Dating Concurrent Objects: Real-Time Modeling and Schedulability Analysis [⋆]

Frank S. de Boer[1,2], Mohammad Mahdi Jaghoori[2,1], and Einar Broch Johnsen[3]

[1] CWI, Amsterdam, The Netherlands
[2] LIACS, Leiden, The Netherlands
[3] University of Oslo, Norway

Abstract. In this paper we introduce a real-time extension of the concurrent object modeling language Creol which is based on duration statements indicating best and worst case execution times and deadlines. We show how to analyze schedulability of an abstraction of real-time concurrent objects in terms of timed automata. Further, we introduce techniques for testing the conformance between these behavioral abstractions and the executable semantics of Real-Time Creol in Real-Time Maude. As a case study we model and analyze the schedulability of thread pools in an industrial communication platform.

1 Introduction

In the object-oriented modeling language Creol [9, 3], 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 primitives, 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.

In this paper we extend Creol with real-time information about the deadlines of messages and the best and worst execution times of the (sequential) control statements. We formalize the semantics of Real-Time Creol with respect to given intra-object scheduling policies in the real-time extension of Maude [5]. This formalization of a Real-Time Creol model provides a refinement of the underlying untimed model in the sense that it only restricts its behaviors.

Schedulability analysis In general analyzing schedulability of a real time system consists of checking whether all tasks are accomplished within their deadlines.

[⋆] This research is partly funded by the EU projects IST-33826 CREDO: Modeling and Analysis of Evolutionary Structures for Distributed Services (`http://credo.cwi.nl`) and FP7-231620 HATS: Highly Adaptable and Trustworthy Software using Formal Methods (`http://www.hats-project.eu`).

We employed automata theory in our previous work [7, 8] to provide a high-level framework for modular schedulability analysis of concurrent objects. In order to analyze the schedulability of an open system of concurrent objects, we need some assumptions about the real-time arrival patterns of the incoming messages; in our framework, this is contained in the timed automata [1] modeling the *behavioral interface* of the open system. A behavioral interface captures the overall real-time input/output behavior of an object while abstracting from its detailed implementation in its methods; a deadline is assigned to each message specifying the time before which the corresponding method has to be completed. Further, we use timed automata to describe an abstraction of the system of objects itself including its message queues and a given scheduling policy (e.g., Earliest Deadline First). The analysis of the schedulability of an open system of concurrent objects can then be reduced to model-checking a timed automaton describing the interactions between the behavioral abstraction of the system and its behavioral interface (representing the environment).

Conformance We test conformance between the Real-Time Creol model of an open system of concurrent objects and its behavioral abstraction in timed automata with respect to a given behavioral interface. Our method is based on generating a timed trace (i.e., a sequence of time-stamped messages) from the automaton constructed from its behavioral abstraction and interface. Using model-checking techniques we next generate for each time specified in the trace additional real-time information about all possible observable messages. This additional information allows us to find *counter-examples* to the conformance. To do so, we use the Real-Time Maude semantics as a language interpreter to execute the real-rime Creol model driven by the given trace. Then we look for counter-examples by incrementally searching the execution space for possible timed observations that are not covered in the extended timed trace.

Case Study 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 [2]. 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.

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. We associate real-time information to the tasks concerning their deadlines and best and worst case execution times.

We perform schedulability analysis on a network of timed automata constructed from the automata abstraction and behavioral interface of the thread pool model in order to verify 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 [12] for this purpose. Further, we test conformance between the Real-Time Creol model of a thread pool and its behavioral abstractions as described above.

Related work We extend Creol with explicit scheduling strategies and a duration statement to specify execution delays. Creol$_{\text{RT}}$ [11] is another real-time extension of Creol with clocks and time-outs. Our work follows a descriptive approach to specifying real-time information suitable for schedulability analysis, whereas the prescriptive nature of time in Creol$_{\text{RT}}$ can affect the functional behavior of an object.

Schedulability analysis in this paper can be seen as the continuation of our previous work [7] on modular analysis of a single-threaded concurrent object with respect to its behavioral interface. In this paper, we extend the schedulability analysis to an open system of concurrent objects in a way similar to [4].

The work of [6, 10] 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 models. Therefore, our main focus is on studying different scheduling policies and design decisions.

We test conformance between a Creol implementation and abstract automata models. Our notion of conformance is similar to **tioco** introduced by Schmaltz and Tretmans [16, 15], but we do not directly work with timed input/output transition systems; an innovation of our work is dealing with conformance between different formalisms, namely Creol semantics in rewrite logic on one hand and timed automata on the other hand. Furthermore, we focus on generating counterexamples during testing along the lines of our previous work [8], which is novel in testing.

Outline The real-time extension of the concurrent object language Creol is explained in Section 2. As explained in Section 3, abstract models of concurrent objects, specified in timed automata, are analyzed to be schedulable. To be able to argue about the schedulability of Real-Time Creol models, we need to establish conformance between our Creol and automata models; this is explained in Section 4. We conclude in section 5.

2 Concurrent Objects in Real-Time Creol

Creol is an abstract behavioral modeling language for distributed active objects, based on asynchronous method calls and processor release points. In Creol, ob-

Syntactic categories.	*Definitions.*
g in Guard	$IF ::= \textbf{interface}\ I\ \{\ [\overline{Sg}]\ \}$
s in Stmt	$CL ::= \textbf{class}\ C\ [(\overline{I\ x})]\ [\textbf{implements}\ \overline{I}]\ \{\ [\overline{I\ x};]\ \overline{M}\}$
x in Var	$Sg ::= I\ m\ ([\overline{I\ x}])$
e in Expr	$M ::= Sg\ \{\ [\overline{I\ x};]\ s\ \}$
o in ObjExpr	$g ::= b \mid x? \mid g \wedge g \mid g \vee g$
b in BoolExpr	$s ::= x := e \mid x := e.\textbf{get} \mid \textbf{skip} \mid \textbf{release} \mid \textbf{await}\ g \mid \textbf{return}\ e$
d in Time	$\mid s; s \mid [o]!m(\overline{e}, d) \mid \textbf{if}\ b\ \textbf{then}\ \{\ s\ \}\ \textbf{else}\ \{\ s\ \} \mid \textbf{duration}(d, d)$
	$e ::= x \mid o \mid b \mid \textbf{new}\ C\,(\overline{e}) \mid [o]!m(\overline{e}, d) \mid \textbf{this} \mid \textbf{deadline}$

