# A Specification Language for Coordinated Objects

Gabriel Ciobanu
Romanian Academy
Institute of Computer Science
Iaşi, Romania
gabriel@iit.tuiasi.ro

Dorel Lucanu
"A.I. Cuza" University
Faculty of Computer Science
Iaşi, Romania
dlucanu@info.uaic.ro

## ABSTRACT
The paper presents a specification language of autonomous objects supervised by a coordinating process. The coordination is defined by means of an interaction wrapper. The coordination semantics is described in the terms of bisimulation relations. The properties of the coordinated objects are expressed as temporal formulas, and verified by specific model-checking algorithms. We use the alternating bit protocol to exemplify our specification language and its semantics.

This approach allows a clear separation of concerns: the same coordinating process can be used with different concurrent objects, and the same objects can be used with a different coordinator. Thus our specification language allows easy modifications and customization. The method is effective in assembling increasingly complex systems from components. Moreover, composing different coordinating processes can be done without changing the code of the coordinated objects. In this way, the difficult task of implementing the mechanism of coordination becomes substantially easier.

## Keywords
coordination, process algebra, classes, objects, bisimulation, temporal logic.

## 1. INTRODUCTION
Coordination of concurrent activities is an important goal of object-oriented concurrent programming languages, as well for component-based software community. As far as there was no support for components abstraction and high-level coordination, it is difficult to ignore the mismatch between conceptual designs and the implementation. Object-oriented languages offer little support for synchronization of concurrent objects. While in theory the set of provided constructs is sufficient to solve the coordination problems, in practice only good programmers are able to handle non trivial tasks. Another difficulty was given by the fact that a low-level approach does not allow the composition of different coordination policies without changing the implementation of the coordinated entities. The main problems are represented by no separation of concerns (expressing coordination abstraction is difficult because the code of coordination is strongly tied to the implementation of the coordinated objects), the absence of abstraction (no declarative means to specify coordination), the lack of compositionality and flexibility, and the difficulties for a programmer to implement the desired coordination.

As a possible solution, this paper introduces and studies a specification language where the components are described as objects, coordination is defined as a process, and their integration is given by a wrapper. Semantic integration of the coordinating process and coordinated entities is based on bisimulation. Coordinating process and coordinated components are rather independent. The explicit description of collaboration between components defines interaction policies and rely on the methods of the objects they coordinate. We use a wrapper which, together with the processes associated to objects provides an interface of the underlying component. The three languages corresponding to objects, coordinating process and wrapper reflect the intuitions and practices of software engineers. In order to support formal manipulation and reasoning, our implementation provides a semantics based on the models of class specification in hidden algebra, labelled transition systems represented as coalgebras, and some theoretical results expressing the coordination in terms of bisimulations and coalgebra homomorphisms. These formal aspects are not visible to the user; the implementation hides them, but ensures both a good matching between intuitions and practices of users, and a sound executable system.

## 2. CLASSES AND OBJECTS
In this section we present the specification of classes and their objects. We propose a specification language with syntax closer to that of object-oriented programming language and with semantics described in hidden algebra [9]. A *class specification* consists of specification of *attributes* and specification of operations. An operation specification includes the signature of the operation and its behavioural specification expressed in the terms of its parameters and attributes values before and after its execution. The grammar supplying the syntax for class specification is given in Figure 1. The decoration of the attributes names with the prime symbol `'` in a method specification is similar to that used in Z, and a decorated attribute name refers the value of the attribute after the execution of the method. A given set of primitive data types including `Bool`, `Int`, ... is assumed.

EXAMPLE 1. *Alternating Bit Protocol*
*The alternating bit protocol (ABP) is a communication protocol consisting of four components (see fig. 2): a sender, a receiver, and two communication channels. Here is a brief description of each component.*

$\langle class\_spec\rangle ::= \langle header\rangle\ \{\langle body\rangle\}$

$\langle header\rangle ::= \texttt{class}\langle class\_name\rangle\ |$

$\qquad \texttt{class}\langle class\_name\rangle\ \texttt{extends}\ \langle class\_list\rangle$

$\langle class\_list\rangle ::= \langle class\_name\rangle\ |\ \langle class\_name\rangle,\ \langle class\_list\rangle$

$\langle body\rangle ::= \langle att\_spec\_list\rangle_{\rm opt}\ \langle opn\_spec\_list\rangle_{\rm opt}$

$\langle att\_spec\_list\rangle ::= \langle att\_spec\rangle\ |\ \langle att\_spec\_list\rangle\ \langle att\_spec\rangle$

$\langle att\_spec\rangle ::= \langle type\rangle\ \langle att\_name;\rangle$

$\langle opn\_spec\_list\rangle ::= \langle opn\_spec\rangle\ |\ \langle opn\_spec\_list\rangle\ \langle opn\_spec\rangle$

$\langle opn\_spec\rangle ::= \langle type\rangle\ \langle opn\_name\rangle()\ \{\langle assert\_list_{\rm opt}\rangle\}\ |$

$\qquad \langle type\rangle\ \langle opn\_name\rangle(\langle param\_list\rangle)\ \{\langle assert\_list\rangle_{\rm opt}\}$

$\langle param\_list\rangle ::= \langle param\rangle\ |\ \langle param\rangle, \langle param\_list\rangle$

$\langle param\rangle ::= \langle type\rangle\ \langle param\_name\rangle$

$\langle assert\_list\rangle ::= \langle assert\rangle\ |\ \langle assert\rangle; \langle assert\_list\rangle$

$\langle assert\rangle ::=$ boolean expression over attributes names, parameters, and decorated attributes names

$\langle type\rangle ::= \langle class\_name\rangle\ |\ \texttt{Bool}\ |\ \texttt{Int}\ |\ \cdots$

**Figure 1: Class specification grammar**

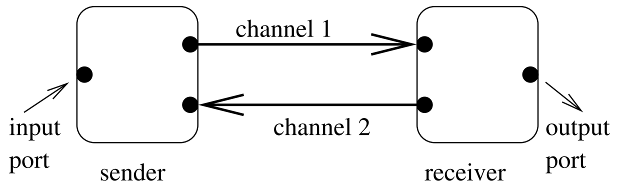


**Figure 2: Alternate Bit Protocol**

**Sender.** *The sender starts by reading a data element at the input port. Then a frame consisting of the read data element and a control bit (= a boolean value) is transmitted via channel 1 to the receiver, until a correct acknowledgment has arrived via channel 2. An acknowledgement is interpreted as being correct if the boolean value read via channel 2 is the same with the control bit. Each time a correct acknowledgement is arrived, the control bit change its value.*

