Electronic Communications of the EASST
Volume 20 (2009)

Proceedings of the Third International Workshop on Foundations and Techniques for Open Source Software Certification (OpenCert 2009)

Model-based Testing and Analysis of Coordinated Components

Gabriel Ciobanu and Dorel Lucanu

11 pages

ISSN 1863-2122

Model-based Testing and Analysis of Coordinated Components

Gabriel Ciobanu[1] **and Dorel Lucanu**[2]

[1] gabriel@info.uaic.ro
Institute of Computer Science, Romanian Academy
and A.I.Cuza University of Iaşi, Romania
[2] dlucanu@info.uaic.ro
Faculty of Computer Science, A.I.Cuza University
Iaşi, Romania

Abstract: Software components are common in the open source community. These components can be specified in model languages like AsmL or JML by using contracts (preconditions, postconditions). Starting from an integrated specification (components, coordinating process, wrapper), a model program is defined and used to define the formal semantics of the whole system. The relationship between coordinator and components are expressed as a bisimulation. The model program can be used for conformance testing and generating test case suites when working with closed systems, and for scenario-based testing when working with reactive systems.

Keywords: components, coordination, process algebra, model program, testing

1 Introduction

We refer to components interacting with each other and with their environment according to a coordinating process. Languages like Java can model this kind of coordination at a low level of abstraction (threads communicate through shared variables). Such a low level approach does not allow the composition of different coordination policies without changing the implementation of the coordinated entities. If the level of abstraction is higher, the integration of the coordination and computation modules becomes non-trivial, and only few good programmers are able to handle it. These difficulties are triggered 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), by the absence of abstraction (no declarative means to specify coordination), and by the lack of compositionality and flexibility.

As a possible solution, we have introduced and studied a specification formalism for coordinated concurrent objects [4, 5, 6, 9]. This formalism allows explicit specifications for classes and objects, quite similar to the existing object-oriented programming notation. A process algebra (CCS) is used to describe the coordination between the concurrent objects; a coordinator describes the global goal of the system, orchestrating the complexity of the local computational goals provided by the objects. Such an approach supports a clear separation between the coordinated objects and their coordinator. Coordination process and coordinated components are rather independent; the coordinator can be composed and replaced easily, some objects working under the same coordinator can be refined or modified. The explicit description of coordination between components can define various interaction patterns and policies. In [4] we have de-

fined a formal framework for systems of components coordinated by a process. The components (objects) are specified in hidden algebra [10], the coordinating process is described in process algebra [11], and a wrapper is expressed as a function mapping the coordinating process actions to sequences of method calls. The integrated semantics of the components, coordinating process and wrapper is given by a bisimulation relation which exhibits how the transitions of the coordinating process are related to the transitions between the configurations given by components. Such an approach is independent of the way how the components are specified.

This paper describes formal methods that can be used to the certification of component-based software systems; we consider components specified in a modelling language like AsmL[1] (see http://www.codeplex.com/AsmL) or JML (see http://sourceforge.net/projects/jmlspecs), both considered as open source tools. An advantage of a modelling language is given by the use of contracts (preconditions, postconditions). Such a contract establishes both the obligations of the method caller (precondition) and what a method guarantees for a correct call (postcondition). The design by contracts of the object oriented systems also uses the class invariants. In order to keep the presentation as simple as possible, the invariants are not considered in this paper. We specify the components via pre- and post-conditions on object methods, a coordination process via a CCS process algebra, and the wrapper as a mapping of (parameterized) pairs of actions to sequences of components methods.

We use a notion of wrapper to express the coordination, and build formally a model program defining the state variables and the update rules of an abstract state machine. The model program can be used for conformance testing and generating test case suites for closed systems, and for scenario-based testing for reactive systems. The synchronization between components and the coordinator can be defined as a bisimulation by using the labelled transition systems described as coalgebras. The model together with the way how the components can work concurrently under the coordination of a process expressed in a process algebra style are illustrated by an example describing the interaction between an ATM (Automatic Teller Machine) and a bank.

