The Credo Methodology*
(Extended Version)

Immo Grabe[1], Mohammad Mahdi Jaghoori[1], Joachim Klein[3], Sascha Klüppelholz[3], Andries Stam[6], Christel Baier[3], Tobias Blechmann[3], Bernhard K. Aichernig[5], Frank de Boer[1], Andreas Griesmayer[5], Einar Broch Johnsen[2], Marcel Kyas[9], Wolfgang Leister[8], Rudolf Schlatte[2], Martin Steffen[2], Simon Tschirner[4], Liang Xuedong[7], and Wang Yi[4]

[1] CWI, Amsterdam, The Netherlands
[2] University of Oslo, Norway
[3] Technische Universität Dresden, Germany
[4] University of Uppsala, Sweden
[5] UNU - IIST, Macau, China
[6] Almende, The Netherlands
[7] RRHF, Oslo, Norway
[8] NR, Oslo, Norway
[9] Freie Universität Berlin, Germany

Abstract. This paper is an extended version of the *Credo* Methodology [16]. *Credo* offers tools and techniques to model and analyze highly reconfigurable distributed systems. In a previous version we presented an integrated methodology to use the *Credo* tool suite. Following a compositional, component–based approach to model and analyze distributed systems, we presented a separation of the system into components and the network. A high–level, abstract representation of the dataflow level on the network was given in terms of behavioral interface automata and a detailed model of the components in terms of Creol models. Here we extend the methodology with a detailed model of the network connecting these components. The *Vereofy* tool set is used to model and analyze the dataflow of the network in detail. The behavioral automata connect the detailed model of the network and the detailed model of the components. We apply the extended methodology to our running example, a peer-to-peer file-sharing system.

1 Introduction

Current software development methodologies follow a component-based approach in modeling distributed systems. A major shortcoming of the existing methods is the lack of an integrated formalism to model highly reconfigurable distributed systems at different phases of design, i.e., systems that can be reconfigured in

* This work has been funded by the European IST-33826 STREP project CREDO on Modeling and Analysis of Evolutionary Structures for Distributed Services. (`http://credo.cwi.nl`)

F.S. de Boer et al. (Eds.): FMCO 2009, LNCS 6286, pp. 41–69, 2010.

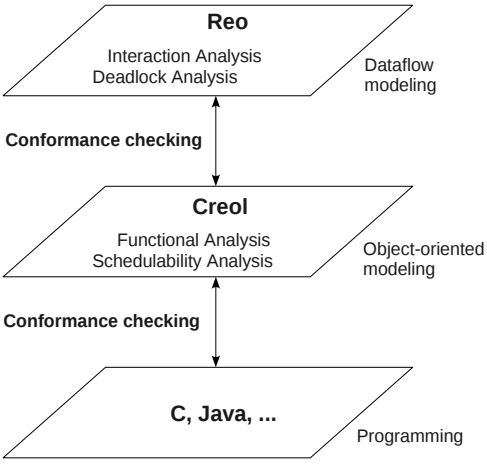


Fig. 1. Overview of modeling levels and analysis in *Credo*

terms of a change to the network structure or an update to the components. Moreover, the high complexity of such systems requires tool-supported analysis techniques.

The *Credo* methodology allows modeling on two different levels of abstraction (cf. Fig. 1). At the abstract level, i.e., the dataflow level, constraint automata [7] are used to represent the interface behavior of components and Reo [3], an executable *dataflow* language for high-level description of dynamic reconfigurable *networks*, is used to describe the glue code to connect the components. The modeling languages CARML (constraint automata reactive module language) and RSL (Reo scripting language) [6] are used for a hierarchical specification of the network and components in a compositional manner. At the concrete level, the concurrent *object-oriented* modeling language Creol [22] is used to provide an executable model of the implementation for the individual components. At this level, *Credo* offers a timed-automata framework for real-time modeling of concurrent objects. Fig. 1 illustrates the relation between the modeling languages and their relation to existing programming languages and different kinds of analysis the *Credo* tool suite provides on the chosen levels of abstraction.

In a previous version of this paper [16] we integrated the *Credo* tools and techniques into the software development life-cycle and illustrated how and when to use them during the design and analysis phases. The tools and methods presented covered a high–level model of the network and a detailed model of the components. In this paper, we extend the methodology introduced in [16] with tools and methods to model and analyze the dataflow of the network in detail. The high–level model of the network in terms of behavioral interface automata connects the detailed network model and the detailed component model; behavioral interfaces are also central to the schedulability analysis of real-time object-oriented description of a detailed component model. The connection

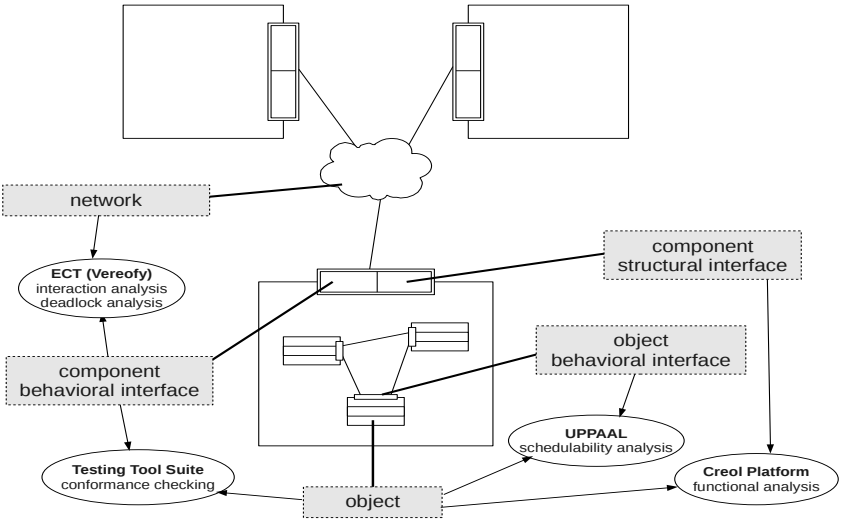


Fig. 2. End user perspective of the *Credo* Tools

between the high–level network model and the detailed component model was checked by conformance testing. In a similar approach conformance between the high–level network model and the detailed network model is established.

At the dataflow-level, which is the most abstract characterization of a system, behavioral interfaces (cf. Fig. 2) are used to describe components and the dataflow between components of a composite system. These interfaces abstract from the details of the (object-oriented) implementation of components. Instead they describe the components and the connections they use to communicate and interact with each other. *Credo* provides as an Eclipse plug-in an integrated tool-suite, ECT (Eclipse Coordination Tools) [13], including a plug-in for the model checker *Vereofy* [6,5]. *Vereofy* uses CARML and RSL as input languages and provides model checking of branching-time properties via a CTL-like logic with regular expressions to specify the observable dataflow as well as alternating-time and linear time versions thereof and bisimulation checking. The logics allow to reason about the coordination principles and the dataflow in the network as well as about the internal states of the components and behavioral interfaces.

The functional behavior of the objects within a component is modeled in Creol. Furthermore, we use the *timed automata* of UPPAAL [25,8] to create real-time models of objects and their behavioral interfaces. The *Credo* tool suite offers an automated technique for *schedulability analysis* of individual objects [21,20]. Given a specification of a scheduling policy (e.g., earliest deadline first) for an object, we use UPPAAL to analyze the object with respect to its behavioral interface in order to ensure that tasks are accomplished within their specified deadlines.

Conformance between a model of a component implementation and its behavioral interface specification is checked by the *Credo* tools [17]. Moreover, given an implementation of a component in a programming language like C, *Credo* also provides a technique to check conformance between the implementation and the Creol model [18,1]. Both techniques are based on *testing*. The abstract behavioral interface model is used to generate test cases to steer the execution.

To illustrate the *Credo* methodology we will give a running example. Throughout the paper we model and analyze a file-sharing system with hybrid peer-to-peer architecture (like in Napster), where a central server keeps track of the data in every peer node.

In Section 2, we develop the structural and behavioral interfaces of the components (peer nodes of the P2P system) and the network (the network manager managing the dynamic connections between peer nodes); and prove some example properties of different kinds. In Section 3, we give a detailed model of the network using the *Vereofy* tool suite and analyze it by means of simulation and model checking. In Section 4, we give executable object-oriented models for the components and analyze them by means of simulation and testing for conformance both with respect to the behavioral interfaces and a Creol implementation. We demonstrate schedulability analysis by analyzing the central server of the peer-to-peer example. Section 5 concludes the paper.

