

Replace this file with `prentcsmacro.sty` for your meeting,  
or with `entcsmacro.sty` for your meeting. Both can be  
found at the [ENTCS Macro Home Page](#).

# Credo Methodology <sup>\*</sup>

## Modeling and Analyzing A Peer-to-Peer System in Credo

Immo Grabe<sup>1,9</sup>, Mohammad Mahdi Jaghoori<sup>1</sup>

Bernhard Aichernig<sup>5</sup>, Christel Baier<sup>3</sup>, Tobias Blechmann<sup>3</sup>, Frank de Boer<sup>1</sup>, Andreas Griesmayer<sup>5</sup>,  
Einar Broch Johnsen<sup>2</sup>, Joachim Klein<sup>3</sup>, Sascha Klüppelholz<sup>3</sup>, Marcel Kyas<sup>2</sup>, Wolfgang Leister<sup>8</sup>,  
Rudolf Schlatte<sup>5</sup>, Andries Stam<sup>6</sup>, Martin Steffen<sup>2</sup>, Simon Tschirner<sup>4</sup>, Liang Xuedong<sup>7</sup>, Wang Yi<sup>4</sup>

<sup>1</sup>*CWI, Amsterdam, The Netherlands* <sup>2</sup>*University of Oslo, Norway* <sup>3</sup>*TU Dresden, Germany*  
<sup>4</sup>*University of Uppsala, Sweden* <sup>5</sup>*UNU - IIST, Macau, China* <sup>6</sup>*Almende, The Netherlands*  
<sup>7</sup>*RRHF, Oslo, Norway* <sup>8</sup>*NR, Oslo, Norway* <sup>9</sup>*CAU Kiel, Germany*

---

### Abstract

*Credo* offers tools and techniques to model and analyze highly reconfigurable distributed systems. In this paper, we present an integrated methodology to use the *Credo* tool suite. In this methodology, we advertise the use of top-down design, component-based modeling and compositional analysis to address the complexity of highly reconfigurable distributed systems. As a running example, we model a peer-to-peer file-sharing system and show how and when to apply the different modeling and analysis techniques of *Credo*.

---

## 1 Introduction

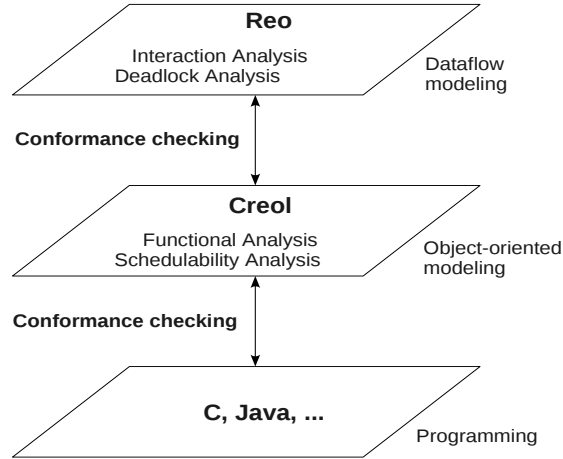
Current software development methodologies follow a component-based approach in modeling distributed systems. A major shortcoming of the existing methods is the lack of an integrated formalism to model highly reconfigurable distributed systems at different phases of design, i.e., systems that can be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

In this paper, we integrate the *Credo* tools and techniques into the software development life-cycle. We illustrate how and when they should be used during the design and analysis phases. Thus, software engineers can benefit by enriching their preferred methodology with the *Credo* tool suite.