2 Specification of Coordinated Components

We deal with systems consisting of three main parts: coordinated components, a coordinator, and a means by which the coordinator controls the activity of the coordinated components. We model the coordinator as a term of process algebra, the coordinated components as objects, and the means of coordination as a wrapper.

We consider an example where the interaction between an ATM and a bank is described. The interaction between a customer and an ATM is not included here; we abstractly represent a customer by a card, a user PIN (personal identification number) and a user amount. Using an ATM, customers can access their bank accounts in order to make cash withdrawals and check their account balances. Typically, a user inserts into the cash machine a card encoded with information on a magnetic strip. To prevent unauthorized transactions, a PIN must be entered by the user. If the account is accessible, the bank and the ATM complete the transaction; most ATMs can dispense cash, and provide information on account balances.

[1] AsmL is the specification language of Abstract State Machines.

2.1 Components Specified by Contracts

The component specification includes information about how to use the component, and what it does from the clients viewpoint. These aspects could be described by a collection of attributes and methods. The attributes are represented by a collection of variables, and the methods are specified by preconditions and postconditions.

We start by specifying an ATM component. An ATM has five attributes: `availAmount` = the amount of money available in the machine, `isCardInserted` = a boolean attribute which is true if and only if a card is inserted in the ATM, `cardPIN` = the PIN number of the inserted card (if any), `cardNumber` = the number of the inserted card (if any), and `amount` = the amount of money introduced by a customer (if any). Another possible attribute is given by the messages displayed on the ATM screen. The interface with a customer is specified by five methods. A method `readCard` describes the action performed by the machine when a card is inserted; its precondition is given by the `require` expression, and its postcondition is given by the conjunction of `ensure` expressions. A card is represented by a structure with two fields: `PIN` and `number`. The other methods of the interface with a user are `enterPIN()`, `askBalance()`, `enterAmount()`, and `releaseCard()`. The full specification of these methods is omitted here. On the other hand, the interface with a bank is specified by means of four methods. Since in what follows we describe the interaction between an ATM and a bank, we provide a full specification for these methods – their description using the AsmL syntax is almost self-explanatory.

```
class ATM
  var availAmount as Integer
  var insertedCard as Card?
  var enteredAmount as Integer
  var enteredPin as Integer

  isCardInserted() as Boolean
    return insertedCard <> null

  // interaction with a user
  readCard(newCard as Card)
    require isCardInserted() = false
    ensure resulting insertedCard = newCard

  readPin(newPin as Integer)
    ...
  wrongPin()
    ...
  askBalance()
    ...
  displayBalance()
    ...
  readAmount(newAmount as Integer)
    ...
  releaseCard()
    ...

  // interaction with a bank
  displayMessage(msg as MessageType)
    require isCardInserted() = true

  displayBalance(bal as Integer)
    require isCardInserted() = true
    ensure resulting enteredAmount = 0
```

```
  cash()
    require isCardInserted() = true
    require enteredAmount <= availAmount
    ensure resulting availAmount = availAmount - enteredAmount
```

A bank is abstractly specified as a set of accounts. The specification of an account is given by a structure with three fields: the card number, the balance, and a boolean attribute `isAccessible` which is true if and only if the account is not blocked or closed. The value of this attribute can be changed by the interaction with other components (not included here).

The interface of a bank with an ATM uses an auxiliary function `getAccount` which returns the bank account corresponding to a given card number. The first two methods are rather constraints than operations: they can be executed only if their preconditions are satisfied by the current state. They do not change the state, and neither return a value.

```
class Bank
  var accounts as Set of BankAccount

  // auxiliary methods
  getAccount(aCardNumber as Integer) as BankAccount
    return the A | A in accounts where A.number = aCardNumber

  // interface with an ATM
  notAccessible(aCardNumber as Integer)
    require getAccount(aCardNumber).isAccessible = false

  notEnoughMoney(aCardNumber as Integer, anAmount as Integer)
    require getAccount(aCardNumber).balance < anAmount

  getBalance(aCardNumber as Integer) as Integer
    require getAccount(aCardNumber).isAccessible = true
    ensure result = getAccount(aCardNumber).balance

  grantMoney(aCardNumber as Integer, anAmount as Integer)
    require getAccount(aCardNumber).balance >= anAmount and
            getAccount(aCardNumber).isAccessible = true
    ensure resulting getAccount(aCardNumber).balance =
                     getAccount(aCardNumber).balance - amount
```

Preconditions and postconditions describe properties of individual methods. Additional information is necessary with respect to the global properties or the interaction between objects.

