Project Number:33826

CREDO

Modeling and Analysis of Evolutionary Structures for Distributed Services

Deliverable D6.3
Final Modelling

Due Date: 01-07-2009
Submission Date: 31-06-2009

Start date of project: 01-09-2006 **Duration:** 3 years

Lead Participant: Almende **Revision:** Draft

Project funded by the European Commission within the Sixth Framework Programme (2002-2006)
Dissemination Level: PU Public

Project Participants

Role	No	Name	Acronym	Country
CO	1	Stichting Centrum voor Wiskunde en Informatica	CWI	NL
CR	2	Universitetet i Oslo	UIO	N
CR	3	Christian-Albrechts-Universität zu Kiel	CAU	DE
CR	4	Dresden University of Technology	TUD	DE
CR	5	Uppsala Universitet	UU	S
CR	6	United Nations University, International Institute for Software and Technology	UNU-IIST	JP
CR	7	Almende B. V.	ALMENDE	NL
CR	8	Rikshospitalet - Radiumhospitalet HF	RRHF	N
CR	9	Norsk Regnesentral	NR	N

C0 = Coordinator   CR = Contractor
NL = Netherlands   N = Norway
DE = Germany   S = Sweden
JP = Japan

Principal Contributors:

Names	Affiliation
Wolfgang Leister	NR
Xuedong Liang	RRHF
Andries Stam	Almende
Sascha Klüppelholz	TUD
Mahdi Jaghoori	CWI

Contents

1 Introduction **5**

2 Case study 1: The ASK System **6**
2.1 Final Models 7
2.2 Relation with Requirements 16
2.3 Lessons Learnt and Conclusion 17

3 Case study 2: Biomedical sensor networks **19**
3.1 Final Models 19
3.2 Relation with Requirements 21
3.3 Lessons Learnt and Conclusion 22

1 Introduction

In this deliverable, the final models for the case study systems ASK and BSN are presented. As far as applicable, we have used the CREDO methodology to structure the final modeling phase: The Creol language has been used to model parts of the systems in terms of object-oriented Creol components, while Reo and constraint automata have been used for modeling the networks. The resulting models can be used directly in the forthcoming validation phase of the project (D6.4), to validate whether the CREDO modeling, simulation, verification and analysis tools meet the user driven requirements identified (See deliverable D6.1 and the Methodology Document).

Section 2 is devoted to the ASK system. We explain what has been modeled and why, briefly present the final models (more details on them can be found in the technical annexes provided with this deliverable), and comment on the modeling trajectory and lessons learnt. In Section 3, we present the final models for the BSN case study, which focus on different aspects of a biomedical sensor network. We present a selection of models related to BSNbriefly, while the details for each model are presented in Annex D6.3.3. Then we relate these models to the requirements, and conclude with lessons learnt for this case study.

Live-CD

All models presented in Deliverable D6.3 and its annexes can be found on the latest version of the *Credo* Live-CD. The Live-CD contains the tools and the documentation necessary to view, execute and/or analyze the models.

2 Case study 1: The ASK System

Within the ASK case study, the final modeling exercise serves three different purposes. Firstly, we gain more insight into the complexity of creating abstract *Credo* models of concrete software. Secondly, we assess the quality and usefulness of the basic *Credo* languages (Creol and REO) and the *Credo* methodology. Finally, the final models can directly be used for the testing and validation of the *Credo* tool suite (Deliverable 6.4).

Overview of the ASK System

In order for the reader to understand the ASK final models and the notions used in them, we first provide a short revised overview of the ASK system.

As we explained in Deliverable D6.2, the ASK system can be technically divided into three parts: the *web front-end*, the *database* and the *contact engine* (see Figure 1). The *web front-end* acts as a configuration dashboard, via which typical domain data like users, groups, phone numbers, mail addresses, interactive voice response menus, services and scheduled jobs can be created, edited and deleted. This data is stored in a *database*, one for each configuration of ASK. The feedback of users and the knowledge derived from earlier established contacts are also stored in this database. Finally, the *contact engine* consists of a quintuple of components *Reception*, *Matcher*, *Executer*, *Resource Manager* and *Scheduler*, which handle inbound and outbound communication with the system and provide the intelligent matching and scheduling functionality.

The "heartbeat" of the contact engine is the *Request loop*, indicated with thick arrows. Requests loop through the system until they are fully completed. The *Reception* component determines which steps must be taken by ASK in order to fulfil (part of) a request. The *Matcher* component searches for appropriate participants for a request. The *Executer* component determines the best way in which the participants can be connected. ASK clearly separates the medium and resource independent request loop from the level of media-specific resources needed for fulfilling the request, called *connectoids* (e.g., a connected phone line, a sound file being played, an email being written, an SMS message to be sent). The *Resource Manager* component acts as a bridge between these two levels. Finally, a separate *Scheduler* component schedules requests based on job descriptions in the database.

Each of the components in the *contact engine* of ASK is able to handle multiple requests "at the same time", i.e. in an interleaved manner, as to adequately support the real-time communication taking place via ASK. To this end, each component implements a *thread-pool*. The threads in each thread-pool execute *tasks* from a *task queue*, which correspond with e.g. the handling of an incoming request or the execution of an outbound telephone call.