The core of the *Credo* tool suite consists of two different *executable* modeling languages: Reo [1] is an executable *dataflow* language for high-level description of the dynamic reconfigurable *network* of connections between the components; Creol [2]

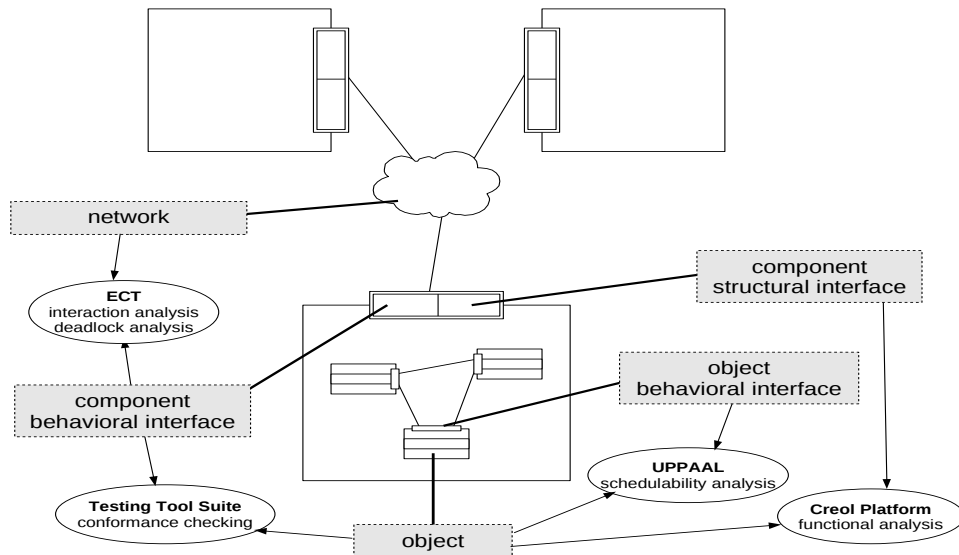
---

<sup>\*</sup> 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>)


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

is an *object-oriented* modeling language, used to provide an abstract but executable model of the implementation of the individual components. Fig. 1 illustrates the relation between these modeling languages and their relation to existing programming languages. It also indicates the kind of analysis the *Credo* tool suite provides for each modeling language.

To support top-down design and compositional analysis, we make use of *behavioral interfaces* for the different abstraction levels of the design (cf. Fig. 2). At the top-level, behavioral interfaces are used to describe the dataflow between the components of a system. These interfaces abstract from the details of the internal object-oriented model of components. Instead they describe the kind of connections components use to communicate and interact. *Credo* provides as an Eclipse plug-in an integrated tool-suite, ECT (Eclipse Coordination Tools) [3], to model and analyze the interactions between the components in a given network, e.g., absence of deadlock can be checked at this early stage of design.


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

The functional behavior of the objects within a component is modeled in Creol. The conformance between such a model of a component and its behavioral interface can be checked in *Credo* [4]. On the other hand, given an implementation of a component in a programming language like C, *Credo* also provides a technique to check for conformance between the implementation and the model [5,6]. Both techniques are based on *testing* and use the behavioral interface as an abstract model to generate test cases and to control the execution of the test cases.

Furthermore, the *Credo* tool suite offers an automated technique for *schedulability analysis* of individual objects [7]. We use the *timed automata* of UPPAAL to model objects and their behavioral interfaces. Given a specification of a scheduling policy (e.g., shortest 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.

We illustrate the *Credo* methodology by an example. 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 node. In Section 2, we develop the structural and behavioral interfaces for the components (nodes of the peer-to-peer system) and the network (the broker managing the dynamic connections between nodes); and prove our model of the network to be deadlock free. In Section 3, we give executable Creol models for the components and analyze them by means of simulation and testing for conformance both with respect to the behavioral interfaces and an implementation. We demonstrate schedulability analysis by analyzing the broker. Section 4 concludes the paper.

## 2 High-Level Dataflow Modeling

Reo [1] is a channel-based coordination model for component composition. As the formal semantics of Reo, we use constraint automata [8]. In Reo, a system consists of a set of components connected by a network. The network exogenously controls the data-flow between the components and may be dynamically reconfigured to connect different components. At this level of abstraction, only a *facade* is visible from each component. A facade consists of port and event declarations, and its abstract behavior is specified using constraint automata. In this paper, we do not go into the details of composing Reo channels for obtaining complex networks. Instead, we model the network behavior directly using constraint automata.

