Project Number:33826

CREDO

Modeling and Analysis of Evolutionary Structures for Distributed Services

Deliverable D6.2
Initial modelling with service interfaces

Due Date: 14-04-2008
Submission Date: 14-04-2008

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

Lead Participant name here **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	NO
CR	3	Christian-Albrechts-Universität zu Kiel	CAU	DE
CR	4	Dresden University of Technology	TUD	DE
CR	5	Uppsala Universitet	UU	SE
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	NO
CR	9	Norsk Regnesentral	NR	NO

CO = Coordinator
CR = Contractor
NL = The Netherlands
NO = Norway
DE = Germany
SE = Sweden
JP = Japan

Document History

Principal Contributors:

Names	Affiliation
Alfons Salden	Almende
Andries Stam	Almende
Tom Chothia	CWI
Wolfgang Leister	NR
Bjarte M. Østvold	NR
Xuedong Liang	RRHF
Marcel Kyas	UIO

Contents

1 Introduction

In this deliverable, the case study systems ASK CS and BSN are modeled in terms of service interfaces. The deliverable forms input to further model those systems as object-oriented Creol components. Therewith, the case partners obtain high-level and refined reference models of both systems, which can ultimately be used to validate whether the CREDO simulation, verification and performance analysis tools meet the user driven requirements identified (See deliverable D6.1 and the Methodology Document). Furthermore, these models help case partners to up to some degree reliably adapt and extend their systems.

In Section 2, the high-level architecture of ASK CS is explained and initial models for part of the system are presented in terms of network automata (see Deliverable D1.2). The initial models will specifically be used in the final modeling (D6.3) to analyze the exploitation of meta-information in the ASK system.

Section 3 presents a generic architecture for biomedical sensor nodes. This is followed by a model of biomedical sensor networks (BSNs), based on the architecture. The model uses Creol interfaces and timed automata. It is demonstrated that a model in terms of Reo is inapplicable for the BSN domain and interfaces of network automata become too large to handle efficiently. The BSN requirements concerning, e.g., real-time operation or resource availability, are represented as interface properties. A transceiver model and a wireless channel model have been developed in Uppaal. We have tested the models in 1-hop and multi-hop communication networks respectively.

2 Case study 1: ASK Community Systems

For a high-level introduction to the ASK CS functionality, we refer to the *Credo* Description of Work. In this document, we focus on the high-level technical and low-level functional aspects of the system.

2.1 Purpose and Context

The initial modeling of ASK CS serves three different purposes. Firstly, we gain insight into the complexity of creating abstract *Credo* models of the concrete software as to tune our expectations with regard to the final modeling. Secondly, we assess the quality and usefulness of the service interface specification language as proposed in Deliverable D1.2 "Specifying Service Interfaces". Finally, the developers of the *Credo* tools can use the ASK CS models for testing and requirement verification.

2.1.1 Initial Modeling Scope

The size and complexity of the ASK CS system as-a-whole makes it necessary to divide the entire modeling effort into multiple separately addressable parts, which each serve a clear purpose. Our starting point for this division can be found in the Methodology Document, where we formulated four individual case study scenarios. For the initial modeling of ASK CS, we restrict ourselves to Scenario **SC.1**: "better exploitation of meta-information within ASK CS". This scenario is centered around the communication of so-called *happiness information*, an indicator for the amount of free phone lines in ASK CS at a certain moment. This information is used inside ASK CS to decide if calls to users can be performed directly or should be postponed. A management strategy on the basis of happiness information has a positive influence on the completion times for so-called *availability jobs*, which are jobs carried out by ASK CS to automatically recruit a number of experts for a certain service in a certain time window.

In this document, we present the high-level architecture of ASK CS and the initial models for Scenario **SC.1** in terms of network automata with attached timing information. We plan to use these automata with the *Credo* tools in the near future to assess the effectiveness of several management strategies, by combining the resulting automata models into a single product automaton, and analyzing several timing properties of this product automaton with the probabilistic model checker PRISM. For the initial modeling, however, we restrict ourselves to the description of ASK CS and the modeling of individual network automata.

2.1.2 Relation with Requirements

Scenario **SC.1** addresses the following three requirement categories (see the Methodology Document):

- Requirement Category 1: The *Credo* model should support the modeling of ASK CS *structure* (component, process, thread (pool), task, request), ASK CS *behavior* (states / actions, communication, function calls), *location* (of structure) and *time* (of behavior);
- Requirement Category 4: It must be possible to verify *non-functional* properties of a reconfigurable system based on a *Credo* model created for it;
- Requirement Category 5: The *Credo* tools must be applicable to individual components as well as compositions of components.

The requirements of Category 1 are all addressed within the scope of the initial modeling, with the exception of the *location* concept. We focus on the structural concepts in Section 2.2, where we give a high-level overview of the ASK CS system, zoom in onto its core application components and model the structure parts relevant to Scenario **SC.1**. With regard to Category 4, the primary non-functional property addressed in **SC.1** will be task handling completion time, as we will explain in Section 2.6 where we provide a series of network automata for several parts of ASK CS. However, the requirements of Category 4 can only be assessed *after* the completion of the entire **SC.1** scenario, i.e. after the analysis planned for the final modeling. Finally, as we will show in Section 2.6 as well, Category 5 is addressed by requiring the possibility to create a single network automaton for an individual ASK CS component as well as for sets of combined ASK CS components. This requirement is already fully addressed within the scope of the initial modeling.

2.2 High-level Technical Architecture

The software of the ASK CS system can be technically divided into three parts: the *web front-end*, the *database* and the *core application* (see Figure 1).
The *web front-end* is an ingenious web interface via which nearly every aspect of the ASK CS system is manually configurable. This includes not only the domain entities of the system, like users, groups, phone numbers, mail addresses, IVR menus, services and scheduled jobs, etc., but also highly technical aspects, like the URL of an asterisk hub (i.e., where the phone card is attached to) or the port numbers of the core application components. Each configuration of the ASK CS system, which we call a running ASK CS *instance*, has a single monolithic *database*, in which almost all domain data and technical data is stored, including e.g. sound files (.wav). *System administrators* can access and edit a large part of the contents of the database via the web front-end: they configure the technical aspects of the system (which is often done only once and never changed afterwards), but also (more importantly) the domain information (users, groups, IVR menus, etc.). The latter contents are used by the *core application*, which consists of a quintuple of *components*, as visualized in Figure 1. The components act as *daemon processes*: after they have been started, the components run forever and together handle communication with

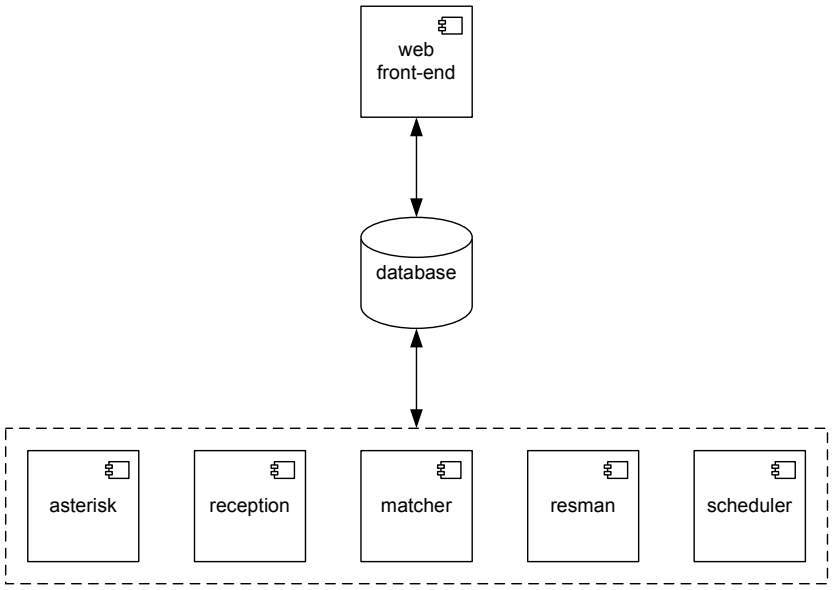

Figure 1: ASK CS System Overview

users by phone, email, or SMS. They make considerable use of the database, e.g. for identifying users, handling scheduled activities and storing IVR menu choices, but also for technical issues like retrieving port numbers of their fellow components.

2.3 Core Application Components

For the initial modeling scenario **SC.1**, an understanding of the workings of the core application components is of considerable importance: the exploitation of meta-information in ASK CS (the happiness information mentioned earlier) takes place inside these components. A technical overview of the five core application components is given in Figure 2. Each component is depicted together with its *tasks* (rounded rectangles), its *ports* (small squares with arrows and port numbers) and its *task triggers* (dashed arrows between tasks). We depict only those entities which are relevant for scenario **SC.1**).