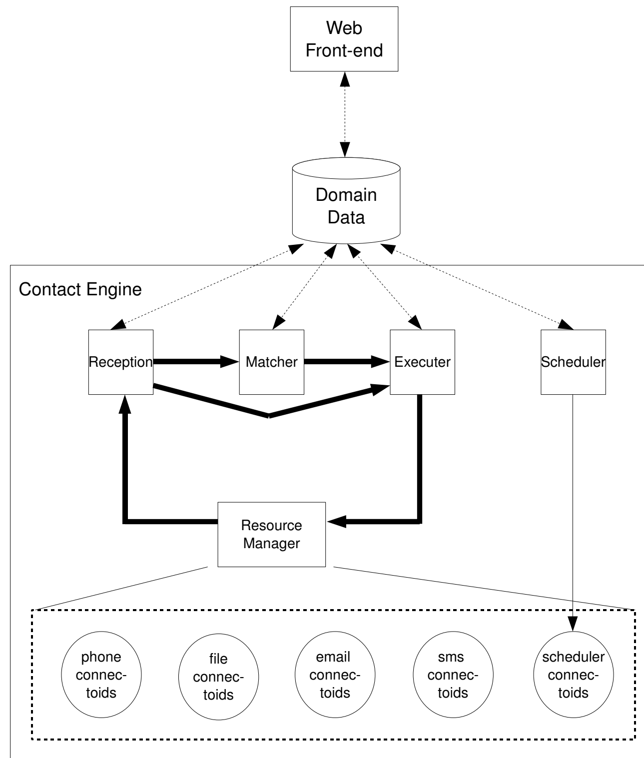


Figure 1: ASK System Overview

Section Organization

The setup of the rest of this section is as follows. We present an overview of the final models in Section 2.1. After that, we indicate the relationship between the final modeling effort and the requirements set at the beginning of the project in Section 2.2. Finally, we provide lessons learnt and draw conclusions in Section 2.3.

2.1 Final Models

We have split the final modeling effort for Case Study 1 (ASK) into three "subprojects": *ReASK*, *CreASK* and *tASK*.

In the *ReASK* project, which builds upon the initial modeling presented earlier in Deliverable D6.2, we have focused on modeling the ASK system in terms of a REO network at various levels of abstraction. At the lowest level of abstraction, we modeled the components in terms of automata, which resemble the initial model of ASK. The REO network and constraint automata have been straightforwardly converted into RSL and CARML specifications for combined verification with the Vereofy tool (see Deliverable D5.3).

In the *CreASK* project, we have focused on a specific part of the ASK core system, namely the *thread-pools*, which are present in each of the components of the system. We have used Creol to create object-oriented models of these thread-pools at two different levels of “declarativeness”: in the first attempts to create models, we sticked heavily to the implementation level and represented each implementation construct explicitly in Creol. After careful analysis, we were able to abstract away from the implementation level and to really use Creol as a *modeling* language, to create a precise but abstract model of the ASK thread-pools. An interesting exercise for the validation phase is to verify whether the more abstract model is indeed a valid abstraction of the more concrete model. With the *Credo* testing tool suite, we plan to validate the Creol models against the real implementations in the ASK system.

Finally, in the *tASK* project, we have used the Creol models of the thread-pools as a basis for timed automata models in UPPAAL. These latter models can be used to assess the schedulability of a particular amount of tasks with strict deadlines.

2.1.1 *ReASK*: REO Models of ASK

The entire set of REO models for the ASK system can be found in Annex D6.3.2.

The REO Models serve as a vehicle to assess *reconfigurability* of the ASK system. By modeling the system in terms of REO circuits or networks, we are forced to think about the system components in a purely compositional manner and with pure exogenous coordination. By the modeling exercise itself, the appropriate “building blocks” for reconfiguration almost automatically pop up in the modeler’s mind.

REO Networks The most difficult part of the modeling exercise was to identify useful levels of abstraction. In the end, we found that the *implementation* of ASK provides the right abstractions (we restricted ourselves to the ASK core, the contact engine). We identified the following four levels of abstraction, of which we used the upper three levels to model the ASK system.

- At the highest level of abstraction, the *Context Level*, we model the system and components in their context (e.g. the database, the communication with the outside world) as system parts within a small network.
- At a lower level, the *System Level*, we model the system as a network which connects the five ASK components (*executables* or *processes*) to each other and to the outer world (in terms of ports exposed by the system).
- A lower level, the *Process Level*, focuses on the organization within each component. At this level, the network connects *threads* or *shared variables* to each other, and the thread-pools within ASK are modeled explicitly.

- At the *Thread Level*, we zoom in onto a single thread within a process. Now, control flow is purely sequential: the network implements low-level control statements like if-then and while-do, connecting *C functions* to each other.

- A lower *Function Level* is possible, zooming in on a specific function. This level can even be applied recursively. In our model, we did not use this level of abstraction.

We show two examples of REO networks. In Figure 2, a network at the System Level is shown. Connections to the outer world are shown at the border of the system: the database (DBIn, DBOut) and the Asterisk IAX telephony communication channels (IAXIn, IAXOut). The five components in the system are connected to each other by means of asynchronous channels between ports. This highest level already shows possibilities for reconfiguration, like replication of the Resource Manager, the creation of a single component which replaces the Reception, Matcher and Executer, etc.

