpoint. Breakpoints can be described as

- **Drive-to-state**: The user specifies a single state of a state machine he would like to examine.
- **Drive-to-configuration**: In this case the user specifies a set of states (i. e. states of different concurrent substates) and check whether a configuration, which contains all specified states, can be reached.

Example: *Is it possible to reach a configuration where concurrent sub state s1 is in state s12 and the concurrent sub state s2 is in state s23? (see Figure 117).*

- **Drive-to-property**: This is the most general case where the user specifies a boolean condition and asks whether the system can be driven into a configuration satisfying the given boolean condition.

Example: *Is it possible to reach a configuration where the value of the attribute x is equal to zero? Is it possible to reach a situation where the system is in state s12 and x is not equal to zero?*

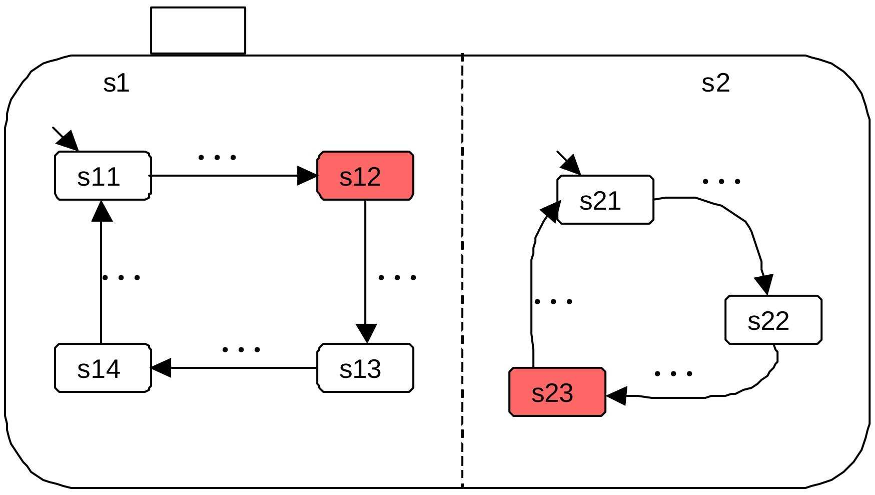


Figure 117: Drive-to-configuration (s12, s23)

Model Checking as a Certification Tool

In a certification use-case, model checking is used to formally validate a system model against given requirement specifications. Given the system model and a set of safety critical requirements the model checker will provide certification evidence that the requirements are never violated by the model. In this case we would like to prove properties like:

- *Will the central locking system unlock the car in a crash even when ...?*
- *Under no circumstances the steering will be locked when ignition is on.*
- *When the speed regulator is switched on, the speed will never exceed the specified limit.*

The model checker expects that the system requirements are given as temporal logic formulae. As in the debugging mode, where the user only specifies a state, a configuration, or a boolean property, which is then translated internally into a temporal logic query for the model checker, we will provide a set of predefined patterns to describe requirements in a formal way. Some of the patterns will be given in a textual form, e.g.

whenever P then Q.

The user selects the pattern and defines properties for its parameters. E.g.

P: *regulator is switched on*

Q: *speed is within the given bound*

Other patterns will be given in a graphical form using LSC (Life Sequence Charts).

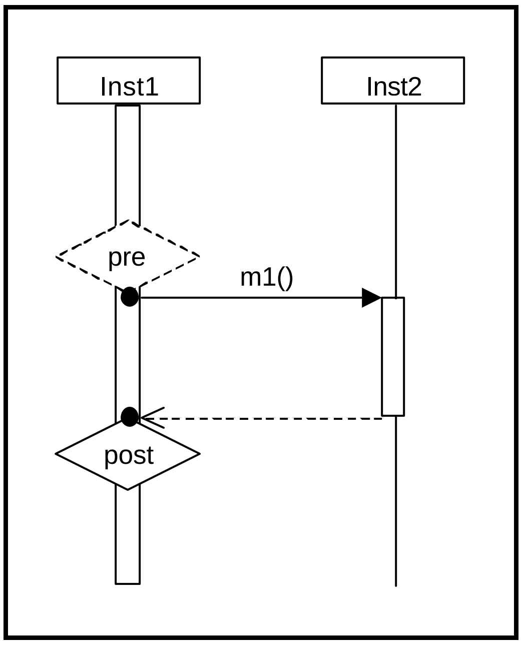


Figure 118: Specifiying pre- and post-conditions of a method call

Figure 118 shows a pattern to specify the effect of a method calls. Whenever the instance *Inst1* calls method *m1()* in a situation satisfying the precondition *pre*[41] it is expected that the post condition *post* holds when *Inst1* receives the result of the method call. In the LSC of Figure 118 the black circle

[41] In the LSC notation (see [29]) the precondition is indicated as a *cold* condition (dashed line) as it is not required that a situation satisfying *pre* occurs.

specifies a co-region, i.e. the condition *pre* is evaluated at the same time instance as the method call is initiated.