In general, the application components work as follows. In fact, the components are executable *daemon processes*: once started up, they run forever. Starting up the ASK CS core application hence is equal to starting up each of the ASK CS components. The tasks in each component are executed by threads of the component's *thread pool*. Each component has a *task queue*, which contains the tasks waiting to be executed by a thread in the thread pool. A task is in fact the combination of a *function pointer* and a *pointer to a function argument*. Threads execute a task by calling the function pointer's function with the argument pointer's argument. There is a clear distinction between *finite tasks* (tasks

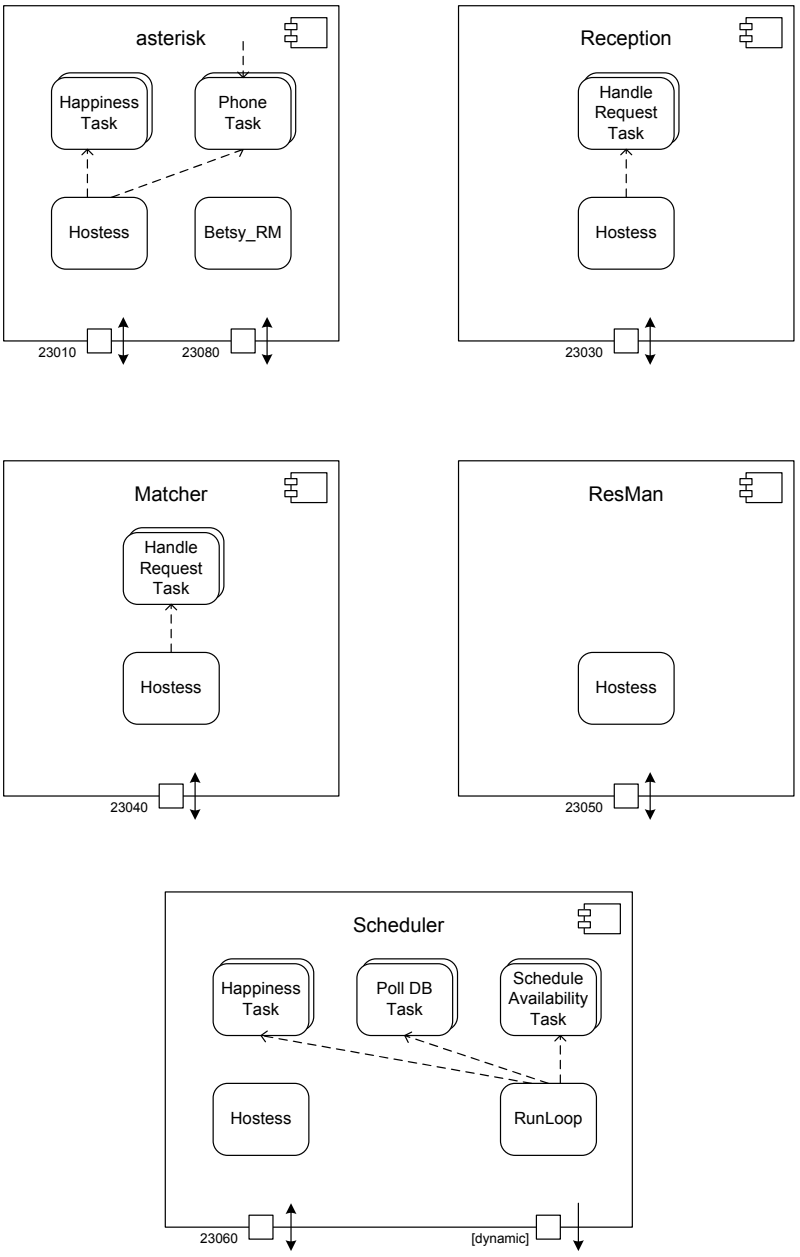

Figure 2: ASK CS Core Application Components

which are finished at some moment in time) and *infinite tasks* (tasks which run forever). The *finite* tasks are depicted in Figure 2 with a double rounded rectangle. All *infinite* tasks are started once the components are started up. The *task triggers* indicate which tasks are created by existing tasks, the direction of the arrows pointing at the tasks being created. Tasks can communicate with each other by either sending *outgoing requests* via a (local) *source port* to a (remote) *destination port*, or listening to a (local) port for *incoming requests*. This type of communication is performed in the ASK CS system via well-known standardized distributable socket communication. Tasks can also interact with the (MySQL) database by sending SQL queries to the MySQL server.

asterisk The asterisk component of ASK CS is mainly based on the open source Asterisk PBX, but tailored for purposes of integration with the four other components. The asterisk component manages telephone conversations, receives incoming calls and sets up outgoing calls.

Reception The major role of the Reception component is to determine which action should be taken by the ASK CS system based upon an incoming event. The incoming events are received as requests from the asterisk component. They can be of various types. To give an example, if a request is received containing an incoming call event from a certain telephone number, the Reception component can decide to present an IVR menu to the caller.

Matcher The role of the Matcher component for scenario **SC.1** is very limited: in this scenario, it merely forwards requests from the Reception component to the ResMan component.

ResMan The role of the ResMan component is *in general* very limited: next to some database logging, it merely forwards requests from the Reception or the Matcher component to the asterisk component.

Scheduler The Scheduler component carries out jobs which are entered by a system administrator in the database. The Scheduler component does not *schedule* any jobs: it *performs* jobs scheduled by someone else.

For Scenario **SC.1**, the only considered job is the so-called *availability job*, in which, at a certain moment in time, a group of users is called by the ASK CS system. Each user is asked via an IVR menu if he or she is available for a certain service. For example, suppose a post-office would like to recruit a group of six mailmen out of a known group of twenty for performing the delivery of mail during next saturday. In that case, the system administrator of ASK CS schedules a schedule availability job, stating that the Scheduler component must try to recruit six mailmen from the group of twenty known mailmen by calling them and presenting to them an IVR menu in which they are asked if they are able to deliver mail for next saturday. The called mailmen reply to the call of

ASK CS by pressing a DTMF number (e.g. 1 = I am available, 2 = I am not available).

Because the amount of phone lines is limited, it is not possible in general to call all users at the same time. Instead, the Scheduler component frequently asks the asterisk component for *happiness information*, an indicator for the amount of free phone lines, based on which it decides if a set of calls can be made or should be postponed. Currently, it is this type of "low-level scheduling" that the Scheduler carries out based on the happiness information.

2.4 Relevant Communication Sequences

We now zoom in onto the precise communication which takes place between the application components. We identify four *communication sequences*, i.e. sequences of directly related requests between components. The following communication sequences are relevant for Scenario **SC.1**:

1. *Availability Job Execution*, a sequence specifying how availability jobs are initiated and executed by the Scheduler component;
2. *Conversation Handling*, a sequence specifying how ASK CS handles conversations with users, including DTMF number input and hang-up;
3. *Incoming Call Handling*, a sequence specifying how ASK CS handles incoming calls from users;
4. *Happiness Information Exchange*, a sequence specifying how the Scheduler component requests happiness information from the asterisk component.

2.4.1 Availability Job Execution

The first communication sequence is depicted in Figure 3 and starts in the Scheduler component. This component has a local *job queue*, which contains all known jobs scheduled at a certain time. The *job queue* should not be confused with the *task queue*, in which the tasks for the thread pool are placed. At start-up, an infinite *RunLoop task* is started, which continuously checks the job queue. As soon as a certain job in the queue is scheduled on or before the current time, it removes that job and puts it *as a task* in the task queue. Hence, at that moment the job is taken up by a thread and executed.

The job queue of the Scheduler component initially contains a *Poll DB* job, which polls the database to see if the local job queue is *dirty*, i.e. not up-to-date with the contents of the database. If this is the case, the local job queue is updated with the jobs from the database. The Poll DB job always ends with scheduling a new Poll DB job in the local job queue one minute after it finishes. This way, the Scheduler component checks the dirtiness of its local job queue each minute.

The ASK CS database can contain scheduled jobs of many types. One of them is the *availability job*, the recruitment job introduced earlier. Slightly simplified, this job has the following goal: "recruit m out of n people by presenting

