Modelling a Secure, Mobile, and Transactional System with CO-OPN

Didier Buchs, Stanislav Chachkov, David Hurzeler
Software Engineering Laboratory
Swiss Federal Institute of Technology Lausanne
1015 Lausanne, SWITZERLAND
{Didier.Buchs, Stanislav.Chachkov, David.Hurzeler}@epfl.ch

Abstract

Modelling complex concurrent systems is often difficult and error-prone, in particular when new concepts coming from advanced practical applications are considered. These new application domains include dynamicity, mobility, security, and localization dependent computing. In order to fully model and prototype such systems we propose to use several concepts introduced in our specification language CO-OPN, like context, dynamicity, mobility, subtyping, and inheritance. CO-OPN (Concurrent Object Oriented Petri Net) is a formal specification language for modelling distributed systems; it is based on coordinated algebraic Petri nets. This paper focuses on the use of several basic mechanisms of CO-OPN for modelling mobile systems and the generation of corresponding Java code. A significant example of distributors accessible through mobile devices (for example, PDA with Bluetooth) is fully modelled and implemented with our technique.

1. Introduction

In the world of complex distributed reactive software systems, mobility represents a new step towards what we might call ubiquitous computing, and has become a major issue in software engineering.

The development of such systems requires modelling tools able to capture their properties as well as the structure of the interactions between the software and its environment. We also want these tools to allow for extensibility and maintenance, and to facilitate the design choices which will enable us to guarantee some system properties.

In this paper, we present a formal framework for the development of mobile distributed systems from the modelling phase to the implementation. The approach we propose has the object-oriented paradigm as a structuring principle. Our general formalism can express both abstract and concrete aspects of systems, with emphasis on the description of concurrency and abstract data types. This formalism is called Concurrent Object-Oriented Petri Nets (CO-OPN)[5][2]. A coordination layer has been developed on top of this formalism [6] so as to deal with a distributed architecture taking into account information about localization and mobility. This is what we plan to detail in this paper.

We will explain our model by using a top-down approach: we will describe the evolution from a high-level interconnection diagram between the major actors of our model, to the internal machinery of these actors, the description of the mobile agents, and the data types used. We will then also address the security and extensibility issues through some semantic concepts.

The paper is organized as follows: In section 2, we informally describe the example we are going to model, and show its interest. In section 3, we explain the model, using the top-down approach mentioned above. We also use the model to give a little bit more detail on the CO-OPN semantics, as far as transactions and mobility are concerned. We end the section by justifying our modelling choices (in the part concerning security) and by talking about subtyping and extensibility. Section 4 deals with automatic (but configurable) code generation from a CO-OPN specification to a Java program, with a particular emphasis on the mobility aspects.

2. The mobile shopping example.

Let us present our example.

The system is composed of three kinds of entities, namely the product dispensers, the mobile phones, and a bank (see figure 1). The bank may communicate with any other entity. A mobile phone and a product dispenser may communicate if and only if the former is located in what we might call the "Product Dispenser zone", i.e. if and only if they are close enough. In the picture, mobile phone 1 can see both dispensers,

whereas mobile phone 2 can only see the cigarette dispenser, and mobile phone 3 can see no commercial dispenser. The bank sees and can be seen by every actor of the system.

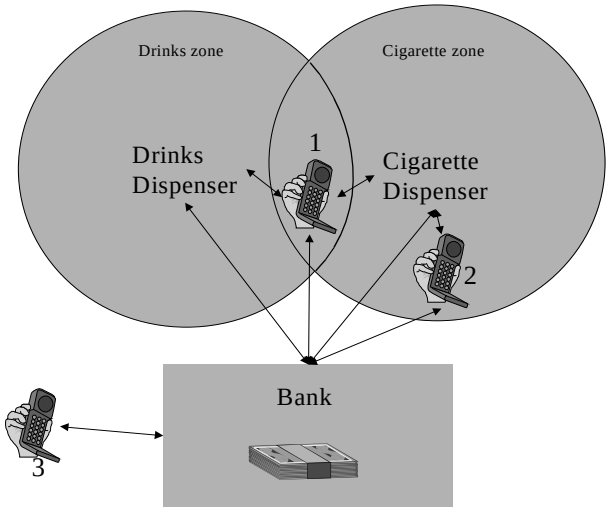

Figure 1: The actors (real life view)

Both the product dispensers and the mobile phone owners have an account in the bank; an owner of a mobile phone may ask the bank to make a transfer from its account to a product dispenser's account, provided the former has enough money to do so.