The pre- and post-conditions are not restricted to the attributes visible at *Inst1* but may contain also references of other instances for example of *Inst2*. An example could be that the method call *m1()* is invoked in a situation where the state machine defining the behaviour of *Inst2* is in a specific configuration. Imagine for example that *Inst2* has two operating modes, normal mode and emergency mode. Then the pattern could be used to specify the result of the method call provided *Inst2* is in normal mode. In the example chart (see Figure 119) the call of *m1* in normal mode results in the execution of action *a1*. But if *m1* is called in a situation where *Inst2* is in the emergency mode action *a2* is executed which may yield to another result and thus violating the given post condition.

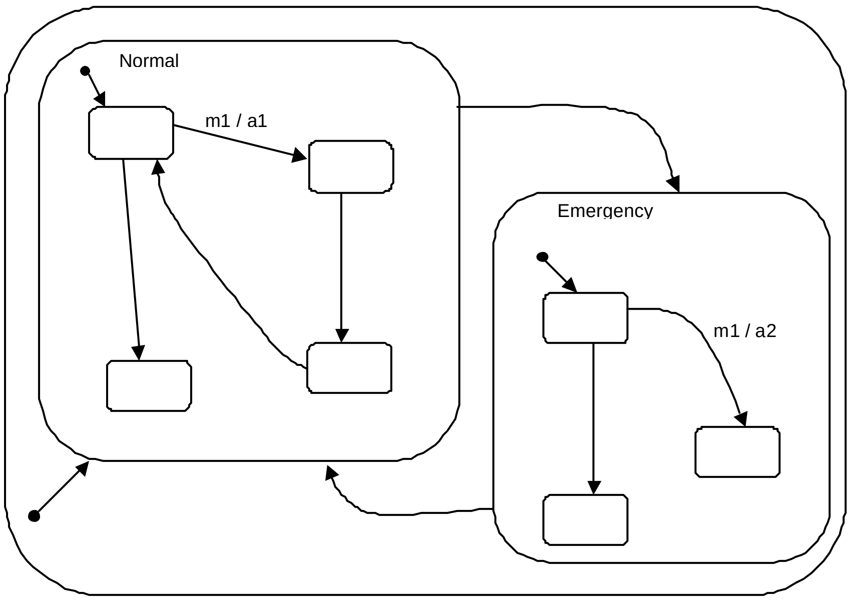


Figure 119: State machine with two operating modes: Normal and Emergency

In the general case we can use the methods to formalize a case splitting as indicated in Figure 120.

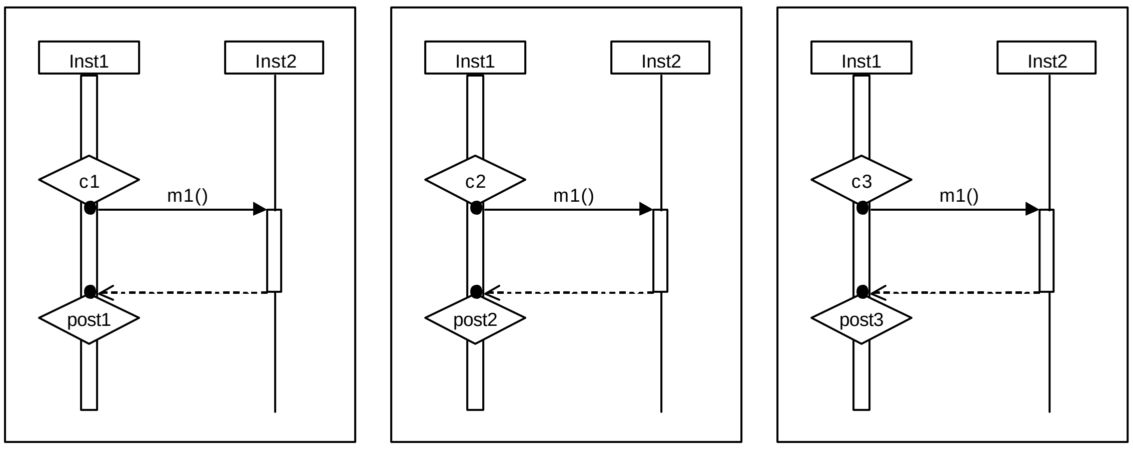


Figure 120: Case splitting

A typical use case for verification is to check that of invariant properties. An invariant property can be applied to the complete runs of a system, i.e. this would be of the form invariance (P)