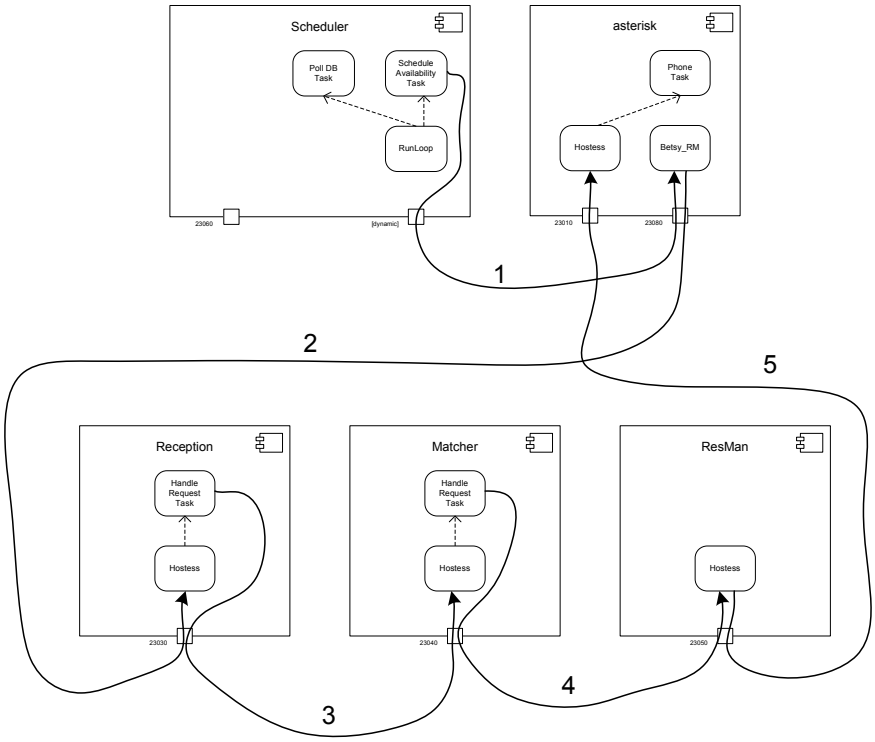

Figure 3: *Availability Job Execution* communication sequence

them IVR menu i", where $m <= n$. Again slightly simplified, it deals with this job as follows. Firstly, it checks the number of users already recruited for the service. Suppose this number is r. Hence, the job is to recruit $m - r$ people out of $n - r$. If $(m - r) > 0$, $m - r$ people are selected from the $n - r$ remaining people. After that, we are about to encounter the first request in the communication sequence, from the Scheduler component to the asterisk component. For each of the $m - r$ people, an individual outgoing call should be made. This is done as follows: **for each** person p to be called, **if** the locally known happiness information indicates that all phone lines are busy **then exit for**, **else** send a request to call person p with IVR menu i to the asterisk component **end for**. Hence, depending on the happiness information, a number of requests is sent to the asterisk component, at least 0, at max $m - r$. We explain the details of the happiness information in Section 2.4.4, where we discuss the Happiness Information Exchange communication sequence. After the schedule availability job has sent the requests, it is finished.

We can now move on to the asterisk component. Requests sent to this component are received either in the infinite *Hostess* task, or in the infinite *Betsy_RM* task (where RM stands for Resource Manager), depending on the port to which the request is sent. In this sequence, it is the Betsy_RM task

which receives the request and translates it into a request for the Reception component, stating: "Scheduler asks for a call to user p with IVR menu i". The Reception component, on its turn, receives the request in its infinite *Hostess* task and creates a new *Handle Request* task which handles the request. The request now receives an ID (in order of retrieval) and a Sub-ID (0) and the request to play IVR menu i is translated into a request to play a *sound file* w. Then, it is forwarded to the Matcher component. Similar to the Reception component, the Matcher component receives the request in its infinite *Hostess* task and creates a new *Handle Request* task which takes care of the request. In this case, it adds the phone number $p.t$ of user p to the request and then forwards it to the ResMan component, in which the *Hostess* task adds 1 to the Sub-ID of the request, stores the request status in the database, and forwards it to the asterisk component.

Finally, we are back at the asterisk component. The request is now received in the *Hostess* task. The contents of the request are now as follows: "Perform a call to phone number $p.t$ and play sound file w when the call is answered.". The asterisk component takes care of this in a fresh new *Phone* task and the communication sequence finishes.

2.4.2 Conversation Handling

A conversation between ASK CS and a user can be roughly characterized as follows:

1. call setup by either the system or the user;
2. a sequence of sound files played by the system and DTMF number inputs from the user;
3. hang-up by either the system or the user.

We have already seen the first step for an *outgoing* call to a user in the previous section. In fact, this sequence also includes the playing of the first sound file by the system once the user takes up the phone. The second step, the actual conversation, is handled via the *Conversation Handling* communication sequence, as shown in Figure 4. Based on input from the user, that is, either a DTMF number pressed or a hang-up, the system has to determine which action to take, for example: play another sound file, store some value in the database, hang-up, connect the user to another user, etc.

The communication sequence starts at the asterisk component. The asterisk subsystem receives a user input in an existing Phone task. In this task, the input is converted into a request. Note that the asterisk component remembers the ID and Sub-ID of the initial call-out request received in the *Availability Job Execution* sequence. The new request hence gets the same ID and Sub-ID, whereafter it is sent to the Reception component, which determines in a new Handle Request task which action to carry out next. If the user input is a DTMF number, the Reception could decide to store a value in the database.

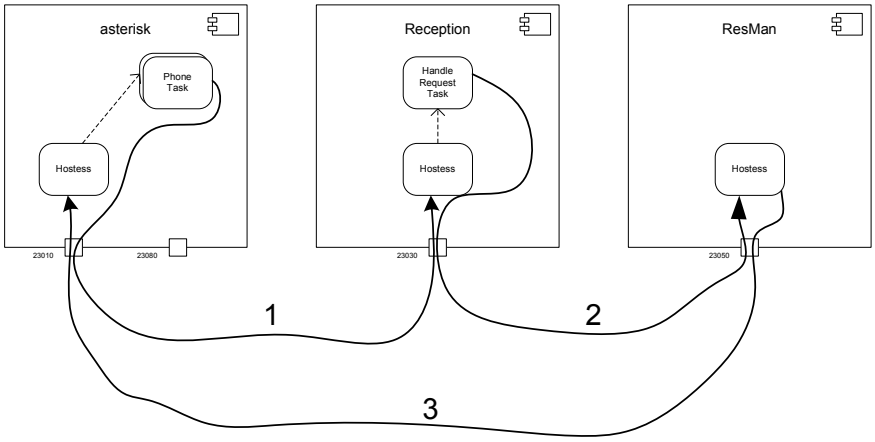

Figure 4: *Conversation Handling* communication sequence

For example, if a user p gets the question "Are you available for service s in time window t? Press 1 for available, 2 for not available" and this user presses number 1 on her phone, the Reception component updates the database with the fact that user p is available for service s in time window t. The Reception component could decide to perform more than one action, for example, it could also forward a request to play a sound file with the text "Thank you!" and hang-up after the file has been played. The IVR menu used for the call determines which actions to carry out given a certain input. Hence, the system administrator is able to configure the precise sequence of actions to take.

In case the user input leads to further conversation or hang-up, the communication sequence hence continues with a request from the Reception component to the ResMan component and eventually a request to the asterisk component. This sequence works in the same way as the final part of the *Availability Job Execution* sequence.

2.4.3 Incoming Call Handling

The third relevant communication sequence is depicted in Figure 5, via which an incoming call from a user is handled. The difference with the former communication sequence is twofold. Firstly, the sequence starts with the asterisk subsystem answering the incoming call and creating a *new* Phone task, which converts the incoming call to a request for the Reception component. Secondly, the Handle Request task in the Reception component now uses the database to determine which IVR menu to use, based on the phone number of the incoming call. After that, the sequence continues in the same way as in the previous two sequences.

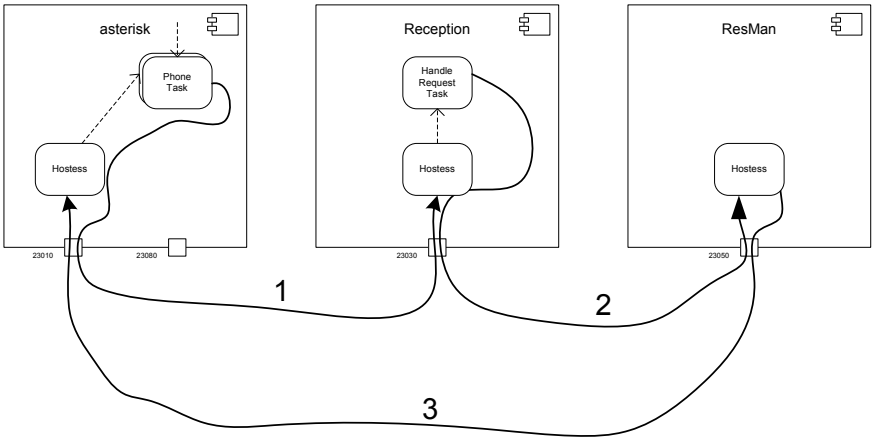

Figure 5: *Incoming Call Handling* communication sequence

2.4.4 Happiness Information Exchange

In the *Availability Job Execution* communication sequence, the Scheduler component uses happiness information to determine whether or not to continue sending requests to the asterisk component. As we explained earlier, this happiness information is an indication for the amount of free phone lines out of the total amount of phone lines. Slightly simplified, it is computed as follows: given a total amount of phone lines l and an amount of b busy phone lines, then the happiness $h = 1 - ((2/l) * b)$. Hence, the happiness ranges from -1 (all lines busy) to 1 (all lines free).