2 High-Level Dataflow Model

We use the exogenous coordination language Reo [3] for the high-level dataflow modeling. Reo is a channel-based formalism that supports compositional design of the network that yields the *glue-code* for a given set of components. In Reo, a system consists of a set of components connected by a network. The network exogenously controls the dataflow between the components and may be dynamically reconfigured to alter the connections between the components. At this level of abstraction, only a *facade* of each component is visible. A facade consists of port and event declarations, and its abstract behavior is specified using an automata model called constraint automata [7]. Constraint automata are variants of labeled transition systems where the transitions are labeled by sets of read and write operations on I/O-ports of components and dataflow locations of the

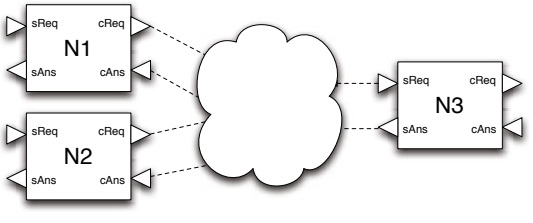


Fig. 3. Peer nodes in the P2P system

network, possibly together with data constraints for the written or read data values. Besides describing the interface behavior of the components, constraint automata also serve as a formal semantics for Reo [7]. In this section, we do not go into the details of how to compose Reo channels. Instead, we use constraint automata as a model for the network behavior directly.

Components use ports to communicate with each other via the network. Fig. 3 shows a system of components (as rectangles), their ports (as small triangles), and the network (as a cloud). Ports can be either input or output ports (implied by the direction of the triangles). By exogenous coordination, we mean that a component has no direct control on how its ports are connected. A component can only indirectly influence its connections by raising events. Events include requests/announcements of services, time-outs, or acknowledgments. These events can trigger reconfigurations of the *context-aware* network. A network manager handles the events and reconfigures the network connections according to the events. At this moment we consider the network manager to be a part of the network and we model the peer nodes independent of a concrete implementation of the network manager.

In this section, we model the peer nodes of the P2P system as components. Each peer node has two sides, a client side and a server side. Each side has a pair of request and answer ports. As a client, a peer node writes a request (a 'key' identifying the requested data) to its cReq–port and expects the result on its cAns–port. As a server, a peer node reads a request from its sReq–port and writes the result to its sAns–port. For two peer nodes to communicate, the network manager has to connect the corresponding ports of the client and the server, i.e., the cReq–port of the client with the sReq–port of the server and the cAns–port of the client with sAns–port of the server.

2.1 Structural Interface Description

To describe the facade of a component, we declare its ports and the events the component may raise. Below, we define two facades, ClientSide and ServerSide. The facade Peer inherits the ports and events declared in these two and adds another event that is needed when the two sides are combined.

```
facade ClientSide begin
  port cReq : outport
  port cAns : inport
  sync_event openCS<req:outport,ans:inport>(in k:Data; out f:Bool)
  sync_event closeCS<req:outport,ans:inport>()
end
```

```
facade ServerSide begin
  port sReq : inport
  port sAns : outport
  sync_event openSS<req:inport,ans:outport>()
  sync_event closeSS<req:inport,ans:outport>()
  register<>(in keyList : List[Data]) // async_event
end
```

```
facade Peer inherits ClientSide, ServerSide begin
  update<>(in keyList : List[Data])   // async_event
end
```

The network manager does not keep a centralized account of all port bindings; these are locally stored at each component. A component cannot directly change its port bindings. Before using ports, the component must request a connection by raising an open session event. An event for closing the session implies that the ports are ready to be disconnected. When requesting to open a session or reporting the end of a session the ports used in that session are send as parameters. In addition to the ports, events can have extra parameters, e.g., the 'open client session' event (written as openCS) provides the key to the data it is looking for as additional information to steer the connection process. Based on the data key the network manager can set up a connection to a server that holds the requested data.

Events are by default asynchronous. However, when expecting return values (e.g., opening or closing a session), we declare events to be synchronous (using the keyword **sync_event**). All events raised by the components are handled by the network. This is reflected in the structural interface description of the network.

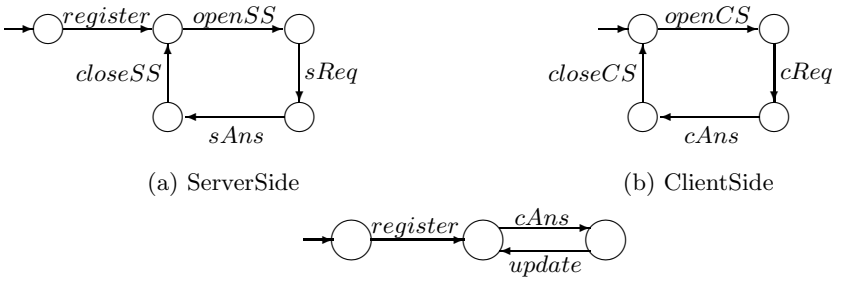
Network. We give the structural interface description of a particular network manager called Broker. The keyword **networkmanager** is used to identify such interfaces (and distinguish them from those characterizing component facades). The *Credo* methodology distinguishes between the concept of a network manager and the network itself because a network in general consists of a network manager and additional coordination artifacts like *channels*, as described later in this section.

The description of the Broker declares the event handlers that it provides. For each event handler, it specifies the facade (representing a component) from which the handled event originated using the keyword **with**.

```
networkmanager Broker begin
  with ServerSide
    register<>(in keyList : List[Data])
    sync_event openSS<in req:inport,ans:outport>()
    sync_event closeSS<in req:inport,ans:outport>()
  with ClientSide
    sync_event openCS<in req:outport,ans:inport>(in k:Data; out f:Bool)
    sync_event closeCS<in req:outport,ans:inport>()
  with Peer
    update<>(in keyList : List[Data])
end
```

2.2 Behavioral Interface Description

The behavioral description for a component facade specifies the order of raising events and the port operations. This is modeled using constraint automata [4]. In these automata, we denote port operations by port names. The corresponding



(a) ServerSide (b) ClientSide

(c) Peer

Fig. 4. Behavioral interfaces for facades

action (read or write) is understood from the port type (given in the structural facade description).

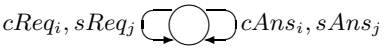
Fig. 4 shows the behavioral specification for the facades in our example. As mentioned earlier, the port actions are enclosed by opening and closing session events in Fig. 4(a) and Fig. 4(b). A server registers its data with the network manager at initialization. We opt for a simple scenario, i.e., each server or client handles only one request at a time. We also assume at this level of abstraction, that openCS is always successful, i.e., every data item searched for is available.

The Peer facade inherits the behavior specified for ClientSide and ServerSide facades. The Peer facade introduces some additional behavior, i.e., an update to the data stored at the broker. The Peer automaton (see Fig. 4(c)) synchronizes with the ServerSide automaton (see Fig. 4(a)) to ensure that an update only takes place after the data is registered. Moreover, the data at the broker is updated after receiving new information (on the ClientSide). This is modeled by synchronization on the read operations on the *cAns*–port.

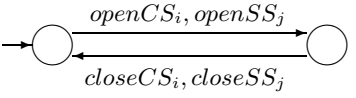
The behavior of the sub-type has to be a refinement of the behavior of its super-type [28]. This is achieved by computing the product of the automata describing the inherited behavior (ServerSide and ClientSide) and the automaton synchronizing them (Peer). In this product [4] transitions with different action names are interleaved while those with common action names are synchronized.

Network. The Broker in a peer-to-peer system connects the ports and handles the events of the components. We show how to model the synchronization of a system consisting of a fixed number of components, say n, for some $n > 0$. The observable actions of the i–th component ($i \in \{1, \ldots, n\}$), i.e., the communications on its ports and its events, are denoted by $openCS_i$, $openSS_i$, $closeCS_i$, $closeSS_i$, $cReq_i$, $sReq_i$, $cAns_i$, and $sAns_i$. Synchronization of actions is modeled in the following automata by a transition labeled with the participating actions.

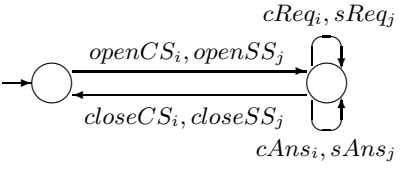
For clarity, we start with different automata for the synchronization of ports and events. Synchronization between the ports of a pair of components i and j is described by the following automaton.



For each pair of components i and j, the following automaton synchronizes the events $openCS_i$ and $openSS_j$ to establish a connection between components i and j and the events $closeCS_i$ and $closeSS_j$ to release the connection again. These two consecutive synchronizations together thus model one session between the client of component i and the server of component j.

