# Modeling and Analysis of Thread-Pools *
## in an Industrial Communication Platform

Frank S. de Boer[1], Immo Grabe[2,1], Mohammad Mahdi Jaghoori[1], Andries Stam[3], and Wang Yi[4]

[1] CWI, Amsterdam, The Netherlands
[2] Christian-Albrechts-University Kiel, Germany
[3] Almende, The Netherlands
[4] University of Uppsala, Sweden

**Abstract.** Thread pools are often used as a pattern to increase the throughput and responsiveness of software systems. Implementations of thread pools may differ considerably from each other, which urges the need to analyze these differences in a formal manner. We use an object-oriented paradigm to model different thread pools in the context of the ASK system, an industrial communication platform. We use *behavioral interfaces*, high-level behavioral specifications for the objects, as a starting-point for analysis. Based on these behavioral interfaces, functional aspects are modeled in Creol, a high-level modeling language for concurrent objects. We use UPPAAL to create real-time models and to perform schedulability analysis with respect to the behavioral interfaces. We finally check conformance between the real-time and Creol models using test-cases generated from the behavioral interfaces.

## 1 Introduction

Thread pools are an important design pattern used frequently in industrial practice to increase the throughput and responsiveness of software systems, as for instance in the ASK system [4]. The ASK system is an industrial communication platform providing mechanisms for matching users requiring information or services with potential suppliers. A thread pool administrates a collection of computation units referred to as threads and assigns tasks to them. This administration includes dynamic creation or removal of such units, as well as scheduling the tasks based on a given strategy like 'first come first served' or priority based scheduling.

In this paper, we propose the use of the *Credo* tool suite in order to capture the various aspects of thread pools and provide a general framework for their analysis. The *Credo* tool suite offers a methodology for the top-down design and compositional analysis of dynamically reconfigurable systems of concurrent objects [11]. We tailor *Credo* methodology to model and analyze the thread

* This work has been supported by the EU-project IST-33826 *Credo: Modeling and analysis of evolutionary structures for distributed services.*

pools in ASK. The core of this methodology consists of two different *executable* modeling languages:

**Creol [13]** is a high-level object-oriented modeling language for describing the interactions between concurrent objects. Creol focuses on modeling the data and control flow thus reflecting the architectural issues of the implementation at a high level of abstraction. It abstracts from scheduling issues.

**Timed Automata [3]** are used to model scheduling policies as well as the behavioral interfaces of objects which describe the timings of incoming messages and their deadlines. At this level, we abstract from architectural details of the model and focus on schedulability analysis (no deadline miss).

After modeling in Creol and analyzing schedulability with timed automata, we need to establish the conformance between the two models. This is achieved by testing. Test cases are generated from behavioral interface specifications. As observed, the behavioral interfaces are central to the analyses in *Credo*.

*Modeling the Architecture* The ASK system has been developed and evolved over years; different subsystems of ASK use specialized thread pools to address issues like the size of the pool, dynamic creation of threads, load balancing, etc. The implementation of ASK contains thousands of lines of C code, that are difficult to understand and analyze. In this paper, we provide a high-level Creol model that is only tens of lines of Creol code with less distracting implementation details, and is thus more amenable to analysis.

The intended use of the Creol modeling language is to provide a formal object-oriented solution to modeling distributed software systems [13, 8]. The Creol modeling language is implemented by means of an interpreter given in Maude [5] and supported by an Eclipse modeling and analysis environment (developed in the *Credo* project [7]) which includes a compiler and type-checker, a simulation platform that allows both closed world and open world simulation as well as guided simulation, and a graphic display of the simulations.

In Creol, objects are concurrent, i.e., conceptually, each object encapsulates its own processor. Therefore, each object has a single thread of execution. Creol objects communicate by asynchronous message passing. The message queue is implicit in the objects. Furthermore, the scheduling policy is underspecified, i.e., messages in the queue are processed in a nondeterministic order. The running method can voluntarily release the processor using special commands allowing another message to be scheduled. For example, a method can test whether an asynchronous call has been completed, and if not, release the processor; thus modeling synchronous calls.

The abstraction from the internal message queue of each object and the related scheduling policies is one of the most important characteristics of Creol which allows for abstractly modeling a variety of thread pools. In this paper, we give an example of an abstract model in Creol of a basic pool where the threads share the task queue. The shared task queue is naturally represented *implicitly* inside a Creol object (called a resource-pool) that basically forwards the queued tasks to its associated threads also represented as Creol objects (called monks).

*Analyzing Schedulability* We perform schedulability analysis on the automata models of thread pools; this verifies whether tasks are performed within their deadlines. In the context of the ASK system, schedulability ensures that the response times for service requests are always bounded by the deadlines. We use UPPAAL [16] for this purpose. To analyze the *schedulability* of thread pools, their behavioral interfaces are modeled with timed automata. A behavioral interface describes the (expected) arrival times and the deadlines of the tasks; namely, the workload on the thread pool. A given scheduling policy, e.g., earliest deadline first (EDF), is also specified with timed automata. This determines where to insert a newly generated task in the message queue of the resource-pool object. The tasks correspond to the monks in the Creol model.

We provide two approaches to schedulability analysis of thread pools. Once the threads are assumed to run in parallel. This is in line with the assumption in Creol that monk objects, representing the threads, have dedicated processors. Next, we model a situation in which all threads share the same processor. In this case, we model a time-sharing CPU allocation scheme to the concurrent threads. To this end, we use one extra clock for each thread to compute the idle times when it is preempted.

Finally, we test conformance between the timed automata models and the underlying Creol models by generating test cases from the behavioral interfaces. We use the test cases to drive the execution of the Creol model extended with an abstract implementation of the given scheduling policy on the simulation platform.

**Related work** The schedulability analysis in this paper can be seen as the continuation of our previous work [12] on modular analysis of a single-threaded concurrent object with respect to its behavioral interface. In this paper, we extend the schedulability analysis to the case of a multi-threaded scheduler (representing an object-oriented thread pool).