Fig. 1. Syntax of the Real-Time Creol kernel. Terms such as $\overline{e}$ and $\overline{x}$ denote lists over the corresponding syntactic categories and square brackets denote optional elements.

jects conceptually have dedicated processors and live in a distributed environment with asynchronous and unordered communication between objects. Communication is between named objects by means of asynchronous method calls; these may be seen as triggers of concurrent activity, resulting in new activities (tasks) in the called object. Objects are dynamically created instances of classes, declared attributes are initialized to some arbitrary type-correct values. An optional *init* method may be used to redefine the attributes during object creation. An object has a set of tasks to be executed, which stem from method activations. Among these, at most one task is *active* and the others are *suspended* on a task queue. The scheduling of tasks is by default non-deterministic, but controlled by *processor release points* in a cooperative way. Creol is strongly typed: for well-typed programs, invoked methods are supported by the called object (when not `null`), such that formal and actual parameters match. In this paper, programs are assumed to be well-typed. This section introduces *Real-Time Creol*, explaining Creol constructs (for further details, see, e.g., [9, 3]) and their relation to real-time scheduling policies.

Figure 1 gives the syntax for a kernel of Real-Time Creol, extending a subset of Creol (omitting, e.g., inheritance). A *program* consists of interface and class definitions and a `main` method to configure the initial state. Let C, I, and m be in the syntactic category of Names. IF defines an interface with name I and method signatures Sg. A class implements a list $\overline{I}$ of interfaces, specifying types for its instances. CL defines a class with name C, interfaces $\overline{I}$, class parameters and state variables $\overline{x}$ (of types $\overline{I}$), and methods $\overline{M}$. (The *attributes* of the class are both its parameters and state variables.) A method signature Sg declares the return type I of a method with name m and formal parameters $\overline{x}$ of types $\overline{I}$. M defines a method with signature Sg and a list of local variable declarations $\overline{x}$ of types $\overline{I}$ and a statement s. Statements may access class attributes, locally defined variables, and the method's formal parameters. Statements for assignment $x := e$, sequential composition $s_1; s_2$, **skip**, **if**, **while**, and **return** e are standard. The statement **release** unconditionally releases the processor by suspending the active task. In contrast, the guard g controls processor release in the statement **await** g, and consists of Boolean conditions which may contain return tests $x?$ (see below). If g evaluates to false, the current task is *suspended* and the execution thread becomes idle. When the execution thread is idle, an enabled task may be selected from the pool of suspended tasks by means of a user-provided scheduling policy.

Expressions e include declared variables x, object identifiers o, Boolean expressions b, and object creation **new** $C(\overline{e})$. As usual, the reserved read-only variable **this** refers to the identifier of the object. Note that remote access to attributes is not allowed. (The full language includes a functional expression language with standard operators for data types such as strings integers lists, sets, maps, and tuples. These are omitted in the core syntax, and explained when used in the examples.)

Time. In Real-Time Creol, the local passage of time is expressed in terms of **duration** statements. We consider a dense time model represented by the sort Time which ranges over non-negative real numbers and is totally ordered by the less-than operator. Furthermore, we denote by ∞ a term of sort Time such that for all $t_1, t_2 \neq \infty$, $t_1 + t_2 < \infty$. The statement **duration**(b, w) expresses the passage of time, given in terms of an interval between the best case b and the worst case w (assuming $b \leq w$). All other statements are assumed to be instantaneous, except the **get** statement which lets time pass while it is blocking (see below).

Communication in Real-Time Creol is based on asynchronous method calls, denoted by expressions $o!m(\overline{e}, d)$, and future variables. (Local calls are written $!m(\overline{e}, d)$.) Thus, after making an asynchronous method call $x := o!m(\overline{e}, d)$, the caller may proceed with its execution without blocking on the method reply. Here x is a future variable, o is an object expression, $\overline{e}$ are (data value or object) expressions, and d is a *deadline* for the method invocation. This deadline specifies the relative time before which the corresponding method should be scheduled and executed. The local variable **deadline** refers to the *remaining permitted execution time* of the current method activation. We assume that message transmission is instantaneous, so the deadline expresses the time until a reply is received; i.e., it corresponds to an *end-to-end* deadline. As usual, if the return value of a call is of no interest, the call may occur as a statement. The future variable x refers to a return value which has yet to be computed. There are two operations on future variables, which control synchronization in Creol. First, the guard **await** $x?$ suspends the active task unless a return to the call associated with x has arrived, allowing other tasks in the object to execute. Second, the return value is retrieved by the expression x.**get**, which blocks all execution in the object until the return value is available. Standard usages of asynchronous method calls include the statement sequence $x := o!m(\overline{e}, d);\ v := x$.**get** which encodes a *blocking call*, abbreviated $v := o.m(\overline{e}, d)$ (often referred to as a synchronous call), and the statement sequence $x := o!m(\overline{e}, d)$; **await** $x?$; $v := x$.**get** which encodes a non-blocking, *preemptible call*, abbreviated **await** $v := o.m(\overline{e}, d)$.

```
class Thread(ResourcePool myPool) implements Thread {
  Void run() { myPool!finish(this) }
  Void start() { skip; duration(5,6); myPool!finish(this) }
}

class ResourcePool(Int size) implements ResourcePool {
  Set[Thread] pool;

  Void init() { Thread thr; pool := {};
    while (size>0) { thr := new Thread(this); size := size-1 }
  }
  Void invoke() {
    Thread thread; await ¬isempty(pool);
    thread := choose(pool); pool := remove(pool,thread);
    thread!start(deadline)
  }
  Void finish (Thread thr) { pool := add(pool,thr) }
}
```

