Models of Rate Restricted Communication for Concurrent Objects[⋆]

Rudolf Schlatte, Einar Broch Johnsen,
Fatemeh Kazemeyni, and S. Lizeth Tapia Tarifa

Department of Informatics, University of Oslo, Norway
`{rudi,einarj,fatemehk,sltarifa}@ifi.uio.no`

Abstract

Many software systems today are designed for deployment on a range of architectures. However, in formal models it is typically assumed that the architecture is known and fixed; for example, that the software is sequential or concurrent, that the communication environment is synchronous or asynchronous but ordered, etc. In order to specify and analyze models which range over different deployment scenarios, it is interesting to lift aspects of low-level deployment variability to the abstraction level of the modeling language. In this paper, we propose a technique for introducing explicit resource constraints on concurrent objects in a timed extension of Creol, a formally defined high-level object-oriented modeling language. The technique is demonstrated by examples concerning rate restrictions on communication between objects. These restrictions are compositional and non-invasive: no change to the functional parts of the model is required, and restrictions can be selectively applied to parts of the model. In fact, the rate restrictions are captured by parameters in the model, which allows timed simulations to be performed with varying rate restrictions. We demonstrate the usefulness of explicit rate restrictions on communication in the model by a case study of wireless sensor networks. In this domain, rate restrictions may be understood as an abstraction over the collision patterns of broadcast data packets. Simulation results with different rate restrictions show how the timed throughput of data to the sink node in the network varies depending on the available rates.

Keywords: bandwidth modeling, simulation, object-oriented models, deployment scenarios

1 Introduction

Software systems today are often designed for a range of different deployment scenarios, which may even evolve over time. Examples of such systems

[⋆] Partly funded by the EU project FP7-231620 HATS: Highly Adaptable and Trustworthy Software using Formal Models (http://www.hats-project.eu).

range from operating systems which may be deployed on sequential, multi-core, or distributed architectures, to sensor-based monitoring systems which may be deployed using various means of inter-sensor communication, such as wired, radio, or even acoustic communication channels. Depending on the deployment scenario, a distributed system may vary in the available processing power or memory provided to its nodes, as well as in the available bandwidth of the communication channels between nodes. In modern architectures such as cloud solutions, virtual infrastructures allow aspects of the deployment scenario to be configured to the needs of the software.

In order to model and reason about such highly configurable software systems, one cannot assume a uniform underlying architecture with given properties. Rather, it is desirable to capture this deployment variability at the abstraction level of the modeling language and subsequently to reason about the model's behavior in the context of different deployment scenarios. In this paper, we develop a general model of resource restrictions between observable time intervals in the executions of the object-oriented modeling language Creol. The restrictions are expressed inside the language itself, and are used to impose rate restrictions on the communication environment of concurrent objects in Creol. The communication environment of a Creol model is parametric in its rate restriction, which allows the rate to be set without altering the functional part of the model.

Creol is a high-level executable modeling language [14, 21] based on asynchronously communicating concurrent objects. The language abstracts from specific scheduling strategies inside the concurrent objects, and from particular properties of the communication environment. Creol has a formal semantics given in rewriting logic [25], which can be used directly as a language interpreter in the rewrite system Maude [13]. In order to observe variations in behavior depending on rate restrictions between observable time intervals in the execution, we work with a timed extension to this semantics [6], which allows the timed behavior of Creol models to be simulated using Maude. This enables us to see the temporal effect of rate restrictions in the model, and compare the timed behavior of the model varying in rates.

We demonstrate by examples how rate restrictions in Creol models can be used to capture properties of radio-based message broadcast as well as of point-to-point communication channels. *Channel-based* communication concerns itself with communication between two objects, modeling e.g. a low-bandwidth connection between two hardware systems. *Arrival-based* restrictions describe a single component with its limits of accepting communication from anyone, capturing interference and resending in the context of radio communication. We further show how we can simulate the system behavior, ranging over rate restrictions, using Maude as an interpreter for Creol models.

The paper is structured as follows: Section 2 introduces Timed Creol, a

$$\begin{array}{lrcl}
\textit{Syntactic categories.} & \textit{Definitions.} & & \\
C, I, m \text{ in } \mathsf{Names} & \mathit{IF} & ::= & \mathbf{interface}\ I\,[\mathbf{extends}\,\overline{I}]\,\{\,[\overline{Sg}]\,\} \\
g \text{ in } \mathsf{Guard} & \mathit{CL} & ::= & \mathbf{class}\,C\,[(\overline{I\ x})]\,[\mathbf{implements}\,\overline{I}]\,\{\,[\overline{I\ x};]\ \overline{M}\} \\
s \text{ in } \mathsf{Stmt} & \mathit{Sg} & ::= & I\ m\ ([\overline{I\ x}]) \\
x \text{ in } \mathsf{Var} & M & ::= & \mathit{Sg} == [\overline{I\ x};]\ \{\,s\,\} \\
e \text{ in } \mathsf{Expr} & g & ::= & b \mid x? \mid g \wedge g \\
b \text{ in } \mathsf{BoolExpr} & s & ::= & s; s \mid x := \mathit{rhs} \mid \mathbf{suspend} \mid \mathbf{await}\ g \mid \mathbf{return}\ e \\
 & & \mid & \mathbf{if}\ b\ \mathbf{then}\,\{\,s\,\}\,[\mathbf{else}\,\{\,s\,\}] \mid \mathbf{while}\ b\,\{\,s\,\} \mid \mathbf{skip} \\
 & e & ::= & x \mid b \mid \mathbf{this} \mid \mathbf{now} \mid \mathbf{null} \\
 & \mathit{rhs} & ::= & e \mid \mathbf{new}\ C(\overline{e}) \mid e!m(\overline{e}) \mid e.m(\overline{e}) \mid x.\mathbf{get}
\end{array}$$

