The unified coordination language UnCL

F.S. de Boer, J.V. Guillen-Scholten, J.F. Jacob

REPORT SEN-E0511 SEPTEMBER 2005

ISSN 1386-369X

The unified coordination language UnCL

ABSTRACT
In this paper we show how to use a (subset) of UML as an Unified Coordination Language (UnCL) that is based on a separation of concerns between coordination and computation. As such UnCL provides a general language for the coordination of, in particular, object-oriented applications. The basic idea of UnCL is to use UML as a formalism to specify the `glue code' in terms of state-machines which are added to the classes of the underlying applications. These state-machines describe the coordination of the objects of the underlying applications in terms of sending and receiving events. We introduce a formal semantics of UnCL and discuss its implementation using a new tool for the transformation of XML data which is based on a new Rule Markup Language (RML). Finally, we discuss the incorporation of a more high-level coordination mechanism called MoCha, an exogenous coordination framework for (distributed) communication and collaboration using mobile channels as its medium.

1998 ACM Computing Classification System: D.1.5;C.2.4;D.2.1;D.2.2;D.3.2;D.3.3
Keywords and Phrases: Coordination;UML;XML;XML transformations;Mobile Channels

The Unified Coordination Language UnCL

Frank de Boer, Juan Guillen-Scholten, and Joost Jacob [⋆]

Centrum voor Wiskunde en Informatica (CWI),
Amsterdam,
The Netherlands.
{frb, juan, jacob}@cwi.nl

Abstract. In this paper we show how to use a (subset) of UML as an *Unified Coordination Language* (UnCL) that is based on a separation of concerns between coordination and computation. As such UnCL provides a general language for the coordination of, in particular, object-oriented applications. The basic idea of UnCL is to use UML as a formalism to specify the 'glue code' in terms of state-machines which are added to the classes of the underlying applications. These state-machines describe the coordination of the objects of the underlying applications in terms of sending and receiving events.
We introduce a formal semantics of UnCL and discuss its implementation using a new tool for the transformation of XML data which is based on a new *Rule Markup Language* (RML). Finally, we discuss the incorporation of a more high-level coordination mechanism called *MoCha*, an exogenous coordination framework for (distributed) communication and collaboration using *mobile channels* as its medium.

1 Introduction

In this paper we introduce a subset of UML called the Unified Coordination Language (UnCL) as an unified model for exogenous coordination. The main purpose of UnCL is to provide a separation of concerns between coordination and computation. In Fig. 1 we give an overview of our coordination model. UnCL provides a special class for every UML class (or group of UML classes) that is relevant for the coordination. This UnCL class imports operations (methods) of the application's class(es). It contains only attributes with references to objects within UnCL or to objects of the application.

In UnCL the additional coordination behavior is specified by associating state-machines with its classes. Such a state-machine in UnCL consists of transitions that involve two kinds of operations. The first kind are the application operations (*app-ops*) that are implemented (and executed) by the coordinated application. The second kind are coordination operations (*co-ops*). Objects in UnCL coordinate their interaction by means of these operations which are communicated via an event-queue system. Every UnCL object is associated with a queue which stores the messages involving its co-ops and that are sent to it.

[⋆] Part of this research was funded by the IST project OMEGA IST-2001-33522, sponsored by the European Commission

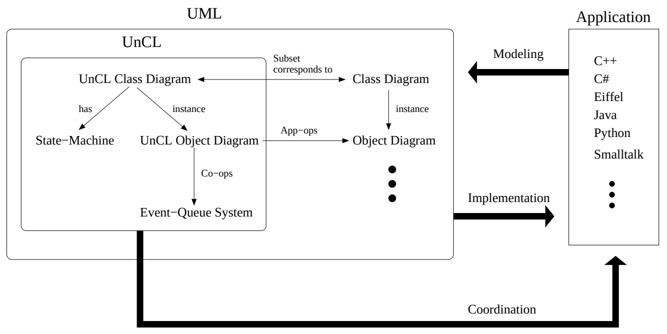

Fig. 1. UnCL.

There are three dynamic aspects to UML models: (1) *app-op* semantics, (2) *co-op* semantics, and (3) *scheduling*; namely, which transition is to be fired at a precise moment. The computations inside an application involve (1), and the coordination involves aspects (2) and (3). In this paper we abstract away from the scheduling because we want to provide a separation of concerns between (2) and (3). This gives us the possibility to choose different scheduling algorithms for the same UnCL model. Thus, the scheduling semantics becomes a parameter of the model itself.