2.2 Coordinating Process Specification

The state of the art in coordination models for systems of agents is presented in [3, 13]. Our model is a channel-based coordination model; Manifold [1] is a prototype for this class. Manifold is based on the Ideal Worker Ideal Manager (IWIM) model, and it has basically two kinds of processes: manager and worker. The manager coordinates the workers and the communications among them. The workers are computational processes which are not aware of who needs the results of their work, or to whom they communicate to. Manifold is also event-driven: managers wait for some specific event to trigger some actions; these actions determine the manager to change its state.

We use a coordinator providing a high level description of the interaction between objects. Its syntax is inspired by process algebras as CCS and π-calculus [11]. 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

some equations, as you can see in Figure 1. Process A corresponds to an arbitrary ATM. Interaction is started by inserting the card, i.e. by an equation $A = ins.A_1$ meaning that an action ins is followed by a process expression A_1. A_1 is a nondeterministic choice $rel.A + ep.A_2$, where the first expression is releasing the card (when the user decides to press "Cancel" button) followed by starting a new process A, and the second expression describes the action ep of entering a PIN followed by a process A_2. A_2 represents a nondeterministic choice between $na.A$, $rel.A$, $wp.A_1$, $ab.A_3$ and $ea.A_4$. This means that it is possible to get either a non-accessible account message (na), or to cancel the whole process (rel), or to enter a wrong pin (wp), or to ask balance (ab), or to enter an amount in order to cash it (ea). Both actions na and rel are followed by A, while ab is followed by an action gb of getting the balance and then executing process A_2 again. Finally ea is followed by a nondeterministic choice between either nem action of receiving a message "Not Enough Money" or getting money (gm). A local action act can involve the existence of its

$$A = ins.A_1 \quad \text{and} \quad B = \overline{na}.B + \overline{gb}.B + \overline{nem}.B + \overline{gm}.B$$

where

$$A_1 = rel.A + ep.A_2 \qquad A_2 = (na + rel).A + wp.A_1 + ab.A_3 + ea.A_4$$

$$A_3 = gb.A_2 \qquad A_4 = nem.A_2 + gm.A_5 \quad , \quad A_5 = rel.A$$

Figure 1: Coordinating process for an ATM and a bank

complementary local action denoted by $\overline{act}$ (also, the complementary action of $\overline{act}$ is act). These two complementary local actions establish a synchronization between components. Process B represents a bank. In our description, a bank can either send a "Not Accessible" message ($\overline{na}$), offering the balance ($\overline{gb}$), send a message "Not Enough Money" ($\overline{nem}$), or offering the required amount ($\overline{gm}$).

The interaction between an arbitrary ATM and an arbitrary bank is described by $A|B$. We prefer to see the above process specification rather as a parametric one. Given a concrete ATM denoted by atm and a concrete bank $bank$, then their interaction is given by $A\langle atm\rangle|B\langle bank\rangle$. A coordinating process specification is finally given by equations of parametric process expressions. For example, the specification of $A = ins.A_1$ related to a specific cash machine atm is given by $A\langle atm\rangle = ins\langle atm\rangle.A_1\langle atm\rangle$. This allows to extend the specification of many ATMs and one bank by $A\langle atm_1\rangle|A\langle atm_2\rangle|\ldots|B\langle bank\rangle$. The case of two banks and their ATMs can be described as $(A\langle atm_1\rangle|\ldots|B\langle bank\rangle) + (A\langle atm'_1\rangle|\ldots|B\langle bank'\rangle)$. Assuming that we have a specification for an arbitrary user U, then the interaction between a specific $user$, a specific atm, and a specific $bank$ is given by $U\langle user\rangle|A\langle atm\rangle|B\langle bank\rangle$. The evolution of such a system is described by a labelled transition system defined by the operational semantics of the process algebra [11].