Figure 1. The syntax of core Timed Creol. Terms such as $\overline{e}$ and $\overline{x}$ denote lists over the corresponding syntactic categories, square brackets [] denote optional elements. Expressions e and guards g are side-effect free; Boolean expressions b include comparison by means of equality, greater- and less-than operators. Expressions on other datatypes (strings, numbers) are written in the usual way and not contained in this figure.

timed extension of the Creol modeling language and illustrates the language by an example of client server communication. Section 3 proposes a modeling pattern for resource-constrained behavior and introduces a rate restricted point-to-point communication channel for the client server example. Section 4 develops a Creol model of a bandwidth restricted wireless sensor network. Section 5 discusses related work and Section 6 concludes the paper.

2 Concurrent Objects in Creol

Creol is an abstract behavioral modeling language for distributed active objects, based on asynchronous method calls and processor release points. In Creol, objects 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 (so-called *processes*) in the called object. This section briefly introduces Creol (for further details see, e.g., [14, 21]).

Creol objects are dynamically created instances of classes, their declared attributes are initialized to some arbitrary type-correct values. An optional *init* method may be used to redefine the attributes. Active behavior, triggered by an optional *run* method, is interleaved with passive behavior, triggered by method calls. Thus, an object has a set of processes to be executed, which stem from method activations. Among these, at most one process is *active* and the others are *suspended* on a process queue. Process scheduling 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. This paper assumes that programs are well-typed.

Figure 1 gives the syntax for a core subset of Timed Creol (omitting, e.g., class inheritance). A *program* consists of interface and class definitions and a `main` method to configure the initial state. 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 x (of type I), and methods M. (The *attributes* of the class comprise 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, 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. Assignment $x := rhs$, sequential composition $s_1; s_2$, and **if**, **skip**, **while**, and **return** constructs are standard. The statement **suspend** unconditionally releases the processor by suspending the active process. In contrast, the guard g controls processor release in the statement **await** g, and consists of Boolean conditions b and return tests $x?$ (see below). If g evaluates to false, the current process is *suspended* and the execution thread becomes idle. In that case, any enabled process from the pool of suspended processes may be scheduled. Explicit signaling is therefore redundant.

Expressions rhs include declared variables x, object identifiers o, Boolean expressions b, and object creation **new** $C(\overline{e})$ and **null**. The specially reserved read-only variable **this** refers to the identifier of the object and **now** refers to the current clock value (explained below). Note that pure expressions are denoted by e and 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.)

Communication in Creol is based on asynchronous method calls, denoted $o!m(\overline{e})$, and future variables. (Local calls are written **this**$!m(\overline{e})$.) After making an asynchronous call $x := o!m(\overline{e})$, the caller may proceed with its execution without blocking on the call. Here x is a future variable, o is an object expression, and $\overline{e}$ are (data value or object) expressions. A future variable x refers to a return value which has yet to be computed. There are two operations on future variables, controlling external synchronization in Creol. First, the guard **await** $x?$ suspends the active process unless a return to the call associated with x has arrived (allowing other processes in the object to be scheduled). Second, the return value is retrieved by the expression x.**get**, which blocks all execution in the object until the return value is available. The statement sequence $x := o!m(\overline{e});\ v := x$.**get** encodes a *blocking call*, abbreviated $v := o.m(\overline{e})$ (often referred to as a synchronous call), whereas the statement sequence $x := o!m(\overline{e})$; **await** $x?$; $v := x$.**get** encodes a non-blocking, *preemptable call*. Synchronous self-calls **this**$.m(\overline{e})$ are handled specially in the semantics to avoid the trivial deadlock case.

Time. In this paper we work with an extended version of the language Creol which includes an implicit time model [6], comparable to a system clock which updates every n milliseconds. In this extension, a datatype `Time` is included in the language. A value of type `Time` can be obtained by evaluating the expression **`now`**, which returns the current time, i.e., the value of the global clock in the current state. Time values form a total order, with the usual less-than operator. Hence, two time values can be compared with each other, resulting in a Boolean value suitable for guards in **`await`** statements. While all other time values are constant, the result of comparing the expression **`now`** with another time value will change with the passage of time. From an object's local perspective in this model of timed behavior, the passage of time is indirectly observable via **`await`** statements, and time is advanced when no other activity may occur. Note that a global clock is not mandatory in the time model, but is employed in this work to make bandwidth measurements across objects possible. (I.e., in our case the clock models "real" time, not a physical clock that is subject to drift.) The semantics of this model of time, combined with Creol's blocking and non-blocking synchronization semantics, are powerful enough to express both activity and progress of time.