**Channels.** *The channel 1 transports frames consisting a data element and a boolean value from the sender to the receiver and the channel 2 transports boolean values from the receiver to the sender. The problem with the two channels is that they are unreliable, that is the message could be damaged in transit. We assume that if something goes wrong with the message, the receiver/sender will detect this by computing a checksum. The channel are supposed to be fair in the sense that they will not produce an infinite consecutive sequence of erroneous outputs.*

**Receiver.** *The receiver starts by receiving a frame via channel 1. If the control bit of the frame is correct (i.e., different from the control bit of the receiver), then the data element is sent to the output port. Each time the frame received is not damaged (checksum is OK) and the control bit is correct, the receiver changes the value of its control bit.*

*We specify first an abstract class including only the common attributes of the sender and receiver:*

```
class AbsComp
{
  Bool bit;
  Data data;
  Bool ack;
}
```

*The class corresponding to the sender is derived from this abstract class, the class corresponding to the receiver being similar:*

```
class Sender extends AbsComp
{
  Bool chBit() {
    bit' = not bit;
    data' = data;
    ack' = ack;
  }
  void read() {
    bit' = bit;
    ack' = ack;
  }
  void sendFrame() {
    bit' = bit;
    data' = data;
    ack' = ack;
  }
  void recAck(Bool pack) {
    bit' = bit;
    data' = data;
    ack' = pack;
  }
}
```

*An assertion of the form "*$\texttt{bit}' = \texttt{not bit};$*" in* $\texttt{chBit}()$ *says that the value of the attribute* `bit` *is changed by the method. For each method, even if an attribute is not changed by its execution, this is explicitly specified. For instance, the method* `read` *is underspecified because we know nothing about the attribute* `data` *after its execution; it may have any* `Data` *value.*

Every object is an autonomous unit of execution which is either executing the sequential code of exactly one method, or passively maintaining its state. An *object instance* is a pair $(R \mid state)$, where $R$ is an object reference and *state* is an ordered sequence of pairs $(attribute, value)$. It is not necessary to have all attributes included in a particular object instance. We consider that an object instance is a canonical element of an behavioural equivalence class, where the behavioural operations are those included in the description of the instance. The result of an execution of a method $R.m(\mathbf{d})$ over a state $st$ consists of a new state $st'$ whose attributes values are computed according to the behavioural specification of $m$; we write $st' = R.m(\mathbf{d})(st)$. For instance, we have

$$\begin{aligned}\texttt{S.chBit}()((\texttt{bit},\texttt{true}),(\texttt{ack},\texttt{false}),(\texttt{data},d)) = \\ ((\texttt{bit},\texttt{false}),(\texttt{ack},\texttt{false}),(\texttt{data},d)).\end{aligned}$$

We suppose that an object reference uniquely determines the class it belongs to. A *configuration* is a commutative sequence of object instances such that an object reference occurs at most once in the sequence. We also consider a special configuration *err* for signaling the occurrence of an exception.

We consider a simple set of commands:

$$\begin{aligned}\langle cmd\rangle ::= R\ =\ \texttt{new}\ C(\mathbf{d})\ |\ \texttt{delete}\ R\ |\ R.m(\mathbf{d})\ | \\ R_1.m_1(\mathbf{d}_1)\|R_2.m_2(\mathbf{d}_2)\ \ |\ \ \langle cmd\rangle;\langle cmd\rangle\ | \\ \texttt{if}\ \langle bexpr\rangle\ \texttt{then}\ \langle cmd\rangle\ \texttt{else}\ \langle cmd\rangle\ |\ \texttt{throw error}()\end{aligned}$$

where $R, R_i$ range over object references, $m, m_i$ over methods, $\mathbf{d}, \mathbf{d}_i$ over data value sequences, and $C$ over class names. The metavariable $\langle bexpr\rangle$ denotes the boolean expressions. We omit here their formal definition; intuitively, a boolean expression is a propositional formula written in the terms of data values, attributes, and relational operators. A boolean expression $e$ is *satisfied* by a configuration *cnfg*, written $cnfg \models e$, if and only if the evaluation of the boolean expression $e$ in the configuration *cnfg* returns true. We also assume that the sequential composition ; is associative.

The *operational semantics* is given by the labelled transition system defined by following rules:

$cnfg \xrightarrow{R = \mathtt{new}\, C(d_1,\ldots,d_n)} cnfg, (R|(att_1,d_1),\ldots,(att_n,d_n));$

$cnfg, (R|state) \xrightarrow{\mathtt{delete}\ R} cnfg$
(we recall that a configuration is a commutative sequence);

$cnfg \xrightarrow{R.m(\mathbf{d})} cnfg'$ if and only if $cnfg'$ is obtained from *cnfg* by replacing the object instance $(R \mid state)$ with $(R \mid state')$, where $state' = R.m(\mathbf{d})(state)$;

$cnfg \xrightarrow{R_1.m_1(\mathbf{d}_1)\|R_2.m_2(\mathbf{d}_2)} cnfg_2$ if and only if $R_1 \neq R_2$, $cnfg'$ is obtained from *cnfg* by replacing the object instance $(R_i \mid state_i)$ with $(R_i \mid state'_i)$, where $state'_i = R_i.m_i(\mathbf{d}_i)(state_i)$, $i = 1,2$;

$cnfg \xrightarrow{cmd_1;cmd_2} cnfg''$ if and only if there is $cnfg'$ such that
$$cnfg \xrightarrow{cmd_1} cnfg' \text{ and } cnfg' \xrightarrow{cmd_2} cnfg'';$$

if $cnfg \xrightarrow{cmd_1} cnfg'$ and $cnfg \models e$, then
$$cnfg \xrightarrow{\mathtt{if}\ e\ \mathtt{then}\ cmd_1\ \mathtt{else}\ cmd_2} cnfg';$$

if $cnfg \xrightarrow{cmd_2} cnfg'$ and $cnfg \not\models e$, then
$$cnfg \xrightarrow{\mathtt{if}\ e\ \mathtt{then}\ cmd_1\ \mathtt{else}\ cmd_2} cnfg';$$

$cnfg \xrightarrow{\mathtt{throw\ error()}} err.$

# 3. COORDINATION

We introduce a coordinating process providing a high-level description of the interaction between objects. Its syntax is inspired by process algebras as CCS and $\pi$-calculus [14]. Interaction with the environment is given by some global actions, and interaction between components is given by a nondeterministic matching between complementary local actions. Each process is described by a set of equations. In some sense, we can think such a description as an abstract system interface. The computational world of our coordinator contains processes and messages. Local actions represents interaction channels. In this way, the coordinating process models a network in which messages are sent from one object to another. This formalism is not suitable to describe state changes (the state changes are described by objects).