Schedulability has been studied for actor languages [18] and event driven distributed systems [10]. Unlike these works, we work with non-uniformly recurring tasks as in task automata [9] which fits better the nature of message passing in object-oriented languages. The main difference is that in our work, multiple objects share the same task queue. These objects are once modeled as using the same processor, therefore scheduled using a time-sharing policy; next we model them as using independent processors, therefore each object runs in parallel to the others.

The work of [6, 15] is based on extracting automata from code for schedulability analysis. However, they deal with programming languages and timings are usually obtained by profiling the real system. Our work is applied on high-level model. Therefore, our main focus is on studying different scheduling policies and design decisions. *Credo* offers techniques for testing conformance with the C code [1], which is not covered in this paper.

**Outline** In section 2 we give a short introduction to timed automata and the Creol language. The current implementation of the ASK system is explained in section 3. We model the different features and the scheduling of selected thread pools of the ASK system in section 4. Schedulability analysis and testing of conformance between the Creol model of a thread-pool and it behavioral interface is discussed in section 5. We conclude with section 6.

## 2 Preliminaries

### 2.1 Timed Automata

In this section, we define timed automata. We use timed automata for specify behavioral interfaces and perform schedulability analysis.

**Definition 1 (Timed Automata).** *Suppose $\mathcal{B}(C)$ is the set of all clock constraints on the set of clocks $C$. A timed automaton over actions $\Sigma$ and clocks $C$ is a tuple $\langle L, l_0, \longrightarrow, I\rangle$ representing*

- *a finite set of locations $L$ (including an initial location $l_0$);*
- *the set of edges $\longrightarrow \subseteq L \times \mathcal{B}(C) \times \Sigma \times 2^C \times L$; and,*
- *a function $I : L \mapsto \mathcal{B}(C)$ assigning an invariant to each location.*

An edge $(l, g, a, r, l')$ implies that action '$a$' may change the location $l$ to $l'$ by resetting the clocks in $r$, if the clock constraints in $g$ (as well as the invariant of $l'$) hold. Since we use Uppaal [16], we allow defining variables of type boolean and bounded integers. Variables can appear in guards and updates.

A timed automaton is called *deterministic* if and only if for each $a \in \Sigma$, if there are two edges $(l, g, a, r, l')$ and $(l, g', a, r', l'')$ from $l$ labeled by the same action $a$ then the guards $g$ and $g'$ are disjoint (i.e., $g \wedge g'$ is unsatisfiable).

*Networks of timed automata.* A system may be described as a collection of timed automata communicating with each other. In these automata, the action set is partitioned into input, output and internal actions. The behavior of the system is defined as the parallel composition of those automata $A_1 \parallel \cdots \parallel A_n$. Semantically, the system can delay if all automata can delay and can perform an action if one of the automata can perform an internal action or if two automata can synchronize on complementary actions (inputs and outputs are complementary). In a network of timed automata, variables can be defined locally for one automaton, globally (shared between all automata), or as parameters to the automata.

A location can be marked *urgent* in an automaton to indicate that the automaton cannot spend any time in that location. This is equivalent to resetting a fresh clock $x$ in all of its incoming edges and adding an invariant $x \leq 0$ to the location. In a network of timed automata, the enabled transitions from an urgent location may be interleaved with the enabled transitions from other automata (while time is frozen). Like urgent locations, *committed* locations freeze time; furthermore, if any process is in a committed location, the next step must involve an edge from one of the committed locations.

$$
\begin{array}{lll}
IF & ::= & \textbf{interface } N\{(Par)\}^?\{\textbf{inherits } Inh\}^? \\
 & & \textbf{begin } \{\textbf{with } N\ Msig^+\}^? \textbf{ end} \\
Inh & ::= & \{N\{(E)\}^?\}^+_, \\
Par & ::= & \{\{v\}^+_, : N\}^+_, \\
Msig & ::= & \textbf{op } N\{(\{\textbf{in } Par\}^?\ \{\textbf{out } Par\}^?)\}^? \\
CL & ::= & \textbf{class } N\{(Par)\}^? \\
 & & \{\textbf{contracts } Inh\}^?\ \{\textbf{inherits } Inh\}^? \\
 & & \textbf{begin } Vdcl^?\{\{\textbf{with } N\}^?\ Mtd\}^* \textbf{ end} \\
Vdcl & ::= & \textbf{var } \{\{v\}^+_, : N\{= e\}^?\}^+_,
\end{array}
$$

$$
\begin{array}{lll}
Mtd & ::= & \{Msig == \{\,Vdcl;\}^?\ S\}^+ \\
g & ::= & b \mid t? \mid \neg g \mid g \wedge g \\
p & ::= & x.m \mid m \\
S & ::= & \epsilon \mid s; S \\
s & ::= & (S) \mid V := E \mid \textbf{skip} \\
 & & \mid v := \textbf{new } N(E) \mid !p(E) \\
 & & \mid t!p(E) \mid t?(V) \mid p(E;V) \\
 & & \mid \textbf{if } b \textbf{ then } S \textbf{ else } S \textbf{ end} \\
 & & \mid \textbf{await } g \mid \textbf{await } t?(V) \\
 & & \mid \textbf{await } p(E;V) \mid \textbf{release}
\end{array}
$$

**Fig. 1.** BNF grammar for Creol. Curly brackets are used as meta parenthesis, superscript ? for optional parts, superscript * for repetition zero or more times, whereas $\{...\}^+_,$ denotes repetition one or more times with , as delimiter. Identifiers N denote interface, class, type, or method names. Capitalized terms such as $E$, $V$, and $S$, denote lists of the syntactic categories of the corresponding lower-case terms [13, 14].

### 2.2 Creol

The (simplified) syntax of Creol is given in Fig. 1. Here we introduce the basic concepts of Creol. A comprehensive presentation of the formal semantics of Creol (given in rewrite logic, see [13]) is beyond the scope of this paper.