without further constraints. On the other hand the invariant property can be restricted to a specific phase of a run. In Figure 121 the invariant *inv* should hold between two execution instances specified by receiving events *e1* and *e2*. The interval in which the invariance should hold can be defined by different actions: sending or receiving events, method calls, or by specifying conditions. Conditions should only be used in combination with other observable items defining a concrete status of the system where the condition has to be evaluated. Alternatively, a starting condition could be given as an *activation condition* (see Figure 122).

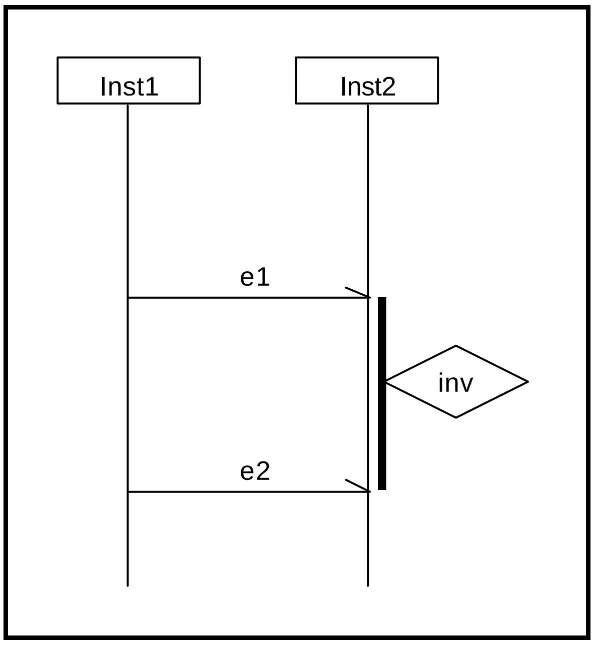


Figure 121: Specifiying an invariance

For example the events *e1* and *e2* may coincide with switching the speed regulator on and off. If the speed regulator is switched on, the speed of the vehicle should be in the range as specified unless the stop signal is received.

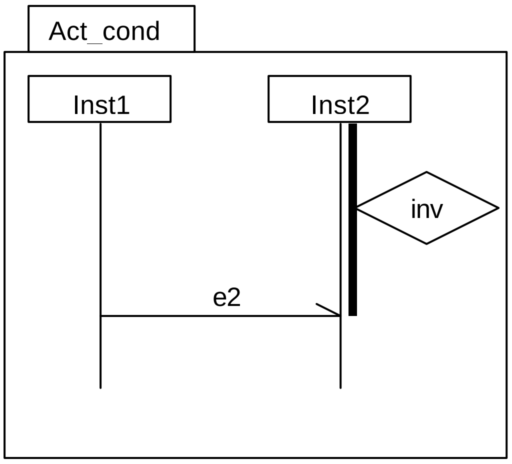


Figure 122: Invariant property invoked by an activation condition

Deadlock Detection

Deadlocks correspond to situations where no further progress is possible - the system "freezes". This class of undesired behaviors can be verified automatically to a large extend.

Deadlock detection can be performed in the design model (and observed in the implementation model).

Live-Lock Detection

When live-locked, a system continues to change its state without doing any useful work wrt. some notion of "progress". Live-locks are harder to detect than deadlocks, but in the reach of algorithmic treatment.

Live-lock detection should be applied on the analysis model and the design model.

Time-Stop and Zeno Detection

Real-time models possibly reach situations where no progress of time is possible (so called time-stops), or time progresses but never exceeds a certain limit (called Zeno behavior). These are errors in the model that do not correspond to any real-life behavior of an implementation. It is therefore important to establish time-stop freedom and non-zenoness in any model with time, otherwise some established model properties can not be trusted to be present in the implementation as well.

Time-stop and Zeno detection should be applied to the analysis model and the design model.