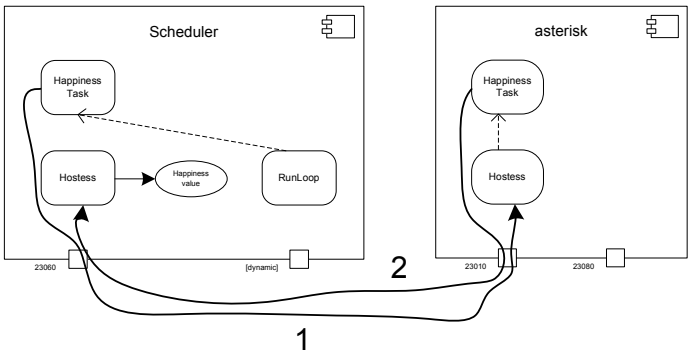

Figure 6: *Happiness Information Exchange* communication sequence

The amount of free phone lines is known inside the asterisk component. The Scheduler component updates its happiness information by regularly polling the asterisk component, via the communication sequence depicted in Figure 6. Hence, the sequence starts at the Scheduler component. Next to the *Poll DB* job explained in the *Availability Job Execution* sequence, the job queue ini-

tially contains another job: the *Gauge Happiness* job. This job sends a "gauge happiness" request to the asterisk component, whereafter it reschedules itself for 30 seconds after its completion. The Hostess task of the asterisk component receives the request and creates a new Happiness task, in which the current happiness is computed and sent to the Scheduler component. Once the Scheduler component receives this value in its Hostess task, it updates the local happiness information.

2.5 Modeling Components as Network Automata

We have chosen to base our modeling work on automata. These provide a concrete, intuitively clear, model of computation, and a structural approach to the analysis of the behavior of components and their composition. There is also a large amount of theoretical and implementational work on using automata for representing components in distributed and reactive systems, which may be of use. The automata model that we will use for the initial modeling of the ASK system was presented in Deliverable 1.2. We will review some of the main features of the automata model in this section and we refer the reader to Deliverable 1.2 for a full account. The formal definition of the automata is as follows:

Definition 1 (Network Automaton: Syntax)**.** *A* network automaton P *is a structure* $P = \langle S, t, A, T \rangle$ *where:*

- S *is a finite set of* states*,*
- $t \in S$ *is the* initial state*,*
- A *is a finite set of* action names*, and*
- $T \subseteq S \times \mathbb{M}(Act) \times S$ *is the set of* transitions*.*

  Each transition is labeled with a multiset of actions $m \subseteq \mathbb{M}(Act)$*, where* $Act = A^\tau \cup A^I \cup A^O$*, consists of*

  - $A^\tau = \{a \mid a \in A\}$ *the set of* internal actions of P*,*
  - $A^I = \{a? \mid a \in A\}$ *the set of* input actions of P*, and*
  - $A^O = \{a! \mid a \in A\}$ *the set of* output actions of P*.*

  Note that A^τ*,* A^I *and* A^O *are pairwise disjoint.*

We can link two automata together using a product function. This function returns a single automaton that acts in the same way as the two input automata run together. In particular it matches the inputs and outputs of the transitions (see Deliverable 1.2 for a full definition).

We augment the basic network automata, as defined above, with the ability to pass values from a finite data domain. This means that instead of just inputting and outputting on a channel name we can send data values, for instance

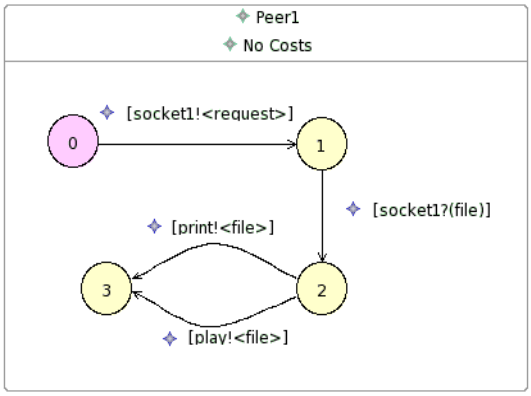

Figure 7: An automaton model of a component

$a!\langle 5\rangle$ means output the value 5 on the channel a and $a?(x)$ means listen on the channel a for a value and bind it to the variable x in the rest of the automaton. In addition, transitions may be labeled with guards to test the values that a variable might have. We define this extension of the automata model by providing a mapping back into the original model. This allows us to use value-passing actions as a convenient short hand for the larger basic automata.

When we model components as automata, we abstract away from most of the details and only put the key features of what we are trying to model in the automaton. An example model of a component is given in Figure 7. The first action that this component performs is to request a file on a socket. It then receives a file and either plays it or prints it. The real component that this automata models will be much more complex, it would probably time out if it did not get the file quickly enough, for instance. How the choice between playing and printing is made is not specified in the model, this could in fact be a complex process, involving user input, however we keep the model simple. It is the modeler's job to ensure that their abstract model matches their real code; if it does then we will produce accurate results when modeling a collection of these automata running together.

Our tool support allows us to model and combine automata. To analyze the automata we can export them to a range of other tools. We can export the automata to Maude for model checking and to allow them to interact with components written in the Creol language. We are currently working on exporting the automata to the PRISM model checker, which will allow us to check the completion times for availability jobs within the Scheduler of the ASK system and compare certain happiness information-based management strategies.

2.6 Network Automata for ASK CS

The goal of the network automata models presented in this section is to determine an optimal strategy for the Scheduler to retrieve and use happiness information to perform its availability jobs. We plan to report on the actual application of the *Credo* tools for finding the optimal strategy in Deliverable D6.3 "Final Modeling". In this document, we limit ourselves to the modeling effort only. We base the models on the following scenario. We consider a small ASK CS configuration with two phone lines and five users[1]. The users are divided into two groups. The first group is a group of three *passive scheduled users*: these users are to be called by ASK CS in an availability job. The second group is a group of two *active non-scheduled users*: these users call the system for irrelevant purposes, but the point is that they thereby occupy phone lines, which potentially causes delays in the completion time of the availability job. The goal is to optimize this completion time without affecting the behavior of the users, e.g. by tailoring the frequency of happiness information exchange or the strategy used for postponing outgoing calls.

2.6.1 Design Principles

We have balanced between keeping the size and the amount of automata as small as possible on one hand, and keeping the model as realistic as possible on the other hand. This has resulted in the following simplifications in comparison with the real ASK CS:

- We limit value passing as far as possible;
- Users always pick up the phone when called;
- Users always answer "yes" to an availability call;
- Users have fixed waiting and listening times;
- Hang-up can be done only by users, not by ASK CS;
- We model the Hostess tasks and the TaskHandler tasks as one automaton;
- We do not model the Betsy RM task of the asterisk component;
- The IVR menu is built into the Reception automata;
- The Availability job is built into the Scheduler automata;
- We leave out the Matcher component and the ResMan component.

[1] the small number of phone lines and users are chosen as a convenient starting point only – further research in the "Final Modeling" phase is needed in order to determine realistic values.

2.6.2 Automata Overview

The result of our modeling effort is a set of 10 network automata, which we will explain individually in the next sections:

- **User Automata**: a *Passive Scheduled User* automaton and an *Active Non-scheduled User* automaton, used for modeling the behavior of users. The idea is to use three instances of the passive and two of the active automaton, thereby having five users of two different types.
- **Application Component Automata** which model the behavior of the ASK CS application components: an *Asterisk Task Handler* and an *Asterisk Incoming Handler* automaton, a *Reception Task Handler* automaton, a *Scheduler Task Handler* and a *Scheduler RunLoop* automaton.
- **Happiness Information Automata**: An *Asterisk Resource Meter* automaton and a *Scheduler Happiness Value* automaton, which model the happiness information as it is present inside the asterisk component and the Scheduler component, respectively.
- A **Database Automaton**, in which the results of the Scheduler's availability job are stored.

2.6.3 User Automata

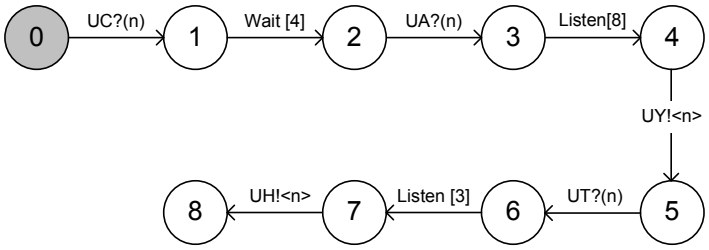

Figure 8: Passive Scheduled User Automaton

The two types of user automata are shown in Figures 8 and 9. The *Passive Scheduled User* of Figure 8 starts with waiting for an incoming call from ASK CS. After that, she waits for 4 seconds, takes up the phone and listens to a message (sound file) for 8 seconds, in which she is asked if she is available for a certain service in a certain time window. She answers "yes" to the message by pressing a DTMF number on her phone. After the user has listened to a "Thank you!" message for 3 seconds, she hangs up. The network automaton in Figure 8 models this behavior by a set of actions, which all communicate a value n. This value ($1 \leq n \leq 3$) represents the unique number of the user. The intuitive meaning of the actions is as follows:

- UC?(n): phone is ringing, ASK CS calls the user

- Wait[s]: user waits for s seconds
- UA?(n): user answers phone and listens to a message (availability question)
- Listen[s]: user listens for s seconds
- UY!<n>: user answers "yes" to the message
- UT?(n): user listens to a "Thank you!" message
- UH!<n>: user hangs up the phone