The process expressions $E$ are defined by guarded processes, nondeterministic choice $E_1 + E_2$, and parallel composition $E_1 \mid E_2$. We have also an empty process 0. Guarded processes are presented by either a global action followed by a process expression, an object guard followed by a process expression, or by a local action followed by a process expression. The first case describes a global action involving a state change execution of an object. The second case describes a link between the actions of an object over a certain guard, followed or not by a process expression depending on the truth evaluation of the guard. Finally, each local action *act* involves automatically the existence of its complementary local action denoted by $\tilde{}act$; these two complementary local actions establish a synchronization and a communication between objects.

A process is described as a sequence of declarations (global actions, local actions, processes and guards) followed by a set of equations. The syntax grammar for processes is:

`proc` $\langle proc\_spec\_name\rangle$
{
  `global actions :` $\langle lact\_list\rangle$;
  `local actions :` $\langle gact\_list\rangle$;
  `processes :` $\langle proc\_id\_list\rangle$;
  `guards :` $\langle guard\_id\_list\rangle$;
  `equations :`
    $\langle eqn\_list\rangle$
}

where

$\langle lact\_list\rangle ::= \langle label\_list\rangle$
$\langle gact\_list\rangle ::= \langle label\_list\rangle$
$\langle label\_list\rangle ::= \langle label\rangle \mid \langle label\rangle,\ \langle label\_list\rangle$
$\langle label\rangle ::= \langle identifier\rangle \mid \tilde{}\langle identifier\rangle$
$\langle proc\_id\_list\rangle ::= \langle id\_list\rangle$
$\langle guard\_id\_list\rangle ::= \langle id\_list\rangle$
$\langle id\_list\rangle ::= \langle identifier\rangle \mid \langle identifier\rangle,\ \langle id\_list\rangle$
$\langle eqn\_list\rangle ::= \langle eqn\rangle \mid \langle eqn\rangle;\ \langle eqn\_list\rangle$
$\langle eqn\rangle ::= \langle proc\_id\rangle = \langle pexpr\rangle;$
$\langle pexpr\rangle ::= 0 \mid \langle label\rangle.\langle pexpr\rangle \mid [\langle guard\_id\rangle]\langle pexpr\rangle \mid$
$[\mathtt{not}\ \langle guard\_id\rangle]\langle pexpr\rangle \mid \langle pexpr\rangle + \langle pexpr\rangle \mid$
$\langle pexpr\rangle|\langle pexpr\rangle$

The metavariable $\langle proc\_id\rangle$ denotes the identifiers occurring in processes list, and $\langle guard\_id\rangle$ denotes the identifiers occurring in guards list.

A coordinating process specification is finally given by equations of parametric process expressions. For example, the specification of ABP communication protocol as a coordination between a Sender and a Receiver can be described in the following way:

```
proc ABP
{
  global actions: in, out, alterS, alterR;
  local actions: ch1, ch2;
  processes: A, A', V, B, B', T;
  guards: sok, rok;
  equations:
    A  = in.A';
    A' = ~ch1.ch2.V;
    V  = [sok]alterS.A + [not sok]A';
    B  = ch1.T;
    T  = [rok]B' + [not rok]out.alterR.B;
    B' = ~ch2.B;
}
```

The structural operational semantics of a coordinating process specification is given by a labelled transition system. The semantic rules are presented in Figure 3. In these rules, $\gamma$ is a function mapping each guard_id into a boolean value, *gact* ranges over the labels occurring in the global actions list, *lact* ranges over the labels occurring in the local actions list, and *act* can be both *gact* or $\tau(lact)$. Based on these rules, the operational dynamics of the previous ABP

$$\frac{}{gact.E \xrightarrow{gact} E} \qquad \frac{E \xrightarrow{act} E'}{E+F \xrightarrow{act} E'}$$

$$\frac{E \xrightarrow{act} E'}{E|F \xrightarrow{act} E'|F} \qquad \frac{E_A \xrightarrow{act} E',\ A = E_A}{A \xrightarrow{act} E'}$$

$$\frac{E \xrightarrow{act} E',\ \gamma(guard\_id) = true}{[guard\_id]E \xrightarrow{act} E'}$$

$$\frac{}{\tilde{\ }lact.E \ | \ lact.E' \xrightarrow{\tau(lact)} E|E'}$$

**Figure 3: The coordinating process operational rules**

process with reliable communication, modelled by $\gamma(sok) = true$ and $\gamma(rok) = false$:

$$A \mid B \xrightarrow{in} A' \mid B \xrightarrow{\tau(ch1)} ch2.V \ \mid \ T \xrightarrow{out}$$

$$ch2.V \ \mid \ alterR.B' \xrightarrow{alterR} ch2.V \mid B' \xrightarrow{\tau(ch2)}$$

$$V \ \mid \ B \xrightarrow{alterS} A \mid B.$$

We use the notation $\tau(lact)$ for the interaction between two processes prefixed by local actions $lact$ and $\tilde{\ }lact$, respectively. Interaction is therefore provided by pairs of actions $lact$ and $\tilde{\ }lact$ corresponding to some methods by means of an interaction wrapper.

# 4. INTERACTION WRAPPER

If we consider the coordinating process as an abstract interface of the system, then an interaction wrapper describes an implementation of this interface by means of a collection of objects. The coordinating process gives some directives, and the coordinated objects interpret these directives by using an interaction wrapper providing the appropriate link between the high level coordinating process and the lower level executing objects. This is the way we get a desirable separation of concerns, ensuring a suitable abstract level for designing large component-based systems without losing the details of low-level implementation of components. In some sense, our specification language has similarities with a symphonic orchestra, where independent players are synchronized by a conductor. The concert sheet followed by the conductor represent a high-level approach of the concert, and the instrumental sheet of the orchestra players are usually larger, containing more details. The link between the players and the coordinating conductor is given by certain entry moments and orchestral scores. The wrapper provides the players, and the entry scores implementing the desired resulting music at a certain moment. Therefore the wrapper provides the objects, and the necessary information for their executions in order to realize a coordinated interaction.

The syntax for the interaction wrappers is given by the grammar presented in Figure 4.