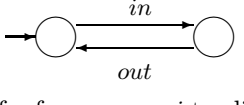


Combining the automata above models the port connections in a session (shown below). As stated before communication between components is only possible after requesting a session to be opened. After the components have finished their communication the session is closed. The *interleaving product* of these combined automata for all pairs of components results in an automaton describing the behavioral interface of the Broker.

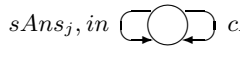


Notice that interleaving allows for components to be involved in more than one session at a time. The *synchronized product* of the network manager automaton with the component automata (from the previous subsection) describes the overall behavior of the system. The product restricts the network manager and the components to exclusive sessions, i.e. a component is involved in at most one session at a time.

Channels. We further refine the network model by introducing *channels* (which are primitive connectors) [3,19]. In general, a channel provides two (channel)-*ends*. We distinguish between input-ends (to which a component can *write*) and output-ends (from which a component can *read*). We also describe the synchronization between the two channel-ends by an automaton. For example, the automaton below models a 1-place buffer. It provides an input-end *in* and an output-end *out*. In state e the buffer is empty and in state f it is full (for simplicity, we abstract from the data transferred and stored).



We model the data-transfer from server j to client i, i.e., the connection between the answer ports, by replacing the synchronization of $cAns_i$ and $sAns_j$ by the following synchronization with the above 1-place buffer.



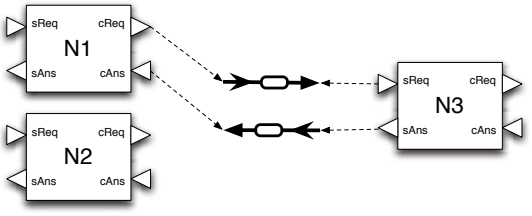


Fig. 5. Using Reo channels for modeling the network

The overall behavior of the system is described by the synchronized product of the Broker, the component automata, and the channel automata. The network itself consists of the Broker and the channels. Fig. 5 shows a configuration in which two buffer channels are used as the network connecting the components. The dashed arrows in this figure show port bindings, i.e., the channel-end to which a port is bound. The bold arrows represent the channels.

3 Dataflow Model

In this section we give a detailed model of the dataflow of our peer–to–peer example. We use the *Vereofy* [10,6,11] (see Fig. 6) tool suite for modeling and analyzing the detailed dataflow model. The *Vereofy* tool suite supports model checking and equivalence checking of components, connectors, and the composite system. Constraint automata serve as a generic operational semantics, which is used for the service interfaces of the components, the network that provides the glue code, and the composite system.

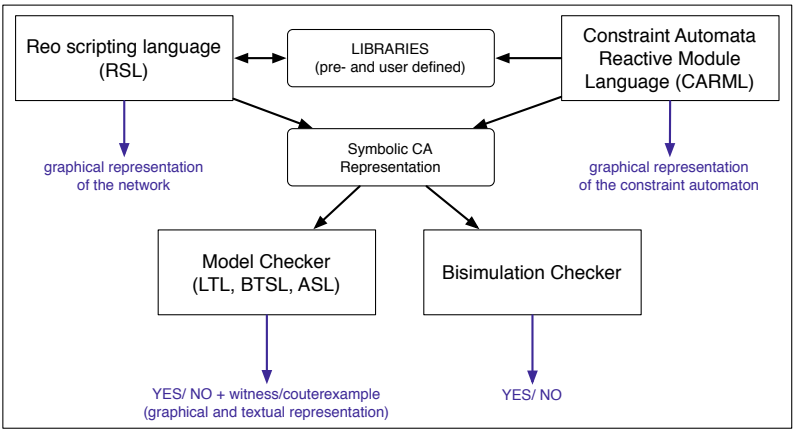


Fig. 6. The *Vereofy* tool suite

We use the specification languages Reo Scripting Language (RSL) for Reo and the Constraint Automata Reactive Module Language (CARML) to specify service interfaces of the components. While the scripting language RSL is used to specify exogenous or endogenous coordination mechanisms, the guarded command language CARML is used to specify behavioral component interfaces and component connectors. Both languages rely on the same semantic automata model. This hybrid approach allows nesting of the two specification languages, supports compositional design, modular verification and reusability of components and component connectors. *Vereofy* includes symbolic model checking tools for linear-time, branching-time and alternating-time temporal logics [24,23,5] with special operators to reason about the events and dataflow at I/O-ports of components and internal nodes of the connecting network. Furthermore *Vereofy* includes a bisimulation checker [9] for components, component connectors, and the composite system.

In the following we show how to model the network manager establishing connections in our running peer-to-peer example. We use CARML to provide textual specifications of the facades of server and client side and RSL to specify the network manager. Finally we explain how the model checking engine of *Vereofy* is used to validate the composite system. The full source code for the P2P model is available on the web:

`http://www.vereofy.de/download/examples/vereofy_p2p_example.zip`

3.1 Modeling in *Vereofy*

The facades from Section 2 serve as a starting point to model the server and client side. Facades define the interface ports together with the possible events. In this section we follow an exogenous modeling approach where the communication and coordination of the peers is handled completely outside the components by the connecting network. Thus, there are no complex events inside the component specifications, i.e., the CARML code for the server and client side. Instead, events are handled by the network manager using synchronous message passing via I/O-ports. The specification of I/O-ports in CARML differs only syntactically from the facade definition presented earlier.

The Server side and client side facades in *Vereofy*. The automata from Section 2 for the server side and client side facades are directly translated into CARML modules. A CARML specification consists of a (possibly empty) list of parameter (e.g. the number of I/O-ports), the interface declaration where source ports (for the input-ends) and sink ports (for the output-ends) of a component and its local variables are defined followed by the transition definitions specifying the behavioral interface. The evaluations of the local variables represent automata states. The transition definitions have the form

$$state\_guard -[\, I/O\_guard \,]\mapsto state\_assignments;$$

where the *state_guard* represents a boolean expression on the current evaluation of the variables, *I/O_guard* is a boolean expression on the dataflow observed

at the interface ports, and *state_assignments* describe the effect on the local variables. An I/O-guard specifies the list of active ports as well as restrictions to the data observed at the active ports. E.g., the I/O-guard "$\{\text{A}\}$ & $\#_{\text{A}} == \text{k}$" states that port A is the only port active during the transition and the observable data value at A is equal to k.

To reduce the complexity of our model for demonstration purposes we (1) abstract from the update events, (2) assume that all peers have all data, and (3) the network manager establishes a connection to server_i if data_i is requested. Furthermore, we use a global data domain

$$\begin{aligned}\text{Data} = \{&0, 1, 2, 3,\\ &\text{key}_0, \text{key}_1, \text{key}_2,\\ &\text{data}_0, \text{data}_1, \text{data}_2,\\ &\text{openSignal}, \text{closeSignal}, \text{registerSignal},\\ &\text{undefined}\}\end{aligned}$$

for the requests, the data and all signals. The numbers $0, 1, 2, 3 \in \text{Data}$ are used as signals triggering a reconfiguration in the network topology. Please note, that for each message type a distinct input or output port has been introduced according to the facades definition from Section 2.1. An alternative way of modeling uses

```
1  MODULE ClientSide{
2    // interface declaration (specification of I/O-ports):
3    in: openCS;
4    in: closeCS;
5    out: myReq;
6    in: myAns;

8    // local variables:
9    var: enum{idle,open,waiting,done} status:=idle;
10   var: Data tans := undefined;

12   // transition definitions:
13   status==idle    -[ {openCS} & #openCS==openSignal ]->
14     status:=open;

16   status==open    -[ {myReq} & #myReq==key0 ]-> status:=waiting;
17   status==open    -[ {myReq} & #myReq==key1 ]-> status:=waiting;
18   status==open    -[ {myReq} & #myReq==key2 ]-> status:=waiting;

20   status==waiting -[ {myAns} ]-> status:=done & tans:=#myAns;

22   status==done    -[ {closeCS} & #closeCS==closeSignal ]->
23     status:=idle & tans:=undefined;
24 }
```

Fig. 7. CARML module for client facade of a peer

less (or even single) input and output ports and structured data types, such as disjoint unions. For the usage of structured data types we refer to the *Vereofy* user manual [11]. Fig. 7 depicts the CARML code for the client facade of a peer.

The interface of the client side facade has three input ports (openCS, closeCS, and myAns) and one output port (myReq). The variable status stores the current state of the peer (initially idle), while the variable trans stores an element from the global data domain Data when the peer receives one (initially undefined). The server side facade of the peer is modeled analogously.

