# 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. It can be used to simulate the behaviors of the ASK system for debugging, testing, and formal verification. Based on the Creol model, we have constructed a real-time model of the ASK system in Uppaal. The real-time model has been used to check the schedulability of the thread pools with respect to 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

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

objects [11]. We tailor *Credo* methodology to model and analyze the thread 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**

```
interface Simple begin
    with Simple op callMe
    with Any op response
end

class Easy contracts Simple begin
    op run == await this.callMe()
    with Simple op callMe == ! caller.response()
    with Any op response == skip
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

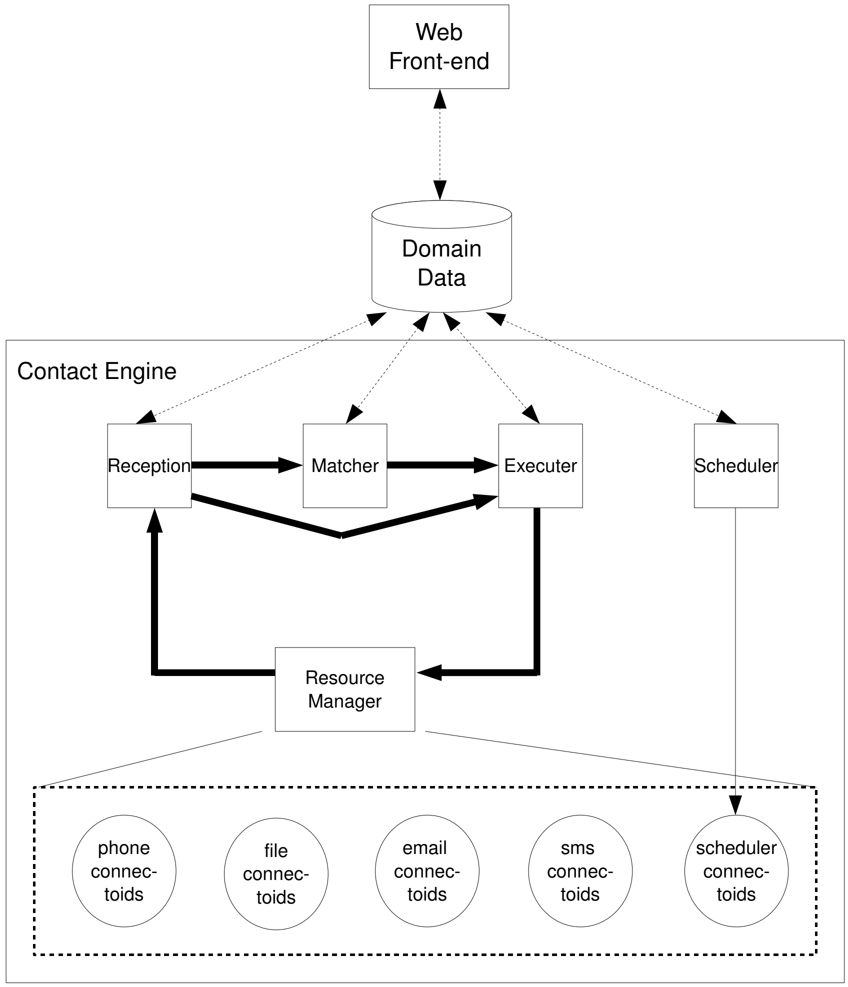



**Fig. 3.** ASK System Overview

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

The low-level models explicitly express all implementation-level details like locks on global variables, explicit tasks and explicit task queues, etc. In this section, we show the low-level Creol model of the Determinate Abbey in order to illustrate the need abstraction.

In the Determinate Abbey, tasks and monks are kept in explicit lists, and tasks are explicitly implemented. This can be seen in the code fragment shown in Figure 4: 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 will see in the more abstract model, it is possible to use the *message queue* of an object to implicitly represent a task queue. Also, we can easily abstract away from an explicit list of monks.

In the low-level model, an array of tasks is “mimicked” in Creol by using a list, an index for the list and a replace method to replace values at a specific index in the list. This is shown in Figure 5. The methods *testAndSetCreating*, *testAndSetBusy*, *setTaskOpen*, *setTaskReady* and *setTask* all contain replace statements. In this manner, we realize an explicit representation of an array in terms of a simple list. For purposes of functional analysis of the thread-pool, the fact that we use an array is an implementation detail and irrelevant for analyzing its functionality. Figure 5 also shows that the task list contains methods which correspond to primitive “test-and-set” operations. Thereby, we model locks on the array. At a functional modeling level, this behavior is in fact irrelevant as well.

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,