Fig. 2. The thread pool

2.1 Object-Oriented Modeling of Thread-Pools

Figure 2 shows a Creol model of a thread pool. The model defines a `Thread` class and the `ResourcePool` class. The task list is modeled implicitly in terms of the message queue of an instance of the `ResourcePool` class. The variable `size` represents the number of available threads, i.e., instances of the `Thread` class. The variable `pool` is used to hold a reference to those threads that are currently not executing a task. Tasks are modeled in terms of the method start inside the `Thread` class. For our analysis the functional differences between tasks is irrelevant, so the method is specified in terms of its duration only and a subsequent call to the method `finish` of the `ResourcePool` object which adds that thread to its pool of available threads.

Tasks are generated (by the environment) with (asynchronous) calls of the `invoke` method of the `ResourcePool` object. In case there are no available threads, the execution of the `invoke` method suspends by the execution of the **`await`** statement which releases control (so that a call of the `finish` method can be executed). When multiple tasks are pending and a thread becomes available, the scheduling strategy of the `ResourcePool` object determines which task should be executed next when the current task has been completed.

2.2 Real-Time Execution in Real-Time Maude

Real-Time Maude [14] defines real-time rewrite theories (Σ, E, IR, TR), where:

– (Σ, E) is a theory in membership equational logic [13] with a signature Σ and a set E of conditional equations. The system's state space is defined as an algebraic data type in (Σ, E), which is assumed to contain a specification of a sort `Time` capturing the (dense or discrete) time domain.

– IR is a set of labeled conditional *instantaneous rewrite rules* specifying the system's local transitions, written **crl** $[l] : t \longrightarrow t'$ **if** *cond*, where l is a name for the rule. Such a rule specifies a one-step transition (in zero-time) from an instance of a pattern t to the corresponding instance of a pattern t', provided the condition *cond* holds. As usual in rewriting logic [13], the rules are applied modulo the equations in E.
– TR is a set of *timed rewrite rules* (or tick rules) capturing the progression of time, written **crl** $[l] : \{t\} \longrightarrow \{t'\}$ **in time** τ **if** *cond* where τ is a term of sort `Time` which denotes the duration of the rewrite. Observe that `{_}` is the built-in constructor of sort `GlobalSystem`, so tick rules apply to the *entire* state of the system which allows time to advance uniformly.

Initial states must be ground terms of sort `GlobalSystem`, which reduce to terms of the form `{t}` by the equations in E. The form of the tick rules then ensures that time advances uniformly throughout the system. Real-time rewrite theories are executable under reasonable assumptions and Real-Time Maude provides different analysis methods [14]. For example, timed "fair" rewrite simulates one behavior of the system up to a certain duration and is written

```
(tfrew t in time ≤ τ .)
```

for an initial state `t` and a ground term τ of sort `Time`. Furthermore, timed search searches breadth-first for states that are reachable from a given initial state `t` within time τ, match a search pattern, and satisfy a search condition. The command which searches for one state satisfying the search criteria is written

```
(tsearch [1] t ⟶* pattern such that cond in time ≤ τ .)
```

Creol's semantics in Maude. Creol has a semantics defined in Rewriting logic [13] which can be used directly as a language interpreter in Maude [5]. The semantics is explained in detail in [9] and can be used for the analysis of Creol programs. In this section we focus on the extension of Creol's semantics in order to define a semantics for Real-Time Creol in Real-Time Maude.

The state space of Creol's operational semantics is given by terms of the sort `Configuration` which is a set of objects, messages, and futures. The empty configuration is denoted `none` and whitespace denotes the associative and commutative union operator on configurations. Objects are defined as tuples

$\langle o,\ a,\ q\ \rangle$

where o is the identifier of the object, a is a map which defines the values of the attributes of the object, and q is the task queue. Tasks are of sort `Task` and consist of a statement s and the task's local variables l. We denote by $\{l|s\} \circ q$ the result of appending the task $\{l|s\}$ to the queue q. For a given object, the first task in the queue is the *active task* and the first statement of the active task to be executed is called the *active statement*.

Let σ and σ' be maps, x a variable name, and v a value. Then $\sigma(x)$ denotes the lookup for the value of x in σ, $\sigma[x \mapsto v]$ the update of σ such that x maps to v, $\sigma \circ \sigma'$ the composition of σ and σ', and $\mathrm{dom}(\sigma)$ the domain of σ. Given

rl [*skip*]: $\langle o,\ a,\ \{l \mid \texttt{skip};s\} \circ q\rangle \longrightarrow \langle o,\ a,\ \{l \mid s\} \circ q\rangle$.

rl [*assign*]: $\langle o,\ a,\ \{l \mid x{:=}e;s\} \circ q\rangle$
$\longrightarrow$ **if** $x \in \mathrm{dom}(l)$ **then** $\langle o,\ a,\ \{l[x \mapsto [\![e]\!]^{none}_{aol}] \mid s\} \circ q\rangle$
else $\langle o,\ a[x \mapsto [\![e]\!]^{none}_{aol}],\ \{l \mid s\} \circ q\rangle$ **fi** .

rl [*release*]: $\langle o,\ a,\ \{l \mid \texttt{release};s\} \circ q\rangle \longrightarrow \langle o,\ a,\ \texttt{schedule}(\{l \mid s\},q)\rangle$.

crl [*await1*]: $\{\langle o,\ a,\ \{l \mid \texttt{await}\ e;s\} \circ q\rangle\ \ c\}$
$\longrightarrow \{\langle o,\ a,\ \{l \mid s\} \circ q\rangle\ \ c\}$ **if** $[\![e]\!]^{c}_{aol}$.

crl [*await2*]: $\{\langle o,\ a, \{l \mid \texttt{await}\ e;s\} \circ q\rangle\ \ c\}$
$\longrightarrow \{\langle o,\ a,\ \{l \mid \texttt{release};\texttt{await}\ e;s\} \circ q\rangle\ \ c\}$ **if** $\neg[\![e]\!]^{c}_{aol}$.