The network manager. We model the network manager in the scripting language RSL which is inspired by the exogenous, channel-based coordination language Reo [3]. Both Reo and RSL yield elegant declarative frameworks for the specification of circuits, i.e., for the compositional construction of (dynamically changing) component connectors by creating channels and gluing their channels ends, the I/O-ports of components, or sub-connectors together. RSL's core language features are (1) instantiation of modules and sub-circuits (via the RSL command new); (2) gluing instances together (explicitly via the RSL command join, or implicitly by reusing port names); (3) forming a new prototype for entities for a higher modeling level (via the arrays source and sink); (4) defining networks with dynamically changing topologies (via the RSL keyword TOPO); (5) and scripting features such as variables, loops, and conditional branching.

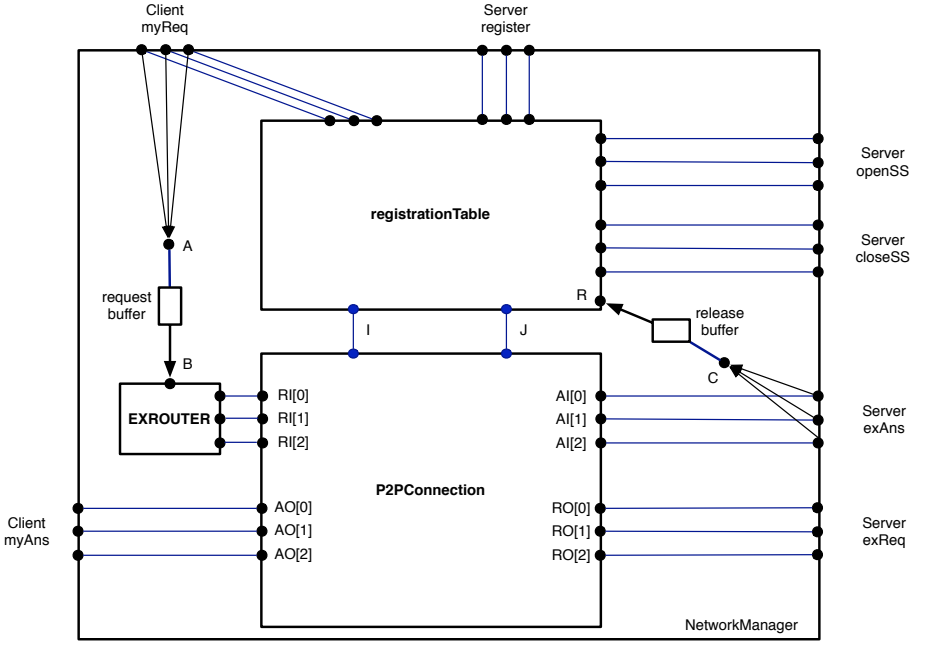


Fig. 8. The network manager

```
#include "builtin"
#include "registrationtable.carml"
#include "p2pconnection.rsl"

CIRCUIT NetworkManager{
  // create registration table
  Table = new registrationTable(source[0],source[1],source[2],
                                source[3],source[4],source[5],R;
                                I,J,
                                sink[0],sink[1],sink[2],
                                sink[6],sink[7],sink[8]);

  // node A merges the requests
  for (i=0;i<3;i=i+1){
    Sync[i] = new SYNC(source[i+3];A);
  }

  // the requests are beeing buffered in the FIFO1
  request_buffer = new FIFO1(A;B);

  // the buffered request later goes via an exclusive router
  // into the connections matrix (P2PConnection)
  new EXROUTER<3>(B;RI[0],RI[1],RI[2]);

  // create P2PConnection to direct requests and answers
  Connections = new P2PConnection(I,J,RI[0],RI[1],RI[2],
                                  AI[0],AI[1],AI[2];
                                  RO[0],RO[1],RO[2],
                                  AO[0],AO[1],AO[2]);

  // the answers are merged into a single node C
  for(i=0;i<3;i=i+1){
    Sync[i+3] = new SYNC(AI[i];C);
  }

  // and buffered in the release_buffer for
  // the later release (in the registration table)
  release_buffer = new FIFO1(C;R);

  // rest of the interface declaration
  source[6] = AI[0]; source[7] = AI[1]; source[8] = AI[2];
  sink[3] = RO[0]; sink[4] = RO[1]; sink[5] = RO[2];
  sink[9] = AO[0]; sink[10] = AO[1]; sink[11] = AO[2];
}
```

Fig. 9. RSL script composing a network manager

An overview on the structure of the network manager is shown in Fig. 8. The network manager consists of several distinct entities, some of them are modeled in CARML while others are specified using RSL. The RSL code composing these entities to form the network manager is presented in Fig. 9. The RSL main program, which is not shown here, composes the peers and the network manager the same way.

The basic idea behind the model of the network manager is that registrationTable – specified in CARML – keeps track of the server registrations, notices requests from clients and generates indices $i, j \in \{0, 1, 2\}$ serving as reconfiguration signals for the dynamically changing P2PConnection – specified in RSL. When peer i sends the key for the ℓ-th data package (key_ℓ) indicating the request for data_ℓ, then the registrationTable is aware which of the registered servers has the requested data. If peer j is the one whose server side has already registered and has the requested data, the registrationTable opens a server session by sending the signal openSignal via the I/O-port openSS_j. Moreover it sends the indices i and j via the internal ports I and J to the P2PConnection. The P2PConnection then establishes a bidirectional connection between peer i and peer j first for the requests and then for the answers. The requests are kept in the request_buffer and delivered to exactly one of the ports $RI[0], RI[1]$ or $RI[2]$ of the P2PConnection using an exclusive router component (EXROUTER). After the connection has been established the request is routed through the P2PConnection from $RI[i]$ to $RO[j]$, i.e, to exReq of peer j. When the request is answered by the server side the data is delivered through the P2PConnection from $AI[j]$ to $AO[i]$, i.e, from port exAns of peer j to port myAns of peer i. A copy of the data package is kept in the release_buffer. In the next step the copy is forwarded to the registrationTable generating new reconfiguration signals on the internal I/O-ports I and J disconnecting the peers. Moreover, it sends the signal closeSignal via the I/O-port closeSS_j to close the server session. The network manager is now back in its initial configuration, ready for a new request-answer-cycle.

The synchronous channels (Sync), the buffers (FIFO1), and the exclusive router (EXROUTER) are part of *Vereofy*'s built-in library. The predefined channels and component connectors from the library can be instantiated like any other component. The composition of channels and components is done implicitly during the instantiation by reusing port names in the new statements. If a port name is used more than once the corresponding ports are joined. E.g. we write new FIFO1$(A; B)$; new FIFO1$(B; C)$ instead of new FIFO$(A; B1)$; new FIFO1$(B2; C)$; $B = \text{join}(B1, B2)$. If the name of a port is source$[i]$ (or sink$[j]$) the port will be the i-th source port (j-th sink port, respectively) of the interface of the network manager. I.e., the network manager provides the interface shown in Table 1.

Dynamically changing network topologies. We now focus on the dynamically changing part of the network, i.e., the P2PConnection. As described above the registrationTable triggers a reconfiguration of the network topology. A P2PConnection manages a bidirectional communication between peer_i and peer_j on the basis of the incoming signals at the I/O-ports I and J. These signals are simultaneously

Table 1. Interface of a network manager

I/O-ports	port type	usage	data values
register $_i$	input	register servers sides	registerSignal
openSS$_i$	output	opening a server session	openSignal
myReq$_i$	input	handling client requests	key$_0$, key$_1$, key$_2$
exReq$_i$	output	forwarding requests	key$_0$, key$_1$, key$_2$
exAns$_i$	input	accepting answers from servers	data$_0$, data$_1$, data$_2$
myAns$_i$	output	forwarding answers	data$_0$, data$_1$, data$_2$
closeSS$_i$	output	closing a server session	closeSignal

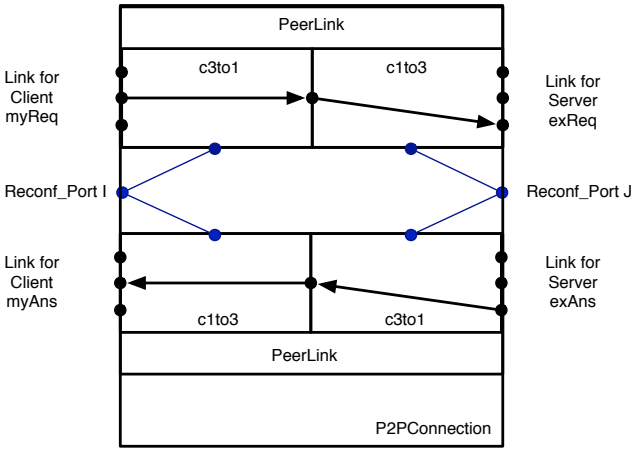


Fig. 10. P2PConnection

forwarded into two sub-circuits – one peerLink for the requests and another peerLink for the answers (see Fig. 10).