2.1 Example: Channel-Based Communication

This subsection illustrates how client-server communication may be modeled in Creol. This model will be extended to capture rate restricted communication in Sec. 3, without changing the functionality of the model. The interfaces of the client and server are given in Figure 2 (top). The `getData` method of the `Server` interface returns some data, the method `authenticate` of the `Client` interface represents an abstraction of an authentication handshake protocol (in the implementation, this would be based e.g. on a shared key, but in this model we are only concerned with the communication patterns).

The implementations of the Client and Server classes are given in Figure 2 (bottom). We see that objects of class `IClient` are active: upon creation they will request three packets from the server, assembling them in the client. Objects of class `IServer` are passive. They respond to `getData` calls by challenging the caller to authenticate itself, and then return a data packet. Running a simulation with one client and one server object will result in the client object containing the string `"0␣1␣2"`. Figure 5 (left) shows an initial state of the model, which can be executed and tested independently of any communication constraints.

```
interface Client {
  Bool authenticate();
}
```

```
interface Server {
  String getData(Client caller);
}
```

```
class IClient(Server server)
implements Client {
  String content = "";

  Unit run() {
    Int i = 0;
    while (i < 3) {
      Fut<String> packet;
      packet = server!getData(this);
      await packet?;
      content = content + packet.get;
      i = i + 1;
    }
  }

  Bool authenticate() { return True; }
}
```

```
class IServer()
implements Server {
  Int packet = 0;

  String getData(Client caller) {
    String result = "";
    Bool auth = caller.authenticate();
    if (auth) {
      result = intToString(packet)+"␣";
      packet = packet + 1;
    }
    return result;
  }
}
```

Figure 2. Client and Server interfaces and implementations

```
interface  MyInterface {
  Unit resourceConsumingMethod();
}

class MyClass(Int resourceLimit)
implements MyInterface {
  Int resourceUsed = 0;
  ... // Other fields

  Unit run() {
    while (True) {
      Time t = now;
      await now > t;
      resourceUsed = 0;
    }
  }

  Unit resourceConsumingMethod() {
    await resourceUsed < resourceLimit;
    resourceUsed = resourceUsed + 1;
    ... // Implement behavior here
  }
}
```

Figure 3. A modeling pattern to capture resource restriction

3 Modeling Resource-Constrained Behavior

This section presents a technique to capture time-sensitive resource-constrained behavior. The technique consists of imposing explicit resource constraints on concurrent objects in Timed Creol. These resource constraints are expressed by a modeling pattern which extends a given model with a class that is parametric in the available resources per time interval. This class contains an active behavior which resets the consumed resources when time has advanced, captured by the `run` method. In the examples of this paper, we apply this technique in order to impose rate restrictions on communication between objects (see Sec. 2.1 and Sec. 4).

Figure 3 depicts the class introduced by the proposed modeling pattern; the parameter `resourceLimit` represents the number of available resources per time interval for the restricted resource and the variable `resourceUsed` represents the number of resources consumed within the current time interval. The latter variable will never exceed the value of the `resourceLimit` parameter and it is reset to its initial value by the `run` method when time advances (see Fig. 3, Line 14).

Each behavior which needs to be constrained with respect to the resource consumption is modeled by a method which checks the availability of the constrained resource by means of an await-statement (Fig. 3, Line 19), and increments the number of consumed resources within the time interval appropriately when it proceeds (Fig. 3, Line 20). Following this technique, it is straightforward to extend a given model with resource constraints.

We now show how the example of Sec. 2.1 may be extended using the proposed technique in order to add a rate restricted communication channel behavior. The basic idea is to model the channel using the proposed modeling pattern and to leave the original client and server objects unchanged and unaware of the rate restriction imposed on their communication. Figure 4 gives the *channel* class modeled using the proposed pattern, where rate limits are imposed on all communication between the client and the server. The `Channel` interface extends both `Client` and `Server` interfaces, since channel objects will act in both roles.

Objects of class `IChannel` are initialized with a parameter `rate` denoting the "resource limit" per time interval that the channel is prepared to handle; the variable `nMessages` denotes the "consumed resources" within a time interval. Here, the `run` method resets the channel capacity when time advances (see Fig. 4, Line 23). The proxy methods implementing the interfaces, starting at Line 24, pass on the method call if the channel has enough capacity left in the current time interval. Otherwise, the call must wait for time to advance. Note that a proxy method is introduced for every method that is restricted by the channel (as explained above). It may sometimes be desirable to fine-tune the bandwidth consumption for the different methods; e.g., a higher bandwidth may be needed in order to transmit data than for completing an authentication handshake. This can be done easily by using the desired values in the proxy methods.