EXAMPLE 2. *The wrapper for previous described protocol ABP instruct a Sender* S *and a Receiver* R *in order to correctly follow the directives of the protocol:*

$\langle wrap\_spec \rangle$ ::= $\langle wrap\_name \rangle (\langle wparam\_list \rangle)$
  `implementing` $\langle proc\_spec\_name \rangle$
  $\{\langle amap\_list \rangle\ \langle gmap\_list \rangle\}$
$\langle wparam\_list \rangle$ ::= $\langle wparam \rangle \mid \langle wparam\_list \rangle;\ \langle wparam \rangle$
$\langle wparam \rangle$ ::= $\langle class\_name \rangle\ \langle object\_ref \rangle$
$\langle amap\_list \rangle$ ::= $\langle amap \rangle \mid \langle amap\_list \rangle\ \langle amap \rangle$
$\langle amap \rangle$ ::= $\langle action\_name \rangle$ -> $\langle cmd \rangle$;
$\langle gmap\_list \rangle$ ::= $\langle gmap \rangle \mid \langle gmap\_list \rangle\ \langle gmap \rangle$
$\langle gmap \rangle$ ::= $\langle guard\_name \rangle$ -> $\langle bexpr \rangle$;

**Figure 4: Wrapper syntax grammar**

```
wrapper w(Sender S, Receiver R) implementing ABP
{
  in -> S.read();
  alterS -> S.chBit();
  alterR -> R.chAck();
  tau(ch1) ->
    R.recFrame(S.data, S.bit) ||
    S.sendFrame();
  tau(ch2) ->
    S.recAck(R.ack()) || R.sendAck();
  out -> R.write();
  sok -> S.bit == S.ack;
  rok -> R.bit =/= R.ack;
}
```

*A directive* `in` *received from the coordinating process is translated into an execution of method* `read` *by* S. *The directives* `alterS` *and* `alterR` *are translated into executions of methods* `chBit` *and* `chAck` *by* S *and* R*, respectively. Whenever a* $\tau(ch1)$ *directive is possible at the level of the coordinating process, it is translated into a synchronization of the methods* `sendFrame` *of* S *and* `recFrame` *of* R*. This synchronization of the autonomous objects is accompanied by a communication between them; this is given by the fact that the arguments of the receiver method* `recFrame` *are attributes of the sender. A similar translation is done for* $\tau(ch2)$*. Finally, the last two lines of the interaction wrapper for ABP emphasize a nice feature related to the concerns separation. Instead of using a matching or a mismatching process algebra to compare the sending bit and the received acknowledge, we clearly separate the computational and coordinating aspects by moving the comparisons at the object level, followed by a true/false result to the coordination process.*

If $act$ is an action name, and $\mathbf{R}$ a sequence of object references, then $w(\mathbf{R})(act)$ denotes the command associated to $act$ by the particular wrapper $w(\mathbf{R})$. The *operational semantics* of such a wrapper $w(\mathbf{R})$ is given by the labelled transition system of the objects configurations, namely

$$cnfg \xrightarrow{act} cnfg' \text{ iff } cnfg \xrightarrow{w(\mathbf{R})(act)} cnfg',$$

where $cnfg$ and $cnfg'$ are configurations including the instances of the objects referred by $\mathbf{R}$ and related by the command corresponding to the action name $act$ via the interaction wrapper $w$.

The definition of the interaction wrapper (and its strong relationship to the dynamics of the involved objects) allows us to define an *integrated semantics* in a nice and advanced way. Taking in consideration that each configuration is supervised by a coordinating process, the whole coordination activity is described as follows:

from the current configuration $cnfg$ supervised by the process $P$, the transition $cnfg \xrightarrow{w(\mathbf{R})(act)} cnfg'$ is valid if and only if there is a process $P'$ such that $P \xrightarrow{act} P'$ and $P'$ supervises $cnfg'$.

In other words, the supervision relation is a bisimulation between the labelled transition system defined by the wrapper, and the labelled transition system defined by the coordinating process specification. This is formally defined in Section 6.

# 5. IMPLEMENTATION

We use hidden algebra [8] and Maude [5] for obtaining executable specifications for classes and their objects. In hidden algebra, an object is specified by:

1. a set $V$ of *visible sorts* with a standard interpretation; the meaning of a visible sort $v$ is a given set $D_v$ of data values;
2. a hidden sort $\mathsf{St}$ called *state sort*;
3. a set $\Sigma$ of operations including:
   (a) constants $init \in \mathsf{St}$, or generalized constants $init : v_1 \cdots v_n \to \mathsf{St}$ indicating initial states,
   (b) *methods* $g : \mathsf{St}\, v_1 \cdots v_n \to \mathsf{St}$ with $v_1, \cdots, v_n \in V$,
   (c) *attributes* $q : \mathsf{St}\, v_1 \cdots v_n \to v$ with $v, v_1, \cdots, v_n \in V$.

The properties of methods and attributes are described by equations. The main feature of hidden algebra is given by *behavioural equivalence* which abstractly characterizes an object state. Two states are behavioural equivalent if and only if they cannot be distinguished by experiments. An experiment is represented by a $\Gamma$-term with the result sort visible and with a place-holder for the state, where $\Gamma$ is a subsignature of $\Sigma$ including the *behavioural operations*. In other words, the result of an experiment is given by an attribute value after the execution of a sequence of methods over the current configuration; all the methods and attributes are in $\Gamma$. If the behavioural equivalence can be decided using only attributes, then an abstract state is characterized by its attributes values.

Each class is implemented in Maude by a module defining a hidden sort for the state of the objects, together with attributes and methods of the class. Here is the module for the sender:

```
fmod SENDER is
  sort Sender .
  inc ABP-DATA .
  op bit : Sender -> Bit .
  op data : Sender -> Data .
  op ack : Sender -> Bit .
  op send : Sender -> Sender .
  op chBit : Sender -> Sender .
  op read : Sender -> Sender .
  op recAck : Sender Bit -> Sender .
  *** equations defining properties
  *** of the methods
endfm
```

The sort `Sender` is hidden and it is used to describe the instances of the class. The sorts `Bit` and `Data` are visible and they models the data values.

A configuration of objects is a set of pairs (object reference, object state). We use the sort `ObjectReference` for object references, and the sort `ObjectState` for object states. For each class we add a distinguished subsort of `ObjectReference` and a distinguished subsort of `ObjectState`.

```
fmod CONFIG is
  sorts ObjectReference ObjectState EmptyConfig Config .
  subsort EmptyConfig < Config .
  op empty : -> EmptyConfig .
  op `(_|_`) : ObjectRef ObjectState -> Config .
  op _`,_ : Config Config -> Config
          [assoc comm id: empty] .
  op _.read$`(`)_ : ObjectRef Config -> ObjectState .
  op _.update$`(_`)_ :
     ObjectRef ObjectState Config -> Config .
  *** equations defining properties of read$()_
  *** and update$(_)_
endfm
```

The composition _,_ of two configurations is specified as being commutative, associative, and having the identity `empty`.