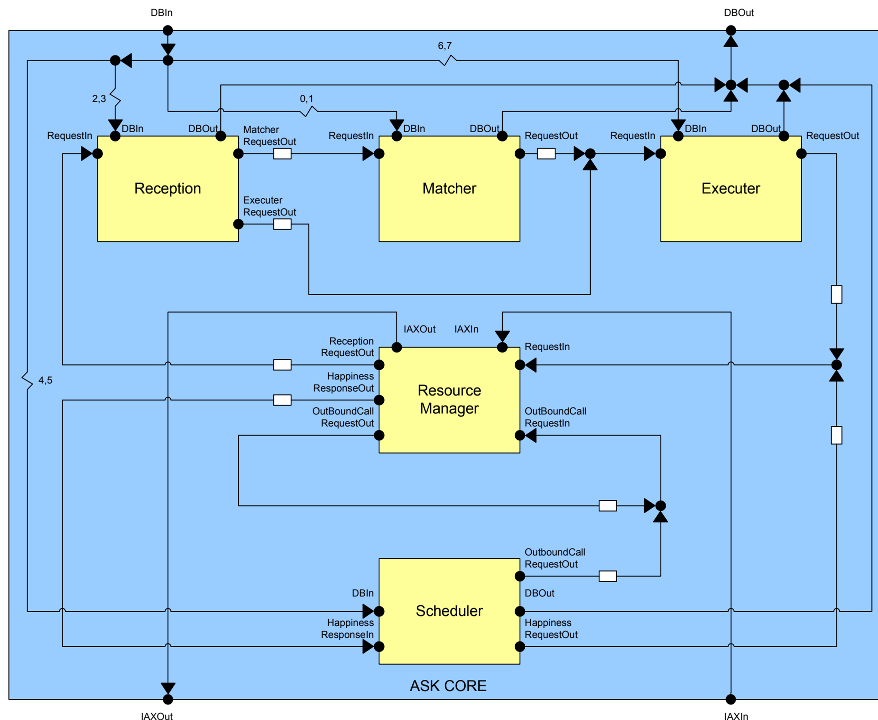


Figure 2: A REO network for the ASK System

A lower level of abstraction is shown in Figure 3. Here, the internals of the *Scheduler* component are depicted. Within the Scheduler, individual threads are shown: *Main* and *SchedulerMonk(1-n)*. In addition, shared variables are explicitly represented: *Job Queue* and *Happiness Value* (details about their meaning can be found in Annex D6.3.2). Note that the entire Scheduler component can be directly combined with the highest-level model (the ports are compatible).

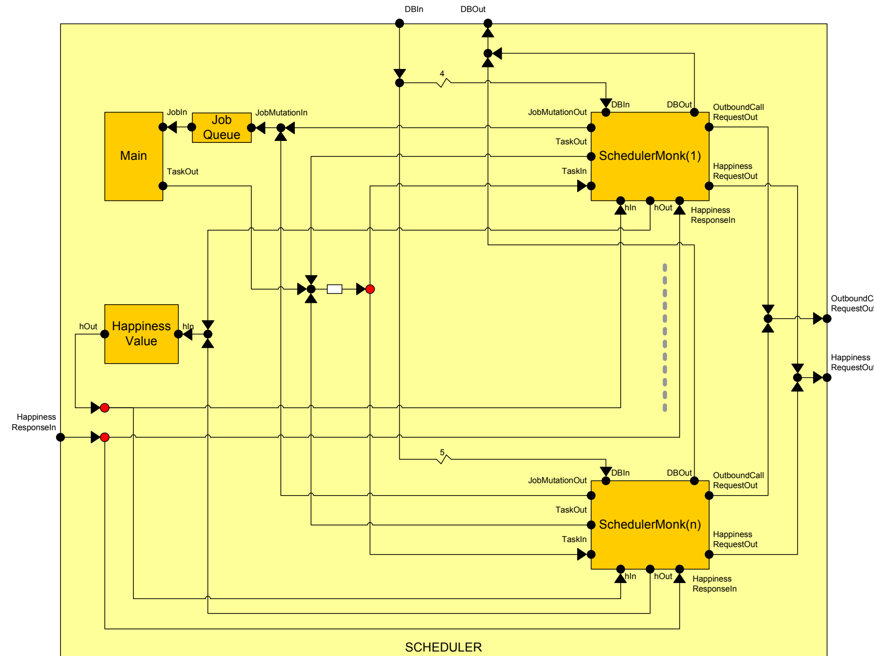


Figure 3: A REO network for the ASK Scheduler component

All REO networks for the ASK system created in the *ReASK* project have been drawn with a general-purpose drawing tool, instead of with the Eclipse Coordination Tools Plugin and REO network editor. The primary reason for this is that the current REO editor does not yet adequately support top-down modeling – it is currently not possible to draw networks which contain abstract component "types" for which no refined representation exists. Future versions of the editor should support top-down modeling, since we are convinced that this is a common and useful way of working, in line with the *Credo* methodology. In addition, the drawing of complex REO networks is a tedious and error-prone task.

Automata During the final modeling phase, we created the REO networks from top to bottom, starting at the highest level of abstraction, continuously refining "black boxes" in the models until we reach a level of abstraction which is detailed enough for assessing reconfiguration. At the lowest level of abstraction, we still keep black boxes, though. For these black boxes, we created abstract *behavioral* models in terms of automata. An example is given in Figure 4, showing an *AsteriskChannel* thread inside the Resource Manager component. As can be seen, the automaton specifies in what order reads and writes on ports happen, and with which values. We created similar models for the *Drivers* of the model: the external entities which send and receive values to and from ports *DBIn, DBOut, IAXIn and IAXOut.*

In Vereofy, these abstract behavioral models can be readily combined with the REO networks to form an *executable model* of the ASK system. To this end, the REO networks and automata have been converted into RSL and CARML specifications. It turned out that this could be done straightforwardly. The only important and more complicated issue was to establish of a useful and abstract enough *Data Domain*: a specification of the range of values which may be passed through the channels. The size of the data domain heavily determines the size of the state space for the combined executable model. Also note that the conversion to CARML and RSL had to be done completely manually, which is in itself error-prone.