```
interface Dabbey inherits Abbey begin
end

class Dabbey(size: Int) contracts Dabbey begin
    var taskList: DabbeyTaskList;
    var monkList: DabbeyMonkList;
    op init ==
        taskList := new DabbeyTaskList(size);
        monkList := new DabbeyMonkList(size, taskList)
    with Any op dispatchTask(in task: Task) ==
        var i: Int;
        taskList.testAndSetCreating(;i);
        taskList.setTask(i,task;);
        taskList.setTaskOpen(i;)
end
```

**Fig. 4.** A class for the Low-level Determinate Abbey

```
class DabbeyTaskList(size: Int) contracts DabbeyTaskList
begin
    ...
    with Dabbey op testAndSetCreating(out index: Int) ==
        await readyCounter > 0;
        index := index(states, ”READY”);
        states := replace(states, ”CREATING”, index);
        readyCounter := readyCounter − 1
    with DabbeyMonk op testAndSetBusy(out index: Int) ==
        await openCounter > 0;
        index := index(states, ”OPEN”);
        states := replace(states, ”BUSY”, index);
        openCounter := openCounter − 1
    with Dabbey op setTaskOpen(in index: Int) ==
        states := replace(states, ”OPEN”, index);
        openCounter := openCounter + 1
    with DabbeyMonk op setTaskReady(in index: Int) ==
        states := replace(states, ”READY”, index);
        readyCounter := readyCounter + 1
    ...
end
```

**Fig. 5.** The TaskList class for the Low-level Determinate Abbey

```
class ShepherdTask(
  taskId: Int, taskCounter: Counter, monkCounter: Counter,
  busyCounter: Counter, mmax: Int, mrate: Int,
  taskQueue: SabbeyTaskQueue, monkQueue: SabbeyMonkQueue)
  contracts ShepherdTask
begin
   op shepherdLoop ==
      ...
      taskCounter.val(;t); monkCounter.val(;m);
      busyCounter.val(;mbusy); mfree := m − mbusy;
      if ((m < mmax) && ((mfree − t) < (m / mrate))) then
         amountToCreate := t − mfree + (m / mrate);
         if (amountToCreate > (mmax − m)) then
            amountToCreate := mmax −m
         end;
         monkQueue.createMonks(amountToCreate;)
      end;
      if (mfree > (m / 2)) then
         task := new PoisonTask(0);
         taskQueue.enqueueTask(task;)
      end;
      release;
      shepherdLoop(;)
    ...
end
```

**Fig. 6.** The Shepherd task class of the Self-scaling Abbey

which causes the monk to cease to exist. In Figure 6, the infinite 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 for managing 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 far more implicit and 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 7. The Mabbey acts as the "mother" of all abbeys

```
class Monk(myPool: ResourcePool) contracts Monk begin
 op run == !myPool.request()
 with ResourcePool op task == skip; run(;)
end
class ResourcePool(nofMonks: Int) contracts ResourcePool
begin
 var freeMonks: Set[Monk];
 op init == var monk: Monk;var n: Int := nofMonks;
    freeMonks := {};
    while (n>0) do monk:=new Monk(this); n:=n-1 end
 op chooseMonk(out monk:Monk) == await ~isempty(freeMonks);
    monk := choose(freeMonks);
    freeMonks := remove(freeMonks,monk)
 op task == var monk: Monk; chooseMonk(;monk); !monk.task()
 with Monk op request == freeMonks := add(freeMonks,caller)
 with Outside op addTask == !task()
end
```

**Fig. 7.** The high-level Minimal Abbey