6.2.4 Testing

Testing is the process of exercising a model to identify differences between expected and actual behavior. Testing involves one or more test runs, each of which explores one possible evolution of the system.

Glass Box Testing

Test generation via approaches based on structural coverage tests aiming to a single test per possible behaviour of an application.

Glass box testing is applicable to the analysis model and the design model.

Regression Testing

The problem addressed is that each time an error is corrected or a change is made in the specification, it must be checked that no new errors have been introduced. This can be realised by means of MSCs that will define all the inputs sent to (or all the outputs received from) the system specification; a simulator will then be piloted by these MSCs, and will check that the outputs correspond.

Regression testing will detect inconsistencies between the specification and the reference MSCs. Additional checks can be made on existence of process instances or value of process variables.

Regression testing can be applied to the design model.

6.2.5 Other Methods

Other methods for validating the models include: dead-code detection, performance evaluation, exhaustive test case generation, and co-simulation and co-execution. These methods are described in more detail in the next section devoted to methods that mainly applicable to implementations.

6.3 Validation Methods for Implementations

In the following we describe the validation and verification methods used in the development process to validate implementations derived from UML models.. The presented techniques can be applied manually or with tool support by (at least) one of the tools in the consortium[42]. The methods are divided into three parts: Sanity Checks, Simulation, and Testing. In addition we list some validation methods for UML models that may also be applied to implemenations.

6.3.1 Sanity Checks

Sanity checks are used to validate assumptions that went into the development processes and are possibly violated at later stages. The sanity properties contribute only indirectly to the correctness of the model.

Code Sanity Checks

Code sanity checks searches generated code for errors that originate from version shifts, lax typing, or known compiler bugs. Special techniques from compiler theory and (strong) type checking can be applied here.

Code sanity checks should be applied to the source code of the implementation model.

Dead-Code Detection

A part of a model or an implementation that is guaranteed to never be reached or executed is called "Dead-Code". In the modeling case, dead code is an unreachable state or transition (i.e. a transition that is never taken). In general dead-code detection can be treated by reachability analysis (see Verification below), but also syntactic techniques exist.

Dead-code detection should primarily be applied to the design model but also to the implementation model.

6.3.2 Simulation

Simulation is the process of virtually executing or imitating of a model by means of software. This is useful for debugging the model thereby increasing the developer's confidence. Simulation is also useful for visualizing error scenarios found by other validation and verification methods.

[42] Some of the presented validation and verification techniques are not yet supported by the tools, but are parts of current or planed developments and will become available in the near future.

Performance Evaluation

Performance analysis is a cross-border domain, as it requires resource description, which is actually not available in UML. Current work in progress aims at introducing timed extensions to UML in order to address this kind of analysis.

On the technical side, performance analysis requires extensions to current verification techniques. We need to deal precisely with time, without worsening the state explosion problem. For this, state-of-the-art techniques may use symbolic representation of clocks.

Performance evaluation can be performed on all models during the development.

6.3.3 Testing

Testing is the process of exercising a model to identify differences between expected and actual behavior. Testing involves one or more test runs, each of which explores one possible evolution of the system.

Conformance Testing

This is the process of establishing the extent to which an Implementation Under Test (IUT) satisfies both static and dynamic conformance requirements (i.e. features given in a standard with which a product implementing that standard must conform), consistent with the capabilities stated in the implementation conformance statement (i.e. the document supplied by the manufacturer of the product).

A conformance test consists of two parts: the static conformance review (checking that implementation choices are permitted by the standard) and the dynamic conformance test (execution of test cases to determine whether the product has implemented the standard correctly. Each test case within a test suite is related to one conformance requirement of the standard.

Conformance testing should be done on the implementation model (and the design model???).

Exhaustive Test Case Generation

As a basic solution, generation of scenarios by simulation can provide reference to test implementations. These scenarios will complete the specification from a dynamic point of view. Because they are generated by simulation, they are sure to be consistent with the specification.

A first step consists in finding pertinent test objectives based on the requirements specification. A second step consists in formalizing test objectives described informally. For this one can use MSCs, by which partial scenarios are described which will be completed by simulation. The third and final step consists in running the simulation to obtain all the test scenarios (if any) corresponding to the test objectives.

Exhaustive test case generation can be applied to the design models and the implementation models.

Co-Simulation and Co-Execution

This type of simulation (or execution) consists in running (executing) the system specification (implementation) and the test specification (executable tests) simultaneously. Most benefit is achieved when the system specification and the test specification come from two different sources. In this case, if the co-simulation runs (co-execution executes) without errors, then the system fulfils its requirements.