For ABP we have a configuration containing objects of classes Sender, Receiver and Channel. The hidden sorts defined in `SENDER`, `RECEIVER`, and `CHANNEL` are the state sorts and we make them subsorts of the ObjectState sort. We also add three distinguished sorts for the references to these objects. The ABP configurations expose the methods and attributes of three classes using attributes and methods with similar names and arguments. We show here only the `bit` attribute and the `recFrame` method.

```
fmod ABP-CONFIG is
  inc CONFIG + SENDER + RECEIVER + CHANNEL .
  subsort Sender < ObjectState .
  subsort Receiver < ObjectState .
  subsort Channel < ObjectState .
  sort ObjectRef<SENDER> .
  subsort ObjectRef<SENDER> < ObjectRef .
  sort ObjectRef<RECEIVER> .
  subsort ObjectRef<RECEIVER> < ObjectRef .
  sort ObjectRef<CHANNEL> .
  subsort ObjectRef<CHANNEL> < ObjectRef .

  op _.bit`(`)_ : ObjectRef<SENDER> Config -> Bool .
  eq S .bit() C = bit(S .read$() C) .

  op _.recFrame`(_`,_`)_ :
    ObjectRef<RECEIVER> Data Bit Config -> Config .
  eq S .recFrame(D, B) C = recFrame(S .read$() C, D, B) .

  *** other methods and attributes,
  *** and their equations
endfm
```

The initial ABP configuration contains a `Sender` object referred by the reference `S`, a `Receiver` object referred by the reference `R`, a data channel referred by the reference `CHD`, and an acknowledge channel referred by the reference `CHA`. We have the constants `initS`, `initR`, `initCHD` and `initCHA` as the initial states for each object, and `init` as the initial configuration.

```
fmod ABP is
  inc CONFIG<SENDER+RECEIVER> .
  op S : -> ObjectRef<SENDER> .
  op R : -> ObjectRef<RECEIVER> .
  ops CHD CHA : -> ObjectRef<CHANNEL> .
  op initS : -> Sender .
  op initR : -> Receiver .
  ops initCHD initCHA : -> Channel .

  eq bit(initS) = b1 .
  eq ack(initS) = b0 .
  eq data(initS) = d1 .
  *** other equations for initR, initCHD, initCHA

  op init : -> Config .

  eq init = < S | initS >, < R | initR >,
    < CHD | initCHD >, < CHA | initCHA > .
endfm
```

Since Maude implements rewriting logic, it is capable to specify a process algebra [18]. The following Maude module defines the syntax of our process algebra:

```
mod PROC is
  sorts Action Process Guard
        ActionProcess ActionProcessSeq .
  subsort ActionProcess < ActionProcessSeq .
  op ~_ : Action -> Action .
  op tau_ : Action -> Action .
  op _._ : Action Process -> Process  [strat(0) frozen] .
  op _|_ : Process Process -> Process [frozen assoc comm] .
  op _+_ : Process Process -> Process [frozen assoc comm] .
  op [_]_ : Guard Process -> Process  [frozen] .
  op _\_ : Process Action -> Process  [frozen] .
  op next : Process Config -> ActionProcessSeq .
  *** equations
endm
```

The operations are declared as being "frozen", i.e., we forbid the use of the rewriting rules in the evaluation of the arguments. An action is represented by a constant of sort `Action`, or by a term of sort `Action` as it is ∼`a` which identifies the complemented action of `a`. We denote by `tau(a)` the special action $\tau$ generated by a pair `(a, ∼a)`. The distinction between local and global actions is given by the restriction operator $P \setminus L$, where $L$ is (a subset of) the set of local actions. This syntax is closer to that used for CCS [18].

Given a process, we want to have the list of actions which can be executed. For this we introduce `next` operation; `next` builds a list of pairs (action, process) for a specific process and configuration. For concurrent processes, the list of pairs (action, process) contains the possible interleavings of actions which can be executed by each process, together with the possible communications between each pair of processes.

```
mod ABP-PROC is
  inc PROC .
  ops ABP S S1 V CHD CHA R T R1 : -> Process .
  ops in alterS alterR out schd
    scha rchd rcha chdt chat : -> Action .
  ops rOK sOK : -> Guard .

  rl S => in . S1 .
  rl S1 => ~ schd . scha  . V .
  rl V => ([sOK](alterS . S)) + ([not sOK]S1) .
  rl CHD => schd . chdt . ~ rchd . CHD .
  rl CHA => rcha . chat . ~ scha . CHA .
  rl R => rchd . T .
  rl T => ([not rOK]R1) + ([rOK](~ out . alterR . R1)) .
  rl R1 => ~ rcha . R  .
  rl ABP => ( S | CHD | CHA | R )
    \ scha \ schd \ rcha \ rchd .
endm
```

The interaction wrapper is described by the following module:

```
mod ABP-COORDINATED is
  inc ABP + ABP-PROC .
  op w : Action Config -> Config .
  var C : Config .
  eq w(in, C) = S .read() C .
  eq w(tau schd, C) =
     CHD .read( S .bit() C,
                S .data() C ) S .send() C .
  eq w(chdt, C) = CHD .transfer() C .
  eq w(tau rchd, C) =
     R .recFrame( CHD .data() C,
                  CHD .bit() C ) CHD .send() C .
  eq w(~ out, C) = R .write() C .
  eq w(alterR, C) = R .chAck() C .
  eq w(tau rcha, C) =
     CHA .read( R .ack() C , null ) R .sendAck() C .
  eq w(chat, C) = CHA .transfer() C .
  eq w(tau scha, C) =
     S .recAck( CHA .bit() C ) CHA .send() C .
  eq w(alterS, C) = S .chBit() C .

  eq eval(sOK, C) = (S .bit() C == S .ack() C) and
                    (CHA .error() C == b0) .
  eq eval(rOK, C) = (CHD .error() C == b0) and
                    (R .bit() C =/= R .ack() C) .
endm
```

# 6. FORMAL SEMANTICS OF THE IMPLEMENTATION

When designing a software system it is important to clarify its intended purpose. In [17], it is expressed that if the main purpose is to support reasoning and formal approach, then the system should strive for minimality; on the other hand if the main purpose is to be intuitive for the practitioners, then it should reflect the expected intuitions and practices. We want to put these purposes together, offering both intuitive languages for components, coordinator and wrapper, and supporting formal approaches and reasoning. We achieve this goal by using the formal models of object specification in hidden algebra, and a coalgebraic treatment of the labelled transition systems for the coordinator and wrapper.