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

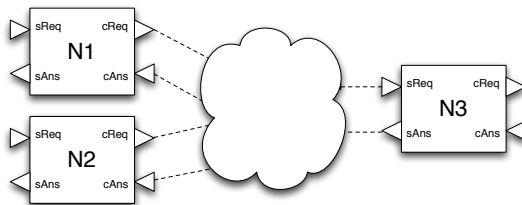


Fig. 3. Nodes in the peer-to-peer system

the network (as a cloud). Ports can be either inports or outports (implied by the direction of the triangles). By exogenous coordination, we mean that a component has no 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. The network includes a network manager that handles events and reconfigures the network according to the events.

In this section, we model the nodes of the peer-to-peer system as the components. The network consists of the broker that manages the connections between the component ports. Each node has two sides, a client side and a server side. On each side, a pair of request and answer ports is needed. As a client, a node writes its request to `cReq` and expects the result on `cAns`. A component on its server side reads a request (‘key’ to some data) from `sReq` and writes the data corresponding to the given key to `sAns`. For two nodes to communicate, the broker has to connect the corresponding ports of the client and 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.

```

1 facade ClientSide begin
2   port cReq : outport
3   port cAns : inport
4   sync_event openCS<req:outport , ans:inport >(in k:Data; out f:Bool)
5   sync_event closeCS<req:outport , ans:inport >()
6 end

1 facade ServerSide begin
2   port sReq : inport
3   port sAns : outport
4   sync_event openSS<req:inport , ans:outport >()
5   sync_event closeSS<req:inport , ans:outport >()
6   register<>(in keyList : List[Data]) // async_event
7 end

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

```

The network manager in a system 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 now safe to be disconnected. These events must provide the ports to be used in the session as parameters. In addition, they can have extra parameters, e.g., the ‘open client session’ event (written as `openCS`) guides the connection by providing the key it is looking for, and in return it is informed whether such a node is found.

Events are by default asynchronous. However, events expecting return values (e.g., open and close session events) should be declared to be synchronous (using the keyword `sync_event`). All events raised by the components in a system have to be handled by the network. This has to be reflected in the structural interface description of the network.

### 2.1.1 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 also 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 is originated using the keyword **with**.

```

1 networkmanager Broker begin
2   with ServerSide
3     register<>(in keyList : List[Data])
4     sync_event openSS<in req:inport ,ans:outport >()
5     sync_event closeSS<in req:inport ,ans:outport >()
6   with ClientSide
7     sync_event openCS<in req:outport ,ans:inport >(in k:Data; out f:Bool)
8     sync_event closeCS<in req:outport ,ans:inport >()
9   with Peer
10    update<>(in keyList : List[Data])
11 end

```

## 2.2 Behavioral Interface Description

The behavioral description for a component facade comprises of specifying the order of raising events and the port operations. This is modeled using constraint automata [8]. In these automata, we denote port operations by specifying the port names. The corresponding 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 surrounded by opening and closing session events in parts (a) and (b) of this figure. A server registers its data with the broker to initialize its operation. 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 somewhere in the system.

The `Peer` facade inherits the behavior specified for `ClientSide` and `ServerSide` facades. The `Peer` facade additionally introduces extra behavior involving an update to the data stored at the broker. This automaton synchronizes with the `ServerSide` facade to make sure that an update cannot take place before the data is initially registered. Moreover, the data at the broker should be updated after receiving new information (on the `ClientSide`). This is modeled by synchronizing on the read operation on `cAns`.

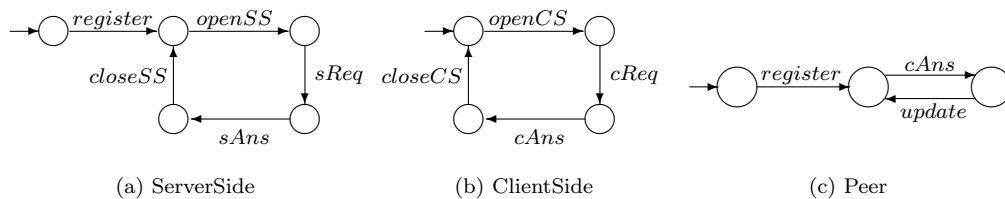


Fig. 4. Behavioral interfaces for facades

In general, the behavior of the sub-type has to be a refinement of the behavior of its super-type [9]. 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 [8] the transitions of different automata are interleaved while those with common action names are synchronized.