Co-simulation (or co-execution) is performed on the design model or the implementation model.

6.3.4 Other Methods

In addition to the methods described in this section, conformance checking can also be applied to implementations. For more information about conformance checks see section 3.2, which describes validation methods that are mainly applicable to UML models.

APPENDIX 1 - UML BASICS

1 Use case modeling

The main model elements manipulated in the use case diagrams are:

- **UseCase** that describes the behavior of a system or other semantics entities without revealing its internal structure;
- **Actor** that models a coherent set of "roles" played by users with respect to an entity with which they are interacting.

In *UML*, use cases can be linked by generalization, inclusion (« ***include »***) or extension (« ***extend »***) relationships [11] :

– the generalization relationship between two use cases has the same meaning as for two related classes;

– the inclusion relationship is a means for factorizing system behaviour. This means that the initial use case explicitly includes the behaviour of another use case at a given point in its sequence;

– the extension relationship is used to model optional behaviours for a given use case. This case is then said to have a "variation point". Assignment of a variation point results in one of two possible configurations: In the first, the extension is not taken into account; in the second, the use case described by the extension is included in the initial case.

The speed regulator example given in Figure 123 illustrates possible relationships between use cases. In this example, the *"**stop regulation**"* function serves as a use case for the various types of relationships described above.

The diagram in Figure 123 depicts the following:

– the "stop regulation" use case has two variants – "stop regulation by braking" and "stop regulation by actuating on/off button"– which correspond to the different regulator stop cases described in project specifications.

– the "stop regulation" use case has a variation point labeled "emergency action". This use case thus comprises both a so-called "normal" regulator stop scenario (most common type of behaviour) and a second scenario that also incorporates the scenario contained in the use case associated with the initial scenario via the extension relationship. This relationship presents a variant with respect to "normal" system stop;

– the "stop regulation" use case has two included use cases furnishing a more detailed description of certain behavioural aspects ("refresh display" and "stop speed acquisition").

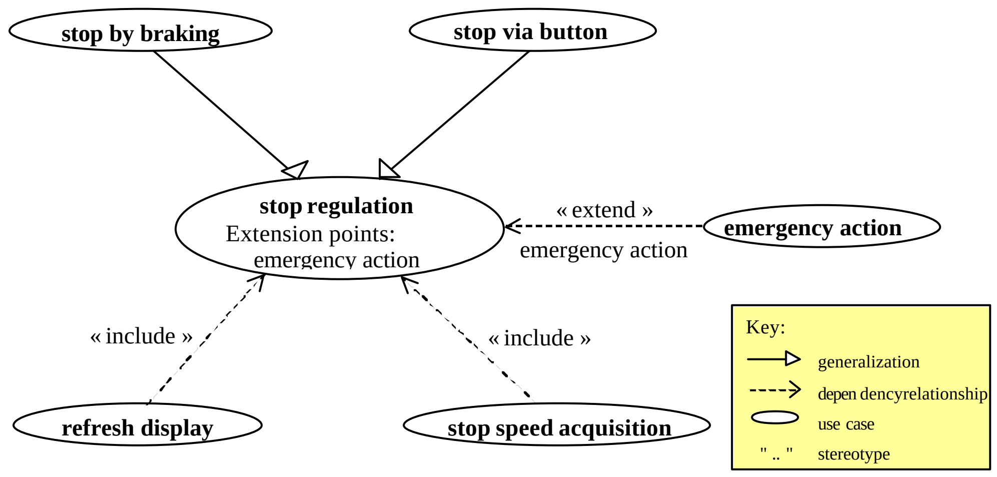


Figure 123: Examples of Use Case Relationships.

Although not the case in the example given here, two actors may be linked by what is known as an inheritance relationship. This means that, with respect to the system, the "inheriting" actor will exhibit the same behaviour as its "parent" (actor).

GLOSSARY

Activity	A work definition describing what a role performs. Activities are part of a stage and/or Iteration and may contains Step
DAM	Detailed Analysis Model
Goal	A goal is a specific condition that is satisfied at the end of a given Work Definition such as the end of an activity group or a phase.
IRD	Initial Requirements Document
Milestone	Synonym for the goal of a phase.
PAM	Preliminary Analysis Model
Phase	A phase is the highest level of granularity in describing the work performed in a process. The process lifecycle describes the phase sequel of a process. It may be decomposed in different activities.
Work Product	Work Product is a description of a piece of information or physical entity produced or used by the activities of the software engineering process. Examples of work products include models, plans, code, executables, documents, databases, and so on.