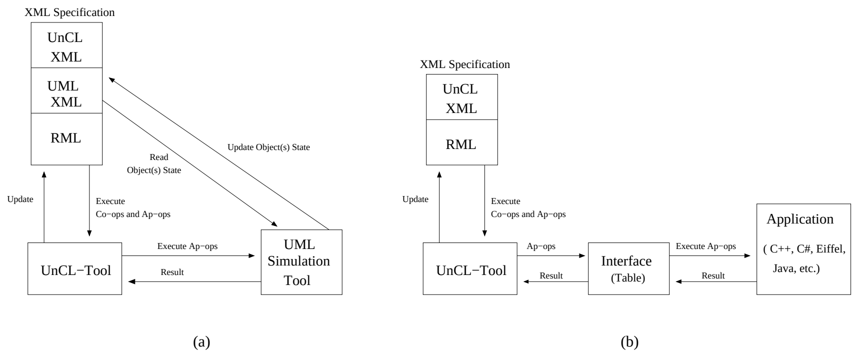

Fig. 2. UnCL Tool Architecture.

In this paper we introduce a precise semantics of the coordination behavior of an UnCL model and its formalization in a new extension of XML [6] called the *Rule Markup Language* (RML). RML is designed for the specification and execution of general transformations of XML data and is therefore very well suited for the specification and execution of the semantics of UML models. The application of RML to UnCL allows for both simulation within UML as well as the coordination of external applications at run-time.

In figure 2a we give the tool architecture for the simulation. The UnCL classes, state-machines, objects, and coordination mechanism are fully specified and given in XML. For every *co-op* a transformation rule in RML is given that describes how the UnCL XML specification of the input object-diagram changes by performing the operation. However, for executing an *app-op* the simulation needs a *UML simulation tool* that reads the relevant parts of the object-diagram, performs the operation, and changes the object diagram accordingly. The advantage of this architecture is that both the execution steps and the state of the UML application are fully given in XML, making it easy to see step-by-step how the state of the application evolves.

Using the same UnCL specification we can also coordinate run-time application(s). Such an application then basically serves as a kind of library whose objects are driven and coordinated by the UnCL model. Fig. 2b shows the corresponding coordination architecture. The attributes of UnCL classes now indirectly refer to run-time objects of the underlying application instead of UML objects specified by XML. The state of the application is hidden in the application itself and not part anymore of the XML specification. App-ops are now being performed by the run-time objects of the application. This means that we need an *interface* that binds the UnCL object references in XML to the run-time object references of the application. The interface maintains a table which relates these two different name spaces.

Plan of the paper: after describing UnCL in this section, we continue with the presentation of a precise semantics of UnCL. Then we discuss the execution platform and its main component RML. We finish with related work and conclusions.

2 Semantics of UnCL

In UnCL objects are coordinated by means of state machines. These state machines are associated with classes and consist of *transitions* of the form

$$l \xrightarrow{[g]t/a} l'$$

where l is the *entry* location and l' is the *exit* location of the transition. Furthermore, g denotes its boolean *guard*, t its *trigger*, and a its *action*.

More specifically, given a set of attributes, with typical element A, defined by the associated UnCL class the boolean guard g of a transition involves a call

$$A.op(A_1, \ldots, A_n)$$

to a boolean app-op op of the object denoted by A which is provided by the underlying application. We require that the execution of such a boolean app-op does not affect the values of the attributes defined by the UnCL diagram.

A trigger t is of the form

$$op(A_1, \ldots, A_n)$$

which specifies a co-op op defined by the UnCL class itself and a corresponding parameter list $A_1, \ldots, A_n$ of attributes.

Finally, an action involves a call

$$A.op(A_1, \ldots, A_n)$$

where op is either a co-op defined by the UnCL class of the object denoted by A or op is a app-op provided by its class of the underlying application.

We model object creation by means of a call $C.NEW(A)$ of an app-op NEW provided by the underlying application. This operation has a value/result parameter so the above call with actual parameter A will assign to A the identity of a new object in class C. In general we model assignments by means of value/result parameters, i.e., an assignment $B = A.op(A_1, \ldots, A_n)$ involving an operation-call is modeled by a call $A.op(B, A_1, \ldots, A_n)$ with value/result parameter B.

In order to formally define the *operational* semantics of state machines in UnCL we assume for each class c of a given UnCL class diagram a set O_c of *references* to objects in class c. In case class c extends c' (according to the UnCL diagram) we have that O_c is a subset of $O_{c'}$. (For classes which are not related by the inheritance hierarchy these sets are assumed to be disjoint.)