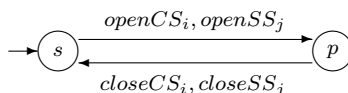
### 2.2.1 Network

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 ( $i \in \{1, \dots, n\}$ ), 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 naturally modeled in the following automata by a transition labeled with the participating actions.

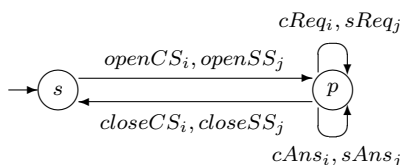
For clarity, we start by different automata for the synchronization of ports and events. Synchronization between the ports of a pair of components  $i$  and  $j$  is described by the following automaton.



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 that need to be made in a session between each pair of components (shown below). The *interleaving product* of these combined automata for all pairs of components results in an automaton describing the behavioral interface of the Broker.



It should be noticed that interleaving allows for components to be involved in more than one session at a time. The *synchronized product* of the Broker automata-

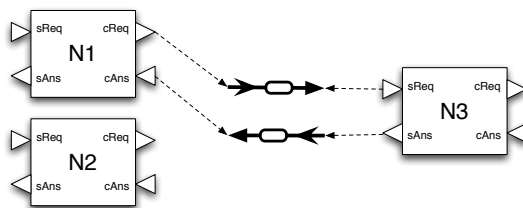
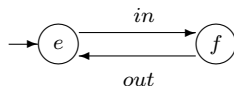


Fig. 5. Using Reo channels for modeling the network.

ton with the component automata (from the previous subsection) finally describes the overall behavior of the system. This product constrains the Broker so that components are involved in at most one session at a time. We can construct this overall behavior modeling the whole system and analyze it with the Vereofy tool [10,11], e.g., to ensure absence of deadlock. Furthermore, Vereofy includes symbolic model checking tools for linear-time, branching-time and alternating-time temporal logics with special operators to reason about the events and data flow at ports of components. Due to lack of space, we do not explain the details of such analyses.

**Channels.** We can further refine the network model by introducing *channels* (which are a specific kind of connectors) [1,12]. 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 can 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.

$$sAns_j, in \quad \begin{array}{c} \circlearrowleft \\ \circlearrowright \end{array} \quad p \quad \begin{array}{c} \circlearrowleft \\ \circlearrowright \end{array} \quad cAns_i, out$$

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 Object-Oriented Modeling

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 finally a Creol implementation of the components. By adding a Creol implementation for 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 can also use such a 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 to check for conformance between the Creol model and an implementation in an actual programming language. The coverage of these test scenarios is improved by symbolic execution of the Creol implementation.

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.