A *model* for a class specification in hidden algebra is a $\Sigma$-algebra $M$ such that $M_v = D_v$ for each visible sort $v \in V$. The $M$-interpretation of a $\Gamma$-context $\mathfrak{c}$ is a function which maps a variable assignment $\vartheta : X \to D$ and a state $st$ to the value $[\![\mathfrak{c}]\!]_M(\vartheta)(st)$ obtained by replacing the occurrences of $\_$ in $\mathfrak{c}$ by $st$, the occurrences of $x \in X$ by $\vartheta(x)$, and evaluating the operations in $\mathfrak{c}$ according to their interpretation in $M$. The *behavioural equivalence* relation $\equiv$ over the set of states $M_{\mathsf{St}}$ is defined as follows: $st \equiv st'$ iff $[\![\mathfrak{c}]\!]_M(\vartheta)(st) = [\![\mathfrak{c}]\!]_M(\vartheta)(st')$ for each $\Gamma$-experiment $c$.

If $\mathcal{B}$ is a specification of concurrent objects, then a $\mathcal{B}$-model consists of a model for each class together with a model for their instances and configurations. This can be expressed in the terms of models for structured specifications [6].

The operational semantics of a process algebra is given by a labelled transition system. We prefer to represent a labelled transition system as a coalgebra. We denote by $\mathsf{Set}$ the category of sets. Let $A$ be a given set of *action names*. Let $\mathsf{T}_{\mathsf{LTS}} : \mathsf{Set} \to \mathsf{Set}$ the functor given by

$$\mathsf{T}_{\mathsf{LTS}}(X) = \{Y \subseteq A \times X \mid Y \text{ finite}\}$$

for each set $X$, and $\mathsf{T}_{\mathsf{LTS}}(f)$ is the function $\mathsf{T}_{\mathsf{LTS}}(f) : \mathsf{T}_{\mathsf{LTS}}(X) \to \mathsf{T}_{\mathsf{LTS}}(X')$ given by

$$\mathsf{T}_{\mathsf{LTS}}(f)(Y) = \{(a, f(x)) \in A \times X' \mid (a,x) \in Y\}$$

for each function $f : X \to X'$. A labelled transition system associated to a process algebra is a coalgebra $\pi : P \to \mathsf{T}_{\mathsf{LTS}}(P)$, where $P$ is a set of *processes*. We have $p \xrightarrow{act} q$ iff $(act, q) \in \pi(p)$.

Considering a wrapper $w(\mathbf{R})$, we define a coalgebra $w(\mathbf{R})_M : M_{\mathsf{Config}} \to \mathsf{T}_{\mathsf{LTS}}(M_{\mathsf{Config}})$ by $(act, cnfg') \in w(\mathbf{R})_M(cnfg)$ iff one of the following two conditions holds:

1. if the result sort of $w(\mathbf{R})(act)$ is $\mathsf{Config}$, then $cnfg' = [\![w(\mathbf{R})(act)]\!]_M(cnfg)$, and
2. if $w(\mathbf{R})(act) = O.q(X_1, \ldots, X_n)(Y)$ with $q$ an attribute, then $cnfg' = [\![Y]\!]_M(cnfg)$.

Moreover, we suppose that there is a coalgebra

$$w(\mathbf{R})_M/_{\equiv} : M_{\mathsf{Config}}/_{\equiv} \to \mathsf{T}_{\mathsf{LTS}}(M_{\mathsf{Config}}/_{\equiv})$$

which commutes the following diagram:

$$\begin{tikzcd} M_{\mathsf{Config}} \arrow[r, "can"] \arrow[d, "w(\mathbf{R})_M"'] & M_{\mathsf{Config}}/_{\equiv} \arrow[d, "w(\mathbf{R})_M/_{\equiv}"] \\ \mathsf{T}_{\mathsf{LTS}}(M_{\mathsf{Config}}) \arrow[r, "\mathsf{T}_{\mathsf{LTS}}(can)"'] & \mathsf{T}_{\mathsf{LTS}}(M_{\mathsf{Config}}/_{\equiv}) \end{tikzcd}$$

where *can* denotes the canonical onto morphism. The behavioural equivalence must be preserved by the transitions defined by $w(\mathbf{R})_M$. Therefore the action terms include only behavioural congruent operations, the transitions are extended to the quotient model, and we require the commutativity of the above diagram. The coalgebra $w(\mathbf{R})_M$ represents the labelled transition system $cnfg \xrightarrow{act} cnfg'$ defined in Section 4.

A $(\pi, w(\mathbf{R}))$-*coordinated* $\mathcal{B}$-*model* is a triple $(M, \gamma, proc)$, where $M$ is a $\mathcal{B}$-model, $proc : M_{\mathsf{Config}}/_{\equiv} \to P$ is a partial colouring operation (supervising operation), and $\gamma : dom(proc) \to \mathsf{T}_{\mathsf{LTS}}(M_{\mathsf{Config}})$ is a coalgebra commuting the following diagram:

$$\begin{tikzcd} M_{\mathsf{Config}}/_{\equiv} \arrow[d, "w(\mathbf{R})_M"'] & dom(proc) \arrow[l, "\mathrm{id}"'] \arrow[r, "proc"] \arrow[d, "\gamma"] & P \arrow[d, "\pi"] \\ \mathsf{T}_{\mathsf{LTS}}(M_{\mathsf{Config}}/_{\equiv}) & \mathsf{T}_{\mathsf{LTS}}(dom(proc)) \arrow[l, "\mathsf{T}_{\mathsf{LTS}}(\mathrm{id})"] \arrow[r, "\mathsf{T}_{\mathsf{LTS}}(proc)"'] & \mathsf{T}_{\mathsf{LTS}}(P) \end{tikzcd}$$

The rectangle from the right-hand side of the diagram says that *proc* is a homomorphism of coalgebras. We take the advantage of using hidden algebra, and we define the configuration supervisor as an attribute. By defining *proc* as a partial function, we restrict the coordinator to be aware of the coordinated configurations, and nothing more.