A product dispenser is an entity which sells products from a predefined list. A mobile phone is notified when it enters or leaves the zone of a product dispenser. It also receives the list of products that the dispenser sells.

The owner of a mobile phone may select a product dispenser from the list of all the «reachable» product dispensers. The mobile phone may then send both a request to the product dispenser to deliver a product, and a transfer order to the bank. Upon receipt of the money on its account, and if the requested product is not out of stock, the product dispenser delivers the product, and displays the identity of the mobile phone owner which has requested (and paid for) it. If, for some reason, the product is not delivered, the money is not withdrawn from the mobile phone owner's account.

This example seems interesting to us for different reasons. First of all, it looks like a realistic system which we may well see and use in a few years; a similar system already exists, where mobile phone owners may pay for parking their car by phone. Second, we believe it addresses many software engineering challenges: It is an embedded system, some of its entities are mobile, it is concurrent, transactional, and it has security as a main issue.

Throughout this paper, we will show how we deal with all these features, arguing that the CO-OPN specification language allows us to model this system by taking these features into account.

3. Modelling with CO-OPN

For obvious simplicity reasons, we will limit the system to one mobile phone and one drinks dispenser. We will explain how we can extend our model, and how it is then treated very similarly.

CO-OPN is an object-oriented modelling language, based on Algebraic Data Types (ADT), Petri nets, and IWIM (Idealized Worker Idealized Manager) coordination models [4]. Hence, CO-OPN specifications are collections of ADT, class and context (i.e. coordination) *modules* [5]. Syntactically, each module has the same overall structure; it includes an *interface section* defining all elements accessible from the outside, and a *body section* including the local aspects private to the module. Moreover, class and context modules have convenient graphical representations which are used in this paper, showing their underlying Petri net model. Low-level mechanisms and other features dealing specifically with object-orientation are out of the scope of this paper, and can be found in [1] [2]. We will, however, show how they have been used in our example.

3.1 The CO-OPN coordination model, and the system entities

In this section, we will describe the CO-OPN specification of the system described in the previous section. We will therefore introduce the various concepts of this language which we use to model the dispenser/mobile phone/bank system, and use a top-down strategy for the modelling. The idea is to start by the highest-level entities interfaces and connections, and to refine them progressively.

Because our system is composed of several computing entities, we use the high-level concept of coordination programming [7] for building our system. In our view, coordination is managing the dependencies among activities. Work has been done to show that coordination patterns are likely to be applied from the beginning of the design phase of the software development [3]. This process involves the use of specific coordination models and languages adapted to our

specific needs during the design phase of the modelling.

Because of their intrinsic nature, IWIM coordination models [7] are very well suited for the coordination of software elements during the design phase [3]. The coordination layer of CO-OPN [2][3][4] is a coordination language based on this model, well adapted to the formal coordination of object-oriented systems. The CO-OPN context modules define the coordination entities [6], while the CO-OPN classes define the basic lowest-level coordinated entities. Finally, as we will see, CO-OPN allows one to cover the formal development of concurrent software from the first formal specification up to the final distributed software architecture [1].

A CO-OPN context is an entity composed of a border, a signature, a finite number of other entities (objects or contexts) and number of connections between the different services and service calls of these entities.

Structure and communication of the three main contexts.

In this subsection we will describe the main contexts of our specification and show how they are connected and communicate. We will also show how they are built on classes (instantiated into objects), and communicate by sending objects to each other. First of all, the most high-level coordination entity is the `World` context.

This context represents the environment in which the entities we want to model are immerged.

Therefore, it contains three other contexts, namely the `Bank`, the `DrinksDispenser`, and the `MobilePhone` contexts, which are the major actors of our system (figure 2). The figure also shows that these contexts are potentially connected through different methods (provided services, black rectangles on the pictures) and gates (required services, white rectangles on the pictures).

It has seemed realistic to make these high-level entities communicate by the means of mobile agents (represented here by mobile objects) which, in a way, represent each context in the other contexts, and act as messengers.