Fig. 3. The semantics of Creol in Maude.

a mapping, we denote by $[\![e]\!]^{c}_{\sigma}$ the evaluation of an expression e in the state given by σ and the global configuration c (the latter is only used to evaluate the polling of futures; e.g., **await** `x?`).

Rewrite rules execute statements in the active task in the context of a configuration, updating the values of attributes or local variables as needed. For an active task $\{l \mid s\}$, these rules are defined inductively over the statement s. Some (representative) rules are presented in Figure 3. Rule *skip* shows the general set-up, where a **skip** statement is consumed by the rewrite rule. Rule *assign* updates either the local variable or the attribute x with the value of an expression e evaluated in the current state. The suspension of tasks is handled by rule *release*, which places the active task in the task queue. The auxiliary function `schedule` in fact defines the (local) *task scheduling policy* of the object, for example first in first out (FIFO) or earliest deadline first (EDF). Rules *await1* and *await2* handle conditional suspension points.

Real-Time Creol's semantics in Real-Time Maude. The rewrite rules of the Real-Time Creol semantics are given in Figure 4. The first rule ensures that a **duration** statement may terminate only if its best case execution time has been reached. In order to facilitate the conformance testing discussed below, we define a global clock `clock(t)` in the configurations (where `t` is of sort `Time`) to time-stamp observable events. These observables are the invocation and return of method calls. Rule *async-call* emits a message to the callee $[\![e]\!]^{none}_{(aol)}$ with method m, actual parameters $[\![\overline{e}]\!]^{none}_{(aol)}$ including the deadline, a fresh future identifier n, which will be bound to the task's so-called `destiny` variable [3], and, finally, a time stamp t. In the (method) *activation* rule, the function `task` transforms such a message into a task which is placed in the task queue of the callee by means of the scheduling function `schedule`. The function `task` creates a map which assigns the values of the actual parameters to the formal parameters (which includes the `deadline` variable) and which assigns the future identity to the `destiny` variable. The statement of the created task consist of the body of the method. Rule *return* adds the return value from a method call to the

crl [*duration*]: $\langle o,\ a,\ \{l \mid \textbf{duration}(b,w); s\} \circ q\rangle$
$\longrightarrow \langle o,\ a,\ \{l \mid s\} \circ q\rangle$ **if** $b \leq 0$.

crl [*async-call*]: $\langle o,\ a, \{l \mid x{:=}e!m(\bar{e}); s\} \circ q\rangle$ clock(t)
$\longrightarrow \langle o,\ a, \{l[x \mapsto n] \mid s\} \circ q\rangle\ \ m(t, [\![e]\!]^{none}_{aol}, [\![\bar{e}]\!]^{none}_{aol}, n)\ \ n$ **if** fresh(n) .

crl [*activation*]: $\langle o,\ a, \{l \mid s\} \circ q\rangle\ \ m(t, o, \bar{v})$
$\longrightarrow \langle o,\ a, \{l \mid s\} \circ$ schedule(task($m(o,\bar{v})$), q) $\rangle$.

crl [*return*]: $\langle o,\ a, \{l \mid$ return(e); $s\} \circ q\rangle\ \ n$ clock(t)
$\longrightarrow \langle o,\ a,\ \{l \mid s\} \circ q\rangle$ clock(t) $\langle n, [\![e]\!]^{none}_{(aol)}, t\rangle$ **if** $n = l$(destiny) .

crl [*get*]: $\langle o,\ a, \{l \mid x := e.$**get**$; s\} \circ q\rangle\ \ \langle n, v, t\rangle$
$\longrightarrow \langle o,\ a, \{l \mid x := v; s\} \circ q\rangle\ \ \langle n, v, t\rangle$ **if** $[\![e]\!]^{none}_{aol} = n$.

crl [*tick*]: $\{C\} \longrightarrow \{\delta(C)\}$ **in time** t **if** $t <$ mte(C) $\wedge$ canAdvance(C) .

Fig. 4. The semantics of Real-Time Creol in Real-Time Maude.

op canAdvance: Configuration $\to$ Bool .
eq canAdvance(C1 C2) = canAdvance(C1) $\wedge$ canAdvance(C2) .
eq canAdvance($\langle o,\ a,\ \{l \mid \textbf{duration}(b,w); s\} \circ q\rangle$) = $w > 0$.
eq canAdvance($\langle o,\ a, \{l \mid x := e.$**get**$; s\} \circ q\rangle\ \ n$) = true **if** $n = [\![x]\!]^{none}_{(\bar{a}o\bar{l})}$.
eq canAdvance(C) = false [**owise**] .

op mte: Configuration $\to$ Time .
eq mte(C1 C2) = min(mte(C1), mte(C2)) .
eq mte($\langle o,\ a, \{l \mid \textbf{duration}(b,w); s\} \circ q\rangle$) = w .
eq mte(C) = ∞ [**owise**] .

op δ_1: Task Time $\to$ Task .
eq $\delta_1(\{l \mid s\}, t) = \{l[\text{deadline} \mapsto l(\text{deadline}) - t] | s\}$.

op δ_2: TaskQueue Time $\to$ TaskQueue .
eq $\delta_2(\{l|s\} \circ q, t) = \delta_1(\{l|s\}, t) \circ \delta_2(q, t)$.
eq $\delta_2(\epsilon, t) = \epsilon$

op δ_3: Task Time $\to$ Task .
eq $\delta_3(\{l|\textbf{duration}(b,w); s\}, t)$
    $= \{l[\text{deadline} \mapsto l(\text{deadline}) - t] \mid \textbf{duration}(b-t, w-t); s\}$.
eq $\delta_3(\{l|s\}, t) = \{l[\text{deadline} \mapsto l(\text{deadline}) - t] | s\}$ [**owise**] .

op δ: Configuration Time $\to$ Configuration .
eq δ(C1 C2, t) = δ(C1,t) δ(C2,t) .
eq δ(clock(t'),t) = clock($t' + t$) .
eq $\delta(\langle o,\ a,\ \{l|s\} \circ q\rangle, t) = \langle o,\ a,\ \delta_3(\{l|s\}) \circ \delta_2(q)\rangle$.
eq δ(C, t) = C [**owise**] .