2.3 Wrapper

We introduce and use a notion of wrapper in order to specify the functionality of a system of coordinated components. Usually the software wrapping allows the data flowing in and out of the components to be intercepted and described. Also communication with other components is examined before passing through. Our wrapper specifies the interaction between the components by using their methods and the coordinating action of the process. We get a desirable

separation of concerns, offering a suitable abstract level for designing large component-based systems without losing the details of low level implementation of components. Such a specification has similarities with an orchestra, where independent players are synchronized by a conductor. The link between the players and the coordinating conductor is given by certain entry moments and orchestral scores. The wrapper instructs the players according to the scores in order to implement the desired resulting music. Therefore the wrapper instructs the components by using necessary information for their executions in order to realize a coordinated interaction. An interaction realized by two complementary actions is denoted by τ. For instance, we denote by $\tau(gb(atm),\overline{gb}(bank))$ the interaction between an action gb of getting the balance at atm and its complementary action $\overline{gb}$ of providing the balance by $bank$. For instance, the wrapper w for the system of coordinated components described previously has the following definitions for the interactions between atm and $bank$:

$$
\begin{aligned}
&w[\tau(gb(atm),\overline{gb}(bank))] = atm.displayBalance(bank.getBalance())\\
&w[\tau(na(atm),\overline{na}(bank))] = bank.notAccessible();atm.displayNotAccessible();\\
&\qquad\qquad\qquad\qquad atm.release()\\
&w[\tau(nem(atm),\overline{nem}(bank))] = atm.displayNotEnoughMoney()\\
&w[\tau(gm(atm),\overline{gm}(bank))] = bank.grantMoney();atm.cash()
\end{aligned}
$$

Formally, a wrapper $w(c_1,\ldots,c_n)$ for a process $P(c_1,\ldots,c_n) = P_1\langle c_1\rangle \,|\, P_2\langle c_2\rangle \,|\, \ldots \,|\, P_n\langle c_n\rangle$ associates a program $w[act]$ for each action label act such that there is a labelled transition $p \xrightarrow{act} q$ in the operational semantics of $P(c_1,\ldots,c_n)$. The program $w[act]$ is expressed in terms of the components involved in such a transition (they do not depend on the particular processes p and q). Recall that act is either of the form $a\langle c_i\rangle$ (action a of component c_i) or of the form $\tau(a\langle c_i\rangle,\overline{a}\langle c_j\rangle)$ (interaction between components c_i and c_j according to their complementary actions a and $\overline{a}$). Such a wrapper provides a computational meaning to each action of the coordinating process. According to the computational meaning behind each action, an interaction between two components can be a synchronization or a communication. A synchronization is provided by the sequential or concurrent executions of methods from the two components, and a communication consists in using the attributes of a component as parameters for methods of the other component (it appears as an interaction between a method and an attribute). Essentially the wrapper binds the actions of the coordinating process to the components methods. For instance, $w[\tau(gm(atm),\overline{gm}(bank))]$ corresponds to the sequence of method calls, namely the method $grantMoney()$ of the component $bank$, followed by $cash()$ and $release()$ of the component atm.

2.4 Formal Semantics

The semantics of a system of coordinated components is given by means of a model program [16]. A *model program* defines the state variables and update rules of an abstract state machine [2]. A state of a model program M is a first-order structure which captures a snapshot of variables values at a given step. A step of M is given by an *action method* which describes an update rule of the abstract state machine. An action method am has formal parameters $\mathbf{x}$, a precondition $Pre(am)$, and an update part $Update(am)$. Mathematically, an action method am is a function which for a given state s and some actual parameters which satisfy $Pre(am)$, it produces a new

state s' where some state variables have changed. A model program can be written using a high level program language as AsmL (http://www.codeplex.com/AsmL). A model program M defines a labelled transition system $\mathsf{LTS}(M)$ obtained by unwinding M (see [16] for more details).