Definition 1. *An* object diagram *of a given UnCL class diagram with classes $c_1, \ldots, c_n$ can be specified mathematically by functions σ_c, for $c \in \{c_1, \ldots, c_n\}$, which specify for each object in class c existing in the object diagram the values of its attributes, i.e., $\sigma_c(o.A)$ denotes the value of attribute A of the object o, i.e., it denotes an object reference in $O_{c'}$, where c' is the (static) type of the attribute A (defined in the class c in the UnCL diagram).*

Often we omit the information about the class and write simply $\sigma(o.A)$. Control information of each object o in an object-diagram is given by $\sigma(o.L)$, assuming for each class an attribute L which is used to refer to the current location of the state machine of o. Furthermore, the event-queue of each object is given by the attribute E.

Given an UnCL class diagram consisting of a finite set of classes $c_1, \ldots, c_n$ and associated state machines, we define its behavior in terms of a *transition relation* on object diagrams. Object diagrams correspond to states in our semantic model. This transition relation is defined parametric in the semantics of the application operations and the way messages are stored and removed from the event-queue. More specifically, we assume for each action $a = A.op(A_1, \ldots, A_n)$ involving an app-op op a *labeled* transition relation

$$\sigma \xrightarrow{o.a} \sigma'$$

which specifies σ' as a possible result of the execution of the call a by the caller object o in σ. Such a labeled transition describes the *observable* effect on the

UnCL object diagram of the execution of the corresponding call by the underlying application. As a special case we assume for each *guard* $g = A.op(A_1, \ldots, A_n)$ involving a boolean app-op op a *labeled* transition relation

$$\sigma \xrightarrow{o.g} b$$

where b denotes a boolean value which indicates the result of the operation (note that we assume that boolean operations does not affect the attributes of the UnCL diagram).

Furthermore, for each trigger $op(A_1, \ldots, A_n)$ we assume the semantic function

$$pop - op(A_1, \ldots, A_n))$$

which, given an input object diagram σ and an executing object o, returns the object diagram σ' that results from *removing* a message $op(o_1, \ldots, o_n)$ from the event-queue $\sigma(o.E)$ of o in σ and *assigning* the object references o_i to $\sigma(o.A_i)$, $i = 1, \ldots, n$, i.e., $\sigma'(o.A_i) = o_i$. In case there does not exist such a message this function is undefined.

On the other hand, given an input object diagram σ and a caller object o, the semantic function

$$push - op(A, A_1, \ldots, A_n)$$

returns the object diagram σ' that results from adding the message $op(o_1, \ldots, o_n)$ involving the co-op op sent by o to the event-queue $\sigma(o'.E)$ of the callee $o' = \sigma(o.A)$, where $o_i = \sigma(o.A_i)$, for $i = 1, \ldots, n$.

Definition 2. *Formally, given an UnCL class-diagram and the semantic interpretations of the app-op's, we have a transition $\sigma \rightarrow \sigma'$ from the object-diagram σ to the object-diagram σ' if the following holds: there exists an object o and a transition*

$$l \xrightarrow{[g]t/a} l'$$

in its state machine such that

Location $\sigma(o.L) = l$ *and* $\sigma'(o.L) = l'$*;*
Guard $\sigma \xrightarrow{o.g}$ *true;*
Trigger $pop{-}op(A_1, \ldots, A_n)(\sigma, o) = \sigma''$, *in case of a trigger* $t = op(A_1, \ldots, A_n)$*;*
Action *We distinguish between the following two cases:*

- *in case of a call* $a = B.op(B_1, \ldots, B_k)$ *involving a co-op op we have*

$$push - op(B, B_1, \ldots, B_k)(\sigma'', o) = \sigma'$$

- *in case of a call a involving an app-op we have*

$$\sigma'' \xrightarrow{o.a} \sigma'.$$

The first clause above describes the flow of control. The second clause states that the guard evaluates to true (without side-effects). The third clause describes the execution of the trigger by the executing object o in the initial object diagram

σ in terms of the corresponding *pop-op* function. Note that the evaluation of the guard and the execution of the trigger are strictly sequentialized. This implies that the guard cannot refer to the new values of the actual parameters of the trigger which are stored in the event-queue. However, a slight modification would suffice to allow for this. For technical convenience only we restricted to a simpler semantic model. Finally, the execution of the action distinguishes between a co-op and an app-op. In both cases, the input diagram is the diagram resulting from the execution of the trigger and the diagram resulting from the execution of the action is the final result of the transition. A call to a co-op *op* is described in terms of the corresponding *push-op* which consists of pushing the message on the event-queue of the callee. Note that a call to a co-op is asynchronous and does not involve a rendez-vous with the callee. However such a synchronization can be modeled easily. Finally, a call to an app-op is described in terms of a corresponding labeled transition which models the execution of the call by the underlying application.