Creol objects are typed by interfaces, whereas classes can implement (indicated by the keyword **contracts**) as many interfaces as necessary. Co-interfaces are used to restrict possible callers, i.e. if a co-interfaces is specified only objects implementing the co-interface are allowed to call methods in the scope of the interface. A co-interface is specified by the keyword **with**. The combination of interfaces as types and co-interfaces enforces type-safe communication. Creol provides the keyword **this** to refer to the actual object and the keyword **caller** to refer to a caller of a method. In Creol concurrent objects communicate via asynchronous method calls. After sending an asynchronous method call, e.g. $t!p(E)$ where $t$ denotes a future to retrieve the value later and $p(E)$ the method call, the process continues execution. The return value of a method call is retrieved via a get operation on the future, e.g. $t?(V)$ where $t$ denotes the future and $V$ the variables to store the result in. Note that get is a blocking operation, i.e. the process blocks until the return value of the method call is computed. A process can also test a method call for termination, e.g. **await** $t?(V)$. In case the future has been calculated the statement is equivalent to $t?(V)$. In case the future has not yet been calculated the statement releases control over the processor. We use $p(E;V)$ as a shorthand for $t!p(E);t?(V)$ and **await** $p(E;V)$ as a shorthand for $t!p(E);$**await** $t?(V)$.

Each object in Creol, upon creation, starts its active behavior by executing its run operation if defined. When receiving a method call a new process is created inside the object to handle the method call. The processes inside an object are interleaved by means of processor release points. A processor release point is reached if a process terminates or reaches a special condition. The **await**

```
1 interface Simple begin
2    with Simple op callMe
3    with Any op response
4 end

6 class Easy contracts Simple begin
7    op run == await this.callMe()
8    with Simple op callMe == ! caller.response()
9    with Any op response == skip
10 end
```

**Fig. 2.** A simple Creol model

keyword opens such a condition. If the condition is false the processor is released otherwise the process continues. The conditions can also query method calls for termination, e.g. **await** $t?(V)$. A process which has not yet started its execution or which is waiting on a condition, that is true, is called enabled. Upon processor release an enabled process is (nondeterministically) chosen to start/continue its execution.

Creol is backed by its formal operational semantics and its strong typing allows for dynamic class upgrades [19]. Since Creol semantics is given in rewrite logic [17], Creol specifications can be executed and analyzed on the Maude [5] platform. Maude is a rewrite engine that can perform analysis like simulation, model checking, etc., on transition systems specified using rewrite logic.

Fig. 2 shows a simple Creol model. The Simple interface defines a callMe operation that can be called only by instances of type Simple, while the response operation does not require any special co-interface. The run method in a class defines its active behavior; thus the class Easy starts with calling its own callMe operation. It waits until the call to callMe has terminated.

## 3 ASK System

ASK is an industrial software system for connecting people to each other. The system uses intelligent matching functionality in order to find effective connections between requesters and responders in a community. ASK has been developed by Almende [2], a Dutch research company focusing on the application of self-organisation techniques in human organisations and agent-oriented software systems. The system is marketed by ASK Community Systems [4]. ASK provides mechanisms for matching users requiring information or services with potential suppliers. Based on information about earlier established contacts and feedback of users, the system learns to bring people into contact with each other in the most effective way. Typical applications for ASK are workforce planning,

customer service, knowledge sharing, social care and emergency response. Customers of ASK include the European mail distribution company TNT Post, the cooperative financial services provider Rabobank and the world's largest pharmaceutical company Pfizer. The amount of people using a single ASK configuration varies from several hundreds to several thousands.

### An Overview of the ASK System

The primary goal of the ASK system is to connect people to other people in the most effective way. The system acts as a *mediator* in establishing the contacts: people can contact the system via various media like telephone or email, and the system itself is also able to contact people via those media. In determining the *effectiveness* of contact establishment, multiple aspects play a role. For example, the rating of *human knowledge and skills* is important in cases where people request contact with specialists or service providers. In these cases, the ASK system is able to ask participants for feedback on the quality of service after the contact. This feedback can be used for optimization of subsequent requests of the same kind. A different role is played by *time schedules*, which indicate when certain people can be reached for certain purposes. The ASK system differentiates between regular plannings and ad-hoc schedules caused by sudden events or delays. Different *communication media* play another role. In most ASK configurations, voice communication (phone, VoIP) is the primary communication medium used, but different media like email and SMS are supported by ASK as well. Moreover, people can own various phone numbers and email addresses, for which they can indicate preferences and time or service dependent usage constraints. The ASK system is able to exploit knowledge about the reacheability of people via specific media, for example in the context of emergency response systems, where people must be contacted within a certain time window. In general, learning from past experiences of all kinds and forecasting based on these experiences plays a crucial role in ASK.

The software of ASK can be technically divided into three parts: the *web front-end*, the *database* and the *contact engine* (see Figure 3). 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