A simple example is given by a model program $M(c)$ for a component c. The state variables are given by the component attributes, and the action methods by (a subset of) the component methods. $Pre(c.m)$ is the precondition of the method m, and $Update(c.m)$ includes the updates of the attributes according to the postconditions. For instance, the postcondition of $ATM::cash()$ produces the update `availAmount := availAmount - enteredAmount`. We assume that the postconditions of the methods can be expressed as updates of the attributes; modelling languages like AsmL are powerful enough to satisfy this requirement for many practical cases. We consider the specification of a component as a non-modal class, i.e., the only constraints over the methods calls sequences are those given by the methods preconditions. In other words, the call of a method is allowed in any state satisfying its precondition.

Another example is given by a model program $M(w,c_1,\ldots,c_n)$ given by a wrapper $w[c_1,\ldots,c_n]$. The state variables are given by $c_1,\ldots,c_n$, and the action methods are given by the guarded programs corresponding to $w[act]$. $Pre(w[act])$ is given by either the weakest precondition $wp(w[act],true)$ or by a verification condition [12, 14]. $Update(w[act])$ modifies the individual state of each c_i according to postconditions of the involved methods. Let us assume that $w[act]$ is written using method calls, parallel composition $\|$ and sequential composition $;$. For a method call $c_i.m(\mathbf{z})$, $Update(c_i.m(\mathbf{z}))$ is the same as that for the model defined by component c_i. For a parallel composition, $Update(S_1 \| S_2) = Update(S_1) \| Update(S_2)$, and for a sequential composition, $Update(S_1;S_2) = Update(S_1);Update(S_2)$.

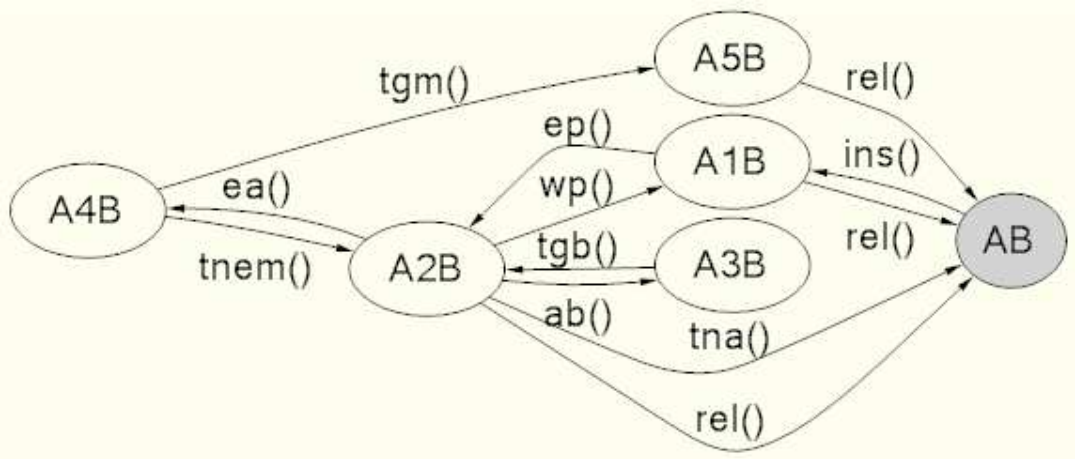

Figure 2: Transition graph for the coordinating process

We describe now the model program $M = M(\mathscr{C},P,w,c_1,\ldots,c_n)$ defined by a specification $\mathscr{C}$ of components, a coordinating process $P(c_1,\ldots,c_n)$ for the components $c_1,\ldots,c_n$, and a wrapper w for $P(c_1,\ldots,c_n)$ with $\mathscr{C}$. We proceed in a reversed order: first we define a labelled transition system LTS, and then we build the model over the skeleton of this LTS. The specification of a process $P(c_1,\ldots,c_n)$ defines a finite $\mathsf{LTS}(P)$ [8]. The transition graph corresponding to the interaction between *atm* and *bank* is given in Figure 2, where AB corresponds to $A\langle atm\rangle \,|\, B\langle bank\rangle$,

$A1B$ to $A1\langle atm\rangle \mid B\langle bank\rangle$, and so on. The transition *tgm()* corresponds to $\tau(gm(atm), \overline{gm}(bank))$; similar for *tgb()*, *tna()*, and *tnem()*.