Both peerLinks consist of sub-circuits called c3to1 and c1to3. We select c3to1 as a showcase for circuits with more than one (static) topology. A dynamic circuit needs a static interface, which must not be changed within the topology descriptions. The RSL code for c3to1 consisting of the interface declaration followed by the definition of four possible topologies is shown in Fig. 11. The RSL keyword `NODE` is used to create a new I/O-port. In the RSL code of the c3to1 in Fig. 11 four nodes are created – three nodes for the input ports and one for the output port constituting the interface of the circuit.

In the first three topologies (topology 0, 1, 2) exactly one of the source ports is connected to the sink via a synchronous channel. In the last topology (topology 3) there are no connections between the sources and the sink port.

The circuit dynamically switches to topology i ∈ [0..3] when receiving a reconfiguration signal i. As shown in Fig. 12 the initial topology can be selected on the instantiation of the sub-circuit providing the `initial_topo` option as shown

```
CIRCUIT c3to1{
 outport = NODE;
 sink[0] = outport;

 for (i=0;i<3;i=i+1){
  inport[i] = NODE;
  source[i] = inport[i];
 }

 TOPO(0) = {
  new SYNC(inport[0];outport);
 }

 TOPO(1) = {
  new SYNC(inport[1];outport);
 }

 TOPO(2) = {
  new SYNC(inport[2];outport);
 }

 TOPO(3) = {
  // unconnected
 }
}
```

```
CIRCUIT c1to3{
 inport = NODE;
 source[0] = inport;

 for (i=0;i<3;i=i+1){
  outport[i] = NODE;
  sink[i] = outport[i];
 }

 TOPO(0) = {
  new SYNC(inport;outport[0]);
 }

 TOPO(1) = {
  new SYNC(inport;outport[1]);
 }

 TOPO(2) = {
  new SYNC(inport;outport[2]);
 }

 TOPO(3) = {
  // unconnected
 }
}
```

Fig. 11. RSL script for a building block in the P2PConnection

for the peerLink. From the RSL code one can also see how the reconfiguration port becomes an additional interface port of the sub-circuits c3to1 and c1to3 and how it can be accessed. The P2PConnection is composed out of two peerLinks, one for the requests and one for the answers.

3.2 Analysis of the Model

Vereofy provides model checking for branching time properties via the CTL-like logic BTSL[24] and the alternating-time logic ASL[23], for linear time properties via $\mathsf{LTL}_{\mathsf{IO}}$[5], as well as bisimulation checking [9]. The three logics allow reasoning about the coordination principles and the dataflow in the network, i.e., between components, as well as the internal states of the components and component interfaces. BTSL (Branching Time Stream Logic) is a CTL-like logic with path quantifiers and formulas built by standard temporal operators, extended by special modalities to specify regular properties for data stream prefixes. $\mathsf{LTL}_{\mathsf{IO}}$ is likewise an extended version of LTL adapted to the constraint automata setting, where the atomic propositions are either state predicates or I/O-guards. ASL (Alternating Stream Logic) extends BTSL by means to allow reasoning about compatibility and the existence (and absence) of strategies for (alliances of) components. In this section we illustrate the type of properties expressible in BTSL,

```
#include "c3to1.rsl"
#include "c1to3.rsl"

CIRCUIT peerLink{
  // instantiation with port names including the reconf. ports:
  first = new c3to1(A[0],A[1],A[2],
              here_is_reconf_port_first; C) with initial_topo=3;

  second = new c1to3(C, here_is_reconf_port_second;
                           B[0],B[1],B[2]) with initial_topo=3;

  /*
    defining the interface using
    two differnt ways in accessing
    the reconf_port of dynamic sub-circuits
  */
  source[0] = first.RECONF_PORT;
  source[1] = here_is_reconf_port_second;

  // defining the rest of the interface
  for (i=0;i<3;i=i+1){
    source[i+2] = A[i];
    sink[i] = B[i];
  }
}
```

Fig. 12. RSL script composing a peer link

ASL and $\mathsf{LTL}_{\mathsf{IO}}$ by providing some examples that can be checked with the help of *Vereofy*. For this, we make use of the following notations inside the formulas:

- $\{A, B\}$ indicates that ports A and B are active and no other port is.
- #A refers to the data item observed at port A.
- `step` indicates an arbitrary step with or without observable dataflow.
- the operators ; (concatenation), * (star), and $^+$ (plus) correspond to the standard operators for regular languages.

1. Deadlock freedom, in the sense that on all paths there is always a next step, can be formalized by means of the following CTL formula.

   ```
   AG[EX[true]]
   ```

2. With BTSL we can formalize a condition stating the existence of a path with specific regular form. A sequence of actions is specified by a regular expression, where the atoms are constraints on a single step of the observable dataflow. This can e.g. be instrumented to check the conformance between the behavioral interfaces and the RSL model. The following formula states the existence of a path, where the first server registers, the second client opens a session and sends a request for the data with key0, the data is transferred, and the connections closed. The formula also requires that the path leads to a state where both the request and the release buffer are empty.

```
E<"{register[0]} & #register[0]==registerSignal";"step"*;
     "{openCS[1]} & #openCS[1]==openSignal";
     "{request[1],openSS[0]} & #request[1]==key0
      & #openSS[0]==openSignal";"#sendRequest[0]==key0";
     "{theAnswerIn[0],theAnswerOut[1]}
      & #theAnswerIn[0]==#theAnswerOut[1]
      & #theAnswerOut[1]==data0";
     "{closeSS[0],closeCS[1]} & #closeSS[0]==closeSignal
      & #closeCS[1]==closeSignal">
  "Manager.request_buffer.state==EMPTY
    & Manager.release_buffer.state==EMPTY"
```

3. We now provide a BTSL formula for the requirement stating that for all possible executions whenever the dataflow satisfies the dataflow specification (i.e., it is part of the language defined by the regular expression) then both buffers will be empty at the end of the execution.

```
A["{register[0]} & #register[0]==registerSignal";("step"*;
     "{openCS[1]} & #openCS[1]==openSignal";
     "{request[1],openSS[0]} & #request[1]==key0
       & #openSS[0]==openSignal";"#sendRequest[0]==key0";
     "{theAnswerIn[0],theAnswerOut[1]}
       & #theAnswerIn[0]==#theAnswerOut[1]
       & #theAnswerOut[1]==data0";
     "{closeSS[0],closeCS[1]} & #closeSS[0]==closeSignal
       & #closeCS[1]==closeSignal")+]"
  Manager.request_buffer.state==EMPTY
   & Manager.release_buffer.state==EMPTY"
```

4. The next property given in terms of an $\mathsf{LTL}_{\mathsf{IO}}$ formula asserts that whenever a server session has been closed in the next step the release buffer of the network manager will be empty.

```
G (("#closeSS[0]==closeSignal"
     | "#closeSS[1]==closeSignal"
     | "#closeSS[2]==closeSignal") ->
   X "Manager.release_buffer.state==EMPTY" )
```

5. The following $\mathsf{LTL}_{\mathsf{IO}}$ formula represents a fairness condition and ensures that enabled requests can not be ignored forever. Stated differently, if the request of a client is enabled at infinitely many positions along a path, then the request fires at infinitely many locations.

```
G F "enabled sendRequest[1]" -> G F "sendRequest[1]"
```

6. The ASL formula given below states that whether there is a strategy that controls, i.e. constraints, the possible dataflow at the three ports (theAnswerOut[0], theAnswerOut[1], and theAnswerOut[2]), such that for all remaining paths the release buffer of the network manager stays globally empty.

```
E{theAnswerOut[0], theAnswerOut[1], theAnswerOut[2]}
  G["Manager.release_buffer.state==EMPTY"]
```

All properties that have been presented in this section have successfully been validated for our model of the peer-to-peer network for which the full source code is available on the web [30]. Besides model checking for temporal logics, *Vereofy* supports checking bisimilarity of two automata, e.g., that two implementations of the network manager are bisimilar.

4 Object-Oriented Model of the Components

In this section, we model the components in Creol, an executable modeling language. To model the components, we provide interfaces for the intra-component communication and a Creol implementation of the components. Together with a Creol implementation of the network manager, we get an executable model of the whole system. Since Creol models are executable we use the terms Creol model and Creol implementation interchangeably.