Fig. 5. Definition of Auxiliary Functions

future identified by the task's `destiny` variable and time stamps the future at this time. Rule *get* describes how the **`get`** operation obtains the returned value.

The global advance of time is captured by the rule *tick*. This rule applies to global configurations in which all active statements are duration statements which have not reached their worst execution time or blocking get statements. These conditions are captured by the predicate `canAdvance` in Figure 5. When the *tick* rule is applicable, time may advance by any value t below the limit determined by the auxiliary *maximum time elapse* [14] function `mte`, which finds the lowest remaining worst case execution time for any active task in any object in the configuration. Note that the blocking **`get`** operation allows time to pass arbitrarily while waiting for a return.

When time advances, the function δ, besides advancing the global clock, determines the effect of advancing time on the objects in the configuration, using the auxiliary functions δ_i, for $i = 1, 2, 3$, defined in Figure 5, to update the tasks. The function δ_1 decreases the deadline of a task. The function δ_2 applies δ_1 to all queued tasks; δ_2 has no effect on an empty queue ϵ. The function δ_3 additionally decreases the current best and worst case execution times of the active **`duration`** statements.

3 Schedulability Analysis

Schedulability analysis consists of checking whether tasks can be accomplished before their deadlines. For analysis, Real-Time Maude uses tick rules that advance time in discrete time steps, therefore verification of dense time models in Real-Time Maude is incomplete. Timed automata verification tools, e.g., UPPAAL, use symbolic time and thus cover the whole dense time space. In this section, we explain how to use timed automata for abstractly modeling concurrent objects and performing schedulability analysis. In this abstract modeling, infinite Creol programs are mapped to finite state automata.

We present a generalization of the automata-theoretic framework in [7] for schedulability analysis of concurrent objects. The overall real-time input/output behavior of an object is to be specified in an automaton called its *behavioral interface*. A behavioral interface abstracts from the detailed implementation of the object, which is in turn given in terms of its output behavior, given in the automata modeling the methods; and, the input enabled scheduler automaton that includes a queue to buffer the messages waiting to be executed.

In this paper we extend the schedulability analysis to an open system of concurrent objects. We explain this extension in terms of the thread pool example introduced in Section 2.1. Such a model can be synthesized from the sequence diagram in Figure 6 which depicts the life-cycle of a task from its generation until its completion. To allow communication between different automata, we define a channel for each action in this diagram: Channel `invoke` has three parameters; namely, task name, the receiver and the sender. The parameters to channel `start` capture the task to be executed, the thread assigned to it, and the current object's identifier. Channel `finish` is parameterized in the identifiers of

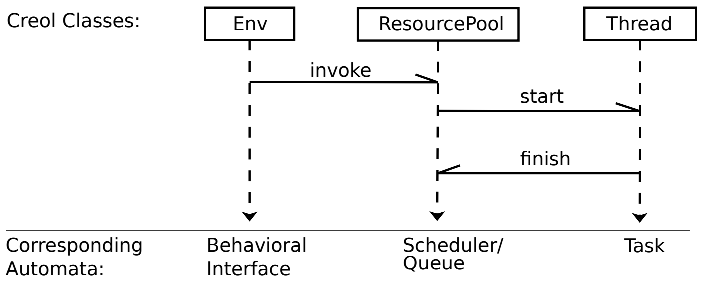


Fig. 6. Sequence diagram of a scenario from generation until completion of a task

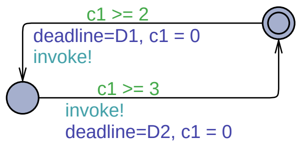


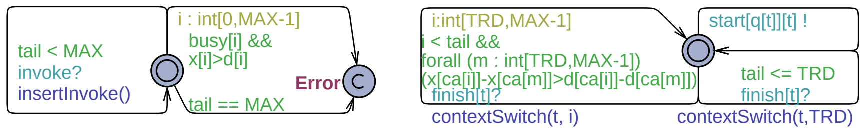
Fig. 7. Modeling a task generation pattern (right)

the executing thread and object. Next we discuss the different automata models corresponding to the three different life-lines in Figure 6.

Behavioral interfaces The behavioral interface captures the overall real-time input/output behavior of an object while abstracting from its detailed implementation. Figure 7 shows a possible behavioral interface for our model of thread pools. This automaton is parameterized in the identifier of the thread pool, written `self`, and an identifier `Env` that represents any entity that may invoke a task on the thread pool. Since we only assume one task type in this example, we define a global constant `task` that will be used to identify this task.

We use a clock `c1` for modeling inter-arrival times and the global variable `deadline` is used for associating deadlines to each task generated. The tasks with different deadlines are interleaved and there is at least 2 and 3 time units of inter-arrival time between two consecutive task instances. This shows an example of non-uniform task generation pattern.

Scheduler and queue The queue and the scheduling strategy are modeled in separate automata; together they represent the ResourcePool class. To model the ResourcePool, every thread is assumed to have a dedicated processing unit, but they share one task queue. We assume a fixed number `TRD` of threads given a priori. We separate the task queue in two parts: an *execution* part, consisting of the slots 0 to `TRD-1`, and a *buffer* part consisting of the rest of the queue. 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. The selection of a task from the buffer part to start its execution is based on a given scheduling strategies, e.g., EDF, FPS, etc.; in our example, we use EDF.