Each $(\pi, w(\mathbf{R}))$-coordinated $\mathcal{B}$-model defines a bisimulation:

PROPOSITION 1. *Let* $\gamma^{\sim} : graph(proc) \to \mathsf{T}_{\mathsf{LTS}}(graph(proc))$ *be the coalgebra given by* $(a, \langle cnfg', p' \rangle) \in \gamma^{\sim}(\langle cnfg, p \rangle)$ *iff* $proc(cnfg) = p$, $proc(cnfg') = p'$, *and* $(act, cnfg') \in \gamma(cnfg)$. *Then* $\gamma^{\sim}$ *is a bisimulation between* $w(\mathbf{R})_M$ *and* $\pi$.

A *homomorphism* of $(\pi, w(\mathbf{R}))$-coordinated $\mathcal{B}$-models from $(M, \gamma, proc)$ to $(M', \gamma', proc')$ consists of a $\Sigma$-homomorphism $h : M \to M'$ such that $proc'(h_{\mathsf{Config}}(cnfg)) = proc(cnfg)$ for all $cnfg \in dom(proc)$.
Let $\mathsf{Mod}(\mathcal{B}, \pi, w(\mathbf{R}))$ denote the category of the $(\pi, w(\mathbf{R}))$-coordinated $\mathcal{B}$-models. The following result shows that a homomorphism preserves the coalgebraic structure:

PROPOSITION 2. *Let* $h : M \to M'$ *be a homomorphism of* $(\pi, w(\mathbf{R}))$-*coordinated* $\mathcal{B}$-*models defined as above.*
*Then* $h_{\mathsf{Config}} : dom(proc) \to dom(proc')$ *is a coalgebra homomorphism from* $\gamma$ *to* $\gamma'$.

# 7. TEMPORAL PROPERTIES OF THE COORDINATED OBJECTS

Since the semantics of the coordinated objects is given by labelled transitional systems, we are able to use the temporal formulas for describing their properties. We use *Computational Tree Logic* (CTL) [4]. CTL is a branching time logic, meaning that its model of time is a tree-like structure in which the future is not determined; there are different paths in the future, any one of which might be the "actual" path that is desired. The CTL *formulas* are inductively defined as follows:

$$p ::= R.att()(\_) = d \;\mid\; R.att(d_1, \ldots, d_n)(\_) = d$$
$$\phi ::= \mathtt{tt} \mid \mathtt{ff} \mid p \mid (\neg\phi) \mid (\phi_1 \wedge \phi_2) \mid \mathtt{EX}\,\phi \mid \mathtt{EG}\,\phi \mid \mathtt{E}[\phi_1\,\mathtt{U}\,\phi_2].$$

The intuitive meaning of an atomic proposition of the form $R.att()(\_) = d$ is "the value of the attribute *att* for the object $R$ in the current configuration is $d$". The meaning for each propositional connective is the usual one. The set of propositional connectives is extended in the standard way. The temporal connectives are pairs of symbols. The first is `E`, meaning "there Exists one path". The second one is `X`, `G`, or `U`, meaning "neXt state", "all future state (Globally)", and "Until", respectively. We can express other five operators, where `A` means "for All paths":

$$\mathtt{EF}\,\phi = \mathtt{E}[\mathtt{tt}\,\mathtt{U}\,\phi] \qquad \mathtt{AX}\,\phi = \neg\mathtt{EX}\,(\neg\phi)$$
$$\mathtt{AG}\,\phi = \neg\mathtt{EF}\,(\neg\phi) \qquad \mathtt{AF}\,\phi = \neg\mathtt{EG}\,(\neg\phi)$$
$$\mathtt{A}[\phi_1\,\mathtt{U}\,\phi_2] = \neg\mathtt{E}[\neg\phi_2\,\mathtt{U}\,(\neg\phi_1 \wedge \neg\phi_2)] \wedge \neg\mathtt{EG}\,\neg\phi_2$$

The satisfaction relation $cnfg \models p$, expressing that a configuration *cnfg satisfies* a CTL atomic proposition $p$, is defined as follows:

1. $cnfg \models R.att()(\_) = d$ if and only if $cnfg = cnfg_1, (R|state)$ and $R.att()(state) = d$.
2. $cnfg \models R.att(d_1, \ldots, d_n)(\_) = d$ if and only if $cnfg = cnfg_1, (R|state)$ and $R.att(d_1, \ldots, d_n)(state) = d$.

Then the satisfaction relation $cnfg \models \phi$ is extended to arbitrary CTL formulas $\phi$ in the standard way [4].

For the ABP example, the following CTL formula:

$$\mathtt{AG}((\mathtt{S.bit}()(\_) = \mathtt{true} \wedge \mathtt{R.ack}()(\_) = \mathtt{false} \wedge \mathtt{S.data}()(\_) = \mathtt{d}) \to$$
$$\mathtt{AF}(\mathtt{S.bit}()(\_) = \mathtt{false} \wedge \mathtt{R.ack}()(\_) = \mathtt{true} \wedge \mathtt{R.data}()(\_) = \mathtt{d}))$$

expresses the fact that if in the current configuration the sender `S` has the `bit` equal to `true` and the sending `data` equal to $d$, the receiver `R` has the `ack` equal to `false`, then always there is in the future a configuration where `S` has the `bit` equal to `falsa`, `R` has the `ack` equal to `false` and the received `data` equal to $d$.

The temporal formulas are verified using a model-checking algorithm [4]. Our approach consists in extracting an SMV description [13] of the labelled transition system in a similar way to that described in [11]. The advantage of this method is that it allows the use of underspecified methods and of CTL formulas.

The algorithm building the Kripke structure is implemented by a Maude "built-in" operation called `writeSmv`. This operation yields a SMV module describing the Kripke structure [13]. The signature of this operation is given as follows:

```
mod SMV-WRITER is
  inc CTL .

  op writeSmv : Config Process FormulaSeq FormulaSeq
    String -> Nat [special (...)] .
endm
```

The first two arguments are the starting configuration and process needed to build the Kripke structure. The third argument is a sequence of CTL formula that represents the fairness constraints. The fourth argument is a sequence of CTL formula that represents the temporal properties to be verified. The last argument is the name of the output file. The result of this operation is the number of states generated.