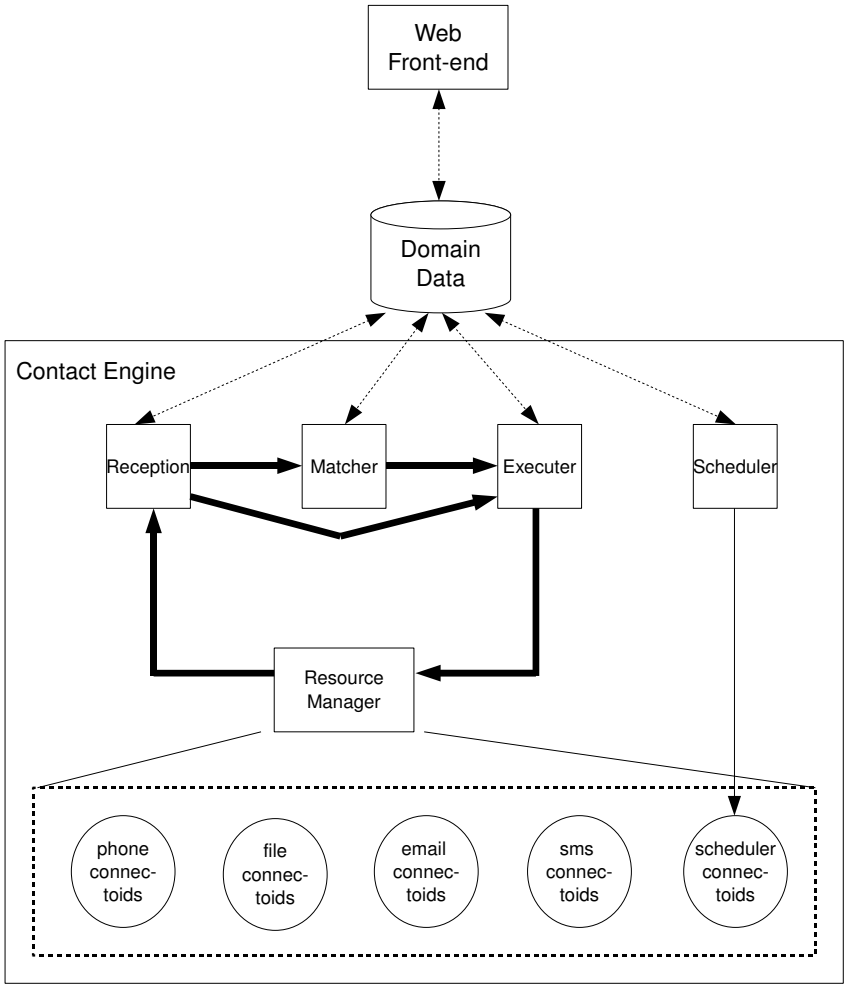



**Fig. 3.** ASK System Overview

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.

**Thread-Pools in ASK**

Each component in the ASK system is equipped with a thread-pool called an *abbey*. 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.

# 4 Modeling

## 4.1 Object-Oriented Modeling

In this section, we first discuss the "low-level" Creol model of the Determinate Abbey in order to illustrate the need for abstraction in modeling (see the appendix for the Creol codes). The low-level models cover all implementation-level details like locks on global variables, explicit tasks and explicit task queues, etc.

In the Determinate Abbey, tasks and monks are kept in explicit lists, and tasks are explicitly implemented (cf. Appendix A). The Dabbey class contains two class variables *DabbeyTaskList* and *DabbeyMonkList*, which are used to hold the tasks-to-be-executed and the (fixed) set of monks. As we see in the more abstract model, it is possible to represent a task queue by the *message queue* of an object. Also, we can abstract away from an explicit list of monks.

In the low-level model, an array of tasks is represented in Creol by using a list, called states. The list is index and the elements of the list model tasks:

- "READY" denotes an empty spot for a task.
- "CREATING" denotes a task in creation.
- "OPEN" denotes a task ready for execution.
- "BUSY" denotes a task that is executed.

We use two functions operating on the list:

- index( list ,status) returns the index of an element with the given status.
- replace( list ,status,index) sets the status of the element at the given index.

A pair of counters keeps track of the number of free spots in the array, readyCounter, and the task waiting for execution, openCounter. The methods *testAndSetCreating*, *testAndSetBusy*, *setTaskOpen*, *setTaskReady* and *setTask* use the replace function model the different phases of the execution of a task. In this manner, we realize an explicit representation of an array in terms of a simple

```
1 class Monk(myPool: ResourcePool) contracts Monk begin
2  op run == !myPool.request()
3  with ResourcePool op task == skip; run(;)
4 end
5 class ResourcePool(nofMonks: Int) contracts ResourcePool
6 begin
7  var freeMonks: Set[Monk];
8  op init == var monk: Monk;var n: Int := nofMonks;
9    freeMonks := {};
10   while (n>0) do monk:=new Monk(this); n:=n-1 end
11 op chooseMonk(out monk:Monk) == await ~isempty(freeMonks);
12   monk := choose(freeMonks);
13   freeMonks := remove(freeMonks,monk)
14 op task == var monk: Monk; chooseMonk(;monk); !monk.task()
15 with Monk op request == freeMonks := add(freeMonks,caller)
16 with Outside op addTask == !task()
17 end
```

**Fig. 4.** The high-level Minimal Abbey

list. For the analysis of the thread-pool, the fact that we use an array is an implementation detail and irrelevant to the analysis. Creol methods are executed mutually exclusive so the methods *testAndSetCreating* and *testAndSetBusy* in fact model "test-and-set" operations. Overall we model a lock on the array providing us with mutual exclusive access and ensuring consistency of the list and the task handling.

The low-level model of the Self-scaling Abbey is even more complex. Tasks and Monks are kept in queues, while a triplet of counters is used to count the number of tasks, monks and busy monks. Creation and deletion of monks is done by a "shepherd" monk. Monks are killed by letting them execute a "poison" task, which causes the monk to terminate. In Appendix C, the looping shepherd task is shown. Once this task is executed by a monk, that monk acts as the shepherd in the abbey. Note in particular the large amount of class parameters, which are needed inside the task to manage the amount of monks in the monk queue. In fact, the dynamicity of the amount of monks, which is in itself an important property of the Self-scaling Abbey, can be modeled in a more abstract manner. The focus should be on the principle of and constraints on creation and deletion, instead of on the specific solution as implemented in the ASK system.

**High-Level Models** As a high-level base model, we created a *Minimal Abbey* (Mabbey), as shown in Figure 4. The Mabbey acts as the "mother" of all abbeys

– the Determinate Abbey and the Self-scaling Abbey are derived from it, as well as other types of abbeys. The two most important classes are the *Monk* class and the *ResourcePool* class. Their class specifications and interfaces they contract are shown in Figure 4.