```
class ResourcePool(nofMonks: Int, maxNofMonks: Int)
  contracts ResourcePool begin
 var freeMonks: Set[Monk]; var nofTasks: Int;
 var nofMonks: Int;
 ...
 op task == var monk: Monk; chooseMonk(;monk);
    !monk.task(); nofTasks := nofTasks - 1
 op poisonTask ==
    var monk: Monk; chooseMonk(;monk); !monk.poisonTask()
 op shepherd == var monk: Monk;
    await (nofTasks>nofMonks*2) || (#(freeMonks)>nofMonks/2);
    if (nofTasks > nofMonks*2) then
      if (nofMonks < maxNofMonks) then
        monk := new Monk(this); nofMonks := nofMonks + 1
      end else if (nofMonks > 1) then
        poisonTask(;); nofMonks := nofMonks - 1
    end end
 ...
 with Outside op addTask ==
  nofTasks := nofTasks + 1; !task(); !shepherd()
end
```

**Fig. 8.** ResourcePool class for the high-level Self-scaling Abbey

– 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 7.

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.

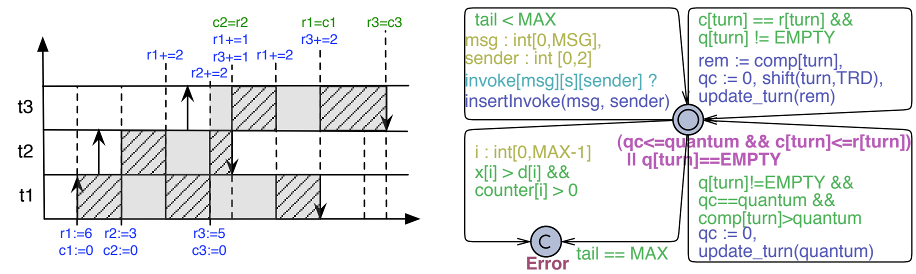
### 4.2 Real-Time Modeling in Uppaal

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 one 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. At its turn, each active




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

**Fig. 9.** Modeling a time-sharing scheduler for a thread-pool

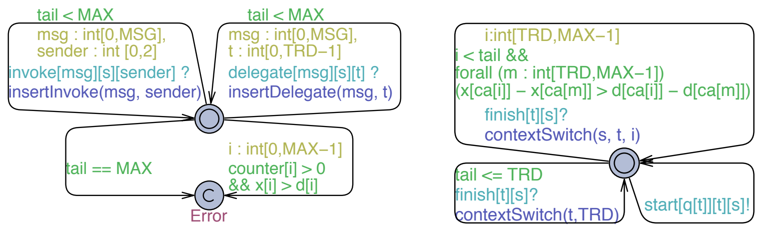
thread gets a fixed time slot (called a *quantum*) for execution. Then the active thread is preempted to give the control to the next thread for execution. Note that before entering the execution part of the queue, no preemption can occur, i.e., once a task is in the execution part it cannot be put back into the buffer part.

In this model, each task is modeled only as a computation time. This abstraction is necessary to enable the modeling of preemption of tasks at any arbitrary time (i.e., the selected quantum). Figure 9.(a) shows how three threads are scheduled. The up-arrows show when a task is assigned to a thread. A down-arrow indicates the completion of the task, after which the thread remains idle 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.

To keep track of execution and idle times of active threads, we associate to each thread a clock `c` and an integer variable `r`; `r` holds the expected response time of the task at every given moment. A task finishes when its clock reaches the expected response time value. As shown in the diagram, when a task is released the corresponding clock and response-time variable are reset at the next quantum; the clock is reset to zero and `r` is given the computation time of the task. The hatched pattern is used in this diagram when a thread has the CPU, whereas the dark solid parts show the idle time of an active thread. At every context-switch (shown by dashed lines), the response-time variables of all idle threads are increased by the quantum value, as shown below:

```
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 we just running. For threads `i != turn` the response time `r[i]` is increased, whereas for the thread that is just stopped we update the remaining computation time, namely