Note that the execution of a transition of a state-machine is atomic. However, more fine-grained modes of execution can be introduced in a straightforward manner.

3 The UnCL Execution Platform

The kernel of the UnCL tool consists of an algorithm for taking transitions in the state-machine, scheduling the transitions, calling the coordinated application and managing a user interface. The part of the algorithm that concerns the coordination, i.e., the processing of the co-op's, is defined using RML rules.

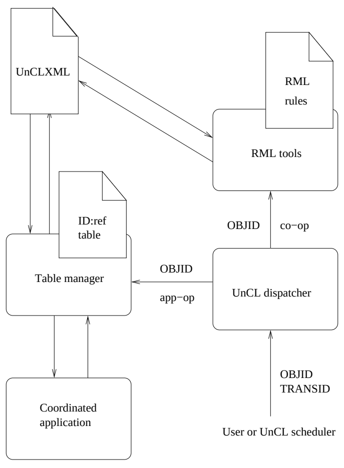

Fig. 3. RML in UnCL

Fig. 3 shows how the RML tools are combined with the UnCL tool, the XML for UnCL models in a new XML vocabulary called UnCLXML, and the coordinated application. A RelaxNG schema for UnCLXML is available at [15]. What is not displayed is a scheduler that decides what transitions to take at a specific moment, this scheduler can be implemented independently, useful for real-time applications, or the scheduler can be a human being using a UnCL system as the web-application at [15]. In the Figure we see a document icon for the UnCLXML document that is publicly visible. We see two other document icons for the RML rules, visible only to the RML tools part of UnCL, and the table relating object references in XML to the run-time object references of the application, visible only to the UnCL table manager that has to be implemented in the same language as the coordinated application. Both the table manager and the RML tools can modify the UnCLXML document, so the document has to be protected by a locking mechanism. The UnCL dispatcher receives an object identifier (OBJID) and a transition identifier (TRANSID) in the case of more than one possible transition for the object, and the dispatcher sends the necessary information to the RML tools in the case of a co-op and to the Table manager in the case of an app-op.

In the following we will show how state-machines and their semantics as defined in Sect. 2 can be encoded in the Rule Markup Language (RML) which is a new extension of XML for specifying and executing XML data. RML can be combined with an XMLvoc in order to define transformations of corresponding XML data using the XMLvoc itself. In the case of UnCL we combine RML with UnCLXML. With RML we can now define also the semantics of UML models in XML. Furthermore, using the RML tool we can *execute* these models. UnCL *users* do **not** have to learn RML, but just write state machines in UnCLXML.

The rules defined with RML consist of an *antecedent* (the input) and a *consequence* (the output). The antecedent and the consequence consist of XML+RML, where XML is the problem domain vocabulary and RML is used to specify *RML-variables* in the XML. The antecedent of a rule will be *matched* with input XML, resulting in a binding for all the RML variables to corresponding XML constructs in the input XML. These constructs can be XML element names, XML attribute names, XML attribute values, whole XML elements including their children, or sequences of XML elements with their children. If a match of the input XML with the antecedent of a rule is possible then there will be a specific XML element in the input XML that matches the antecedent, and this XML element will be *replaced* with the consequence of the rule. The consequence of a rule also contains XML+RML. The RML-variable names will be replaced with the corresponding contents of the RML variables in the output.

The table in Fig. 4 sums up all current RML elements *with a short description of their usage*. Due to a lack of space we have to refer to the RML tutorial at [14] for a longer description of all the RML elements. It is easy to think of many more useful elements in RML than appear in the table. Not everything imaginable is implemented because a design goal of RML is to keep it as concise and elegant as possible. Only constructs that have proven themselves useful in practice are added.

Elements that designate rules	
div	class="rule"
div	class="antecedent" context="yes"
div	class="consequence"

element	attribute	A	C	meaning
Elements that match elements or lists of elements				
rml-tree	name="X"	*		Bind 1 element (and children) at this position to RML variable X.
rml-list	name="X"	*		Bind a sequence of elements (and their children) to X.
rml-use	name="X"		*	Output the contents of the RML variable X at this position.
Matching element names or attribute values				
rml-X	...	*		Bind element name to RML variable X.
rml-X	...		*	Use variable X as element name.
...	...="rml-X"	*		Bind attribute value to X.
...	...="rml-X"		*	Use X as attribute value.
...	rml-others="X"	*		Bind *all* attributes that are not already bound to X.
...	rml-others="X"		*	Use X to output attributes.
...	rml-type="or"	*		If this element does not match, try the next element in the antecedent if that also has rml-type="or".
Elements that add constraints				
rml-if	child="X"	*		Match if X is already bound to 1 element, and occurs *somewhere* in the current sequence of elements.
rml-if	nochild="X"	*		Match if X does not occur in the current sequence.
rml-if	last="true"	*		Match if the younger sibling of this element is the last in the current sequence.
A * in the A column means the construct can appear in a rule antecedent. A * in the C column is for the consequence.				