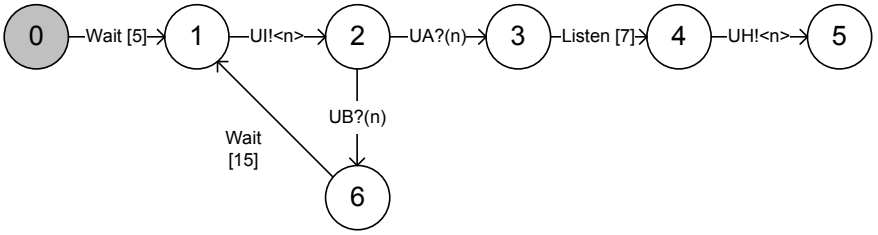

Figure 9: Active Nonscheduled User Automaton

A similar sequence occurs in the behavior of the *Active Non-scheduled User*, shown in Figure 9. However, this user is *active*: she calls the ASK CS herself. We add the possibility that all phone lines of ASK CS are busy and the user retries a call after a while. This is modeled by the loop in the figure. The intuitive meanings of the actions are as follows:

- Wait[s]: user waits for s seconds
- UI!<n>: user calls ASK CS
- UB?(n): all phone lines of ASK CS are busy
- UA?(n): user answers phone and listens to a message
- Listen[s]: user listens for s seconds
- UH!<n>: user hangs up the phone

2.6.4 Application Component Automata

We have modeled five application component automata, two for the asterisk component, one for the Reception component and two for the Scheduler component. For the asterisk component, we distinguish between the handling of requests from the Reception component (outgoing call, "are you available?" question and "thank you" message) and the handling of incoming information from the user (incoming call, "yes"-answer and hang-up). The requests from the Reception are handled by the *Asterisk TaskHandler* as shown in Figure 10,

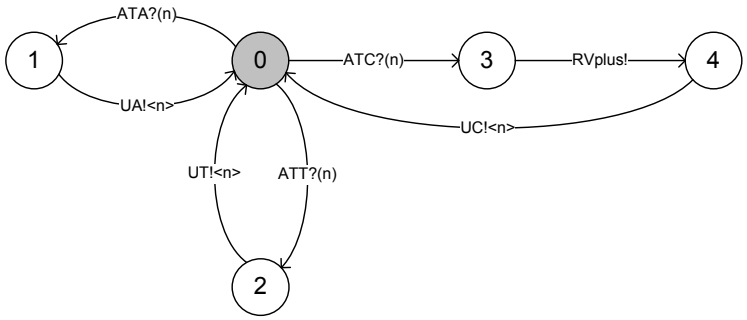

Figure 10: Asterisk TaskHandler Automaton

while the user information is handled by the *Asterisk IncomingHandler* depicted in Figure 11.
The *Asterisk TaskHandler* has straightforward behavior. Note that for an outgoing call, one of the phone lines is used. This is why the RVplus! message is communicated with the *Asterisk Resource Meter* automaton (see Figure 15), which keeps track of the amount of busy phone lines. If there is no free phone line, then the Asterisk TaskHandler blocks until one becomes free. The actions have the following meaning (we repeat previously explained actions for convenience):

- ATA?(n): request to play a message
- UA!<n>: ASK CS plays a message
- ATC?(n): request to perform an outgoing call
- RVplus!: increase the amount of busy phone lines
- UC!<n>: ASK CS calls the user
- ATT?(n): request to play a "Thank you!" message
- UT!<n>: ASK CS plays a "Thank you!" message

In the *Asterisk IncomingHandler*, the added complexity is that we allow the situation that all phone lines are busy and the user tries to call again somewhat later. Therefore, in the reception of an incoming call, extra communication with the *Asterisk Resource Meter* takes place in order to check whether all phone lines are occupied or not. If the automaton synchronizes on RVplus!, this means that one of the phone lines was available, otherwise, the automaton synchronizes on RVmax?, which indicates that all phone lines are busy. The following actions are used:

- UY?(n): user answers "yes" to the message
- RTY!<n>: "yes" answer is propagated to Reception

- UH?(n): user hangs up the phone
- RVminus!: decrease the amount of busy phone lines
- RTH!<n>: hang-up is propagated to Reception
- UI?(n): user calls ASK CS
- RVplus!: increase the amount of busy phone lines
- RTI!<n>: incoming call is propagated to Reception
- RVmax?: all phone lines are busy
- UB!<n>: user hears busy tone

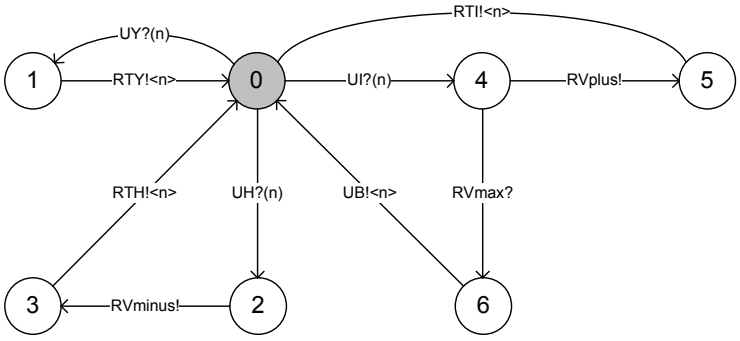

Figure 11: Asterisk IncomingHandler Automaton

The automaton for the Reception component is depicted in Figure 12. It accepts four different inputs: an incoming call request, a "yes" answer and a hang-up request, which are received from the asterisk component, and an outgoing call request, which is received from the Scheduler (and not from the asterisk component, because we leave out the Betsy RM task). Note that a "yes" answer results in an output action to the database, DBY!<n>. The consequence of this is that the amount of users to be recruited for the availability job decreases. The actions of the Reception TaskHandler are straightforward:

- RTH?(n): user hangs up the phone
- RTI?(n): user calls ASK CS
- ATA!<n>: send request to play a message
- RTY?(n): user answers "yes" to the message
- DBY!<n>: store "yes" result in the database
- ATT!<n>: send request to play a "Thank you!" message

- RTC?(n): request to perform an outgoing call
- ATC!<n>: outgoing call request is propagated to asterisk

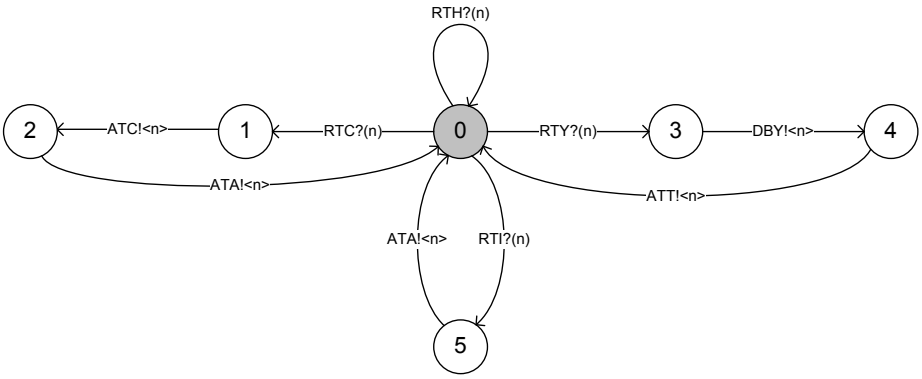

Figure 12: Reception TaskHandler Automaton

For the Scheduler, we modeled two automata. The *RunLoop* automaton takes care of running through a one-minute cycle in which the happiness information is retrieved twice (once per 30 seconds), and the availability job is executed once (the 50th second of each minute). It is shown in Figure 13. As can be seen, actions STH! and STS! are used to trigger the tasks in the *Scheduler TaskHandler* automaton depicted in Figure 14. This latter automaton implements the somewhat more complicated availability job, as follows: it sends a message DBD! to the database, after which it receives a list of identification numbers of users, which is always a (possibly empty) subset of the list $\{1, 2, 3\}$. The end of the list is indicated with the number 0. For each number in the list, the local happiness information is checked with SHVV! and SHV?(h). Since the current implementation of the network automata does not allow for negative numbers, we map the $\{-1, \ldots, 1\}$ range onto a $\{0, \ldots, 2\}$ range. Hence, if the happiness is 0, all phone lines are occupied. If the happiness indicates that there is a free phone line, a request for an outgoing call is sent to the Reception via RTC!<n>, otherwise, the loop is exited and the remaining numbers of the database are consumed without any consequence. The following list of actions is used in the Scheduler RunLoop and the Scheduler TaskHandler:

- Count[s]: delay for s seconds
- STH!: update happiness information
- STS!: perform availability job
- AHVV!: propagate happiness information request to asterisk
- DBD!: ask database for list of users to call

- DBU?(n): get the identification number of a user
- SHVV!: request current happiness value
- SHV?(h): get current happiness value
- RTC!<n>: request to perform an outgoing call