$\mathsf{LTS}(P)$ can be easily described as a model program $M(P)$ with a single state variable vp ranging over the states (as an enumerating type), and with the action methods corresponding to the ones in P. If act is such an action, $Pre(act)$ is $\vee_{p\in Src(a)} vp = p$ and $Update(act)$ is $+_{q\in Tar(a)} vp := q$, where $Src(act) = \{p \mid p \xrightarrow{act} q \in \mathsf{LTS}(P)\}$, $Tar(act) = \{q \mid p \xrightarrow{act} q \in \mathsf{LTS}(P)\}$, and $+$ denotes the nondeterministic choice operator. The state variables of $M(\mathscr{C}, P, w, c_1, \ldots, c_n)$ are those from $M(w, c_1, \ldots, c_n)$ together with vp. The action methods are the ones of $M(P)$ enriched with the preconditions and updates of the corresponding action from $M(w, c_1, \ldots, c_n)$: $Pre_M(act) = Pre_{M(P)}(act) \wedge Pre(w[act])$, and $Update_M(act) = Update_{M(P)}(act) \parallel Update(w[act])$.

The relationship between M, $M(P)$ and $M(w, c_1, \ldots, c_n)$ can be expressed in terms of bisimulations, as it is also presented in [4]. We just mention here the main construction and result. We consider the labelled transition systems as coalgebras $\gamma : X \to \mathsf{T}_{\mathsf{LTS}}(X)$, where

- $\mathsf{T}_{\mathsf{LTS}} : \mathsf{Set} \to \mathsf{Set}$ is the functor given by $\mathsf{T}_{\mathsf{LTS}}(X) = \{Y \subseteq A \times X \mid Y \text{ finite}\}$,
- Set is the category of sets,
- A is the set of action names,
- γ is the labelled transition system given by $x \xrightarrow{act} y$ iff $(a, y) \in \gamma(x)$.

The fact that M is a bisimulation between $M(P)$ and $M(w, c_1, \ldots, c_n)$ is expressed by the commutativity of the following diagram:

$$\begin{array}{ccccc} State & \xleftarrow{\pi_1} & State \times Proc & \xrightarrow{\pi_2} & Proc \\ \downarrow{\scriptstyle \mathsf{LTS}(M(w,c_1,\ldots,c_n))} & & \downarrow{\scriptstyle \mathsf{LTS}(M)} & & \downarrow{\scriptstyle \mathsf{LTS}(P)} \\ \mathsf{T}_{\mathsf{LTS}}(State) & \xleftarrow[\mathsf{T}_{\mathsf{LTS}}(\pi_1)]{} & \mathsf{T}_{\mathsf{LTS}}(State \times Proc) & \xrightarrow[\mathsf{T}_{\mathsf{LTS}}(\pi_2)]{} & \mathsf{T}_{\mathsf{LTS}}(Proc) \end{array}$$

We exemplify the construction of M by considering the action tgm. Recall that $w[tgm] = bank.grantMoney(); atm.cash()$. The method action *tgm()* in M is obtained from the corresponding one given in $M(P)$:

```
[Action]
tgm()
  require (vp = A4B)
  vp := A5B
```

by adding the conjunction of the preconditions of the methods `Bank::grantMoney()` and `ATM::cash()` together with the updates given by their postconditions.
The new obtained method is:

```
tgm()
  // precondition
  require (vp = A4B)
  require atm.enteredAmount <=
          bank.getAccount(atm.insertedCard.number).balance
  require atm.enteredAmount <= atm.availAmount
  require bank.getAccount(atm.insertedCard.number).isAccessible = true
  // updates
  getAccount(atm.insertedCard.number).balance :=
        getAccount(atm.insertedCard.number).balance - atm.enteredAmount
  atm.availAmount := atm.availAmount - atm.enteredAmount
  vp := A5B
```

In order to respect the encapsulation principle, we may include the updates of the balance and available amount in the two methods `Bank::grantMoney()` and `ATM::cash()`, respectively. Then, we replace the two updates in the action $tgm()$ with the calls of the two methods.