Figure 5 (right) shows how to initialize the constrained model. Note that the only difference with respect to the unconstrained model (left) is in the object initialization phase, where the client is connected to the channel instead of directly to the server.

4 Example: Wireless Sensor Networks

This section presents an extended case study to illustrate the technique introduced in Section 3. We illustrate the effects of arrival-based communication restriction on model behavior by a model of a wireless sensor network (WSN).

```
interface Channel
extends Client, Server {
  Unit setClient(Client client);
  Unit setServer(Server server);
}

class IChannel(Int rate)
implements Channel {
  Client client = null;
  Server server = null;
  Int nMessages = 0;
  Unit setClient(Client client) {
    this.client = client;
  }
  Unit setServer(Server server) {
    this.server = server;
  }

  Unit run() {
    while (True) {
      Time t = now;
      await now > t;
      nMessages = 0; } }
  Bool authenticate() {
    await client != null;
    await nMessages < rate;
    Fut<Bool> fauth
      = client!authenticate();
    nMessages = nMessages + 1;
    await fauth?;
    return fauth.get;
  }
  String getData(Client caller) {
    await server != null;
    await nMessages < rate;
    Fut<String> result
      = server!getData(this);
    nMessages = nMessages + 1;
    await result?;
    return result.get;
  }
}
```

Figure 4. Channel interface and implementation

```
{
  // Unconstrained model
  Server server;
  Client client;
  server = new IServer();
  client = new IClient(server);
}
```

```
{
  // Constrained model
  Server server;
  Client client;
  Channel channel;
  channel = new IChannel(1);
  server = new IServer();
  client = new IClient(channel);
  channel.setClient(client);
  channel.setServer(server);
}
```

Figure 5. Initialization of the constrained (left) and unconstrained (right) models

A typical WSN consists of a number of sensor nodes, equipped with wireless transmitters, and a sink which collects data. The sensors record some sort of data and send it towards the sink. Since wireless sensors can be very small, or deployed in inaccessible terrain, not every sensor will be able to reach the sink directly. Hence, sensors have the additional duty of routing messages from other nodes towards the sink. Wireless sensor networks use scheduling algorithms for channel access and bandwidth management, and routing algorithms for power- and bandwidth-efficient transmitting of messages.

For channel access, there are two different choices when operating on a single shared channel (e.g. multiple senders operating on the same frequency): the *Time Division Multiple Access* (TDMA) model and the *Carrier Sense Multiple Access* (CSMA) model with *Collision Avoidance* (CA). The CSMA/CA scheme is somewhat simpler for ad-hoc networks, but cannot provide bounded channel access delay and guaranteed fairness, which is unacceptable for time-critical applications. The TDMA scheme, on the other hand, has

```
type SensorId = Int;
// Data packets. Format: (id of originator, sequence no)
data Packet = Packet(SensorId, Int);

interface Node {
  Unit receive(Packet packet);
  Unit setNetwork(Network network)
}

interface Network {
  // Set the network topology.
  Unit setTopology(Map<Node, List<Node>> topology);
  // Broadcast 'packet' from 'source'
  Unit broadcast(Node source, Packet packet)
}
```

Figure 6. The interfaces of the sensor network model

(after an initial setup / negotiation phase) zero overhead and zero collision during data transmission, and can increase the efficiency of the network [12]. Using TDMA, sensor nodes share the same frequency channel and transmit in succession, each using its own time slot.

Routing of messages towards the sink may be done in many different ways to optimize for time consumption, energy consumption, number of messages sent, etc. The model in this chapter uses a very simple Flooding algorithm; a more involved WSN model in Creol implementing the AODV routing algorithm can be found in [24].

4.1 Structure of the Model

In our model, the objects representing nodes are not directly connected to each other. Instead, each node object has a reference to a *Network* object which models the behavior of the transmission medium between nodes, following the idea of TDMA. This structure makes it possible to model collisions, message loss, selective retransmission, and the node topology (which nodes can be reached from each node) without local knowledge of the topology inside the node objects, by modifying the behavior of the network object. This object also implements the resource restriction pattern described in Section 3 in order to model limited bandwidth. Figure 6 shows the interfaces of both the nodes and the network. The `broadcast` method of the network gets called by nodes when they wish to broadcast data; the `receive` method of a node is called by the network with data that is broadcast by another node.