### 3.1 Executable Creol Model

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[13]. 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 [14], 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 first 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 `porta` 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 by defining the `run` method. It repeatedly reads data requests from its `req` port and reports the results on its `ans` port.

```

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 repeats the same process by calling the `run` method again.

### 3.2 Validation of the Model

Creol programs and models can be *executed* using the rewriting mechanism in Maude [14]. 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 also 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 both describing and analyzing 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.

#### 3.2.1 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* can be 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 [4].

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* [4]. It allows a form of *run-time assertion checking* of the Creol-models, namely for compliance with the abstract specification.

We can derive a specification for an object directly from the structural interfaces and the behavioral interfaces. The specification of the implementation of the `ServerSide` can be 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:

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

To test our executable model `ServerImpl` for conformance with respect to the behavioral interface description, we have to translate this specification to Creol and in the next step to Maude. The specification in Maude is executed together with the model. We can relate the inputs and outputs of the method calls. With the datastore at hand, we can specify via the method parameters that the data delivered along the `sAns` port of the server is actually the data identified by the key. Though 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.

### 3.2.2 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 can even use a modified version of the above testing interpreter to get rid of the error reporting. Please note that the Maude interpreter of Creol is a set of rewrite rules which reduces the modification of the interpreter in this case to deletion of the rules dealing with the error reporting.

Furthermore we can use the facades and behavioral interfaces of section 2 to derive a Creol skeleton of network. Filling in the details of the network manager we get a Creol model of the network. With the Creol models of the components at hand we get a Creol model of the entire system which can be executed in Maude.

We can use Maude to steer the execution of the model on different levels. We can use the different built-in rewriting strategies, use Maude’s search command to search for an execution leading to a designated program state, or use Maude’s meta-level to control the details of the execution by controlling the application of the rewrite rules.

With the above simulation strategies, we can 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, the user gets to see only part of the wished behavior, or, worse still, unwanted, erroneous behavior is missed. Using Maude’s search facility allows 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 to represent the system in a high-level, abstract manner, and concentrate on the crucial design-choices, which furthermore increases the chances to be able to model-check such as model. The use of the Maude-implementation based on rewriting theory finally helps in dealing with the asynchronous nature of communication: as mentioned, the asynchronicity is represented by some form of equivalence on the traces, which can directly be represented as equivalence in the Maude rewriter. 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).

### 3.2.3 Conformance Testing of the Implementation

We use a formal testing process 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 [5].

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 validating both functional and nonfunctional aspects. Functional aspects can be 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. To test this assumption, the implementation is instrumented to record *events* and use this instrumentation to record *traces* of observable events, then instrument the model to restrict its execution flow to the recorded trace. 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

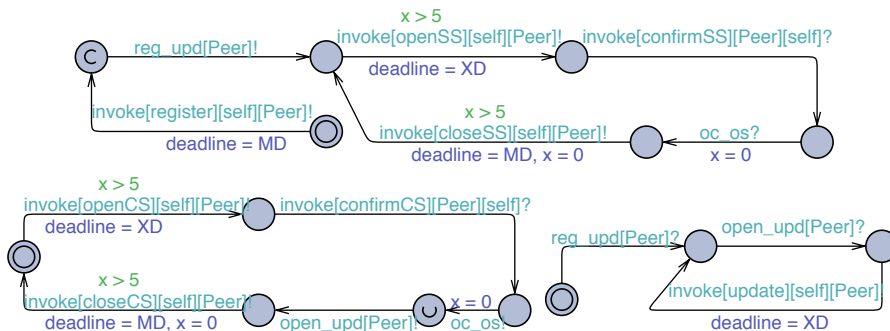


Fig. 6. The behavioral interface of broker modeled in timed automata

as test oracle for the execution of the test cases on the actual implementation [6].

### 3.3 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 [15]. An object or component is called schedulable if it can process all its tasks in time, i.e., within their designated deadlines. We demonstrate the schedulability analysis process [16,7] on the broker 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. 6 shows the behavioral interface of the broker augmented with real-time information. The automata in this figure are derived from the behavioral interface of Peer (in Section 2) by removing the port operations. To send messages, we use the `invoke` channel, with the syntax `invoke[message][sender][receiver]!`. For specifying the deadlines associated to a message, we use the variable `deadline`.

In Fig. 6, 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 broker is not directly involved. This model allows the server side of any peer to be able to match with the client side of any peer (abstracting from the details of matching the peers).