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

Fig. 8. Allowing parallel threads to share a queue

Figure 8(a) shows a queue of size `MAX` which stores the tasks in the order of their arrival; the queue is modeled by the array `q` and `tail` points to the first empty element in the queue. This automaton is parameterized in `s` which holds the identity of this object. This automaton can accept *any* task (whose identifier is between 0 and the constant `MSG`) by *any* caller (whose identifier is between 0 and the constant `OBJ`); this is seen as the Uppaal 'select' statement over `msg` and `caller` on the `invoke` channel. This transition is enabled if the queue is not yet full (`tail < MAX`). 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 queue goes to `Error` state if a task misses its deadline (`x[i] > d[i]`) or the queue is full (`tail == MAX`).

Figure 8(b) shows how a scheduling strategy can be implemented. This automaton should be replicated for every thread, thus parameterized in thread identity `t` as well as the object identity `s`. There will be one instance of this automaton for each slot `q[t]` in the execution part of the queue. 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 (i.e. a synchronization on `finish[t][s]`), it selects the next task from the buffer part of the queue for execution by putting it in `q[t]`; this task is then immediately started (`start[q[t]][t][s]`).

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 [7], a system is schedulable only if it 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 $\lceil D_{max}/B_{min} \rceil$.

Tasks A simple task model is given in Figure 9. In this model, the task has a computation time of between 5 to 6 time units. This corresponds to the model of the task given in the Creol code, which is defined in the start method of the Thread class and contains a skip statement followed by a duration. In general, a task model may be an arbitrarily complex timed automaton.

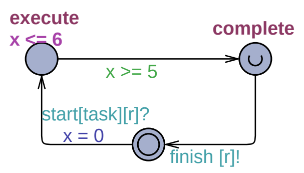


Fig. 9. Modeling a task

For schedulability analysis, one can experiment with different parameters. For example, one can choose different scheduling policies, like FCFS, EDF, etc. Since we assume that threads run in parallel, with more threads, we can handle more task instances (i.e., smaller inter-arrival times). Furthermore, if deadlines are too tight, schedulability cannot be guaranteed. Schedulability analysis amounts to checking whether the `Error` location in the queue automaton is reachable. Analysis shows that in the chosen settings, i.e., the selected inter-arrival times for the tasks and an EDF scheduler, this model cannot be schedulable with 2 parallel threads, no matter how big the deadlines are. Intuitively, every 5 time units, two instances of the task may be inserted in the queue, and each task may take up to 6 time units to compute. With three parallel threads, these tasks can be scheduled correctly even with the deadline value of 6 time units for each task instance.

4 Conformance Testing

Our overall methodology for the schedulability analysis of a Real-Time Creol model consists of the following: We model the real-time pattern of incoming messages in terms of a timed automaton (the behavioral interface of the Creol model). Next we develop on the basis of sequence diagrams, which describe the observable behavior of the Creol model, automata abstractions of its overall real-time behavior. We analyze the schedulability of the product of this abstraction and the given behavioral interface (in for example Uppaal). Further, we define conformance between the Real-Time Creol model and its timed automaton abstraction with respect to the given behavioral interface in terms of inclusion of the timed traces of observable actions.

More specifically, let $\mathbf{C}$ denote a Creol model, i.e., a set of Creol classes, $\mathbf{B}$ a timed automaton specifying its behavioral interface and $\mathbf{A}$ a timed automata abstraction of the overall behavior of $\mathbf{C}$. We denote by $O(\mathbf{A} \parallel \mathbf{B})$ the set of timed traces of observable actions of the product of the timed automata $\mathbf{A}$ and $\mathbf{B}$. The set of timed traces of the timed automaton $\mathbf{B}$ we denote by $T(\mathbf{B})$. Further, given any timed trace $\theta \in T(\mathbf{B})$, the Creol class $\mathit{Tester}(\theta)$ denotes a Creol class which implements θ (see, for example, the class *Tester* in Figure 11). This class simply injects the messages at the times specified by θ. We denote by $O(\mathbf{C}, \mathit{Tester}(\theta))$ the set of timed traces of observable actions generated by the Real-Time Maude

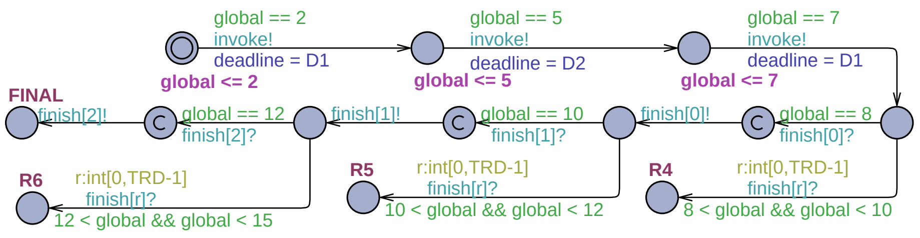


Fig. 10. Generating ready sets.

semantics of the Creol model $\mathbf{C}$ driven by θ. We now can define the conformance relation $\mathbf{C} \leq_{\mathbf{B}} \mathbf{A}$ by

$$O(\mathbf{C}, \mathit{Tester}(\theta)) \subseteq O(\mathbf{A} \parallel \mathbf{B}),$$

for every timed trace of observable actions $\theta \in T(\mathbf{B})$.

In this section we illustrate a method for testing conformance by searching for *counter-examples* to the conformance in terms of our running example. Note that a counter-example to the above conformance consist of a timed trace $\theta \in T(\mathbf{B})$ such that $O(\mathbf{C}, \mathit{Tester}(\theta)) \setminus O(\mathbf{A} \parallel \mathbf{B}) \neq \emptyset$.