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

**Fig. 10.** A scheduler for parallel threads in UPPAAL

`comp`. The value of `comp` is used when a task finishes before a quantum is reached, e.g., tasks `t2` and `t3`. In this case, the response time of other 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;
}
```

Figure 9.(b) shows the model of the queue combined with the time-sharing scheduler. Since tasks are only modeled as a computation time, they cannot generate subtasks and therefore no `delegate` channel is needed (cf. parallel threads). The clock `qc` is used for handling the quantum 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 scheduling policy can be modeled in the `insertInvoke` function. In this model, the deadline or priority values for tasks can be modeled statically. When modeling the task generation pattern, the computation and deadline values for tasks should be given.

*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 10.(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.

The `delegate` channel is dedicated to self calls creating subtasks that inherit the parent's deadline. To identify the parent, it receives the thread identity as `t`. Inheriting the deadline is modeled by reusing the clock assigned to the 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 10.(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

### 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. In order to perform schedulability analysis, we need to specify the tasks as well as their generation pattern. Tasks in this model correspond to the methods of monk objects (cf. the Creol models in Section 4). The task generation pattern or the environment modeling is to capture work load for the ASK system and is specified as a timed automaton. Such a timed automaton specifying the task generation patterns serves as the behavioral interface of the given abbey. In this section, we assume two threads and therefore two instances of the monk object.

Figure 11 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;

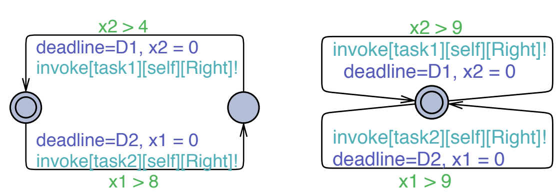



**Fig. 11.** Generating task instances sequentially (left) or in parallel (right)

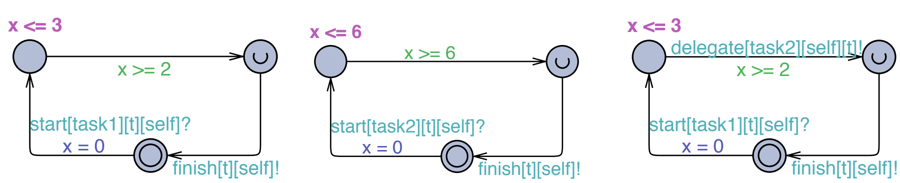
the identity of the thread pool is given as `self`. In one model, the tasks are generated independently with an inter-arrival time of at least 9 time units between every two occurrences of the instances of the same task type. In the other model, tasks are generated one after the other in a sequential manner.

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 12.(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. 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 12.(b) shows a model of task1 which creates an instance of task2. The schedulability




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

**Fig. 12.** Modeling tasks (corresponding to monks) for parallel threads

analysis of this model with a queue length of three fails due to queue overflow. This implies that a deterministic 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 self-scaling abbey can perform better in situations that tasks can generate subtasks.

### 5.2 Conformance Testing

Finally, we sketch a general methodology for testing conformance between a high-level Abbey and its behavioral interface. Consider for example the ResourcePool class for the high-level Minimal Abbey and its behavioral interface which describes in terms of a timed automaton the expected task generation patterns.

From the behavioral interface we can generate sequences of time-stamped tasks with their deadlines of the form

$$(t_1, \mathrm{task}(d_1)), \ldots, (t_n, \mathrm{task}(d_n))$$

where $t_i$ denotes the time at which a task has been queued with deadline $d_i$. Note that the semantics of timed automata are exactly defined in terms of such sequences ([]).

We next apply the test case to the ResourcePool class in the Maude interpreter extended with an implementation of the scheduler. Such an implementation only requires the definition of a sort for representing time. We then can check whether the ResourcePool class does in fact forward the tasks as specified by the scheduler. Assuming the time-sharing model, the actual schedulings of the tasks corresponding to the test case can be obtained by simulation of the (synchronous) product of scheduler and the timed automaton representing the test case.

## 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 concurrent object based Creol models abstract from scheduling concerns.

We bridge the gap between these two levels of modeling by testing conformance. Behavioral interfaces are central to the analyses. Thread pools are analyzed with respect to the task generation pattern given in the behavioral interfaces, modeling the work-load. The test cases for conformance checking are also derived from behavioral interfaces.

Future work consist first of all of an implementation of the method for testing conformence 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, 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, 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.