When an object of class *SensorNode* (see Figure 7) is generated, its `run` method triggers the two behaviors `senseTask` and `routeTask`. Until the sensor has made the number of sensings it is supposed to do (set via parameter `maxSensings`), the `senseTask` generates a data packet to be sent off. The sensor data, which for simplicity is just a counter, is added to the `sendqueue` list together with the sensor's id. The `sendqueue` list contains all messages waiting to be sent by the sensor. If there are elements in the `sendqueue` list,

```
class SensorNode(SensorId id,
                 Int maxSensings)
implements Node {
  Set<Packet> received = EmptySet;
  List<Packet> sendqueue = Nil;
  Int noSensings = 0;
  Network network = null;

  Unit run() {
    await network != null;
    this!senseTask();
    this!routeTask();
  }

  Unit setNetwork(Network network) {
    this.network = network;
  }

  Unit senseTask() {
    while (noSensings < maxSensings){
      this.store(Packet(this.id,
                        noSensings));
      noSensings = noSensings + 1;
      suspend; } }

  Unit routeTask() {
    while (True) {
      // sending a message takes
      // one time unit.
      Time t = now;
      await sendqueue != Nil;
      network!broadcast(
        this, head(sendqueue));
      sendqueue = tail(sendqueue);
      await now > t; } }

  Unit receive(Packet packet) {
    if (~contains(received, packet))
    {
      received = insertElement(
        received, packet);
      this.store(packet); } }

  Unit store(Packet packet) {
    sendqueue = Cons(packet,
                     sendqueue); } }
```

Figure 7. The implementation of the sensor nodes

the sensor's `routeTask` will broadcast the first message in the `sendqueue` list by a call to the `broadcast` method of the network object.

When a sensor receives a message from another sensor, a call to the `receive` method is made by the network. If the sensor has not seen this message before, it is added to the `received` set, and queued for re-sending. This unconditional resending implements a simple flooding routing algorithm: when a sensor senses data, it broadcasts it to all other nodes within range; when a sensor receives a message that it has not seen before, it rebroadcasts this message to all its neighbors. More involved routing algorithms exist where sensors selectively rebroadcast messages depending on whether they are on the path to the sink, but this simple algorithm suffices to illustrate our approach.

The initialization block of the model (not shown) configures the sensor network by first creating all `Sensor` objects and one `Network` object. The network topology is defined by a call `nw.setTopology(...);` that describes which sensors should be able to send to which other sensors. The topology is not necessarily symmetrical; it is possible for a sensor to receive messages from another sensor but not be able to send to it in return. Finally, the nodes start their active behavior once they can see the initialized network (modeled via a call to `setNetwork`).

Timed behavior of the sensor nodes is modeled in the `routeTask` method. When executing that method, the current time is stored. The model assumes that sending a message takes time; after broadcasting, `routeTask` waits until a period of time has passed before continuing execution.

The `SinkNode` class given in Figure 8 implements the same interface as

```
class SinkNode implements Node {
  Time lastReceived = Time(0);
  Int noReceived = 0;
  Set<Packet> received = EmptySet;

  Unit setNetwork(Network network)
  { skip; }
  Unit receive(Packet packet) {
    if (~contains(received, packet))
    {
      noReceived = noReceived + 1;
      received = insertElement(
        received, packet);
      lastReceived = now;
    } } }
```

Figure 8. The implementation of the sink node

```
class INetwork(Int bandwidth)
implements Network {
  Map<Node, List<Node>> topology
    = EmptyMap;
  Int usedbandwidth = 0;

  Unit run() {
    while (True) {
      Time t = now;
      await now > t;
      usedbandwidth = 0;
    }
  }

  Unit setTopology(Map<Node, List<Node>> topology) {
    this.topology = topology; }
  Unit broadcast(
    Node source, Packet packet)
  {
    await usedbandwidth < bandwidth;
    usedbandwidth = usedbandwidth + 1;
    List<Node> targets
      = lookup(topology, source);
    while (~isEmpty(targets)) {
      Node target = head(targets);
      target!receive(packet);
      targets = tail(targets);
    } }
}
```

Figure 9. The implementation of the network

the sensor nodes, but has a different behavior. The major difference is that the sink has no *run* method, and hence no activity of its own. The `receive` method of the sink counts the number of unique messages received and records the time when the last message was received. Figure 9 shows the implementation of the network. Its behavior is implemented by the `broadcast` method.

The bandwidth is the number of time slots in a channel. In general, nodes are scalable in sending the messages, but they are limited by the bandwidth of the sinks or gateways. In addition, the number of sent messages may become limited by the transformation strategy regarding to the power efficiency. Since transmission is the biggest source of energy drain in WSNs, having control over bandwidth consumption is important for the efficient power consumption and longevity of the nodes [15].

4.2 Simulation and Analysis

Creol's rewriting logic semantics allows the Maude rewrite engine to be used as a language interpreter in order to execute Creol models. We run a series of simulations of the WSN, obtained by varying the message arrival rate restriction and the topology of the `Network` object. The number of nodes was constant, with four sensor nodes and one sink node. The results of these simulations are given in Figure 10. We selected two "outlier" topologies: a star topology with every node directly connected to the sink, minimizing message