The `confirmCS` and `confirmSS` messages model the confirmation sent back from the broker to the open session requests by the peers. In the implementation, this will be an implicit reply which is therefore not modeled in the behavioral interfaces of the peers in Section 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. 7

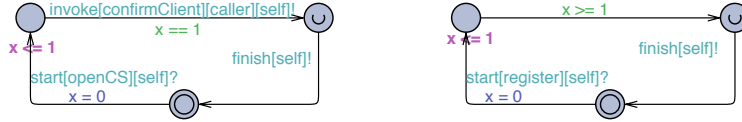


Fig. 7. Method automata for handling openCS and register events

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` 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.

### 3.3.1 Checking Schedulability

When a class is instantiated as an object, an off-the-shelf scheduler can be selected and (possibly) tailored to the particular needs of the object. For an object, one should make 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:

- (i) The scheduler receives a new message when the message queue is already full. In theory [7], 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.
- (ii) 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, a lack 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 [17] for analyzing bigger systems. In simulation, we can measure the worst-case response time for each message, which identifies a lower bound for the deadline value in a schedulable system.

## 4 Conclusions

The *Credo* project has been successful in developing modeling and analysis techniques addressing highly reconfigurable distributed systems. In this paper, we described when and how to use these tools and techniques at the design stage of a software development process. At a high level of abstraction, the dynamic connections between the components are modeled using data-flow networks and verified, e.g., for absence of deadlock. Then an abstract object-oriented model of the implementation is devised in Creol, which has an executable formal semantics. This

model can be used for further analysis of functional as well as non-functional properties, e.g., schedulability. The conformance between the object-oriented and dataflow models as well as the conformance between an implementation in a programming language and the Creol model is tested.

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 future, we intend to broaden the scope of the *Creol* 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.

## References

- [1] Arbab, F.: Reo: A channel-based coordination model for component composition. *Mathematical Structures in Computer Science* **14** (2004) 329–366
- [2] Johnsen, E.B., Owe, O.: An asynchronous communication model for distributed concurrent objects. *Software and Systems Modeling* **6**(1) (2007) 35–58
- [3] : Eclipse coordination tools. <http://reo.project.cwi.nl/cgi-bin/trac.cgi/reo/wiki/Tools>
- [4] Grabe, I., Steffen, M., Torjusen, A.B.: Executable interface specifications for testing asynchronous Creol components. In: *Proc. Fundamentals of Software Engineering (FSEN'09)*. LNCS, Springer-Verlag (2009) To appear.
- [5] Griesmayer, A., Aichernig, B., Johnsen, E., Schlatte, R.: Dynamic symbolic execution for testing distributed objects. In: *Proc. Tests and Proofs*. LNCS, Springer-Verlag (2009) to appear.
- [6] Aichernig, B., Griesmayer, A., Schlatte, R., Stam, A.: Modeling and testing multi-threaded asynchronous systems with Creol. In: *Proc. Harnessing Theories for Tool Support in Software (TTSS'08)*. ENTCS, Elsevier (2008)
- [7] Jaghoori, M.M., de Boer, F.S., Chothia, T., Sirjani, M.: Schedulability of asynchronous real-time concurrent objects. *J. Logic and Alg. Prog.* **78**(5) (2009) 402 – 416
- [8] Arbab, F., Baier, C., Rutten, J.J., Sirjani, M.: Modeling component connectors in Reo by constraint automata. In: *Proceedings of FOCLASA'03*. Volume 97 of ENTCS., Elsevier (2004) 25–46
- [9] Rumpe, B., Klein, C.: Automata describing object behavior. In: *Object-Oriented Behavioral Specifications*, Springer (1996) 265–286
- [10] Klüppelholz, S., Baier, C.: Symbolic model checking for channel-based component connectors. *Electr. Notes Theor. Comput. Sci.* **175**(2) (2007) 19–37
- [11] Blechmann, T., Klein, J., Klüppelholz, S.: Vereofy. <http://www.vereofy.de>
- [12] Jaghoori, M.M.: Coordinating object oriented components using data-flow networks. In: *Proc. Formal Methods for Components and Objects (FMCO'07)*. Volume 5382 of LNCS., Springer (2007) 280–311
- [13] Yu, I.C., Johnsen, E.B., Owe, O.: Type-safe runtime class upgrades in Creol. In Gorrieri, R., Wehrheim, H., eds.: *Proc. 8th International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'06)*. Volume 4037 of LNCS., Springer-Verlag (June 2006) 202–217
- [14] 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)
- [15] Larsen, K.G., Pettersson, P., Yi, W.: UPPAAL in a nutshell. *STTT* **1**(1-2) (1997) 134–152
- [16] de Boer, F., Chothia, T., Jaghoori, M.M.: Modular schedulability analysis of concurrent objects in Creol. In: *Proc. Fundamentals of Software Engineering (FSEN'09)*. LNCS, Springer (2009) To appear.
- [17] Tschirner, S., Xuedong, L., Yi, W.: Model-based validation of QoS properties of biomedical sensor networks. In: *Proc. Embedded software (EMSOFT '08)*, ACM (2008) 69–78