BIBLIOGRAPHY

[1] OMG, “The Software Process Engineering Metamodel (SPEM),” OMG, Revised Submission ad/2001-06-05, 9/27/2001 2001.

[2] L. T. J. Aubry, M.Larborn, N. Voros, S. Batistos, S. Kostavasili, “End users requirements specification,” IST-1999-10069, Report D1/PSA/WP1/V2.0, 2001.

[3] F. T. S. Gérard, E. Palachi, P. Pettersson, D. Alexandre, “UML profile for real time embedded systems development,” IST-1999-10069, public document 2001.

[4] S. Gérard, “Modélisation UML exécutable pour les systèmes embarqués de l'automobile,” in *GLSP*. Paris: Evry, 00.

[5] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modelling and Design*: Prentice Hall, 91.

[6] I. Jacobson, M. Christerson, P. Jonson, and G. Övergaard, *Object-Oriented Software Engineering : A Use Case Driven Approach*, 92.

[7] B. P. Douglass, *Doing Hard Time : Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns*: Addison Wesley, 99.

[8] M. Broy, “Requirements Engineering for Embedded Systems,” presented at FemSys'97, 1997.

[9] P. Desfray, *Modélisation par objets : la fin de la programmation*, MASSON ed, 96.

[10] I. Jacobson, “Formalizing use-case modeling,” *JOOP*, 95.

[11] J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language Reference Manual*: Addison-Wesley, 99.

[12] S. Gérard, F. Terrier, J.-L. Roux, I. Ober, E. Palachi, D. Alexandre, and U. Brockmeyer, “UML notations and semantics for real time embedded systems,” AIT-WOODDES, Paris, Internal Report 01/10/2000 2000.

[13] F. Terrier, G. Fouquier, D. Bras, L. Rioux, P. Vanuxeem, and A. Lanusse, “A Real Time Object Model,” presented at TOOLS Europe'96, Paris, France, 96b.

[14] B. Liskov, “Distributed Programming in Argus,” *Communication of ACM*, vol. 31, pp. 300-312, 1988.

[15] O. M. Nierstratz, “Active Objects in Hybrid,” presented at OOPSLA'87, 1987.

[16] R. Chandra, A. Gupta, and J. Hennessy, “COOL: An Object-Based Language for Parallel Programming,” *IEEE Computer*, vol. 27, 1994.

[17] F. Terrier, D. Bras, G. Fouquier, L. Rioux, and P. Vanuxeem, “Objets temps réel et ordonnancement par échéance,” presented at Fourth Conference on Real-Time Systems and Embedded Systems (RTS&ES'96), Paris, France, 96a.

[18] B. Selic, G. Gullekson, and P. T. Ward, *Real time Object-oriented Modeling*: John Wiley & Sons, Inc., 94.

[19] D. Harel, “Statecharts : A Visual Formalism for Complex Systems.,” *Science of Computer Programming*, vol. 8, pp. p231-274, 87a.

[20] OMG, “Unified Modeling Language Specification (version 1.4),” OMG, Request for proposal formal/01-09-67, September 2001 00.

[21] A. Tanenbaum, *Systèmes d'exploitation, systèmes centralisés, systèmes distribués*, 1994.

[22] P. Pushner and C. Koza, “Calculating the maximum execution time of real-time programs,” *Journal of real-time systems*, vol. 1, pp. 159-176, 1989.

[23] J. P. Babau, S. Gérard, and F. Cottet, “Méthodologie de mesure de durée d'exécution de tâche d'une application temps réel à contraintes strictes,” presented at RTS'96, Paris, 1996.

[24] G. Booch, *Object Oriented Analysis and Design with Applications*, second edition ed. Redwood City, CA, 1994.

[25] I. Krüger, R. Grosu, P. Scholz, and M. Broy, “From MSCs to Statecharts,” presented at Distributed and Parallel Embedded Systems, 99.

[26] ITU-T, “Recommendation Z.120 : Message Sequence Chart (MSC),” ITU-T, Geneva 96c.

[27] W. Damm and D. Harel, “LSCs: Breathing Life into Message Sequence Charts,” Weizmann Institute Report CS98-09, April 1998 98.