4.1 Generating a Test Case

We first generate a timed trace $\theta = (t_1, a_1), \ldots, (t_n, a_n)$ by simulating the abstract timed automaton model $\mathbf{A}$ together with the behavioral interface $\mathbf{B}$. To this end, we add a dummy automaton with a fresh clock `global` and an integer `time` which is incremented every time unit. This way we can find the absolute time interval in which every action in the trace has happened. In order to be able to search for a counter-example to conformance, we generate ready sets of observable actions generated by the behavioral abstraction of the Creol model. For each time interval between t_{i-1} and t_i in this trace and for every observable action a, we are interested in the following timed reachability property:

"`E<>` t_{i-1} `<= global && global <` t_i `&& a_f`",

where `a_f` denotes whether the observable action a has occurred in this interval. Instead of checking this property directly for every action, we encode it into one automaton as explained below (see Figure 10). This way, we avoid the need to add flags like `a_f` for every observable action and to go deep in the model to set it true when the corresponding action happens.

The algorithm to construct the automaton in Figure 10 for generating ready sets is as follows. Given a trace $\theta = (t_1, a_1), \ldots, (t_n, a_n)$, we first create a linear timed automaton T_θ with the locations $L = \{l_i \mid 1 \leq i \leq n+1\}$. By going from l_i to l_{i+1}, this automaton should ensure that action a_i happens at time $x == t_i$. This is done differently for inputs and outputs. Since the abstract UPPAAL model $\mathbf{A}$ (i.e., excluding the behavioral interface) is input-enabled, the input actions only need to inject the task at the required time; namely with a transition from l_i to l_{i+1} with an `invoke` action. This transition should provide the required deadline. The output action `finish` is, however, produced by a task and consumed by the scheduler. To intercept this action, this automaton first mimics the scheduler by accepting the action, i.e., `finish?`, and then it mimics the task by issuing `finish!`.

We add to T_θ a location R_{ij} for each time interval between t_{i-1} and t_i and for each observable output action $o_j \in O(\mathbf{A} \parallel \mathbf{B})$, with one transition from l_i to R_{ij} with a guard $\texttt{global} \leq t_i$ accepting the output action o_j; if a_i is the same as o_j, this transition is guarded by $\texttt{global} < t_i$. In our example, there is only one observable output action namely `finish`, but since a task can be taken by different threads, the `finish` action can be issued by different threads; therefore, the transitions for receiving this action should allow any thread identity `r` between 0 and `TRD-1`.

Finally, the reachability of the location R_{ij} implies that o_j must be included in the ready set R_i. We observe that in our example, only `R3` and `R4` are reachable; this is due to the possibility of finishing the first task instance in the interval $[7, 8]$. The consequent task instances can finish in the intervals $[10, 11]$ and $[12, 13]$, therefore, their completion does not contribute to an action in the ready sets. The test case including the ready sets and deadlines is:
(2, invoke(D1), {}) (5, invoke(D2), {}) (7, invoke(D1), {finish}) (8, finish, {finish}) (10, finish, {}) (12, finish, {})

4.2 Executing a Test Case in RT-Maude

Executing a test case amounts to injecting the inputs at the right times and looking for the right outputs at the right times. The system is input-enabled, so it accepts all the inputs. If the system under test cannot produce the expected output at the right time, the test fails. If along the test execution, the system under test can do an observable action that is neither the expected output nor in the ready-set, it is a counter-example to conformance. If the system can produce all expected outputs and no counter-example is found, the test passes in the sense that we are more confident that refinement holds and that the Creol model is schedulable. Notice that a counterexample to refinement does not necessarily imply non-schedulability in itself, but it shows an execution path that is likely to miss a deadline. We demonstrate this with the test-case from the previous subsection, repeated below:
(2, invoke(D1), {}) (5, invoke(D2), {}) (7, invoke(D1), {finish}) (8, finish, {finish}) (10, finish, {}) (12, finish, {})

We encode the input behavior given in the test-case as a complementary class that calls the methods of the model under test at the required times. For our running example, the code for the trace from previous subsection is given in Figure 11 (assuming $D1 = D2 = 6$).

By generating one instance of the ResourcePool class (with size 3 which gives us a schedulable Uppaal model, cf. Section 3) and one instance of the Tester

```
class Tester (mut:ResourcePool){
  Void run(){
    duration(2,2);
    mut!invoke(6);
    duration(3,3); // 5-2 = 3
    mut!invoke(6);
    duration(2,2); // 7-5 = 2
    mut!invoke(6);
  }
}
```

Fig. 11. Tester Class

tsearch [1] {init} $\longrightarrow^*$ {Conf1 finish(T,M,E,N)} **in time** $\leq$2
If this search is successful, then we have a counter-example; otherwise, we continue with the following search:
tsearch [1] {init} $\longrightarrow^*$ {Conf1 invoke(2,M,E,N)} **in time** $\leq$2
If this search is not successful, then the test fails; otherwise, if Maude answers C1 $\rightarrow$ Conf1 then we continue with the following search:

tsearch [1] {C1 invoke(2,M,E,N)} $\longrightarrow^*$ {Conf2 finish(T,M,E,N)} **in time** $\leq$3
If this search is successful, then we have a counter-example; otherwise, we continue with the following search:
tsearch [1] {C1 invoke(2,M,E,N)} $\longrightarrow^*$ {Conf2 invoke(5,M,E,N)} **in time** $\leq$3
If this search is not successful, then the test fails; otherwise, if Maude answers C2 $\rightarrow$ Conf2 then we continue with the following search:

tsearch [1] {C2 invoke(5,M,E,N)} $\longrightarrow^*$ {Conf3 invoke(7,M,E,N)} **in time** $\leq$2
If this search is not successful, then the test fails; otherwise, if Maude answers C3 $\rightarrow$ Conf3 then we continue with the following search:

tsearch [1] {C3 invoke(7,M,E,N)} $\longrightarrow^*$ {Conf4 finish(8,M,E,N)} **in time** $\leq$1
If this search is not successful, then the test fails; otherwise, if Maude answers C4 $\rightarrow$ Conf4 then we continue with the following search:

tsearch [1] {C4 finish (8,M,E,N)} $\longrightarrow^*$ {Conf5 finish(T,M,E,N)} **in time** <2
If this search is successful, then we have a counter-example; otherwise, we continue with the following search:
tsearch [1] {C4 finish (8,M,E,N)} $\longrightarrow^*$ {Conf5 finish(10,M,E,N)} **in time** $\leq$2
If this search is not successful, then the test fails; otherwise, if Maude answers C5 $\rightarrow$ Conf5 then we continue with the following search:

tsearch [1] {C5 finish (10,M,E,N)} $\longrightarrow^*$ {Conf6 finish(T,M,E,N)} **in time** <2
If this search is successful, then we have a counter-example; otherwise, we continue with the following search:
tsearch [1] {C5 finish (10,M,E,N)} $\longrightarrow^*$ {Conf6 finish(12,M,E,N)} **in time** $\leq$2

Fig. 12. Executing the test-case for the thread-pools

class, we can check the output behavior of the Creol model against the test-case with consecutive search commands in Real-Time Maude as shown in Figure 12. In our case, the only observable output action is `finish`. To find a counter-example along this trace, we need to check whether a `finish` action can happen when it is not expected in the ready set, i.e., before time 2, between 2 and 5, between 8 and 10, or between 10 and 12. For each search command, we need to specify as time bound the duration since its start configuration, e.g., to search from C2 which is at time 5, we only need to search for another 2 time units to reach time 7.

It is possible to write a meta-level Maude script to automate the consecutive execution of these search commands, such that each search starts from the resulting configuration of the previous one. The technical details of how this can be done is beyond the scope of this paper.

5 Conclusion

We bridge the gap between automata theory and object orientation. We exploit the expressive power of Real-Time Maude to define the semantics of Real-Time Creol. We complement it with the analytical power of timed automata analysis tools like UPPAAL. Based on this, we explained a methodology for schedulability analysis of open concurrent systems and applied it to the design and analysis of thread pools in an industrial communication platform. This methodology provides a separation of concerns between high-level modeling of architectural features of concurrent systems (in Creol) and their analysis for schedulability (using timed automata).

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 abstractions and the Creol models, thus bridging the gap between the two levels of abstraction. We described a testing technique that is able to find counter-examples to conformance.

Future work consists, 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, so that one can generate code for application-specific schedulers from Creol models.

References

1. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science 126(2), 183–235 (1994)
2. The ASK community systems. http://www.ask-cs.com/
3. de Boer, F.S., Clarke, D., Johnsen, E.B.: A complete guide to the future. In: de Nicola, R. (ed.) Proc. 16th European Symposium on Programming (ESOP'07). LNCS, vol. 4421, pp. 316–330. Springer-Verlag (Mar 2007)

4. de Boer, F.S., Grabe, I., Jaghoori, M.M., Stam, A., Yi, W.: Modeling and analysis of thread-pools in an industrial communication platform. In: Proc. 11th International Conference on Formal Engineering Methods (ICFEM'09). LNCS, vol. 5885, pp. 367–386. Springer (2009)
5. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Quesada, J.F.: Maude: specification and programming in rewriting logic. Theoretical Computer Science 285(2), 187–243 (2002)
6. Closse, E., Poize, M., Pulou, J., Sifakis, J., Venter, P., Weil, D., Yovine, S.: TAXYS: A tool for the development and verification of real-time embedded systems. In: Berry, G., Comon, H., Finkel, A. (eds.) Proc. Computer Aided Verification (CAV01). LNCS, vol. 2102, pp. 391–395. Springer (2001)
7. Jaghoori, M.M., de Boer, F.S., Chothia, T., Sirjani, M.: Schedulability of asynchronous real-time concurrent objects. J. Logic and Alg. Prog. 78(5), 402 – 416 (2009)
8. Jaghoori, M.M., Longuet, D., de Boer, F.S., Chothia, T.: Schedulability and compatibility of real time asynchronous objects. In: Proc. Real Time Systems Symposium. pp. 70–79. IEEE CS (2008)
9. Johnsen, E.B., Owe, O.: An asynchronous communication model for distributed concurrent objects. Software and Systems Modeling 6(1), 35–58 (2007)
10. Kloukinas, C., Yovine, S.: Synthesis of safe, QoS extendible, application specific schedulers for heterogeneous real-time systems. In: Proc. 15th Euromicro Conference on Real-Time Systems (ECRTS 2003). pp. 287–294. IEEE Computer Society (2003)
11. Kyas, M., Johnsen, E.B.: A real-time extension of creol for modelling biomedical sensors. In: Proc. FMCO'08. LNCS, vol. 5751, pp. 42–60. Springer (2009)
12. Larsen, K.G., Pettersson, P., Yi, W.: UPPAAL in a nutshell. STTT 1(1-2), 134–152 (1997)
13. Meseguer, J.: Conditioned rewriting logic as a united model of concurrency. Theoretical Computer Science 96(1), 73–155 (1992)
14. Ölveczky, P.C., Meseguer, J.: Semantics and pragmatics of Real-Time Maude. Higher-Order and Symbolic Computation 20(1–2), 161–196 (June 2007)
15. Schmaltz, J., Tretmans, J.: On conformance testing for timed systems. In: Cassez, F., Jard, C. (eds.) FORMATS. LNCS, vol. 5215, pp. 250–264. Springer (2008)
16. Tretmans, J.: Model based testing with labelled transition systems. In: Hierons, R.M., Bowen, J.P., Harman, M. (eds.) Formal Methods and Testing. LNCS, vol. 4949, pp. 1–38. Springer (2008)