We use intra-component interfaces together with the behavioral interfaces of Section 2.2 to derive test specifications to check for conformance between the behavioral models and the Creol implementation. We also use this specification to simulate the environment of a component while developing the component.

Given a C implementation of the system, we use the behavioral interfaces of Section 2.2 to derive test scenarios for checking conformance between the Creol model and an implementation in an actual programming language. Dynamic symbolic execution on the Creol implementation is used to compute test inputs for the scenarios for an improved coverage of the model [18].

Finally, we model the real-time aspects of the system using timed automata. In the real-time model, we add scheduling policies to the objects. Here, we check for schedulability, i.e., whether the tasks can be accomplished within their deadlines.

4.1 Modeling in Creol

Creol is an executable modeling language suited for distributed systems. Types are separated from classes, instead (behavioral) interfaces are used to type objects. Objects are concurrent, i.e., conceptually, each object encapsulates its own processor. Creol objects can have active behavior, i.e., during object creation a designated `run` method is invoked.

Creol allows for flexible object interaction based on asynchronous method calls, explicit synchronization points, and underspecified (i.e., nondeterministic) local scheduling of the processes within an object. Creol supports software evolution by means of runtime class updates [31]. This allows for runtime reconfiguration of the components. To facilitate the exogenous coordination of the components we have extended Creol with facades and an event system (cf. Section 2.1).

The modeling language is supported by an Eclipse modeling and analysis environment which includes a compiler and type-checker, a simulation platform

based on Maude [12], which allows both closed world and open world simulation as well as guided simulation, and a graphic display of the simulations.

In the rest of this section, we specify the interfaces of a local data store for a peer syntactically. Then, we implement parts of a peer as an example.

Each peer consists of a client object, a server object and a data-store object. The Client interface provides the user with a search operation. The data-store provides the client object with an add operation to introduce new data and the server object with a find operation to retrieve data. We model these two perspectives on the data-store by two interfaces StoreClientPerspective and StoreServerPerspective.

The interfaces are structured in terms of inheritance and cointerface requirements. The cointerface of a method (denoted by the **with** keyword) is a static restriction on the objects that may call the method. In the model, the cointerface reflects the intended user of an interface. In Creol, object references are always typed by interfaces. The caller of a method is available via the implicit variable **caller**. Specifying a concrete cointerface allows for callbacks. Finally, method parameters are separated into input and output parameters, using **in** and **out** keywords, respectively.

```
1 interface StoreClientPerspective begin
2  with Client
3   op add(in key:Data, info:Data)
4 end

6 interface StoreServerPerspective begin
7  with Server
8   op find(in key:Data; out info:Data)
9 end

11 interface Store
12  inherits StoreClientPerspective, StoreServerPerspective
13 begin end
```

The interfaces cover the intra-component communication while the facades cover the inter-component communication (cf. Section 2.1). To implement a Creol class, we can use only the ports and events specified in the facades. Note that the use of ports is restricted to reading from an inport or writing to an outport. Since the inter-component communication is coordinated exogenously by the network, the components are not allowed to alter the port bindings; instead, they have to raise an event to request a reconfiguration of the communication network structure.

Next, we provide implementation models for the interfaces in terms of Creol classes. The client offers a search method to the user. To perform a search, the client makes a request to the broker. The event openCS<req, ans>(key; found) provides the ports req and ans to be reconfigured, plus the parameters key and found. If the data identified by key is available, the broker connects the given ports to a server holding the data and reports via found the success of the search. Otherwise, the ports are left unchanged and the failure is reported via found. If successful the

client expects its ports to be connected properly and communicates the data via its ports.

For simplicity, a client only operates one search at a time. Nevertheless, the user can issue multiple concurrent search requests. The requests are buffered and served in an arbitrary order (due to the nondeterministic scheduling policy) one at a time.

```
1  class ClientImp (store:StoreClientPerspective, req:outport, ans:inport)
2     inside Peer implements Client begin

4     with User op search(in key:Data out result:Data) ==
5        var found : Boolean;
6        raise_event openCS<req, ans>(key; found);
7        if (found) then
8           req.write(key;);
9           ans.take(;result);
10          ! store.add(key, result)
11       end;
12       raise_event closeCS<req, ans>()
13 end
```

To obtain the result of the search, the client uses a synchronous call to the ans port. The update regarding the new data is sent to the data-store asynchronously ! store.add(key, result). Using asynchronous communication the client can already continue execution while the data-store is busy processing the changes. The client is a passive object, i.e., it does not specify a run method.

The server object is active in the sense that it starts its operation upon creation. The active behavior is specified in the run method. This involves reading data requests from the req port and delivering the results on the ans port. To repeat the process, the run method issues an asynchronous self call before termination.

```
1  class ServerImp (store:StoreServerPerspective, req:inport, ans:outport)
2        inside Peer implements Server
3  begin
4     op run ==
5        var key, result:Data;
6        raise_event openSS<req, ans>();
7        req.take(;key);
8        store.find (key; result);
9        ans.write(result;);
10       raise_event closeSS<req, ans>();
11       ! run ()
12 end
```

By raising the event openSS<req,ans>(), a server announces its availability to the broker. This synchronous event returns whenever a request is made for some data on this server. Having provided the ports along the event, the server object expects to be connected to the requesting client, and reads the key to the

requested data from its req port. The server looks up the data corresponding to the *key* in the data-store using the find operation. The result is sent back on the ans port. The event *closeSS* announces the accomplishment of the transaction. Finally, the server prepares for a new session by calling the run method again.

4.2 Analysis of the Model

Creol programs and models can be *executed* using the rewriting logic of Maude [12]. Maude offers different modes of rewriting and additional capabilities for validation, e.g., a search command and the means for model checking. Credo offers techniques to analyze *parts* of the system in isolation; on the lowest level, to analyze the behavior of a single (active) object in isolation.

Credo offers techniques to analyze, in a black-box manner, the behavior of a component modeled in Creol, by interaction via message passing. This allows for the description and analysis of systems in a divide-and-conquer manner. Thus the developer has the choice of developing the system bottom-up or top-down.

Although Creol allows modeling systems on a high level, the complete model might still be too large to be analyzed or validated as a whole. By building upon the analysis of the individual components, compositional reasoning still allows us to validate the system.

Conformance Testing of the Model. In the context of the Creol concurrency model, especially the *asynchrony* poses a challenge for validation and testing. Following the black-box methodology, an abstract component *specification* is given in terms of its interaction with the environment. However, in a particular execution, the actual order of outputs issued from the component may not be preserved, due to the asynchronous nature of communication. To solve this problem, the conformance of the output to the specification is checked only up-to a notion of observability [17].

The existing Creol interpreter is combined with an interpreter for the abstract behavior specification language to obtain a *specification-driven interpreter for testing and validation* [17]. It allows for *run-time assertion checking* of the Creol-models, namely for compliance with the abstract specification.

We derive a specification for an object directly from the structural interfaces and the behavioral interfaces. The specification of the implementation of the ServerSide is derived from the facade depicted in Section 2.1 and the behavioral interface depicted in Section 2.2. The facade determines the direction of a communication, i.e., whether it is incoming or outgoing communication. For the specification the direction is inverted - the specification 'interacts' with the object to analyze it. The order of the events is determined by the behavioral interface.

The specification language features, among others, choice (between communication in the same direction, i.e., incoming only or outgoing only) and recursion. As an example, we give the specification of a server:

$$\begin{aligned}\varphi_S = {} & \langle \mathit{event}\ \mathit{register}(\mathit{keyList})\rangle? \ .\ \mathsf{rec}\ X\ .\ \langle \mathit{event}\ \mathit{openSS}()\rangle? \ . \\ & \langle \mathit{port}\ \mathit{s.sReq}(\mathit{key})\rangle! \ .\ \langle \mathit{port}\ \mathit{s.sAns}(\mathit{data})\rangle? \ . \\ & \langle \mathit{event}\ \mathit{closeSS}()\rangle? \ .\ X\end{aligned}$$

To test our executable model ServerImpl for conformance with respect to the behavioral interface description, we translate the specification to Creol and in the next step to Maude. The specification in Maude is executed together with the model. With the data-store at hand, we specify via the method parameters that the data delivered along the sAns port of the server is actually the data identified by the key. This needs to be done on the level of the Maude code.