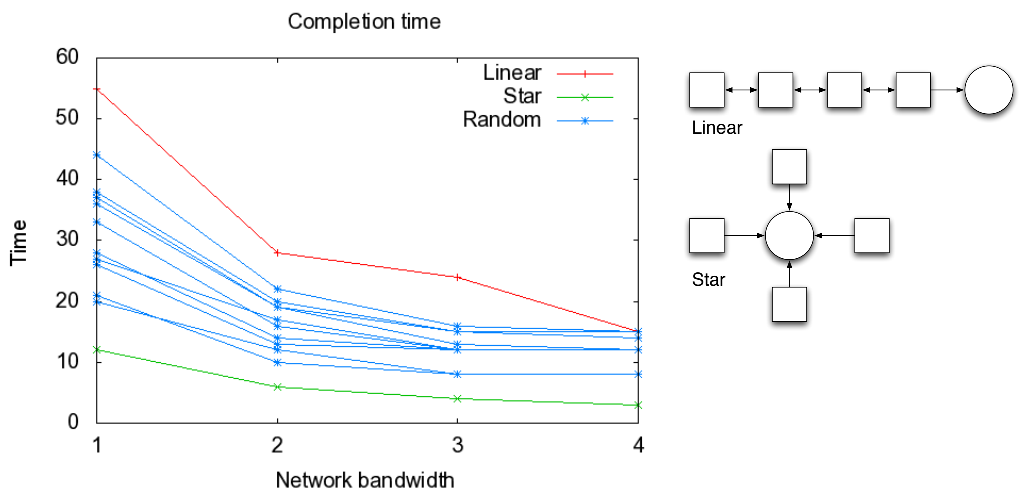

Figure 10. Message arrival times at the sink node as a function of bandwidth and network topology.

travel distance, and a linear topology where the nodes form a chain with the sink at one end, maximizing average message travel time.

We further simulated a series of randomly-generated network topologies with constant four sensor nodes and six connections. Each network contained four sensor nodes, where each node was initialized to create and send three data packets. As expected, the time for all messages to reach the sink goes up as available bandwidth (as modeled by the network arrival rate restriction) goes down. Also, the performance characteristics of the random networks can be observed to lie between those of the two chosen extreme topologies, potentially allowing reasoning about timing behaviors of arbitrary sensor networks of given connectedness based on the behavior of the boundary cases.

Simulation performance has been satisfactory for small to medium-sized models (a few dozen objects). Note that a Creol model contains less instances than the modelled system, since objects are not used as data containers and a Creol object models functionality of a system's component or subsystem.

5 Related Work

The concurrency model provided by concurrent objects and Actor-based computation, in which software units with encapsulated processors communicate asynchronously, is increasingly attracting attention due to its intuitive and compositional nature (e.g., [1–3,9,14,18,32]). A distinguishing feature of Creol is the cooperative scheduling between asynchronously called methods [21], which allows active and reactive behavior to be combined within objects as well as compositional verification of partial correctness properties [2, 14]. Creol's model of cooperative scheduling has recently been generalized to concurrent object groups in Java [30] by restricting to a single activity within each group.

In this paper, we work with Timed Creol [6], a timed extension of Creol in which the passage of time may be observed by means of the await-statements. This allows timing aspects of a model's behavior to be simulated and related to non-functional properties of the model.

Techniques and methodologies for predictions or analysis of non-functional properties are based on either *measurement* or *modelling*. Measurement-based approaches apply to existing implementations, using dedicated profiling or tracing tools such as JMeter or LoadRunner. Model-based approaches allow abstraction from specific system intricacies, but need parameters provided by domain experts [16]. A survey of model-based performance analysis techniques is given in [5]. Experimentation and simulations is the major source for obtaining an initial understanding of non-functional behavior in distributed networks. In the domain of wireless sensor networks, experiments are performed using simulators such as NS-2 and Omnet+. However, different simulators may give vastly different results, even for simple protocols [8], because the simulators make different assumptions about medium access control and physical layers [4]. In contrast, we have shown in this paper that abstract simulations give initial insights into the behavior of distributed algorithms without having to consider particular assumptions about the lower-level layers. Furthermore, formal models allow a more systematic, in-depth exploration of the execution space not only in terms of model-checking and theorem proving techniques, but also in terms of flexibility with respect to the simulation scenarios [27].

Formal approaches using process algebra, Petri Nets, game theory, and timed automata (e.g., [7, 10, 11, 17, 19]) have been applied in the embedded software and multimedia domains. A family of routing tree discovery algorithms for network diffusion protocols has been specified in TLA [26], where the specification is executable (by generating execution traces) and used to simulate individual runs as well as runs for a given set of parameters. The main performance measure of that paper is the cost of data dissemination during a data interval.

Here, we use Maude [13] to simulate Creol models by executing Creol's rewriting logic [25] semantics. Maude provides a high-level framework in which flexible and domain-specific communication forms can be specified. Maude and its real-time extension have been used to model and analyze a wide range of protocols (e.g., [27, 29, 31]), but has to our knowledge not been used to capture parametric resources for, e.g., communication rates. In the domain of WSNs, Ölveczky and Thorvaldsen have shown that simulations in Real-Time Maude can provide fairly accurate performance results compared to NS-2 simulations using high-level formal models which provide greater flexibility than traditional simulators in defining appropriate simulation scenarios [27].