Fig. 4. All the RML constructs

Variable binding of RML-variables during the matching of the antecedent of a rule is attempted in the order of the elements in the input XML tree. If an input XML tree contains more than one match for a variable then only the first match is used for a transformation. If you want to transform all matches then you will have to repeat applying the rule on the input.

Binding of RML-variables can also be done *before* a rule is applied if the RML-variables are supposed to contain string values; in that case the matching will only succeed if the supplied string values appear in the input XML in the position where the RML variable appears. An example of this pre-binding of variables in the UnCL tool is when the user supplies an object ID (variable IDOBJ in the examples that follow later) when the user wants that object to take a transition. To pre-bind a value of `id002` to RML variable `IDOBJ`, the user can supply an extra argument for the RML tools: `IDOBJ=id002`. Such pre-binding can also be done when using the RML libraries instead of the command-line tools.

The RML tutorial at [14] also describes a concise XML vocabulary for defining RML recipes, called Recipe RML (RRML). RRML is used to define *sequences* of, possibly iterated, transformations and has proven itself useful in alleviating the need for writing shell scripts or functions in a programming language containing sequences of calls to the RML tools. The idea is to avoid programming and to define as much as possible in XML in a data driven design.

Figure 5 shows a simple example RML rule. This is a rule that is used after a transition has been taken successfully by an object modeled with UnCL. With this rule the `location` attribute of the object is assigned the value of the `target` attribute. An example of the effect of the rule would be that

```
...
<obj id="id538" location="state_3" target="state_5" ... >
  <queue>
    ...
  </queue>
</obj>
...
```

is changed into

```
...
<obj id="id538" location="state_5" target="None" ... >
  <queue>
    ...
  </queue>
</obj>
...
```

for an object with identifier `id538`.

```
<div class="rule" name="set location">
  <div class="antecedent">
    <obj id="rml-IDOBJ" location="rml-L" target="rml-T"
         rml-others="rml-O" >
      <rml-list name="ObjChildren"/>
    </obj>
  </div>
  <div class="consequence">
    <obj id="rml-IDOBJ" location="rml-T" target="None"
         rml-others="rml-O">
      <rml-use name="ObjChildren"/>
    </obj>
  </div>
</div>
```

Fig. 5. The example RML rule

When applying this rule, the RML transformation tool first searches for an `obj` element in the input, corresponding with the `obj` element in the `antecedent` of the rule. These `obj` elements match if the `obj` in the input has an `id` attribute with the value bound to the RML `IDOBJ` variable mentioned in the antecedent, in the example this value is `id538` and it is bound to the RML variable `IDOBJ` *before* the rule is applied. This pre-binding of some of the variables is how UnCL can manage and schedule the execution of the RML transformation rules. If the `obj` elements match, then the other RML variables (`L`, `T`, `O` and `ObjChildren`) are filled with variables from the input `obj`. The `L`, `T` and `O` variables are bound to strings, the `ObjChildren` variable is bound to the children of the `obj` element: a list of elements and all their children. The `consequence` of the rule creates a new `obj` element, using the values bound to the RML variables, and *replaces* the `obj` element in the input with this new `obj` element.

Due to lack of space we restrict the description of the formalization in RML of the processing of the co-op's to the removal of a message from the event-queue, as shown in Fig. 6. The figure contains some lines with `...` in places where `rml-list` and `rml-use` constructs are used to preserve input context in the output. Here we see that in RML a pattern can be matched that is distributed over remote parts in the XML, the remoteness of the parts is why the rule has so many lines. In short, this rule looks for the name of the trigger that indicates the message that has to be removed from the event-queue, and then simply copies the event-queue without that event. But to find that name of the trigger, a search through the whole UnCLXML model has to take place, involving the following steps.