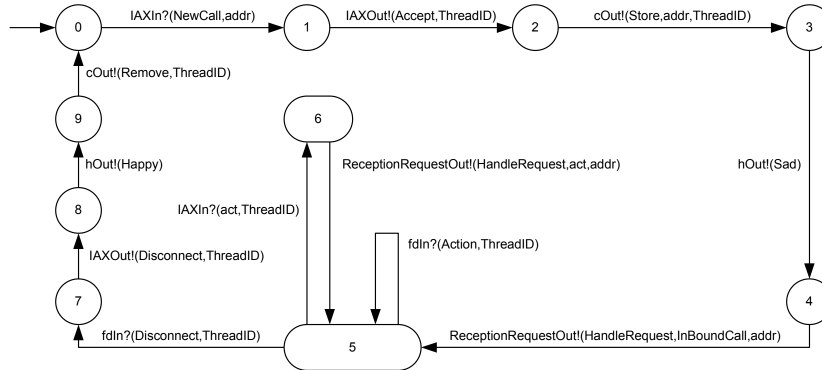


Figure 4: An automaton for the ASK Resource Manager AsteriskChannel

2.1.2 *CreASK*: Creol Models of Thread-Pools in ASK

All Creol models for the ASK thread-pools can be found in Annex D6.3.1.

The Creol Models serve as a vehicle to *analyze the differences* between various types of thread-pools. By choosing a sufficient abstract level of modeling, the essence of these differences becomes apparent and can be studied separately from implementation-level distractions. Furthermore, the *Credo* testing tool suite can be used to verify, to a certain extent, whether the implementation of a thread-pool behaves according to its Creol model. At the same time, an abstract Creol model of a thread-pool can be used as the basis for a (manually created) corresponding UPPAAL model of that thread-pool, with which schedulability analysis is possible (see Section 2.1.3.

Types of Thread-Pools in ASK In the ASK system, thread-pools are called *abbeys*, the threads within the pool are called *monks*. Two types of abbeys are currently in use, although many more have been created in the past at Almende:

- The so-called *Determinate Abbey* (Dabbey) uses a fixed amount of monks, which get their tasks from a task array with an amount of "slots" equal to the number of monks. The operation to put a task in an empty slot in the task array blocks if no empty slot is available.
- Another type of abbey is the *Self-scaling Abbey* (Sabbey). This abbey uses an infinite task queue and a variable amount of monks. Monks are created and "poisoned" at run-time by a special monk called the *shepherd*, which does so by keeping track of the ratio between the amount of tasks to be handled and the amount of available monks.

Creol Models at a Low Level of Abstraction We started our modeling exercise with creating models as if we were "programming" in Creol: our first attempts to create Creol models of the two types of abbeys were heavily related to implementation-specific (distracting) issues, like locks on global variables, explicit tasks and explicit task queues, etc. Figure 5 shows a UML class diagram of this Low-Level Creol model for the Determinate Abbey. Tasks and monks are kept in explicit lists, and tasks are explicitly implemented. An array of tasks is "mimicked" in Creol by using a list, an index for the list and a replace method to replace values at a specific index in the list. The task list also contains methods which correspond to primitive "test-and-set" operations.

Clearly, this Creol model does not abstract away from the implementation. As such, it is not suitable for analyzing the essential differences between various types of thread-pool, although it could serve as a reference model for a concrete implementation in a programming language.