Work on modelling object-oriented systems with resource constraints is more scarce. Using the UML SPT profile for schedulability, performance and time, Petriu and Woodside [28] informally define the Core Scenario Model (CSM) to solve questions that arise in performance model building. CSM has a notion of resource context, which reflects the set of resources used by an operation. CSM aims to bridge the gap between UML specifications and techniques to generate performance models [5]. Closer to our work is Hooman and Verhoef's extension of VDM++ for simulation of embedded real-time systems [20], in which architectures are explicitly modelled using CPUs and buses. Objects are deployed on CPUs and communication between CPUs takes place over buses which impose a delay for message delivery, but no restriction on the amount of messages that can be delivered concurrently.

In previous work [22, 23], the authors have developed a framework based on concurrent object groups [30] using the technique of resource-constrained deployment components which are parametric in the amount of concurrent activity they allow within a time interval. The work reported here complements that work and will be integrated with it in the future.

6 Conclusions and Future Work

In this paper we present a modeling technique which formalizes patterns for time-sensitive resource-constrained concurrent objects, applied to the modeling of communication rate restrictions. With this technique aspects of low-level deployment variability are lifted to the abstraction level of the modeling language, as illustrated here with the Creol language. The examples show how different forms of rate restricted communication can be modeled inside Timed Creol and how simulation techniques in Creol's simulation environment in the Maude rewrite system can be used to observe the effects of parametric rate restrictions on non-functional properties of models, such as message arrival times, throughput, and abstractions of packet collisions in a network of cooperating nodes. The examples illustrate how both broadcast and channel-based communication models can be modeled with parametric rate restrictions using the proposed technique. The examples capture different network behaviors, e.g., collision patterns for broadcasted data packets, and allow the timed throughput of data in high-level models of radio-based message broadcast as well as of point-to-point communication channels to be observed.

In order to observe the effects of different deployment restrictions, a timed model for Creol is utilized; simulation and testing techniques can be used to gain insights into model behavior and how the effect of the parametric rate restrictions affect non-functional properties of the models.

The technique described in this paper allows communication rate restrictions for single objects to be fully expressed in terms of Timed Creol itself. However, the modeling pattern for resource constraints is not limited to expressing rate restrictions. In future work, we plan to apply the proposed technique to other kinds of resources, e.g., in order to capture resource availability. However, the approach seems best suited for restrictions on single objects. Modeling rate-restricted communications for groups of object in a natural way requires an extension to the semantics of the Creol language similar to deployment components [23]. A thorough investigation of communication rate restrictions within the framework of deployment components remains future work. Furthermore, it is interesting to investigate stronger analysis techniques than the simulations presented in this paper by combining symbolic analysis with simulations. For example, by using state abstractions a single simulation run can capture a whole class of concrete runs but evades full model checking.

References

[1] Gul A. Agha. *ACTORS: A Model of Concurrent Computations in Distributed Systems.* The MIT Press, 1986.

[2] Wolfgang Ahrendt and Maximilian Dylla. A verification system for distributed objects with asynchronous method calls. In *Proc. International Conference on Formal Engineering Methods (ICFEM'09)*, volume 5885 of *LNCS*, pages 387–406. Springer, 2009.

[3] Joe Armstrong. *Programming Erlang: Software for a Concurrent World.* Pragmatic Bookshelf, 2007.

[4] Rena Bakhshi, François Bonnet, Wan Fokkink, and Boudewijn R. Haverkort. Formal analysis techniques for gossiping protocols. *Operating Systems Review*, 41(5):28–36, 2007.

[5] Simonetta Balsamo, Antinisca Di Marco, Paola Inverardi, and Marta Simeoni. Model-based performance prediction in software development: A survey. *IEEE Transactions on Software Engineering*, 30(5):295–310, 2004.

[6] Joakim Bjørk, Einar Broch Johnsen, Olaf Owe, and Rudolf Schlatte. Lightweight time modeling in Timed Creol. *Electronic Proceedings in Theoretical Computer Science*, 36:67–81, 2010. *Proceedings of 1st International Workshop on Rewriting Techniques for Real-Time Systems (RTRTS 2010).*

[7] Howard Bowman, Jeremy Bryans, and John Derrick. Analysis of a multimedia stream using stochastic process algebra. *Computer Journal*, 44(4):230–245, 2001.

[8] Rachel Cardell-Oliver. Why flooding is unreliable in multi-hop, wireless networks. Technical Report UWA-CSSE-04-001, University of Western Australia, February 2004.

[9] Denis Caromel and Ludovic Henrio. *A Theory of Distributed Objects.* Springer, 2005.

[10] Arindam Chakrabarti, Luca de Alfaro, Thomas A. Henzinger, and Mariëlle Stoelinga. Resource interfaces. In Rajeev Alur and Insup Lee, editors, *Proc. Third International Conference on Embedded Software (EMSOFT'03)*, volume 2855 of *LNCS*, pages 117–133. Springer, 2003.

[11] Xi Chen, Harry Hsieh, and Felice Balarin. Verification approach of Metropolis design framework for embedded systems. *International Journal of Parallel Programming*, 34(1):3–27, 2006.