The task list is modeled implicitly, in terms of the message queue of the object. By using the proper way of messaging, i.e. synchronous or asynchronous, blocking and non-blocking behavior for inserts in the queue can be modeled. The size of the queue can be limited by means of a class variable *nofTasks* which represents the number of tasks currently in the task queue (this construct is used in the Determinate Abbey). A list for the monks is not modeled: it is not needed at this level of abstraction. A variable *freeMonks* is used to hold all monks which are currently not executing a task. Based on simple requests issued by the monks themselves, the monks are added to the list of free monks. Tasks are modeled in terms of simple methods inside the monk class – this is enough, as for our analysis the functional differences between *tasks*, as opposed to the differences between *thread-pools*, is irrelevant.

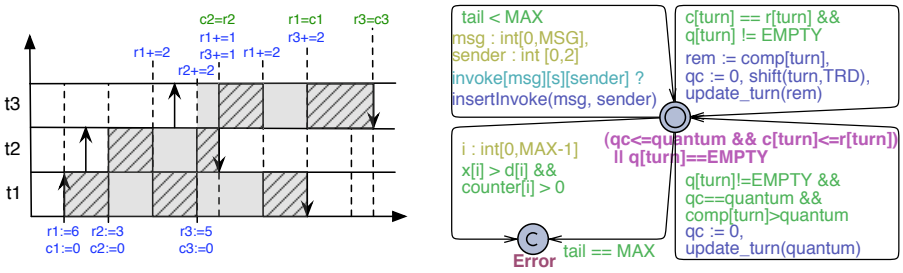
### 4.2 Real-Time Modeling in UPPAAL

In this section, we model a thread-pool using timed automata in UPPAAL. We use these UPPAAL models in the next section for schedulability analysis of real-time models of the ASK system. We model a thread-pool as a scheduler automaton taking tasks from a queue and dispatching them among concurrent threads. This model can be seen as an extension of the framework for schedulability analysis of concurrent objects [12] to a situation in which objects share the message/task queue.

We separate the task queue in two parts: an *execution* part and a *buffer*. The execution part includes the tasks that are being executed. This part needs one slot for each thread and is therefore as big as the number of threads; we assume a fixed number of threads given a priori. Before beginning their execution, tasks are queued based on a given scheduling strategies, e.g., EDF, FPS, etc., in the rest of the queue (i.e., the buffer part).

In the rest of this section, we show two approaches in modeling concurrent threads sharing a task queue. 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 'executing' threads share the processor; therefore, each task runs a period of time before it is interrupted by the scheduler to run the next one. In both cases, when a task reaches the execution part, it will not be put back to the buffer part. We call this *weak non-preemption*, i.e., in the special case of one thread, it behaves like a non-preemptive scheduler. The scheduler (responsible for dispatching methods) and the queue (responsible for receiving messages) can be modeled in the same automaton or separately.

*Time-Sharing.* In this model, execution threads share one CPU. Therefore, the tasks in the execution part of the queue are interleaved. We call a thread active




(a) Executing three tasks (quantum = 2) (b) Scheduler including a queue

**Fig. 5.** Modeling a time-sharing scheduler for a thread-pool

if a task is assigned to it. At its turn, each active thread gets a fixed time slot (called a *quantum*) for execution. If the assigned task does not finish within this quantum, the thread is preempted and the control is given to the next active thread. Recall that we use weak non-preemption, 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 abstraction is necessary to enable the modeling of preemption of tasks at any arbitrary time (i.e., the selected quantum). Figure 5.(a) shows intuitively how three threads are scheduled. The up-arrows show when a task is released. A down-arrow indicates the completion of the task, after which the thread remains inactive in this scenario. The tasks assigned to `t1`, `t2` and `t3` have the computation times of 6, 3 and 5, respectively, and the preemption quantum is 2.

We associate to each thread a clock `c` and an integer variable `r` for response time, i.e., the execution plus idle time, which is updated dynamically while an active thread is idle. A task finishes when its clock reaches the expected response time value (`c==r` shown in green in Figure 5.(a)). When a task is assigned to a thread, only at the next quantum the thread becomes active, i.e., the thread clock is reset to zero and `r` is given the computation time of the task. In Figure 5.(a), the active period of the threads is shown in gray, during which the hatched pattern denotes when the thread has the CPU. At every context-switch (shown by dashed lines in Figure 5.(a)), the response-time variables of all idle threads are increased to reflect their recent idle time.

Figure 5.(b) shows the formal model of the time-sharing scheduler, which includes the message queue. The clock `qc` is used to keep track of time slots. The invariant on the initial location of the automaton ensures progress when a context-switch should occur and on the other hand it doesn't deadlock when the queue is empty (`q[turn]==EMPTY`). The edges on the right-side of the automaton model context-switch; in `update_turn` response-time variables are updated:

```
for (i = 0; i < TRD; i++) {
  if (q[i] != EMPTY) {
    if (i != turn) r[i] += quantum;
    else comp[ca[i]] -= quantum;  // remaining computation time
} }
```

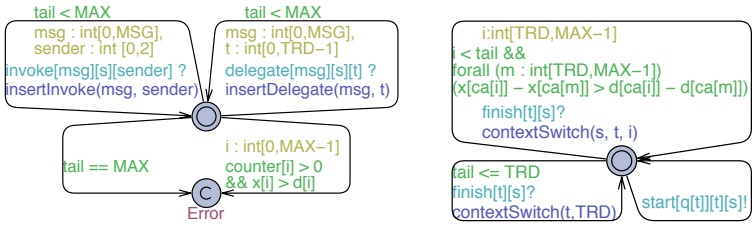
The first check `q[i] != EMPTY` makes sure that the thread `i` is active, i.e., a task is assigned to it. The variable `turn` shows the thread that was just running. For threads `i != turn` the response time `r[i]` is increased to cover their idle time, whereas for the thread that is just stopped we update `comp`, the remaining computation time. The value of `comp` is used when a task finishes before a quantum is reached, e.g., tasks `t2` and `t3` in Figure 5.(a). In this case, the response time of idle threads is increased by `comp` instead of `quantum`.