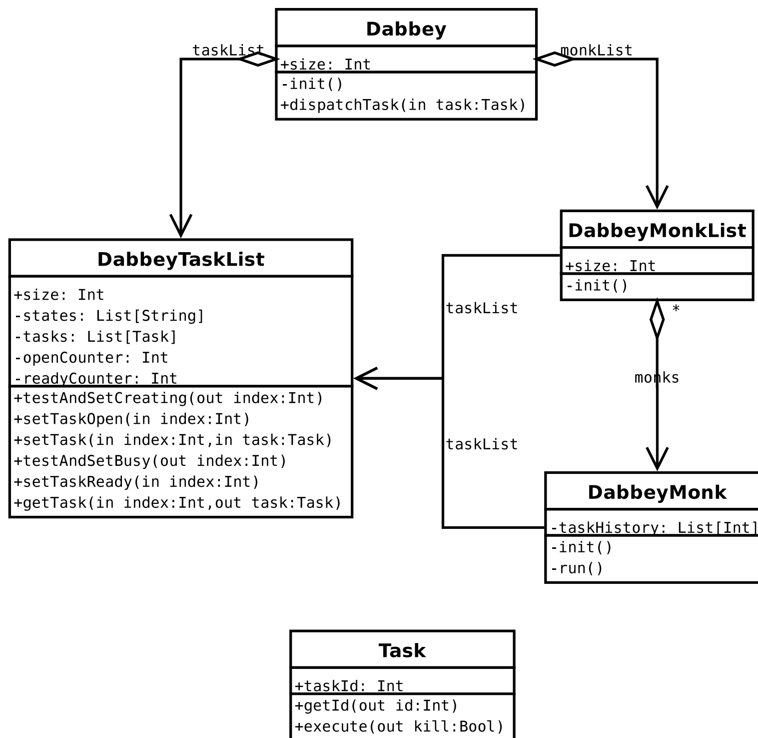


Figure 5: Class Diagram of the Low-Level Determinate Abbey in Creol

Creol Models at a High Level of Abstraction Throughout the modeling exercise, we learnt how to use Creol as a real *modeling* language and how to exploit the characteristics of its underlying execution and messaging model. This resulted in considerably smaller and cleaner models at a higher level of abstraction, better suited for analysis. The UML class diagram of the High-Level Creol model is shown in Figure 6. Only the Dabbey and the monks are modeled as explicit classes. The task queue (or task array) is implicitly modeled, by exploiting the fact that each Creol object contains a *message queue*. The size of this queue can be limited by means of a class variable *nofTasks* which represents the number of tasks currently in the task queue. We have abstracted away from the actual execution of tasks.

To give the reader an idea of the size of the models compared to the size of the implementation, it is interesting to mention the amount of LoC (lines of code) for the implementation of the Determinate Abbey, its low-level and its high-level Creol model. The implementation in the ASK system consists of 188 lines of C code, header files included. The size of the low-level Creol model, including interfaces, is 178 lines, while the high-level Creol model consists of only 64 lines (including interfaces). We also managed to create a *Minimal Abbey* (Mabbey) specification, as the "mother" of all different versions of the abbey, consisting of only 50 lines of Creol code for both classes and interfaces.

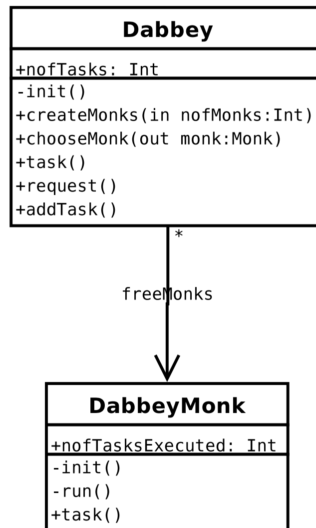


Figure 6: Class Diagram of the Abstract Determinate Abbey in Creol

2.1.3 *tASK*: UPPAAL Models of Thread-Pools in ASK

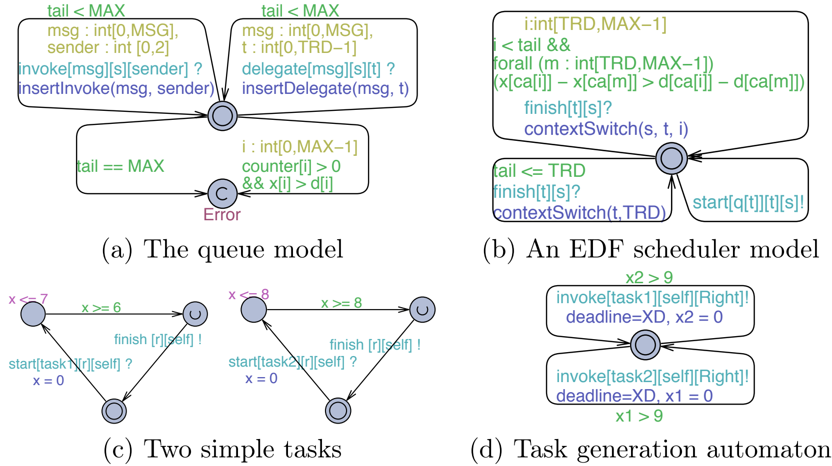
To model a thread-pool, we can take two approaches. At a higher level of abstraction, we can assume that the threads run in parallel as if each has its own processing unit. We can alternatively model a time-sharing scheduling policy where the running threads run a period of time each before they are interrupted by the scheduler to run the next one.

To model a thread pool, we separate the task queue in two parts. The first part is as big as the number of threads and includes the tasks that are being executed. But before execution, tasks can be queued based on different scheduling strategies, e.g., EDF, FPS, etc. in the rest of the queue (called the buffer part below). When a task reaches the execution part, it will not be put back to the buffer part.

Parallel Threads. This model is more accurate when we can rely on the fact that the real system will run on a multi-core CPU and each thread will in fact run in parallel to the others. It would also be a good approximation when the execution time for tasks is obtained by profiling the real system. The reason is that we get a (mean) execution time for each task as if it was the only process.

Figure 7 shows the model of a scheduler where a task queue is shared between parallel processing units. In this model, the queue and the scheduling strategy are modeled separately. Part (a) shows the queue which stores the tasks as they arrive, i.e., in a FIFO order. Part (b) models the scheduling strategy and should be replicated for every processing unit. The different instances of this automaton will be assigned each to one slot in the queue.

In this approach, one can model tasks as timed automata; two simple task models are given in Figure 7(c). More complicated models can include sub-task generation. Finally a model of task generation pattern (by the environment) is given in part (d) of this figure.



(a) The queue model

(b) An EDF scheduler model

(c) Two simple tasks

(d) Task generation automaton

Figure 7: A queue shared between parallel threads

Time-Sharing. In this model, tasks that are executing share the same CPU and get a time slot (called a *quantum*) for execution. Then the running thread is preempted to give a chance to the next thread for execution. Note that before entering the execution part of the queue, no preemption can occur, i.e., once a task is in the execution part it cannot be put back into the buffer part.