[12] Maggie Xiaoyan Cheng, Lu Ruan, and Weili Wu. Coverage breach problems in bandwidth-constrained sensor networks. *ACM Transactions on Sensor Networks*, 3(2):12, 2007.

[13] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott. *All About Maude - A High-Performance Logical Framework*, volume 4350 of *LNCS*. Springer, 2007.

[14] Frank S. de Boer, Dave Clarke, and Einar Broch Johnsen. A complete guide to the future. In Rocco de Nicola, editor, *Proc. 16th European Symposium on Programming (ESOP'07)*, volume 4421 of *LNCS*, pages 316–330. Springer, 2007.

[15] Antonios Deligiannakis, Yannis Kotidis, and Nick Roussopoulos. Bandwidth-constrained queries in sensor networks. *VLDB Journal*, 17(3):443–467, 2008.

[16] Ilenia Epifani, Carlo Ghezzi, Raffaela Mirandola, and Giordano Tamburrelli. Model evolution by run-time parameter adaptation. In *Proc. 31st International Conference on Software Engineering (ICSE'09)*, pages 111–121. IEEE, 2009.

[17] Elena Fersman, Pavel Krcál, Paul Pettersson, and Wang Yi. Task automata: Schedulability, decidability and undecidability. *Information and Computation*, 205(8):1149–1172, 2007.

[18] Philipp Haller and Martin Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2–3):202–220, 2009.

[19] Matthew Hennessy and James Riely. Information flow vs. resource access in the asynchronous pi-calculus. *ACM Transactions on Programming Languages and Systems*, 24(5):566–591, 2002.

[20] Jozef Hooman and Marcel Verhoef. Formal semantics of a VDM extension for distributed embedded systems. In *Concurrency, Compositionality, and Correctness*, volume 5930 of *LNCS*, pages 142–161. Springer, 2010.

[21] Einar Broch Johnsen and Olaf Owe. An asynchronous communication model for distributed concurrent objects. *Software and Systems Modeling*, 6(1):35–58, 2007.

[22] Einar Broch Johnsen, Olaf Owe, Rudolf Schlatte, and S. Lizeth Tapia Tarifa. Validating timed models of deployment components with parametric concurrency. In B. Beckert and C. Marché, editors, *Proc. International Conference on Formal Verification of Object-Oriented Software (FoVeOOS'10)*, volume 6528 of *LNCS*, pages 46–60. Springer, 2011.

[23] Einar Broch Johnsen, Olaf Owe, Rudolf Schlatte, and Silvia Lizeth Tapia Tarifa. Dynamic resource reallocation between deployment components. In Jin Song Dong and Huibiao Zhu, editors, *ICFEM*, volume 6447 of *LNCS*, pages 646–661. Springer, 2010.

[24] Wolfgang Leister, Xuedong Liang, Sascha Klüppelholz, Joachim Klein, Olaf Owe, Fatemeh Kazemeyni, Joakim Bjørk, and Bjarte M. Østvold. Modelling of Biomedical Sensor Networks using the Creol Tools. Technical Report 1022, Norsk Regnesentral (Norwegian Computing Center), Oslo, Norway, July 2009.

[25] José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.

[26] Sule Nair and Rachel Cardell-Oliver. Formal specification and analysis of performance variation in sensor network diffusion protocols. In *MSWiM '04: Proceedings of the 7th ACM international symposium on Modeling, analysis and simulation of wireless and mobile systems*, pages 170–173. ACM, 2004.

[27] Peter Csaba Ölveczky and Stian Thorvaldsen. Formal modeling, performance estimation, and model checking of wireless sensor network algorithms in real-time maude. *Theoretical Computer Science*, 410(2–3):254–280, 2009.

[28] Dorin Bogdan Petriu and C. Murray Woodside. An intermediate metamodel with scenarios and resources for generating performance models from UML designs. *Software and System Modeling*, 6(2):163–184, 2007.

[29] Adrián Riesco and Alberto Verdejo. Implementing and analyzing in maude the enhanced interior gateway routing protocol. In *Proc. 7th International Workshop on Rewriting Logic and its Applications (WRLA'08)*, volume 238 of *ENTCS*, pages 249–266. Elsevier, 2009.

[30] Jan Schäfer and Arnd Poetzsch-Heffter. JCoBox: Generalizing active objects to concurrent components. In Theo D'Hondt, editor, *European Conference on Object-Oriented Programming (ECOOP 2010)*, volume 6183 of *LNCS*, pages 275–299. Springer, 2010.

[31] Bow-Yaw Wang, José Meseguer, and Carl A. Gunter. Specification and formal analysis of a PLAN algorithm in Maude. In Ten-Hwang Lai, editor, *ICDCS Workshop on Distributed System Validation and Verification*, pages E49–E56, 2000.

[32] Adam Welc, Suresh Jagannathan, and Antony Hosking. Safe futures for Java. In *Proc. Object oriented programming, systems, languages, and applications (OOPSLA'05)*, pages 439–453. ACM Press, 2005.