During application of this rule, the matching algorithm first tries to match the input with the antecedent of the rule, where IDOBJ and IDTRANS are pre-bound RML variables. With these pre-bound variables it can find the correct `obj`, then it finds the `ClassName` for that object. With the `ClassName` the `class` of the object can be found in the `classdiagram` in UnCLXML. When the class of the object is found, the transition in that class with id `TRANSID` can be found and in that `transition` element in the input we can finally find the desired `TriggerName`. The algorithm then looks for a message with name `TriggerName` in the event-queue of the `obj`, and binds all other events in the event-queue to RML variables `PreEvents` and `PostEvents`. In the consequence of the rule then, all these bound RML variables are available to produce a copy of the input, with the exception that the correct event is removed.

4 UnCL and Mobile Channels

In UnCL state machines model communication between objects in terms of the coordination operations which involve a simple event-queue mechanism. This provides a separation of concerns between the computational part specified by the application and the coordination part specified by UnCL. Due to this separation of concerns it is possible to replace the event-queue mechanism with any other coordination mechanism. Preferably, with one that preserves the separation of concerns and is easy to implement in concurrent and distributed systems. An example of such coordination mechanisms are shared data spaces like Linda [4]

```
<div class="rule">
  <div class="antecedent">
    <UnCL>
      <classdiagram>
        ...
        <class name="rml-ClassName">
          ...
          <statemachine>
            ...
            <transition id="rml-IDTRANS">
              ...
              <trigger>
                <op name="rml-TriggerName">
                  <rml-list name="Params"/>
                </op>
              </trigger>
              ...
            </transition>
            ...
          </statemachine>
        </class>
        ...
      </classdiagram>
      <objectdiagram>
        ...
        <obj class="rml-ClassName"
             id="rml-IDOBJ"
             rml-others="rml-OtherObjAttrs">
          ...
          <queue>
            <rml-list name="PreEvents"/>
            <op name="rml-TriggerName"/>
            <rml-list name="PostEvents"/>
          </queue>
        </obj>
        ...
      </objectdiagram>
    </UnCL>
  </div>
```

```
  <div class="consequence">
    <UnCL>
      <classdiagram>
        ...
        <class name="rml-ClassName">
          ...
          <statemachine>
            ...
            <transition id="rml-IDTRANS">
              ...
              <trigger>
                <op name="rml-TriggerName">
                  <rml-use name="Params"/>
                </op>
              </trigger>
              ...
            </transition>
            ...
          </statemachine>
        </class>
        ...
      </classdiagram>
      <objectdiagram>
        ...
        <obj class="rml-ClassName"
             id="rml-IDOBJ"
             rml-others="rml-OtherObjAttrs">
          ...
          <queue>
            <rml-use name="PreEvents"/>
            <rml-use name="PostEvents"/>
          </queue>
        </obj>
        ...
      </objectdiagram>
    </UnCL>
  </div>
</div>
```

Fig. 6. RML rule for removing an event from the event-queue

and JavaSpaces [7]. In this section we discuss the replacement of the event-queue by another coordination mechanism called *MoCha* [8, 9]. MoCha is an exogenous coordination framework for (distributed) communication and collaboration using *mobile channels* as its medium.

4.1 MoCha's Mobile Channels

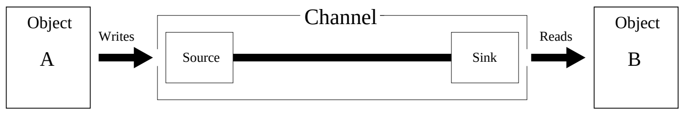

Fig. 7. General View of a Channel.

A channel in MoCha, see figure 7, consists of two distinct ends: usually (*source*, *sink*) for most common channel-types, but also (*source*, *source*) and (*sink*, *sink*) for special types. These channel-ends are available to the objects of an application. Objects can *write* by inserting values to the source-end, and *read* by removing values from the sink-end of a channel; the data-flow is locally *one way*: from an object into a channel or from a channel into an object.

Channels are *point-to-point*, they provide a directed virtual path between the (remote) objects involved in the connection. Therefore, using channels to express the communication carried out within an application is *architecturally very expressive*, because it is easy to see which objects (potentially) exchange data with each other. This makes it easier to apply tools for dependencies and data-flow analysis of an application.

Channels provide *anonymous connections*. This enables objects to exchange messages with other objects without having to know *where* in the network those other objects reside, *who* produces and consumes the exchanged messages, and *when* a particular message was produced or will be consumed. Since the objects do not know each other, it is easy to update or exchange any one of them without the knowledge of the object at the other side of the channel. This provides objects that are loosely coupled in space and time.

The ends of a channel are *mobile*. We introduce here two definitions of mobility: logical and physical. The first is defined as the property of passing on channel-end identities through channels themselves to other objects in the application; spreading the knowledge of channel-ends references by means of channels. The second is defined as physically moving a channel-end from one location to another location in a distributed system, where location is a *logical address space* where objects execute. Both kinds of mobility are supported by MoCha.