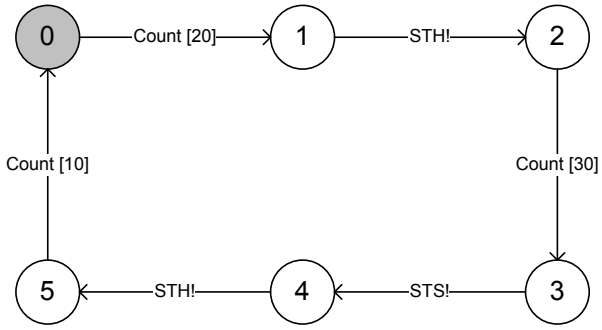

Figure 13: Scheduler RunLoop Automaton

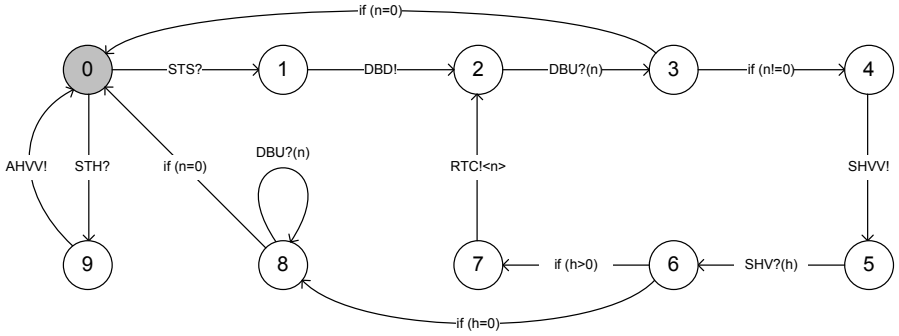

Figure 14: Scheduler TaskHandler Automaton

2.6.5 Happiness Information Automata

In ASK CS, happiness information is exchanged between the asterisk component and the Scheduler component. In order to precisely model the updating mechanism currently implemented in ASK CS, we have modeled two automata which represent the happiness value. The *Asterisk Resource Meter* (see Figure 15) keeps track of the happiness value within the asterisk component. It is updated each time a phone line is occupied or released. The *Scheduler Happiness Value* (see Figure 16) represents the happiness as known by the Scheduler component.

It initially assumes that all lines are free and is updated each time the *RunLoop* automaton performs an STH! action. As we explained earlier, since the current implementation of the network automata does not allow for negative numbers, we map the $\{-1, \ldots, 1\}$ range onto a $\{0, \ldots, 2\}$ range. The used actions in both components are:

- RVplus?: increase the amount of busy phone lines
- RVminus?: decrease the amount of busy phone lines
- RVmax!: all phone lines are busy
- AHVV?: receive a happiness information request from the Scheduler
- AHV!<h>: send happiness value h to the Scheduler
- SHVV?: receive a happiness value request from the Scheduler TaskHandler
- SHV!<h>: send happiness value h to the Scheduler TaskHandler

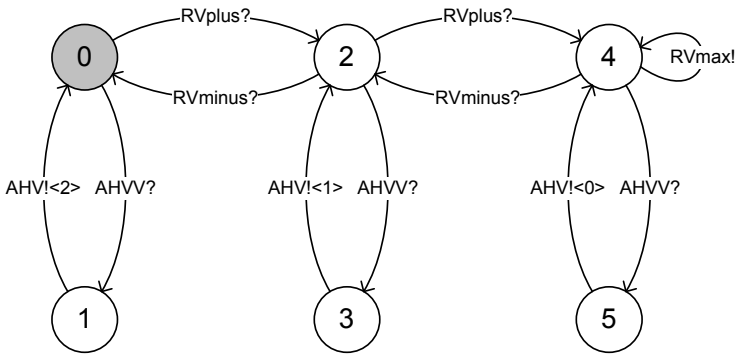

Figure 15: Asterisk Resource Meter Automaton

2.6.6 Database Automaton

The *Database Automaton*, depicted in Figure 17, keeps track of the users which have to be called for the availability job (users 1, 2 and 3). It uses only three different actions:

- DBY?(n): user n has answered "yes"
- DBD?: Scheduler requests a list of users to be called
- DBU!<n>: user n must be called

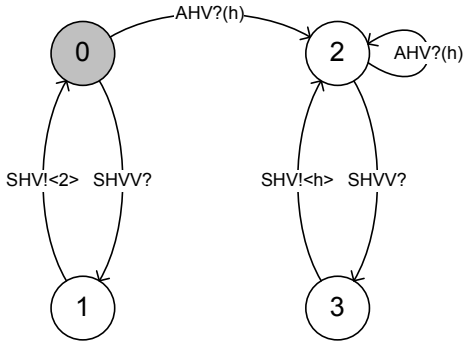

Figure 16: Scheduler Happiness Value Automaton

The list of users to be called is initially $\{1, 2, 3\}$. Suppose user 1 answers "yes", then the database automaton is able to perform input action DBY?(1). From that moment on, the list of users to be called is $\{2, 3\}$. Note that when the list is requested by the Scheduler, the end of the list is indicated with action DBU!<0>. As soon as state 26 of the Database is reached for the first time, this indicates that the availability job has been completed successfully, since users 1, 2 and 3 all have positively answered to the ASK CS call.
If we compute the product automaton of the automata presented in this section, we are thereby able to analyze the average completion time for the availability job by computing how long it takes to reach state 26 of the Database automaton. Changes to the timings in the Scheduler RunLoop or the functioning of the Scheduler TaskHandler could lead to a better average completion time. We plan to report on these expected analysis results in D6.3 "Final Modeling".

2.7 Requirements Assessment

In Section 2.1.2, we pointed out which requirement categories formulated in the Methodology Document are addressed by the initial modeling effort as reported on in this document. Our conclusion is that the expressiveness of Network Automata is sufficient for modeling the behavior of ASK. Also, the automata can be arranged such that they map directly onto their real-world structural counterparts. Finally, timings can be attached to the automata in terms of costs on the transitions between states. Thereby, we conclude that the Network Automata fulfill the requirements of Category 1, the support of all structural and behavioral modeling and the support of time. We also conclude that the requirements of Category 5 are fulfilled: it is possible to model individual components as single automata as well as sets of components in combination.

In Section 2.1.2, we also indicated that the requirements of Category 4, the possibility to verify non-functional properties of a reconfigurable system based on a *Credo* model created for it, can not yet be assessed, since efficient creation of a product automaton and effective analysis of this automaton has not yet

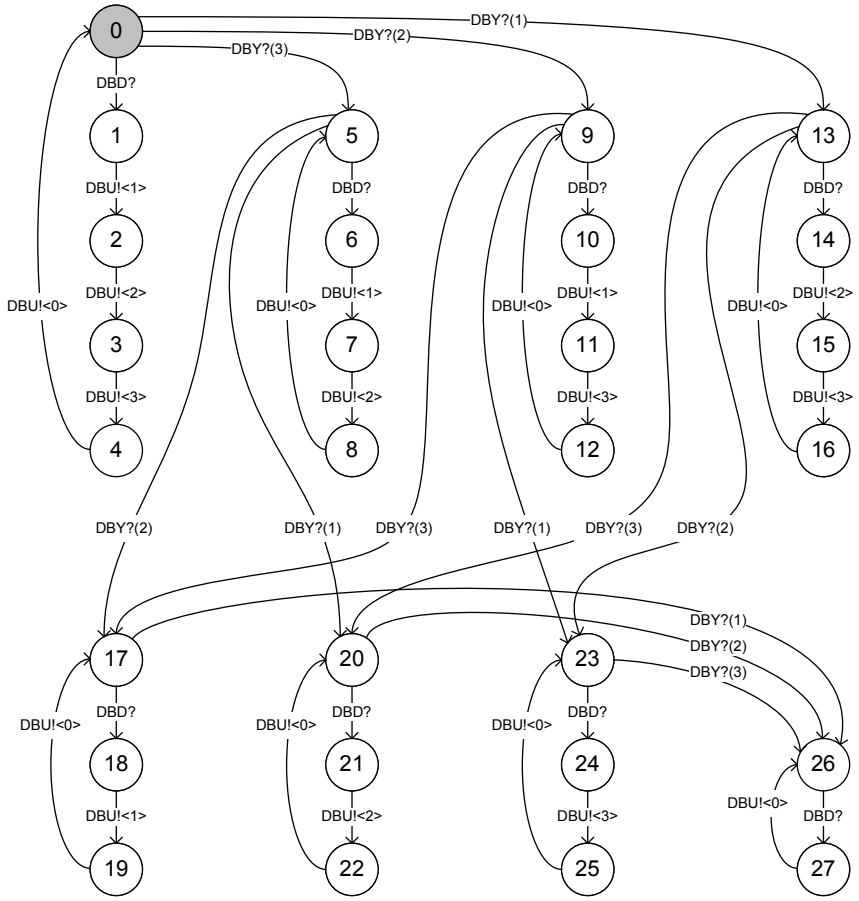

Figure 17: Database Automaton

been carried out. We plan to assess the requirements of Category 4 in D6.3 "Final Modeling".

3 Case study 2: Biomedical sensor networks

The generic architecture of a biomedical sensor network is shown in Figure 18, where each shaded element corresponds to one sensor node. The node n_1 reveals its internal structure, which consists of a radio object r, a controller object c, and (in our example) two sensor objects s_1 and s_2.