Finally the variable `turn` is updated at every context-switch, such that the next active thread is selected. If it happens that there are no more active threads, i.e., the last task just finished, `turn` will keep its old value, as modeled in the for loop below:

```
turn = (turn + 1) % TRD;
for (i=0; q[turn]==EMPTY && i<TRD-1; i++) {
  turn = (turn + 1) % TRD;
}
```

The edges on the left model insertion of tasks into the queue using the `insertInvoke` function, in which the scheduling policy can be modeled. In this model, the deadline or priority values for tasks can be modeled statically. Each queue slot is assigned a clock `x` which shows how long a task sits in that queue slot. The automaton also takes care that when a task misses its deadline (`x[i] > d[i]`) or the queue is full (`tail == MAX`), it goes to the `Error` state.

*Parallel Threads.* In this model, every thread is assumed to have a dedicated processing unit, but they share one task queue. 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. In this model, the queue and the scheduling strategy are modeled in separate automata. Figure 6.(a) shows a queue of size `MAX` which stores the tasks in the order of their arrival. This automaton is parameterized in `s` which holds the identity of the object. It accepts any message from any sender on the `invoke` channel, using the UPPAAL 'select' statement on `msg` and `sender`. To check for deadlines, a clock `x` is assigned to each task in the queue, which is reset when the task is added, i.e., in `insertInvoke` function.




(a) A queue shared between threads (b) An EDF scheduler

**Fig. 6.** A scheduler for parallel threads in UPPAAL

This model allows for specifying tasks as timed automata; therefore, tasks can create subtasks with self-calls. As a result, we don't need `c` and `r`. The `delegate` channel is dedicated to self calls that create subtasks inheriting the parent's deadline. To identify the parent, it receives the thread identity as `t`. Inheriting the deadline is modeled by reusing the clock `x` assigned to the parent task (which is in turn assigned to thread `t`). The number of tasks (and subtasks) assigned to clock `x[i]` is stored in `counter[i]`. This is handled in the `insertDelegate` function. The queue goes to `Error` state if the a task misses its deadline (`x[i] > d[i]`) or the queue is full.

Figure 6.(b) shows how a scheduling strategy can be implemented. This automaton should be replicated for every thread, thus parameterized in `t` as well as the object identity `s`. The different instances of this automaton will be assigned each to one slot in the queue, namely `q[t]`. This example models an EDF (earliest deadline first) scheduling strategy. The remaining time to the deadline of a task at position `i` in the queue is obtained by `x[ca[i]]-d[ca[i]]`. When the thread `t` finishes its current task (`finish[t][s]`), it selects the next task from the buffer part of the queue for execution by putting it in `q[t]`; next, it is started (`start[q[t]][t][s]`).

## 5 Analysis

We use timed automata to do schedulability analysis of our program model in Uppaal. This analysis provides as a result a scheduling for a given strategy like Earliest-Deadline-First. We take the resulting strategy to the Maude level and test the model against it.

### 5.1 Schedulability Analysis of the Automata Model

Schedulability analysis is checking whether tasks can be accomplished before their deadlines. In this section, we analyze the schedulability of the timed automata models of thread pools given in the previous section. Tasks in this model correspond to the methods of monk objects (cf. the Creol models in Section 4). The model of a thread pool is not enough for schedulability analysis, because we need to know how fast tasks are generated and what their deadlines are. The timed automaton specifying the task generation patterns serves as the behavioral interface of the thread pool. The behavioral interface can be seen as a model of the environment and captures the work-load of the ASK system. In this section, we assume two threads and therefore two instances of the monk object.

Figure 7 shows two models of behavioral interfaces for our model of thread pools. In these diagrams, `Right` shows the identity of an object in the environment that sends messages task1 and task2 to the resource pool under analysis; the identity of the thread pool is given as `self`. In one model, the tasks are generated independently with an inter-arrival time of at least 9 time units between every two occurrences of the instances of the same task type. In the other model, tasks are generated one after the other in a sequential manner.

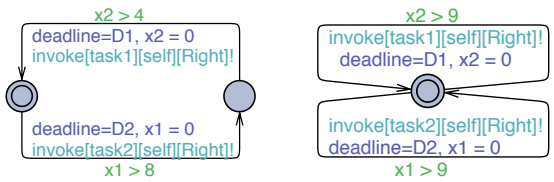
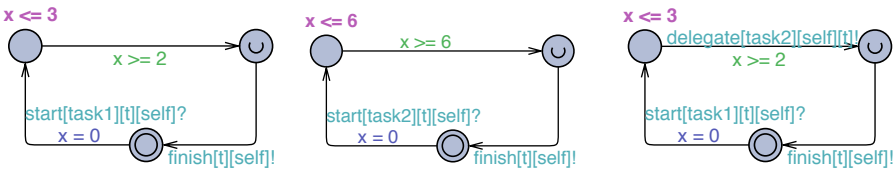



**Fig. 7.** Generating task instances sequentially (left) or in parallel (right)

To perform schedulability analysis by model checking, we need to find a reasonable queue length to make the model finite. The execution part of the queue is as big as the number of threads, and the buffer part is at least of size one. As in single-threaded situation of objects [12], a system is schedulable only if does not put more than $\lceil D_{max}/B_{min} \rceil$ messages in its queue, where $D_{max}$ is the biggest deadline in the system, and $B_{min}$ is the best-case execution time of the shortest task. As a result, schedulability is equivalent to the `Error` state not being reachable with a queue of length $\lceil D_{max}/B_{min} \rceil$. Therefore, schedulability analysis does not depend on whether an upper bound on queue length is assumed (Dabbey) or not (Sabbey). When analyzing the Determinate Abbeys (Dabbeys), one can assume a smaller queue bound if necessary to check for queue overflow situations.

To use the time-sharing model of a thread pool, a task is modeled as a computation time. The two task types are given the computation times of 3 and 6 time units. This model is analyzed with a queue length of three where two concurrent threads are assumed. Given the behavioral interface with parallel task generation, the minimum deadline for which the model is schedulable is 7 and 9 for task1 and task2, respectively. For sequential task generation, the deadlines can be reduced to 5 and 6 for task1 and task2, respectively.