3 Testing and Analysis

A system $\mathscr{S} = (\mathscr{C}, P, w, c_1, \ldots, c_n)$ of coordinated components is called *closed* if the specification of P does not include global actions. This means that the LTS of $P(c_1, \ldots, c_n)$ has only decorated silent transitions $\tau(a, \overline{a})$. The corresponding action methods in M are called *controllable* [2, 16]. In other words, in a closed system we have only controllable actions. For instance, the system described in our examples becomes closed if a component *User* is added. The coordinator indicating how a user can interact with an ATM is

$$U = \overline{ins}.U_1 \qquad U_1 = \overline{rel}.U + \overline{ep}.U_2$$
$$U_2 = \overline{rel}.U + \overline{wp}.U_1 + \overline{ab}.U_2 + \overline{ea}.U_3 \qquad U_3 = \overline{rel}.U$$

The wrapper gives a model program for each interaction (e.g, $\tau(rel(atm), \overline{rel}(user))$. For these closed systems, the model program $M = M(\mathscr{C}, P, w, c_1, \ldots, c_n)$ can be used for *conformance testing* and *generating test case suites*. Both tests are lying on the same basic technique: define a subset of *acceptance states* in M, and compute all the paths connecting the initial state with an acceptance state. Such a path is called a *trace*. An example of such a trace is $tins()$ (insert a card), $tep()$ (enter a pin), $tea()$ (enter an amount), $tnem()$ (the bank sends "not enough money"), $trel()$ (release the card). In conformance testing, the implementation under test (IUT) is checked if it is able to follows all the traces. Of course, we consider only implementations built with components satisfying the specification $\mathscr{C}$. To generate a test case suite, we have to find for each trace suitable instances for components $c_1, \ldots, c_n$ such that IUT having as "input" these instances follows exactly the corresponding trace. For the trace mentioned above, the test case must include a *user*, an *atm*, and a *bank* such that the user has correctly introduces the pin, has an account at the *bank*, but the balance is less than the required amount. This can be reached by computing the *path condition* [15] for the trace, and then using an automated prover for finding a satisfiability witness for this condition.

Given some components $c_1, \ldots, c_n$, $\mathsf{LTS}(M(\mathscr{C}, P, w, c_1, \ldots, c_n)$ is usually a subsystem of $\mathsf{LTS}(P)$ because of the methods preconditions. Thus the traces are defined according to $\mathsf{LTS}(P)$. A problem appears when for each trace we can find suitable $c_1, \ldots, c_n$ such that the trace is in $\mathsf{LTS}(M(\mathscr{C}, P, w, c_1, \ldots, c_n)$. We refer to this problem as a *consistency problem* between the specification $\mathscr{C}$ of the components and the specification of the coordinating process P, because it is equivalent to checking whether the components are able to perform all the interactions specified by the coordinator. This problem can be solved in a similar way to that of the test case generation.

A system $\mathscr{S} = (\mathscr{C}, P, w, c_1, \ldots, c_n)$ of coordinated components is a *reactive system* if the specification of P includes local actions. For instance, if the user is not specified then the system described in our examples is reactive. This means that the LTS of $P(c_1, \ldots, c_n)$ could have transitions decorated with local actions (e.g., $ins()$ - which describes only the ATM behaviour when a card is inserted). The action methods in M corresponding to local actions in P are called *observable* [2, 16]. Thus, in a reactive system we have both controllable actions and observable

actions. The model $M(\mathscr{C}, P, w, c_1, \ldots, c_n)$ for a reactive system could be used for *scenario-based testing* using the technique described in [16]. For instance, the scenario suggested above must be explicitly given by describing the "reactions" of the user by setting the values for *enteredPin* and *enteredAmount*.