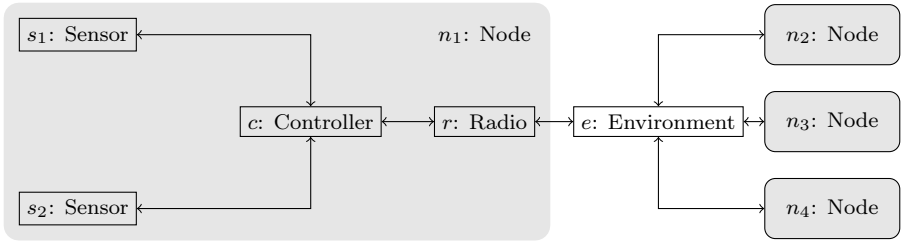

Figure 18: Architecture of a sensor node and its relation to other nodes

The controller object c maintains the main activity of the node. It will read data from each sensor with specific periods π_1 and π_2, collect a number of readings from each sensor in a package, and send it to the radio r. In addition, it will receive messages broadcasted on the network and react to it. If the message is routed via the node itself, forward it. If the node is asked to do so in the packet as long as sufficient power is available, and otherwise ignore it. Control messages are passed between nodes to approximate the current network topology and to update routing tables.

The *Environment* between the sensor nodes models different aspects of (wireless) networks. First, it provides channel communication end points by implementing the *Channel* interface (see below). Then it maintains the interconnections between the nodes in the network: It knows which node is in reach of another node, and consequently records the network topology. The environment object decides, whether a message can be successfully delivered. It will, depending on the type of the object connecting to a channel end, decide whether writing to a channel end is broadcast communication or point-to-point communication. It also controls the mobility of the nodes in that it can decide to move objects out of and into the range of other objects. Depending on the desired level of detail, we can move between abstraction levels by replacing the environment object.

The only means of communication on a wireless channel is by *broadcasting* messages. If several Radio objects in reach of each other send at the same time, their messages are all lost due to *collisions.* Currently, Radio objects cannot influence their location in the topology. We plan to provide methods for this in the future. At first, we consider the case of *one-hop* communication, where all objects are in reach of each other.

In Section 3.1 we explain the syntactic interfaces of the components of a biomedical sensor network. The syntactic interface conveys information of the

```
interface Channel
begin
  with Radio
    op send (in channel: Int, receiver: Radio, data: Data)
    op free (in channel: Int; out yes: Boolean)
    op snoop (in channel: Int; out receiver: Radio, data: Data, strength: Int)
end
```

Figure 19: Interface of Channel

kinds messages exchanged between the components of the network. In Section 3.2 we describe the interface of the environment object in terms of *network automata*. Section 3.3 describes a timed model of the radio component. It serves to explain that a radio component can either send or receive, but it cannot do both at the same time, and describes the time windows of the send and receive phase.

3.1 Syntactic Interfaces

The syntactic interfaces express the features provided by the components of a single sensor node (see node n_1 in Figure 18). These interfaces will be implemented by classes to obtain an executable model of a sensor node.

The interface *Channel*, which is implemented by the Environment and which is shown in Figure 19, provides all methods required for sending on a channel or receiving messages from the channel. Note that this and subsequent interfaces are formulated using the Creol language.

send Send *data* on *channel* to the node identified by *radio*.

free Test whether *channel* is free.

snoop Read whatever message is available on *channel*. The result value *strength* is set to the received signal strength indication of the message. For convenience, this method provides basic package structuring: It extracts the intended *receiver* of the message.

The interface of a *Radio* is shown in Figure 20. In our setting, the Environment is *passive* in the sense that it does not deliver messages to sensor nodes, but instead the Radio component asks the Environment for messages in its reach by the Channel interface. The radio is a passive process, reacting to calls from the Controller object. It tries to obtain a channel and to send messages to the Controller and listens to a channel to pass messages on to the Controller. The radio will only provide operations to the Controller. These are:

write Write a message to the current channel.

```
interface Radio
begin
  with Controller
    op write (in receiver : Radio, data: Data)
    op read (out receiver: Radio, data: Data, strength: Int)
    op setPower (in power: Int)
    op setState (in state : Int)
    op getChannelStatus (out status: Int)
    op getError (out error: Int)
    op getChannel (out channel: Int)
    op setChannel (in channel: Int)
end
```

Figure 20: Interface of Radio

read Read a message received by the radio. This method also provides the signal *strength* of the received message.

setPower Sets the power used to send messages.

setState Allows the controller to set the state to one of "receiving", "transmitting", and "off".

getChannelStatus Allows the controller to read the status of the channel ("free" or "busy").

getError Get a possible error code for the last transaction, where 0 shall represent "the last operation was successful."

getChannel Get the current transmission channel frequency of the radio, which is encoded by a number ranging from 11 to 26.

setChannel Change the transmission channel frequency of the radio, which encoded by a number ranging from 11 to 26.

The interface of a *Controller* is shown in Figure 21. The controller only provides the method "write" to an instance of Sensor, allowing the sensor to push data to the controller. In all other respects, the controller encapsulates all activities of the sensor node: reading from passive sensors, aggregating data, sending it to the radio, instructing the radio to receive messages, and so on.

The interfaces of different sensors are shown in Figure 22. The interface named "Sensor" is common to all sensors. It is used by controller objects to set the sampling resolution (most sensors do not allow it), the sampling frequency, the encoding (most sensors only support one encoding), to switch it on and to switch it off (for power management).

We define two sub-interfaces: A passive sensor provides a "read" method, which allows the controller to read data from that sensor. The second sub-interface exports no additional methods, meaning that the controller cannot

```
interface Controller
begin
  with ActiveSensor
    op write (in value: Data)
end
```

Figure 21: Interface of Controller

```
interface Sensor
begin
  with Controller
    op setResolution (in res: Data)
    op setFrequency (in f: Real)
    op setEncoding (in encoding: Data)
    op switchOn
    op switchOff
end

interface PassiveSensor inherits Sensor
begin
  with Controller
    op read(out value: Data)
end

interface ActiveSensor inherits Sensor
begin
end
```

Figure 22: Interfaces of Sensors

read data from it, but allows the sensor to push data to the controller. An instance of ActiveSensor must implement an activity that pushes the data.

3.2 Network Automata

In this section we describe how a network automata model (as proposed in Deliverable D1.2) for the Environment class (See Figure 18) is computed.

Let D range over all data values we intend to send via the wireless network. Let D include the values none and error, where none represents that no data is available on the network and error represents that there was a transmission error, caused by either a mid-air collision or noise.

For the model of network automata we ignore the possibility of errors introduced by noise. Noise is a stochastic phenomenon and stochastic phenomena are not covered in network automata. But we can model errors caused by mid-air

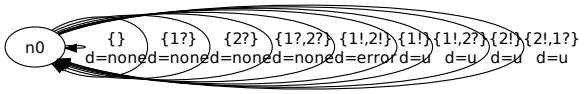

Figure 23: Automaton for two nodes

Figure 24: Automaton for three nodes

collisions.

Let P be a set of node identities. A topology of a network is a simple directed graph $T = (P, E)$, where an edge $(p, q) \in E$ expresses that a node p can send a message to the node q and that q can receive that message. Here, we do *not* assume that the topology is symmetric: if $(p, q) \in E$, then $(q, p) \in E$ need not hold. Let $s \in P$ denote the unique *sink node*. It has the distinguishing property that it can send to *all* nodes in the network, represented by the constraint $\{(s, p) \mid p \in P \setminus \{s\}\} \subseteq E$. The converse is not necessarily true.

To each process identifier $p \in P$ we associate two channel ends p_i, the input channel end, and p_o, the output channel end. Let P_i represent the set of all input channel ends and P_o represent the set of all output channel ends. To each process $p \in P$ we assign a data variable d_p that represents the value read or written during the current transaction.

Then the network automaton modelling their interactions has one state $T = (P, E)$ for each topology, and for each $I \subseteq P_i$ (representing the processes that listen to the network) and $O \subseteq P_o$ (representing the nodes sending on the network) with $I \cap O = \emptyset$ (because a node cannot send and receive as a same time) it contains all transitions from T to T with a label that satisfies all the following label:

1. If $O = \emptyset$, then $\bigwedge_{p \in P_i} d_p = \mathsf{none}$.
2. If $o, o' \in O$ with $(o, o') \in E \vee (o', o) \in E$, then $\bigwedge_{p \in P_i \wedge (o,p) \in E \vee (o',p) \in E} d_p = \mathsf{error}$.
3. Otherwise $\left(\bigwedge_{p \in I \setminus J \wedge o \in O \wedge (o,p) \in E} d_p = d_o\right) \wedge \left(\bigwedge_{p \in J} d_p = \mathsf{error}\right)$ for $J \subseteq I$. The nodes in J have their communication disturbed by noise. If no noise is desired, $J = \emptyset$ is assumed.

Figure 23 and 24 show example automata for two and three nodes, where all are in reach of each other. The channel end marked $p?$ refers to reading from the

input channel of process p, whereas $p!$ refers to writing to p's output channel. We have elided the constraints describing data propagation and represent them as $d = u$, assuming that only the data value u is sent.