In this model, each task is modeled only as a computation time. This simplification is necessary to enable the modeling of preemption of tasks at any arbitrary time (i.e., the selected quantum). Figure 8 shows the model the queue combined with the scheduler (on the left). The scheduling policy can be modeled in the `insertInvoke` function. In this model, the deadline or priority values for tasks can be modeled statically. In this figure, on the right, a model of task generation pattern is given, in which the computation and deadline values for tasks are also given.

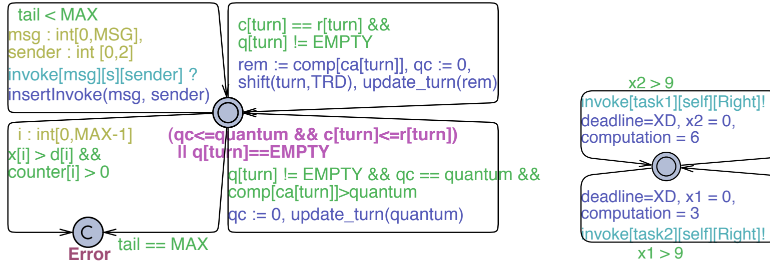


Figure 8: A time-sharing queue and task arrival automata

2.2 Relation with Requirements

In the Addendum on Deliverable D6.1 (Requirements), we formulated the following requirement categories:

1. *Credo* should support the modeling of ASK *structure*, ASK *behavior*, *location* and *time*;
2. *Credo* should support the modeling of constraints on *time*, *memory size* and *network bandwidth*;
3. It must be possible to verify *functional* properties of a reconfigurable system based on a *Credo* model created for it;
4. It must be possible to verify *non-functional* properties of a reconfigurable system based on a *Credo* model created for it;
5. The *Credo* tools must be applicable to individual components as well as compositions of components;
6. The *Credo* tools must provide ways to (semi-)automatically create models based on source code of the ASK system;
7. The *Credo* tools must be able to verify non-functional properties at runtime;

8. The *Credo* tools must provide information about functional and non-functional properties in an attractive visual manner;

Categories 1, 2, 5, 6 and 8 are covered by the final modeling. We will discuss whether these requirements are met in Section 2.3. The assessment of requirement categories 3, 4 and 7 is postponed until after validation (Deliverable 6.4).

2.3 Lessons Learnt and Conclusion

The subprojects within the final modeling phase of the *Credo* project have gained the following insights:

ReASK. The REO network modeling language is very suitable for assessing reconfigurability at several levels of abstraction, especially due to its drive towards *compositionality*. Implementation concepts from the ASK system proved useful to distinguish between multiple levels of abstraction in a REO model. The current version of the Eclipse REO editor does not adequately support top-down modeling as we did in this subproject. Furthermore, conversion to CARML and RSL for verification with Vereofy currently needs to be done completely manually.

CreASK. The Creol language can be easily "abused" as an object-oriented programming language. It takes some time to get acquainted with its underlying execution model and language concepts based on this model. However, if used appropriately, Creol is a very powerful instrument to model functional aspects of software systems at a level of abstraction suitable for analysis and assessment of alternatives. The current versions of the Creol compiler and execution platform satisfied our requirements with regard to the size of the models that can be handled.

tASK. Creol models can be used as a starting point for timed automata in the UPPAAL tool. This opens up the way towards the assessment of non-functional (timing and schedulability) properties of Creol models.

Assessment of Initial Requirements

Insights gained throughout the project make it quite difficult to use the initial requirement categories (RCs) as a validation framework for the project results.

RC1. As we already concluded in our report on the Initial Modeling (Deliverable D6.2), *Credo* supports the modeling of structural and behavioral aspects of software systems, as well as of timing aspects. Location as such cannot be explicitly represented in any of the *Credo* modeling languages.

RC2. Constraints on time can be modeled and analyzed in the UPPAAL tool. However, the *Credo* tools are less suited for modeling constraints on memory size or network bandwidth, in particular any *resource constraint*.

RC5. The compositionality of especially REO makes the *Credo* language applicable to individual components as well as component compositions.

RC6. *Credo* definitely does *not* support (semi-)automatic creation of models based on source code of ASK, a requirement we considered as a necessity for broader adoption of the *Credo* tool suite, but which proved to be infeasible within the resource and time limits of the *Credo* project.

RC8. Finally, the attractive visual manner in which models can be represented was not per se within scope.

3 Case study 2: Biomedical sensor networks

Based on the generic architecture of a biomedical sensor network (BSN) presented in previous deliverables D6.1, D6.1 Addendum, and D6.2 we modelled several aspects and level of detail using the Credo tools, namely in Creol (including several extensions), in UPPAAL, and in Vereofy, following the Credo Methodology Document [3]. In this section we give an overview of the models, relate these models to the requirements document (D6.1 Addendum), and present experiences while working with the models.

3.1 Final Models

The models presented here focus on different aspects of a BSN using different selection of modelled layers in the communication stack. While the models for flooding and the AODV routing algorithm focus entirely on aspects of the Network Layer, we also developed models featuring aspects of the Link Layer. We selected data forwarding and routing strategies to be modelled in the Credo tools, because these models address the behaviour of distributed algorithms in dynamic structures in a intuitive way.

In the following we present the models (I) to (VIII). The in-depth descriptions of these models can be found in Annex D6.3.3 of this document. For a description of Case Study 2 we refer to Deliverables D6.1 and D6.1 Addendum.