4 Conclusion and Related Work

The paper presents a technique that can be used to apply formal methods to the certification of component-based software systems, e.g. for conformance testing and generation of test case suites. A model for a given specification is defined by the components contracts, a specification of the coordinating process, and a wrapper binding the coordination action to sequences of the components method calls. We use a notion of wrapper in order to express the coordination between components. The formal semantics is given by a model program, and the relationship between coordinator and components can be expressed as a bisimulation. Such a model can benefit from the practical model-based testing tools. The model program can be used for conformance testing and generating test case suites for closed systems, and for scenario-based testing for reactive systems. We exemplify our approach by a system consisting of an ATM and a bank.

Previously the authors have introduced and used hiddenCCS in [5, 6] as a formal specification framework based on hidden algebra and CCS. Such a specification extends the object specification with synchronization and communication elements associated with methods and attributes of the objects. Then hiddenCCS is extended in [4] to a specification language with a syntax closer to object-oriented languages. We use Maude in [4] to give an algebraic semantics of this specification language, and show how this semantics can be used to verify the system against temporal properties. The specification of the components for a communication protocol by using a language having strong features of the object-oriented programming is presented in [9]. We also describe the extraction of a Kripke structure from the specification of a system of coordinated objects, and we use it to verify the correctness of a communication protocol.

Bibliography

[1] F. Arbab, I. Herman, P. Spilling. An Overview of Manifold and its Implementation. *Concurrency: Practice and Experience* 5(1), pp.23–70, 1993.

[2] E. Börger, R. Stärk. *Abstract State Machines A Method for High-Level System Design and Analysis*, Springer, 2003.

[3] N. Busi, P. Ciancarini, R. Gorrieri, G. Zavattaro. Coordination Models: A Guided Tour. *Coordination of Internet Agents: Models, Technologies, and Applications*, Springer, pp.6–24, 2001.

[4] G. Ciobanu, D. Lucanu. A Specification Language for Coordinated Objects. *ACM SIGSOFT Software Engineering Notes* 31(2), ACM Press, 2006.

[5] G. Ciobanu, D. Lucanu. Specification and Verification of Synchronizing Concurrent Objects. *Integrated Formal Methods*, LNCS vol.2999, Springer, pp.307–327, 2004.

[6] G. Ciobanu, D. Lucanu. Communicating Concurrent Objects in HiddenCCS. *Electronic Notes in Theor. Comp. Sci.* 117, pp.353–373, 2005.

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

[8] R. Cleaveland, J. Parrow, B. Steffen. The Concurrency Workbench: a semantics-based tool for the verification of concurrent systems. *ACM TOPLAS* 15(1), ACM Press, pp.36–72, 1993.

[9] M. Daneş, G. Ciobanu, D. Lucanu. Specification of Coordinated Objects and Verification of Their Temporal Properties. *7th SYNASC*, IEEE Computer Society, pp.259–266, 2005.

[10] J. Goguen, G. Malcolm. A Hidden Agenda. *Theoretical Computer Science* 245(1), pp.55–101, 2000.

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

[12] G. Nelson. A Generalization of Dijkstra's Calculus. *ACM Transactions on Programming Languages and Systems*, 11(4)m pp. 517-561, 1989.

[13] G.A. Papadopoulos. Models and Technologies for the Coordination of Internet Agents: a Survey. *Coordination of Internet Agents: Models, Technologies, and Applications*, Springer, pp.25–56, 2001.

[14] C. Pierika, F. S. de Boer A Proof Outline Logic for Object-Oriented Programming. *Theoretical Computer Science* 343(3), pp.413–442, 2005.

[15] G. Snelting , T. Robschink , J. Krinke. Efficient Path Conditions in Dependence Graphs for Software Safety Analysis. *ACM Transactions on Software Engineering and Methodology* 15(4), pp.410–457, 2006.

[16] M. Veanes, C. Campbell, W. Grieskamp, W. Schulte, N. Tillmann, L. Nachmanson. Model-Based Testing of Object-Oriented Reactive Systems with SpecExplorer. *Formal Methods and Testing*, pp.39–76, Springer, 2008.