When the thread pool for parallel threads is applied, one can model tasks as timed automata; two simple task models are given in Figure 8.(a). In this model, task1 has a computation time of between 2 to 3 time units, and task2 takes 6 time units to execute. Using either of the two behavioral interfaces above, the model is schedulable with the deadlines 3 and 6 for task1 and task2, respectively.




(a) Modeling tasks as computation times (b) Generating a subtask

**Fig. 8.** Modeling tasks (corresponding to monks) for parallel threads

In parallel generation of tasks, parallel threads can handle the tasks faster, and therefore, smaller deadlines are needed for schedulability. It turns out that the parallelism of the threads does not affect the schedulability of tasks that are created sequentially.

More complicated models can include sub-task generation. Figure 8.(b) shows a model of task1 which creates an instance of task2. The schedulability analysis of this model with a queue length of three fails due to queue overflow. This implies that a determinate abbey with a too small buffer size can fail. By increasing the size of the queue to four, the model will be schedulable given a deadline of 9 to the task1 (and task2 still needs a deadline of 6). This shows that a determinate abbey can still be useful if a big enough buffer size is used.

### 5.2 Conformance Testing

From the analysis of the behavioral interfaces we get a sequence of time-stamped tasks with their deadlines of the form

$$(t_1, \mathrm{task_1}(c_1, d_1)), \ldots, (t_n, \mathrm{task_n}(c_n, d_n))$$

where $t_i$ denotes the time at which $task_i$ has been queued, where $task_i$ has the computation time $c_i$ and s deadline $d_i$. For analysis, we assume a scheduling strategy like Earliest Deadline First (EDF).

We use this information to provide a discrete time model for our Creol model. Instead of modeling continuos time, e.g., by means of ticks, we *jump* from a time-stamp to the next time-stamp. This is implemented by a variation of the Creol interpreter. Assuming $(t_i, \mathrm{task_i}(c_i, d_i))$ is the task with the smallest time-stamp at a given point, $(t_i, \mathrm{task_i}(c_i, d_i))$ is removed from the queue and execution of $\mathrm{task_i}$ starts. The first thing to check is whether $\mathrm{task_i}$ can be computed in time, i.e. whether $t_i + c_i \leq d_i$. If this condition is violated we report a scheduling error indicating a conformance conflict, since the analysis of the timed automata return a *sound* scheduling. Otherwise, assume that $\mathrm{task_i}$ was assigned to $\mathrm{monk_j}$ a new time-stamp $t_i + c_i$ is introduced indicating when $\mathrm{monk_j}$ is available again. Depending on an assumption on the execution times of the tasks (optimistic or pessimistic) the shortest or longest execution time can be chosen or an arbitrary execution time within the given limits.

As long as there are monks available we jump to the next smallest time-stamp and schedule the associated event. In case we run out of monks we *jump* to the next time-stamp $t_f$ at which point a monk is again available. Now every $\mathrm{task_i}$ with time-stamp $t_i \leq t_f$ can be scheduled and the actual task to be scheduled is selected according to the implemented strategy, e.g., in the case of a EDF, the task with the closest deadline among the schedulable tasks is selected.

When all the tasks in the test case have been consumed, we compare the Maude scheduling to the UPPAAL scheduling. The UPPAAL scheduling can be easily obtained by simulating the scheduler automaton (cf. Section 4) put together with a linear timed automaton representing the test-case. If any deviation between the Maude and UPPAAL schedulings is observed, we have found a counter-example to conformance.

## 6 Conclusion

In this paper, we employed the *Credo* methodology for the design and analysis of thread pools in an industrial communication platform. This methodology is based on a separation of concerns between high-level modeling of architectural features of thread pools (in Creol) and their analysis for schedulability (using timed automata). We use timed automata to specify scheduling policies; whereas the high-level Creol models abstract from scheduling concerns and focus on the architectural modeling using concurrent objects.

Behavioral interfaces are central to the analyses. Thread pools are analyzed for schedulability with respect to the task generation pattern given in the behavioral interfaces modeling the work-load. We also derive test cases from the behavioral interfaces for checking conformance between the timed automata and Creol models, thus bridging the gap between the two levels of abstraction.

Future work consist first of all of an implementation of the method for testing conformance between a Creol model of a thread-pool and the timed automata models. Another line of future research consists of real-time extensions of the Creol language itself to support a full development cycle. Thus, one can generate code for application-specific schedulers from Creol models.

## References


1. B. Aichernig, A. Griesmayer, R. Schlatte, and A. Stam. Modeling and testing multi-threaded asynchronous systems with Creol. In *2nd Workshop on Harnessing Theories for Tool Support in Software (TTSS'08)*, ENTCS. Elsevier, 2009. to appear.
2. The Almende research company. http://www.almende.com/.
3. R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.
4. The ASK community systems. http://www.ask-cs.com/.
5. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: specification and programming in rewriting logic. *Theoretical Computer Science*, 285(2):187–243, 2002.
6. E. Closse, M. Poize, J. Pulou, J. Sifakis, P. Venter, D. Weil, and S. Yovine. TAXYS: A tool for the development and verification of real-time embedded systems. In G. Berry, H. Comon, and A. Finkel, editors, *Proc. Computer Aided Verification (CAV01)*, volume 2102 of *LNCS*, pages 391–395. Springer, 2001.
7. Credo - modeling and analysis of evolutionary structures for distributed services. http://credo.cwi.nl/.
8. F. S. de Boer, D. Clarke, and E. B. Johnsen. A complete guide to the future. In R. D. Nicola, editor, *ESOP*, volume 4421 of *LNCS*, pages 316–330. Springer, 2007.
9. E. Fersman, P. Krcal, P. Pettersson, and W. Yi. Task automata: Schedulability, decidability and undecidability. *Information and Computation*, 205(8):1149–1172, 2007.
10. J. J. G. Garcia, J. C. P. Gutierrez, and M. G. Harbour. Schedulability analysis of distributed hard real-time systems with multiple-event synchronization. In *Proc. 12th Euromicro Conference on Real-Time Systems*, pages 15–24. IEEE, 2000.