Modelling BSN using timed automata (I). We modelled a BSN using timed automata [4], where the sensor nodes communicate using the Chipcon CC2420 transceiver according to the IEEE 802.15.4 standard. Based on the model, we have used UPPAAL to validate and tune the temporal configuration parameters of a BSN in order to meet desired QoS requirements on network connectivity, packet delivery ratio and end-to-end delay. The network studied allows dynamic reconfigurations of the network topology due to the temporally switching of sensor nodes to power-down mode for energy-saving or their physical movements. Both the simulator and model-checker of UPPAAL are used to analyse the average-case and worst-case behaviours. To enhance the scalability of the tool, we have implemented a version of the UPPAAL simulator optimised for exploring symbolic traces of automata containing large data structures such as matrices. Our experiments show that even though the main feature of the tool is model checking, it is also a promising and competitive tool for efficient simulation and parameter tuning. The simulator scales well; it can easily handle up to 50 nodes in our experiments. The model checker installed on a notebook can also deal with networks with 5 up to 16 nodes within minutes depending on the properties checked; these are BSNs of reasonable size for medical applications.

Wireless Creol (II). Many distributed applications can be understood in terms of components interacting in an open environment. This interaction can

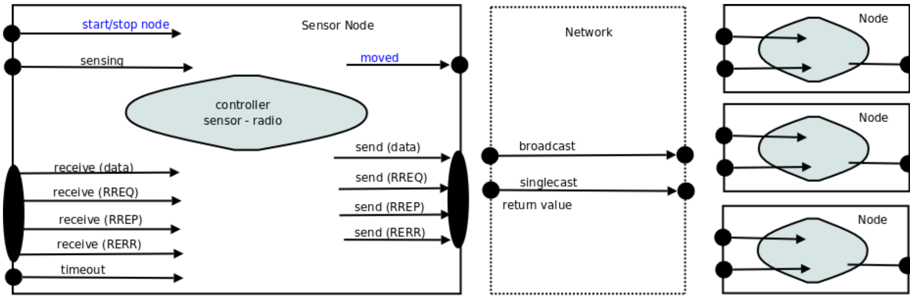


Figure 9: Interfaces for AODV

use components that are tightly connected with order preservation of communicated messages, whereas others are more loosely connected not guaranteeing order preservation, or allowing message loss. Components of a BSN may communicate over wireless networks, where sending and receiving must be synchronised, since the wireless medium cannot buffer messages. We proposed a formal framework for such systems, which allows high-level modelling and formal analysis of distributed systems [1], and introduced this as an extension to Creol.

We introduce a light-weight notion of multi-object network components, where the objects inside a component are tightly connected and communicate directly with each other. Primitives are added for broadcast communication to objects supporting a given interface. An operational semantics for the language is defined in rewriting logic, which directly provides an executable implementation in Maude. We applied this framework to a small wireless sensor network, that is modelled to be evolving, which means that connections may be broken, and new ones may appear. The executable program representing this whole system consists of approximately 70 lines of Creol code.

Modelling of the flooding strategy in Creol (III). We modelled the flooding strategy to forward BSN data in Creol by implementing the nodes and the network as objects. The *Network* implements a broadcast method, while the *Node* class implements methods to *sense*, *receive*, and *send* data. The implementation of this model gave valuable input for establishing a "best practice" for modelling in Creol.

Extended model of the flooding strategy in Creol (IV). We modelled an extended model in Creol implementing distance between the nodes as well as power consumption.

Modelling of AODV in Creol (V). We extended the Creol flooding model to include the mechanisms of the AODV [2] routing protocol. When trying to find routes to a sink node, the AODV protocol broadcasts so-called RREQ

messages, and gets singlecast RREP messages as replies, thus being able to build up routing tables in the nodes. The flooding model had to be extended by the AODV functionality and a suitable representation of messages of different kinds (e.g., payload, RREQ, RREP, RERR). We used multiple inheritance to add the functionality for the routing tables, and caches, while we used behaviourless objects to represent the messages.

The implementation of the model was done from a practical point of view, i.e., the representation of the data was done like in some real implementations, only abstracting away technical details that do not contribute to evaluate the properties. We managed to implement a model of an AODV-based network that can be run in the interpreter mode. However, due to several design choices, and technical limitations in the current version of Creol the model got quite lengthy with about 1500 lines of code. Note that the current implementation addresses both flooding and AODV in one model.

A simplified AODV model (VI). We also modelled an alternative, simplified model of AODV using Creol and compared it with the Promela code of the Spin model checker. This model is about 250 lines of code long in both variations.

Modelling Flooding (VII) and AODV in Vereofy (VIII). We used Vereofy and its input languages CARML and RSL to provide a model for both, the flooding and the AODV routing protocol. The formal semantics of the input relies on constraint automata and thus the model describes the behaviour of the sensor nodes and the network at the interface level. The specification of the interface behaviour of a sensor node is given in terms of CARML sub-modules for sensing, receiving and sending. For unicast and broadcast the communication media have been modelled as dynamic component connector networks composed with the help of RSL.

3.2 Relation with Requirements

The presented models of a BSN follow the prioritised requirements S-1 to S8 of the Deliverable D6.1 Addendum. However, our models are to some extent more abstract, since we do not specifically consider the hospital-specific requirements, such as number of patients (S-1), statistical model of noise (S-6), software update features (S-7), and the infrastructure part of the hospital (S-8). The implemented models concentrate on supporting various number of hops and topologies (S-2); propagation of data from the source to one or several sink nodes (S-3) with both variants; dynamic change of topology (S-4); and dynamic adding and removing of nodes (S-5).