Because the communication via channels is also *anonymous*, when a channel-end moves, the object at the other side of the channel is not aware nor affected by this movement. Mobility allows dynamic reconfiguration of channel connections among the objects in an application, a property that is very useful and even crucial in systems where objects are mobile. An object is called mobile when, in a distributed system, it can move from one location (where its code is executing) to another.

Channels provide transparent *exogenous coordination*. Channels allow several different types of connections among objects without them knowing which channel types they are dealing with. Only the creator of the connection knows the type of the channel, which is either synchronous or asynchronous. This makes it possible to coordinate objects from the 'outside' (exogenous), and, thus, change the application's behavior without having to change the code of it's classes.

4.2 Channel Types

MoCha supports eleven types of channels. All with the same interface, but with different behavior. We give a short description of three major channel types. For more details and the remaining channel types we refer to the MoCha middleware manual [9].

– *Synchronous channel.* The I/O operations on the two ends are synchronized. A *write* on the source-end can succeed only when a *take* operation also atomically succeeds on the sink-end, and vice-versa. A *take* operation is the destructive version of the *read* operation.

– *Lossy synchronous channel.* If there is no I/O operation performed on the sink channel-end while writing a value to the source-end, then the *write* operation always succeeds but the value gets lost. In all other cases, the channel behaves like a normal synchronous type.
– *Asynchronous unbounded FIFO channel.* The I/O operations performed on the two channel-ends succeeds asynchronously. Values written into the source channel-end are stored in the channel in a FIFO distributed buffer until taken from the sink-end.

4.3 Implementation

The MoCha framework is implemented in the *Java* language using the *Remote Method Invocation package* (RMI). This MoCha middleware can be used for both distributed and non-distributed applications. The middleware has a clear and easy high-level application programming interface (API). Full API details can be found in [9].

4.4 UnCL and MoCha

Replacing the event-queues by MoCha channels requires the introduction of channel-ends in UnCL and the definition of their coordination operations in the state-machine semantics. Since channel-ends are also UML classes, we accomplish the first, by allowing the UnCL class attributes to also refer to these channel-end objects. The state-machine coordination operations are defined as:

- $T.new(L, E_1, E_2)$ creates a new channel, where $\{E_1, E_2\}$ are attributes storing the created channel-ends. T is an attribute that refers to the type of the channel, and L is an attribute that refers to a particular location. In the MoCha middleware such a creation is translated into the expression `chan = new MobileChannel(L,T)`, where `chan.E1` and `chan.E2` are the attributes that refer to the ends of the new created channel.
- $E.write(V)$ writes the reference value of attribute V to the source channel-end E.
- $E.take(V)$ takes a reference from the sink channel-end E and stores it in attribute V.
- $E.read(V)$ reads a reference from the sink channel-end E and stores it in attribute V. (read is the non-destructive version of *take*).
- $E.move(L)$ moves a channel-end E to location L.

Observe that, in cases where we are not concerned with modeling locations we can take the same first four operations and remove the location attribute L.

Using MoCha in UnCL has four major advantages. First, since the MoCha framework is implemented in the Java language, there is a straightforward implementation for every UnCL model. Straightforward in the sense that the MoCha middleware implements the same operations and channels as the ones of the

UnCL + MoCha model, providing a one-to-one relation between a UnCL channel and a MoCha middleware channel. Second, since MoCha supports distributed environments, every UnCL model automatically does as well. Third, the UnCL model now provides the means for more high-level exogenous coordination. In addition of changing the state-machines, with MoCha we can also change the application's behavior by simple choosing a different type of channel between objects. And finally four, MoCha enhances the, already present, separation of concerns between the computational part and the coordination part of an application.

5 Conclusions and Related Work

In this paper we presented an *Unified Coordination Language* (UnCL) that is based on a separation of concerns between coordination and computation. UnCL provides a general language for coordination given in UML that can be used both for simulation and coordination of an application at run-time. We discussed a precise semantics of UnCL state machines, the UnCL execution platform, and how to use an executable extension of XML specifications within this platform. Finally, we discussed the possibility of incorporating MoCha into UnCL.

UnCL relates to other coordination languages like *Linda* [4], *JavaSpaces* [7], and MANIFOLD [1]. For the majority of these models UML interfaces are made. However, as far as we know, UnCL is the first coordination language that fully integrates with UML. Besides modeling coordination a UnCL UML-specification can also coordinate an application in runtime. UnCL + MoCha relates to Reo[2], an exogenous coordination language where complex channel connections are compositionally build out of simpler ones.