The object is executed together with the specification in a special version of the Maude interpreter customized for the testing purpose. The programmer can track down the reason for a problem according to the Maude execution. This can be either a mistake in the executable model or a flaw in the behavioral model, i.e., the specification. The interpreter reports an error if unexpected behavior is observed, i.e., an unspecified communication from the object to the specification, or a deadlock occurs.

Simulation. The conformance testing introduced in the previous section is already a simulation of a part of the system, i.e., the object under test. We use a modified version of the above testing interpreter to eliminate the error reporting. Notice that the Maude interpreter of Creol is a set of rewrite rules which reduces the modification of the interpreter in this case to the deletion of the rules dealing with the error reporting.

Furthermore, we use the facades and behavioral interfaces of section 2 to derive a Creol skeleton of the network. By filling in the details of the network manager, we get a Creol model of the network. The model of the network and the models of the components together form a model of the entire system, which can be executed in Maude.

We use Maude to steer the execution of the model on different levels. We use the different built-in rewriting strategies to simulate different executions of the system. We use Maude's search command to search for a specific execution leading to a designated program state. And we use Maude's meta-level to control an execution by controlling the application of the rewrite rules.

To supplement the above simulation strategies, we use Maude's *model-checking* facilities. In general, the simulation is non-deterministic, which means, that only part of the specified behavior is covered. Therefore erroneous behavior might be missed. Maude's search facility allows us to explore the search space systematically. A general limitation of model checkers is the state space explosion, which makes larger systems unmanageable, when it comes to model checking. By analyzing *parts* of the system in isolation we reduce the state space explosion. Furthermore, Creol as a modeling language allows us to represent the system in a high-level, abstract manner, and concentrate on the crucial design-choices, which furthermore increases the chances of being able to model-check such a model. Since Maude is based on rewriting, dealing with the asynchronous nature of communication is natural: the asynchronicity is represented by trace–equivalence, which is directly represented as equivalence in the Maude rewriting system. This allows the execution engine to more efficiently represent the state space (by working on the normal forms instead of exploring all re-orderings one by one).

Conformance Testing of the Implementation. The testing process uses formal methods (e.g., automata and simulation of a model's formal program semantics) to provide the necessary links between behavioral interfaces, Creol models, and the actual implementation.

Behavioral interfaces provide *test scenarios*, patterns of interactions between the components. A test case created according to a test scenario represents a functional description, but does not guarantee a good coverage of the model. To optimize the coverage, *dynamic symbolic execution* is used to analyze execution paths through the Creol model to find representative test cases while avoiding redundancies in the test suite [18].

Once a test suite is created, the next step in testing is executing the tests on the implementation and reaching a test verdict to check the conformance between model and implementation. Testing a concurrent system involves validation of both functional and non–functional aspects. Functional aspects are covered by standard techniques like runtime assertions in the implementation and unit testing. To test the concurrency behavior of an implementation against its model we use the observation that typically the Creol model and the implementation share a common structure with regard to high-level structure and control flow. It is therefore reasonable to assume that, given equivalent stimuli (input data), they will behave in an equivalent way with regard to control flow.

We instrument the implementation to record *events* and use the instrumented implementation to record *traces* of observable events. Then we restrict the execution of the model to these traces. If the model can successfully play back the trace recorded from the implementation (and the implementation produces the correct result(s) without assertion failures), then the test case is successful. The Creol model is used as a test oracle for the execution of the test cases on the actual implementation [1].

4.3 Schedulability Analysis

In this section, we explain how to model the real-time aspects of the peer-to-peer system using timed automata and the Uppaal model checker [25]. An object or component is called schedulable if it can accomplish all its tasks in time, i.e., within their designated deadlines. We demonstrate the schedulability analysis process [14,20] on the network manager object in the peer-to-peer model, which is the most heavily loaded entity in this system.

In the real-time model of an object, we add explicit schedulers to object specifications. For schedulability analysis, the model of an object consists of three parts: the behavioral interface, the methods and the scheduler.

Behavioral interface. To analyze an object in isolation, we use the behavioral interface as an abstract model of the environment. Thus, it triggers the object methods. Fig. 13 shows the behavioral interface of the network manager augmented with real-time information. The automata in this figure are derived from the behavioral interface of Peer (see Section 2.2) by removing the port operations. To send messages, we use the invoke channel, with the syntax

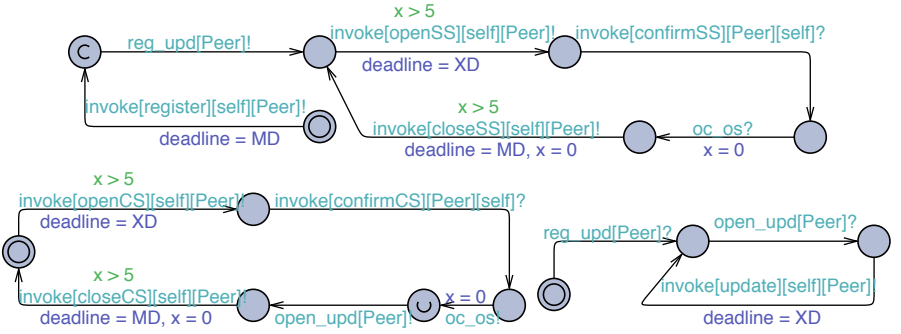


Fig. 13. The behavioral interface of broker modeled in timed automata

invoke[message][sender][receiver]! . To specify the deadlines associated to a message, we use the variable deadline.

In Fig. 13, we use the open_upd and reg_upd channels to synchronize the automata for Peer with ClientSide and ServerSide, respectively. Additionally, the automata for ClientSide and ServerSide are synchronized on the oc_os channel; this abstractly models the synchronization on port communication between the components in which the network manager is not directly involved. This model allows the client side of any peer to connect to the server side of any peer (abstracting from the details of matching the peers).

The confirmCS and confirmSS messages model the confirmation sent back from the network manager to the open session requests by the peers. In the implementation, this is an implicit reply which is therefore not modeled in the behavioral interfaces of the peers in Section 2.2. These edges synchronize with the method implementations (explained next) in order to reduce the nondeterminism in the model.

Methods. The methods also use the invoke channel for sending messages. Fig. 14 shows the automata implementation of two methods for handling the openCS and register events. In openCS, and similarly in every method, the keyword **caller** refers to the object/component that has called this method. The scheduler should be able to start each method and be notified when the method finishes, so that it can start the next method. To this end, method automata start with a synchronization on the start channel, and finish with a transition synchronizing on the finish

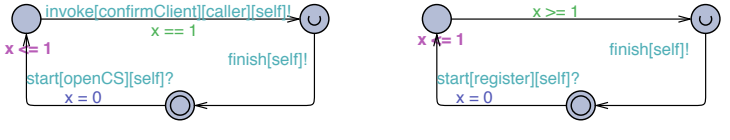


Fig. 14. Method automata for handling openCS and register events

channel leading back to the initial location. The implementation of the openCS method involves sending a message confirmCS back to the sender, while the register method is modeled merely as a time delay.

Checking Schedulability. When an object is instantiated, an off-the-shelf scheduler is selected and tailored to the particular needs of the object. For an object, we get a network of timed automata in UPPAAL by instantiating the automata templates for methods, behavioral interface and the scheduler. There are two conditions indicating that a system is not schedulable:

1. The scheduler receives a new message when the message queue is already full. In theory [20], a schedulable object needs a queue length of at most $\lceil d_{max}/b_{min} \rceil$, where d_{max} is the biggest deadline value used and b_{min} is the smallest execution time of all methods.
2. The deadline of at least one message in the queue is missed.

In either of the above cases, the scheduler automaton goes to a location called `Error`. This location has no outgoing transitions and therefore causes deadlock. Therefore, absence of deadlock implies schedulability, as well as correct output behavior for the object.

Due to the high amount of concurrency in the model, model checking is of limited use. Nevertheless, we can use the simulation feature of UPPAAL [29] to analyze bigger systems. We measure the worst-case response time for each message, which identifies a lower bound for the deadline value in a schedulable system.

5 Conclusions

We presented an extended version of the *Credo* methodology now covering also the detailed modeling and analysis of the network. The *Credo* modeling and analysis techniques addressing highly reconfigurable distributed systems presented cover a broader spectrum of the software development process. The *Vereofy* tool set is added to the picture providing modeling and analysis techniques for detailed network models.

At a high level of abstraction, the dynamic connections between the components are modeled using behavioral interface specifications. The detailed model of the network is given in terms of a Reo model specified in *Vereofy*. The detailed model of the components is given in terms of an object–oriented *Creol* model. Both models are used for analysis of functional as well as non-functional properties, e.g., schedulability, deadlock freedom. The conformance between the component and the network models is established via the behavioral interface specifications. Furthermore we can establish conformance between the *Creol* model and a given implementation by conformance testing.