The use of the `writeSmv` operation for the ABP correctness formula is as follows:

```
red writeSmv(
    init, ABP, none,
    (AG(
        AP(S .bit() C == b1)
        & AP(R .ack() C == b0)
        & AP(S .data() C == d1)
       ->
           AP(S .bit() C == b0)
           & AP(R .ack() C == b1)
           & AP(R .data() C == d1))
        )
    ),
    "abp.smv") .
```

It is worth to note that we use a simplified form of the correctness formula. The execution of the SMV model checker over the input file "abp.smv" provides the following result:

```
-- specification
    !E(1 U (S_bit__ = b1 & R_ack__ = b0 & ... is false
```

SMV says that the correctness property does not hold, and provides a counterexample. The counterexample shows an infinite path where a channel always generates errors. We recall that ABP works properly under the assumption that the channels are fair, i.e., they do not produce such infinite sequences of errors. This assumption can be easily specified by adding a fairness constraint for each channel.

```
red writeSmv(
    init, ABP,
    (AP(CHD .error() C == b0)
        & AP(CHD .procName() C == transfer>),
     AP(CHA .error() C == b0)
        & AP(CHA .procName() C == transfer>)),
    (AG(
        AP(S .bit() C == b1)
        & AP(R .ack() C == b0)
        & AP(S .data() C == d1)
       ->
            AF(AP(S .bit() C == b0)
            & AP(R .ack() C == b1)
            & AP(R .data() C == d1))
        )
    ),
    "abp.smv") .
```

The two fairness constraints are specified by the lines 3-6. Using these constraints, SMV succeeds to prove the correctness of ABP under the fairness assumption:

```
-- specification
    !E(1 U (S_bit__ = b1 & R_ack__ = b0 & ... is true
resources used:
processor time: 0.062 s,
BDD nodes allocated: 10057
Bytes allocated: 1695620
BDD nodes representing transition relation: 1191 + 1
```

# 8. CONCLUSION

Modelling complex systems out of components and building corresponding applications is currently a challenge for software community. This task assumes the description of the whole system from different points of view: data, concurrency, synchronization, communication, coordination. Since each specific aspect related to data, concurrency, and coordinated component-based systems can be described using a specific formalism, the final description of the system could be a multiformalism specification. HiddenCCS formalism introduced in [2, 3] is a formal specification framework based on hidden algebra and CCS. This specification extends the object specification with synchronization and communication elements associated with methods and attributes of the objects, and use a CCS description of the interaction patterns. The operational semantics of hiddenCCS specifications is based on labelled transition systems. Another related multiformalism can also be found in [16]. Frølund and Agha [7] introduce independent support constructs for coordination of concurrent objects using synchronizers. We extend their synchronizers in providing a more general and elegant approach based on process algebra and related notions.

In [1] the authors focus on a formal basis for one aspect of software architecture design, namely the interactions between components. They define architectural connectors as explicit semantic entities characterizing the participants roles in an interaction. A system is described by its components and connectors. A variant of CSP is used to define the roles, ports, and glue specifications. An important motivation for the authors is represented by the potential for automating the analysis of architectural description by using tools to determine whether connectors are well formed, and ports are compatible with their roles. In [12], the authors describe a distributed a software architecture in terms of its components and their interactions, associating behavioural specifications with the components and then checking whether the system satisfies certain properties. The approach is based on labelled transition systems to specify the behaviour, and compositional reachability analysis to check composite system models. Our approach add an important feature to these architecture description languages, namely a useful separation of concerns. Moreover, we use an efficient model checker for verifying temporal properties of a component-based system.

More precisely, in this paper we design a specification language for coordinated objects with a syntax closer to OOP languages, and a semantics of coordination given by an integrating bisimulation. The interaction between the coordinating process and autonomous objects is given via interaction wrapper. Operational semantics of the coordinated objects is given by labelled transition systems, and we express their temporal properties in CTL. These temporal properties can be verified automatically by using adapted algorithms and new specific tools.

# 9. REFERENCES


[1] R. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, vol.6, pp.213-249, 1997.

[2] G. Ciobanu and D. Lucanu. Specification and verification of synchronizing concurrent objects. In J.Derrick E.Boiten and G.Smith (Eds.): *Integrated Formal Methods 2004*, Lecture Notes in Computer Science, vol.2999, pp.307-327, Springer, 2004.

[3] G. Ciobanu and D. Lucanu. Communicating Concurrent Objects in HiddenCCS. *Electronic Notes in Theoretical Computer Science*, vol.117, pp.353-373, Elsevier, 2004.

[4] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. MIT Press, 2000.

[5] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada Maude: Specification and Programming in Rewriting Logic *Theoretical Computer Science*, vol. 285, pp.187-243, 2002.

[6] F. Durán and J. Meseguer. Structured Theories and Institutions. *Theoretical Computer Science*, vol.309, pp.357-380, 2003.

[7] S. Frølund and G. Agha. A language framework for multi-object coordination. In *Proc. ECOOP'93*, Lecture Notes in Computer Science, vol. 707, pp.346-360, Springer, 1993.

[8] J. Goguen, and G. Malcolm. A hidden agenda. *Theoretical Computer Science* vol.245, pp.55-101, 2000.

[9] Gh. Grigoraş and D. Lucanu. On Hidden Algebra Semantics of Object Oriented Languages. *Sci. Ann. of the "A.I.Cuza" Univ. of Iaşi*, 14(Computer Science), pp.51-68, 2004.

[10] B. Jacobs. Coalgebraic Reasoning about Classes in Object-Oriented Languages. In Electronic Notes in Theoretical Computer Science, vol.11, pp.231-242, Elsevier, 1998.

[11] D. Lucanu, and G. Ciobanu. Model Checking for Object Specifications in Hidden Algebra. In B.Steffen, G.Levi (Eds.) *Verification, Model Checking, and Abstract Interpretation*, Lecture Notes in Computer Science vol.2937, Springer, pp.97-109, 2004.

[12] J. Magee, J. Krammer, and D. Giannnakopoulou. Analysing the Behaviour of Distributed Software Architecture: a Case Study. In *Proc. of the 5th IEEE Workshop on Future Trends of Distributed Computing Systems*, Tunis, pp.240-245, 1997.

[13] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[14] R. Milner. *Communicating and Mobile Systems: the π-calculus*. Cambridge University Press, 1999.

[15] J. Rutten. Universal Coalgebra: A Theory of Systems. *Theoretical Computer Science* vol.249, pp.3-80, 2000.

[16] G. Salaün, M. Allemand, and C. Attiogbé. A Formalism Combining CCS and CASL. Research Report 00.14, IRIN, Univ. Nantes, 2001.

[17] M. Shaw and D. Garlan. Formulations and Formalisms in Software Architecture. In J. van Leeuwen (Ed.): *Computer Science Today: Recent Trends and Developments*, Lecture Notes in Computer Science, vol.1000, Springer, 1995.

[18] A. Verdejo, and N. Martí-Oliet. Implementing CCS in Maude 2. In *4th WRLA*, *Electronic Notes in Theoretical Computer Science*, vol.71, pp.239-257, Elsevier, 2002.