Other related work on coordination modeling are SOCCA [5] and CSP-OZ [12]. SOCCA is an object-oriented specification language supporting the arbitrarily fine-grained synchronization of processes. Despite the fact that SOCCA is related to UML it is a separate language and not an extension like UnCL. CSP-OZ is an integrated formal method combining the process algebra CSP with the specification language Object-Z. It provides the means for putting special information (tags) in UML class diagrams. The full CSP-OZ specification is obtained after compiling these class diagrams, unlike UnCL where the specification is fully given in UML. Both CSP-OZ and UnCL + MoCha use channels as the coordination mechanisms. However, CSP-OZ channels are static while UnCL + MoCha channels are dynamic. This enables UnCL to specify dynamic reconfigurable coordination patterns.

In our approach we abstract away from a particular scheduling algorithm. This gives us the advantage to make such an algorithm a parameter of an UnCL model. This is different from other work like [3], [10], and [11] where scheduling is already integrated into the semantics, making it more difficult to change the already present scheduling algorithm (if desired).

Instead of using RML for the UnCL transformation rules we could have used other tools for XML transformations, like XSLT [13]. We chose RML because it was developed with more complex matching patterns in mind: The XML wild-cards defined with RML can be distributed over several places in the input. Such a distributed matching pattern is hard to define with XSLT, because XSLT templates are path oriented instead of pattern oriented.

We have successfully used the UnCL architecture in project OMEGA IST-2001-33522, sponsored by the European Commission, where we formalized the OMEGA subset of UML and will apply it to industrial case studies. A first test case is demonstrated on-line at [15].

References

1. F. Arbab, *Manifold Version 2: Language Reference Manual*, Technical Report, Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands, 1996.
2. F. Arbab, *Reo: A channel-based coordination model for component composition*, Mathematical Structures in Computer Science, Vol. 14, No. 3, pp. 329-366, June 2004.
3. M. von der Beeck, *Formalization of UML-Statecharts.* Proceedings of UML 2001, LNCS 2185, 406-421, Canada, 2001.
4. N. Carriero, D. Gelernter. *How to Write Parallel Programs: a First Course*, MIT press, 1990.
5. G. Engels and L. P.J. Groenewegen. *Specification of Coordinated Behavior by SOCCA.* In B.Warboys, editor, Proc. of the 3rd European Workshop on Software Process Technology (EWSPT '94), Berlin, Germany, February 1994. Springer, LNCS 772, Berlin 1994, pages 128-151.
6. *Extensible Markup Language (XML) 1.0 (Second Edition)*, W3C recommendation, October 2000. url: `http://www.w3c.org/XML/`
7. E. Freeman, S. Hupfer, and K. Arnold, *JavaSpaces TM Principles, Patterns, and Practice*, Chapter 1 of book, Addison-Wesley, September 1999.
8. J.V. Guillen-Scholten, F. Arbab, F.S. de Boer and M.M. Bonsangue, *A Channel-based Coordination Model for Components*, A. Brogi and J. Jacquet, editors, Proceedings of 1st International Workshop on Foundations of Coordination Languages and Software Architectures, ENTCS 68.3, Elsevier Science, 2002.
9. J.V. Guillen-Scholten, *MoCha, easyMoCha, chocoMoCha Electronic Manual beta version 0.96b*, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, May 2005.
10. D. Latella, I. Majzik, and M. Massink. *Towards a formal operational semantics of UML statechart diagrams.* In P. Ciancarini, A. Fantechi, and R. Gorrieri, editors, Proc. FMOODS'99, IFIP TC6/WG6.1, pages 331–347. Kluwer, 1999.
11. J. Lillius and I Paltor. *Formalising UML State Machines for Model Checking.* In R France and B. Rumpe, editors, Proc. UML'99, number 1723 in LNCS. Springer Verlag, Berlin, 1999.
12. M. Möller, E.-R. Olderog, H. Rasch, and H. Wehrheim. *Linking CSP-OZ with UML and Java: A Case Study.* In Integrated Formal Methods, number 2999 in Lecture Notes in Computer Science, pages 267-286. Springer-Verlag, March 2004.
13. *XSL Transformations (XSLT) Version 1.0*, W3C recommendation, Editor James Clark , 16 November 1999, url: `http://www.w3.org/TR/xslt`
14. RML web site. Url: `http://homepages.cwi.nl/∼jacob/uncl/uncl.rnc`
15. Coordinating UML with UnCL web site.
    Url: `http://homepages.cwi.nl/∼jacob/uncl/index.html`