The process described in this paper can be integrated in the existing software development methodologies which support component-based modeling, and thus enhance them with support for formal modeling and analysis of dynamically reconfigurable distributed systems. In the future, we intend to broaden the scope

of the *Credo* modeling language and its corresponding tool suite in order to support the full development life-cycle of large-scale, open systems. This involves, on one hand, integrating models of software architecture into the process; and on the other hand, working further on deployment concerns such as scheduling.

Case Studies

The Credo methodology has been successfully applied to two industrial case studies.

ASK System. The *Credo* methodology has been applied to model and analyze the ASK system, an industrial software system developed by Almende [2]. The purpose of the ASK system is to improve communication between people by providing a mediating communication platform with knowledge about the availability, schedules, skills and past experience of users. Typical applications for ASK are workforce planning, customer service, knowledge sharing, social care and emergency response. Various communication channels can be incorporated. The ASK system is a learning system, trying to improve the quality of service according to self–monitoring and feedback mechanisms. An important part of all core components of the system are thread pools. They are used to manage the (varying) workloads of the system by distribution of individual tasks, creation of new threads to handle tasks, and destruction of threads in case of low workload to minimize the idle time. We have modeled and analyzed the different kinds of thread pools in the ASK system according to the *Credo* methodology [15].

BSN. The *Credo* methodology has been applied to model and analyze a biomedical sensor network (BSN). For the BSN case study we modeled and analyzed different routing protocols for a biomedical sensor network. The BSN case study is focused on the application of the sensor network in a hospital. Patients are monitored via medical sensors which communicate their observations via radio signals to a sink, representing the entry point to the (wired) hospital communication network. The signals are not broadcasted directly to the sink but via other sensor nodes, used as hubs. Among functional properties, like emiting an emergency signal in certain scenarios, non–functional properties, like energy consumption are of interest. Two different routing protocols have been modeled, analyzed, and compared [26,27].

References

1. Aichernig, B., Griesmayer, A., Schlatte, R., Stam, A.: Modeling and testing multi-threaded asynchronous systems with Creol. In: Proc. TTSS 2008. ENTCS, vol. 243, pp. 3–14. Elsevier, Amsterdam (2009)
2. The Almende research company, `http://www.almende.com/`
3. Arbab, F.: Reo: A channel-based coordination model for component composition. Mathematical Structures in Computer Science 14, 329–366 (2004)

4. Arbab, F., Baier, C., Rutten, J.J., Sirjani, M.: Modeling component connectors in Reo by constraint automata. In: Proc. FOCLASA 2003. ENTCS, vol. 97, pp. 25–46. Elsevier, Amsterdam (2004)
5. Baier, C., Blechmann, T., Klein, J., Klüppelholz, S.: Formal Verification for Components and Connectors. In: de Boer, F.S., Bonsangue, M.M., Madelaine, E. (eds.) FMCO 2008. LNCS, vol. 5751, pp. 82–101. Springer, Heidelberg (2009)
6. Baier, C., Blechmann, T., Klein, J., Klüppelholz, S.: A Uniform Framework for Modeling and Verifying Components and Connectors. In: Field, J., Vasconcelos, V.T. (eds.) COORDINATION 2009. LNCS, vol. 5521, pp. 247–267. Springer, Heidelberg (2009)
7. Baier, C., Sirjani, M., Arbab, F., Rutten, J.J.M.M.: Modeling Component Connectors in Reo by Constraint Automata. In: Proceedings of the 2nd International Workshop on Foundations of Coordination Languages and Software Architectures. Science of Computer Programming, vol. 61, pp. 75–113 (2006)
8. Behrmann, G., David, A., Larsen, K.G., Håkansson, J., Pettersson, P., Yi, W., Hendriks, M.: Uppaal 4.0. In: QEST, pp. 125–126. IEEE Computer Society, Los Alamitos (2006)
9. Blechmann, T., Baier, C.: Checking equivalence for Reo networks. In: Electronic Notes in Theoretical Computer Science, vol. 215, pp. 209–226 (2008)
10. Blechmann, T., Klein, J., Klüppelholz, S.: Vereofy, `http://www.vereofy.de`
11. Blechmann, T., Klein, J., Klüppelholz, S.: Vereofy User Manual. TU Dresden (2008 –2009), `http://www.vereofy.de`
12. 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 (2001)
13. CWI Coordination Group. Eclipse coordination tools, `http://reo.project.cwi.nl/cgi-bin/trac.cgi/reo/wiki/Tools`
14. de Boer, F., Chothia, T., Jaghoori, M.M.: Modular schedulability analysis of concurrent objects in Creol. In: Arbab, F., Sirjani, M. (eds.) Fundamentals of Software Engineering. LNCS, vol. 5961, pp. 212–227. Springer, Heidelberg (2010)
15. 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: Breitman, K., Cavalcanti, A. (eds.) ICFEM 2009. LNCS, vol. 5885, pp. 367–386. Springer, Heidelberg (2009)
16. Grabe, I., Jaghoori, M.M., Aichernig, B., Baier, C., Blechmann, T., de Boer, F., Griesmayer, A., Johnsen, E.B., Klein, J., Klüppelholz, S., Kyas, M., Leister, W., Schlatte, R., Stam, A., Steffen, M., Tschirner, S., Liang, X., Yi, W.: Credo methodology. Modeling and analyzing a peer-to-peer system in Credo. In: Johnsen, E.B., Stolz, V. (eds.) Proceedings of the 3nd International Workshop on Harnessing Theories for Tool Support in Software (TTSS 2009), ICTAC 2009 satellite Workshop. Electronic Notes in Theoretical Computer Science. Elsevier, Amsterdam (2010)
17. Grabe, I., Steffen, M., Torjusen, A.B.: Executable Interface Specifications for Testing Asynchronous Creol Components. In: Arbab, F., Sirjani, M. (eds.) Fundamentals of Software Engineering. LNCS, vol. 5961, pp. 324–339. Springer, Heidelberg (2010)
18. Griesmayer, A., Aichernig, B.K., Johnsen, E.B., Schlatte, R.: Dynamic symbolic execution for testing distributed objects. In: Dubois, C. (ed.) Tests and Proofs. LNCS, vol. 5668, pp. 105–120. Springer, Heidelberg (2009)
19. Jaghoori, M.M.: Coordinating object oriented components using data-flow networks. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2007. LNCS, vol. 5382, pp. 280–311. Springer, Heidelberg (2008)

20. 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)
21. 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 Computer Society Press, Los Alamitos (2008)
22. Johnsen, E.B., Owe, O.: An asynchronous communication model for distributed concurrent objects. Software and Systems Modeling 6(1), 35–58 (2007)
23. Klüppelholz, S., Baier, C.: Alternating-time stream logic for multi-agent systems. Science of Computer Programming. Corrected Proof (2009) (in Press)
24. Klüppelholz, S., Baier, C.: Symbolic model checking for channel-based component connectors. Science of Computer Programming 74(9), 688–701 (2009)
25. Larsen, K.G., Pettersson, P., Yi, W.: UPPAAL in a nutshell. STTT 1(1-2), 134–152 (1997)
26. Leister, W., Björk, J., Schlatte, R., Griesmayer, A.: Validation of Creol models for routing algorithms in wireless sensor networks. Report 1024, Norsk Regnesentral, Oslo, Norway (2010)
27. Leister, W., Liang, X., Klüppelholz, S., Klein, J., Owe, O., Kazemeyni, F., Bjørk, J., Østvold, B.M.: Modelling of biomedical sensor networks using the Creol tools. Report 1022, Norsk Regnesentral, Oslo, Norway (2009)
28. Rumpe, B., Klein, C.: Automata describing object behavior. In: Object-Oriented Behavioral Specifications, pp. 265–286. Springer, Heidelberg (1996)
29. Tschirner, S., Xuedong, L., Yi, W.: Model-based validation of QoS properties of biomedical sensor networks. In: Proc. Embedded software (EMSOFT 2008), pp. 69–78. ACM Press, New York (2008)
30. Vereofy source code of the peer-to-peer example (2010), `http://www.vereofy.de/download/examples/vereofy_p2p_example.zip`
31. Yu, I.C., Johnsen, E.B., Owe, O.: Type-safe runtime class upgrades in Creol. In: Gorrieri, R., Wehrheim, H. (eds.) FMOODS 2006. LNCS, vol. 4037, pp. 202–217. Springer, Heidelberg (2006)