We discuss in the following which of the above models address which of the prioritised required properties in Table 1 of Deliverable D6.1. We list in Table 1 (of the current document) the models that address the respective properties.

Some of the prioritised properties are addressed by many of our presented models, like packet delivery ration (ratio between sent and delivered packets

Property	**Model**
Timing, end-to-end delay	(I)
Packet delivery ratio	(I)—(VIII)
Network connectivity, deadlock, isolated node	(I)—(VIII)
Energy consumption	(IV), (I)
Memory and Buffer	(III), (IV), (V), (VII), (VIII)
Wireless channel, collisions, access failure	(I), (II), (VII), (VIII)
Mobility, forwarding, routing, topology changes	(III), (IV), (V), (VI), (VII), (VIII)
Interference, concurrent transmission	(I), (II), (V), (VII), (VIII)

Table 1: Properties related to models

is commonly implemented), network connectivity (all models address network connectivity to some extent) or mobility (we selected flooding and routing algorithms as a main theme of our research), while other properties are addressed specifically by one of the models developed for a specific one of the Credo-tools, like timing or energy consumption. While some of the properties are quite obviously addressed by a model, other properties are addressed more hidden. E.g., interferences in Model (V) are not addressed on the lower layer communication, but in the sense that the possible occurrence of interferences will cause indeterminism in the network layer since messages can arrive or be thrown away in the case of interferences.

3.3 Lessons Learnt and Conclusion

The chosen models have their origin in practical applications in sensor networks. For some of these models real-world implementations or simulation models already existed which we used in our work. Note that some of the real-world implementations of the AODV algorithm are of substantial length, and therefore we could only model the most essential parts of the algorithm. In some cases aspects the real world implementations contain elements that should be modelled stochastically, which is beyond the scope of Credo, and therefore has been left to other modelling tools. We found that algorithms implementing forwarding and routing are on a suitable level with regard to the Credo methodology.

Implementations of algorithms used as an inspiration for our models often use programming paradigms that do not fit to the Credo methodology. However, since the Credo tools offer programming constructs familiar to application programmers, the difference between modelling and programming sometimes is not in the mind of the developers of the models. In the absence of a best practice for modelling with the Credo tools this led to models that were not optimal, and sometimes other ways of modelling an aspect had to be found. In this way we are now about to establish a best practice of modelling with the Credo tools.

The work with modelling the BSN scenario also lead to numerous ideas being taken up and implemented in the Credo tools. Bugs in the implementations were addressed during modelling, limitations were pushed, and extensions of

the languages included, thus leading to a more powerful set of tools.

About modelling in Creol. While modelling we found it rather intuitive to start with modelling in Creol once the run-time system and compiler were installed on the computer. We found a reasonable selection of language constructs suitable for the modelling task. Modelling with Creol is quite close to application development, which sometimes makes the developers forget that they in fact are modelling. While modelling we also found several issues that could be improved from a practical point of view. We mention syntactical issues, absence of local scoping, missing support for multiple inheritance and type casting, and the need for data dictionaries (like `struct` in C) as issues that we came over.

About modelling in Vereofy. Some aspects are rather intuitive to transform into a model, while for others, e.g., the model of broadcast and singlecast, expert knowledge was necessary. Several missing language features were added during the modelling phase.

About modelling in UPPAAL. The C-alike modelling language made it intuitive to model timed automata, and use the integrated tool box. This tool is the most finished tool in the Credo tools family, and therefore a best practice already is established.

Conclusion. In the final modelling we developed models that address several properties of BSN. We concentrated on forwarding and routing aspects, and chose flooding and AODV as studies. These were modelled with a similar scope in several of the Credo tools so that we can compare these tools later in the verification phase to be presented in Deliverable D6.4.

Technical Annexes

D6.3.1 This technical annex presents the Creol models of the ASK system, in which the basic thread-pool architecture of each component is modeled.

D6.3.2 This technical annex presents the REO model of the ASK system, in terms of REO networks and automata.

D6.3.3 This technical annex shows the final modelling of the BSN case study, using both Creol, UPPAAL, and Vereofy.

D6.3.4 This technical annex compares AODV protocol model checking using Spin and Creol; in this document referred to as Model VI.

References

[1] E. B. Johnsen, O. Owe, J. Bjørk, and M. Kyas. An object-oriented component model for heterogeneous nets. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, editors, *FMCO*, volume 5382 of *Lecture Notes in Computer Science*, pages 257–279. Springer, 2007.

[2] C. Perkins, E. Belding-Royer, and S. Das. Ad hoc On-Demand Distance Vector (AODV) Routing. RFC 3561 (Experimental), July 2003.

[3] A. Salden, A. Stam, F. de Boer, I. Balasingham, X. Liang, M. Kyas, M. Steffen, W. Leister, and B. M. Østvold. The Credo methodology: An end-user perspective. Credo Deliverable, Apr. 2008.

[4] S. Tschirner, X. Liang, and W. Yi. Model-based validation of QoS properties of biomedical sensor networks. In *EMSOFT '08: Proceedings of the 7th ACM international conference on Embedded software*, pages 69–78, New York, NY, USA, 2008. ACM.