11. I. Grabe, M. M. Jaghoori, B. Aichernig, C. Baier, T. Blechmann, F. de Boer, A. Griesmayer, E. B. Johnsen, J. Klein, S. Klüppelholz, M. Kyas, W. Leister, R. Schlatte, A. Stam, M. Steffen, S. Tschirner, L. Xuedong, and W. Yi. Credo methodology. Modeling and analyzing a peer-to-peer system in Credo. In *3rd International Workshop on Harnessing Theories for Tool Support in Software (TTSS'09)*, ENTCS. Elsevier, 2009. To appear.
12. M. M. Jaghoori, F. S. de Boer, T. Chothia, and M. Sirjani. Schedulability of asynchronous real-time concurrent objects. *J. Logic and Alg. Prog.*, 78(5):402 – 416, 2009.
13. E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. *Software and Systems Modeling*, 6(1):35–58, 2007.
14. E. B. Johnsen, O. Owe, and I. C. Yu. Creol: A type-safe object-oriented model for distributed concurrent systems. *Theoretical Computer Science*, 365(1-2):23–66, 2006.
15. C. Kloukinas and S. Yovine. Synthesis of safe, QoS extendible, application specific schedulers for heterogeneous real-time systems. In *Proc. 15th Euromicro Conference on Real-Time Systems (ECRTS 2003)*, pages 287–294. IEEE Computer Society, 2003.
16. K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. *STTT*, 1(1-2):134–152, 1997.
17. J. Meseguer. Conditioned rewriting logic as a united model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.
18. L. Nigro and F. Pupo. Schedulability analysis of real time actor systems using coloured petri nets. In *Proc. Concurrent Object-Oriented Programming and Petri Nets*, volume 2001 of *LNCS*, pages 493–513. Springer, 2001.
19. I. C. Yu, E. B. Johnsen, and O. Owe. Type-safe runtime class upgrades in creol. In *Proc. the 8th Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, volume 4037 of *LNCS*, pages 202–217. Springer, 2006.

## A A class for the Low-level Determinate Abbey

```
1 interface Dabbey inherits Abbey begin
2 end

4 class Dabbey(size: Int) contracts Dabbey begin
5    var taskList: DabbeyTaskList;
6    var monkList: DabbeyMonkList;
7    op init ==
8       taskList := new DabbeyTaskList(size);
9       monkList := new DabbeyMonkList(size, taskList)
10   with Any op dispatchTask(in task: Task) ==
11      var i: Int;
12      taskList.testAndSetCreating(;i);
13      taskList.setTask(i,task;);
14      taskList.setTaskOpen(i;)
15 end
```

## B The TaskList class for the Low-level Determinate Abbey

```
1 class DabbeyTaskList(size: Int) contracts DabbeyTaskList
2 begin
3    ...
4    with Dabbey op testAndSetCreating(out index: Int) ==
5       await readyCounter > 0;
6       index := index(states, "READY");
7       states := replace(states, "CREATING", index);
8       readyCounter := readyCounter - 1
9    with DabbeyMonk op testAndSetBusy(out index: Int) ==
10      await openCounter > 0;
11      index := index(states, "OPEN");
12      states := replace(states, "BUSY", index);
13      openCounter := openCounter - 1
14   with Dabbey op setTaskOpen(in index: Int) ==
15      states := replace(states, "OPEN", index);
16      openCounter := openCounter + 1
17   with DabbeyMonk op setTaskReady(in index: Int) ==
18      states := replace(states, "READY", index);
19      readyCounter := readyCounter + 1
20   ...
21 end
```

## C The Shepherd task class of the Self-scaling Abbey

```
1 class ShepherdTask(
2   taskId: Int, taskCounter: Counter, monkCounter: Counter,
3   busyCounter: Counter, mmax: Int, mrate: Int,
4   taskQueue: SabbeyTaskQueue, monkQueue: SabbeyMonkQueue)
5   contracts ShepherdTask
6 begin
7    op shepherdLoop ==
8       ...
9       taskCounter.val(;t); monkCounter.val(;m);
10      busyCounter.val(;mbusy); mfree := m - mbusy;
11      if ((m < mmax) && ((mfree - t) < (m / mrate))) then
12         amountToCreate := t - mfree + (m / mrate);
13         if (amountToCreate > (mmax - m)) then
14            amountToCreate := mmax -m
```

```
15            end;
16            monkQueue.createMonks(amountToCreate;)
17        end;
18        if (mfree > (m / 2)) then
19            task := new PoisonTask(0);
20            taskQueue.enqueueTask(task;)
21        end;
22        release;
23        shepherdLoop(;)
24    ...
25 end
```

## D ResourcePool class for the high-level Self-scaling Abbey

```
1  class ResourcePool(nofMonks: Int, maxNofMonks: Int)
2   contracts ResourcePool begin
3  var freeMonks: Set[Monk]; var nofTasks: Int;
4  ...
5  op task ==
6    var monk: Monk; chooseMonk(;monk);
7    !monk.task(); nofTasks := nofTasks - 1
8  op poisonTask ==
9    var monk: Monk;
10   chooseMonk(;monk);
11   !monk.poisonTask()
12 op shepherd == var monk: Monk;
13   await (nofTasks>nofMonks*2) || (#(freeMonks)>nofMonks/2);
14   if (nofTasks > nofMonks*2) then
15     if (nofMonks < maxNofMonks) then
16       monk := new Monk(this);
17       nofMonks := nofMonks + 1
18     end else if (nofMonks > 1) then
19       poisonTask(;);
20       nofMonks := nofMonks - 1
21   end end
22 ...
23 with Outside op addTask ==
24   nofTasks := nofTasks + 1;
25   !task(); !shepherd()
26 end
```