Network automata modelling the behaviour of the Environment object are exponential in the number of nodes.

Theorem 1. *Let P be a set of process identifiers. The size of the network automaton describing the behaviour of the environment of P has $\Theta(2^{(|P|-1)(|P|-2)})$ states.*

Proof. Let $s \in P$ be the sink node. Each state is represented by a configuration of edges, which includes the set $\{(s,p)|p \in P \setminus \{s\}\}$. In addition, there are $2^{(|P|-1)(|P|-2)}$ simple directed graphs with the node set $P \setminus \{s\}$. To this, we must add the subsets of $\{(p,s)|p \in P \setminus \{s\}\}$, representing the connections from sensor nodes to the sink. This results in $2^{(|P|-1)(|P|-2)} + 2^{|P|-1} \in \Theta(2^{(|P|-1)(|P|-2)})$ states. □

For networks of 40 nodes (which is a realistic size for a biomedical sensor network), the network automaton is not representable: we find positive constants $c > 1$ such that the automaton has $c \cdot 10^{448}$ states. This exceeds the common memory of todays machines by a factor of about 10^{439}.

3.3 Timed Modelling Design

A timed automaton is a finite state automaton extended with real-time clocks. UPPAAL is a tool box for timed automata, which provides a modelling language, a simulator and a model checker. In UPPAAL, timed automata are further extended with data variables of types such as integer and array etc., and networks of timed automata, which are sets of automata communicating with synchronous channels or shared variables, to ease the modelling tasks. The modelling language allows to define templates to model components that have the same control structure, but different parameters, which is a perfect feature for modelling of sensor nodes.

In this section, we develop a UPPAAL model for a biomedical sensor network (BSN), as a network of timed automata where each automaton models a sensor node. As all sensor nodes are implemented with the same chip for wireless communication, running the same protocol, we use a template to model the node behaviour with open timing parameters to be fixed in the validation phase. The network topology is modelled using a matrix declared as an array of integers in UPPAAL. Elements in the matrix denotes the connectivity between pairs of nodes.

3.3.1 Modelling the Chipcon CC2420 Transceiver

To study the network performance, we need to model only the transceiver of a sensor node for wireless communication. We assume that all sensor nodes use the Chipcon CC2420 transceiver. We model the transceiver as a UPPAAL

template based on the radio control state machine of the transceiver, described in its reference manual.

The modelled template is shown in Figure 25. Most of the states are of the same name as the radio control states in the original state machine for the transceiver. The functionality of the transceiver is modelled by the state transitions according to the reference manual.

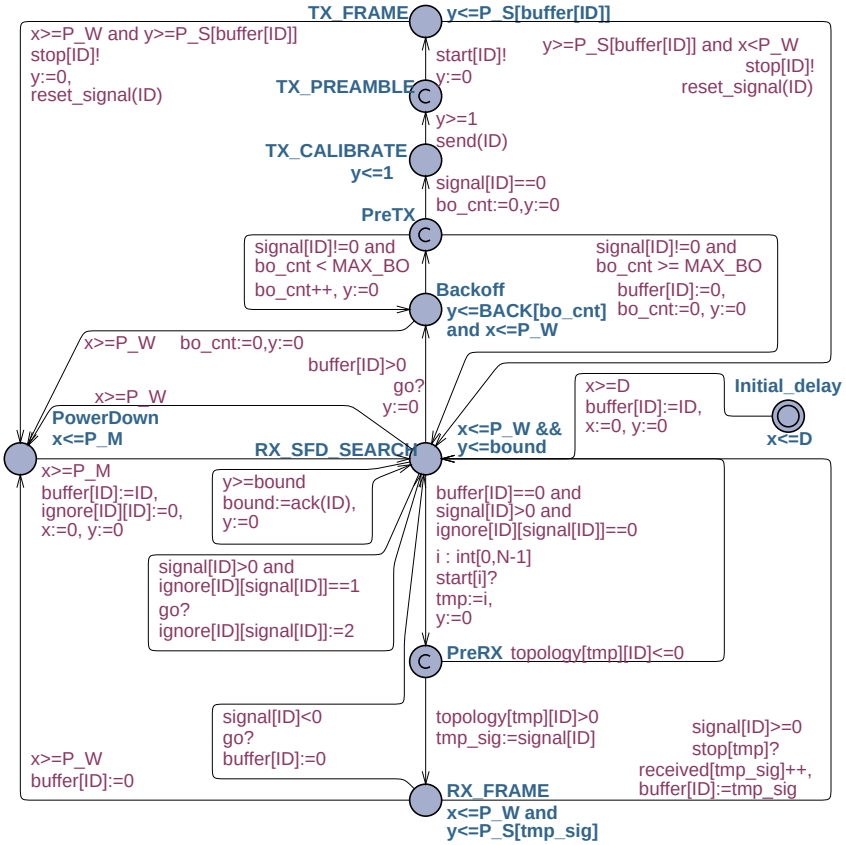

Figure 25: A UPPAAL template for wireless sensor nodes based on the Chipcon CC2420 Transceiver

3.3.2 Modelling the Network and Communication

Now we describe how data packets are transferred between nodes and how errors are modelled, that may occur during packet transmission. The description is mainly on the global data variables used by the template automaton.

The network topology – the spatial distribution of the sensor nodes – represents the direct connections between the nodes. It is the task of the routing protocol to find a path for a packet from one node to the sink. We model the

network topology using a matrix (`topology`) referred as topology matrix. The dimensions of this matrix correspond to the number of nodes in the network. Every element stands for the connectivity from one node (row index) to another (column index). If the matrix should map the topology, negative values can be used, for instance, to represent that a pair of nodes is not connected and positive values can reflect the distance or signal strength between the corresponding nodes. The matrix can also be used to store routing information. In this case, some values can stand for a connection, where a node is in range but not on a routing path.

Using the topology matrix, it is easy to model a fixed routing scheme. The matrix also allows us to model dynamic reconfigurations of the network topology due to the movement of a node or the change of routing information at runtime. To study dynamic reconfigurations, we have modelled controlled flooding which is a dynamic routing scheme. A node broadcasts a packet to all its neighbours and remembers every received packet to control this flooding. If a node receives a packet that has been forwarded earlier, it will be ignored, which avoids cyclic forwarding. The model contains a matrix (`ignore`) with which every node remembers the packets it has received so far. The same matrix is used to remember if an acknowledgment is expected or received. In addition to dynamic routing, the flooding scheme offers the opportunity for an implicit acknowledgment: when a node has transmitted a packet, it will most likely receive it again after a short while, because the receiver(s) will broadcast it again. When a defined time after transmission has passed, a node will call a function (`ack`) to check if a packet has to be retransmitted.

For simplicity, we abstract away from the contents of packets. Every node has an unique identifier and if a node emits a packet, it is named by the identifier of the node. The identifier is also used to determine the length of the packet (`P_S[ID]`). To transmit a packet, a node uses a function named `send`. The function walks through the topology matrix and updates the incoming signal of every node in range, where the incoming signals are modelled by an array named `signal`. Packet collisions that lead to packet losses are modelled with help of the signal array. If a node starts a transmission while another node in range is receiving a signal, the corresponding element in the signal array will be set to a negative value meaning that the packet is corrupted.

Detailed information on the design of the timed automation model, simulation and verification environment settings and results can be found in [1] [2].

4 Conclusion

In this document, we reported on the initial modeling of the service interfaces of the two case study systems ASK CS and BSN. The results presented in this document will be used as input for further modeling activities and verification of the *Credo* language and tools.

In the initial modeling of ASK CS, we limited ourselves to Scenario **SC.1** of the Methodology Document: better exploitation of meta-information within ASK CS. We presented a detailed overview of the structure and behavior of the current ASK CS relevant for this scenario. Based on this overview, we created a set of ten different *network automata*, as proposed in deliverable D1.2 "Specifying Service Interfaces". We conclude that the requirements attached to Scenario **SC.1**, as far as they are addressed in the initial modeling part for this scenario, are completely fulfilled. Moreover, we feel that the modeling approach envisioned at the start of the project is feasible for this case study.

In the case study on biomedical sensor networks the initial timed models were used to analyse packet delivery ratio. In later deliverables we will introduce models based on full Creol-programs, and we will address the other requirements from D6-1 (and its addendum) using Creol tools, e.g., simulation and model checking.

Technical Annexes

D6.2.1 This technical annex surveys the most often used validation tools for wireless sensor networks.

D6.2.2 This technical annex presents a solution for QoS provisioning in biomedical sensor networks.

D6.2.3 This technical annex describes a model-based validation technique, which can be used in QoS validation and parameter tuning in biomedical sensor networks.

References

[1] S. Tschirner, X. Liang, and W. Yi. Model based validation of QoS properties in biomedical sensor networks. *Journal of Logic and Algebraic Programming*, submitted for publication (Feb. 2008).

[2] S. Tschirner and W. Yi. Validating QoS properties in biomedical sensor networks. In *Proc. The 19th Nordic Workshop on Programming Theory (NWPT'07)*, pages 11–15, Oslo, Norway, Oct. 2007.