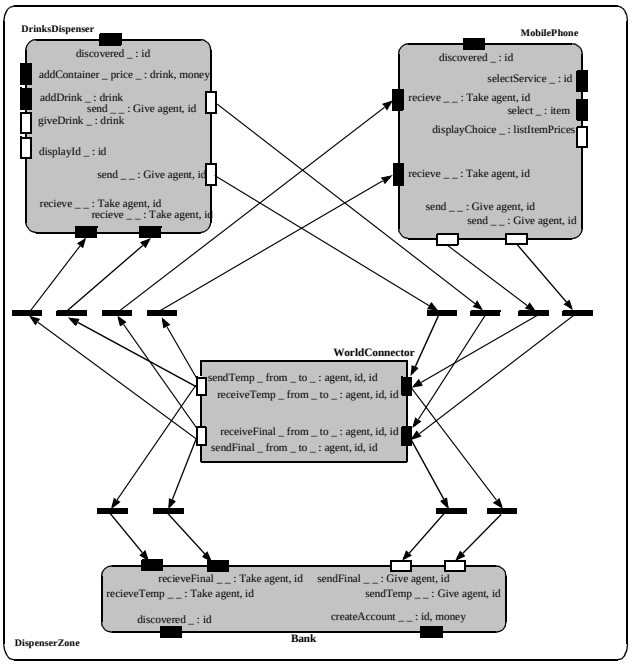

Figure 2: The coordination model

How do we make the three actors communicate? We have chosen to model this by creating a `WorldConnector` class. The use of this class is to centralize all the connections. In other words, the three actors are all connected to the `WorldConnector` class, and the latter manages the connections. Let us give an example. Suppose the `DrinksDispenser` context needs to send a message or an object to the `Bank` context. It then sends the message/object to the connector with the bank's id as parameter, and the connector retransmits it to the `Bank` context with the drinks dispenser's id as parameter to indicate origin.

The interest of such a connector class is that if we replace a component, we only need to change the connector class, and not find out which are the modules referring to this component and change them. Also, it diminishes the number of connections in the system (if we have more than 2 components).

These actors and the `WorldConnector` class are all part of the `World` context. Let us explain how this context is modelled in CO-OPN.

First of all, we define a context called `WorldSegment`, which consists in an object of class `WorldConnector` and an `AbstractContext` context. The `WorldSegment` context is really a generic: `WorldSegment(AbstractContext)`. This means that it depends on a formal parameter

module. The idea is that we will make the `World` context inherit from this context, and when doing this, we need to replace the parameter module `AbstractContext` by an actual module. The genericity is really a way of making specifications more concise.

We will have the `World` context inherit from context `WorldSegment(DrinksDispenser)`, context `WorldSegment(Bank)`, and context `WorldSegment(MobilePhone)`. As context `WorldSegment(AbstractContext)` has a `WorldConnector` object named `wc`, such will be the case with all three "instanciations", and thus, the `World` context will only have one object `wc` of class `WorldConnector`. This way, we have connected all entities!

The `Bank` context can, at any time, send an agent object to any of the other protagonists of the system. The drinks dispenser, however, must wait until the mobile phone is in its "zone" to be able to communicate with it. We have modelled this by a method `discovered` on the `DrinksDispenser` context, which we call as soon as the mobile phone is in the right zone. Because we do not model the mobile phone's movement in space, we will explicitly call this method during simulation. As soon as this method is activated, the communication link is established and the `DrinksDispenser` context sends one of its agents to the `MobilePhone` context.

```
send a iddd With receive a id mobile;
```

The `AbstractContext` context.

The three contexts share a common structure (inherited, as we will see later on, from the `AbstractContext` context) to deal with these agents: They all have an `arrived` method and a `send` gate, as well as three pending request queues to asynchronously deal with the sending and the receiving of agents (see figure 3).

When an agent arrives in the `AbstractContext` context, it is put in the `qIn` queue. When it leaves the context, it is either put in the `qOutFinal` (if it leaves with a **`give`**) or the `qOutTemp` (if it leaves with a **`lend`**) queue.

The contexts also have an identity, used to identify agents' destinations and origins. The identities are also used to attribute a bank account to an actor of the system.

Please note that in the contexts inheriting from this last context, we will not detail or even show (in the graphical representations) all the features which are identical to those of `AbstractContext.`

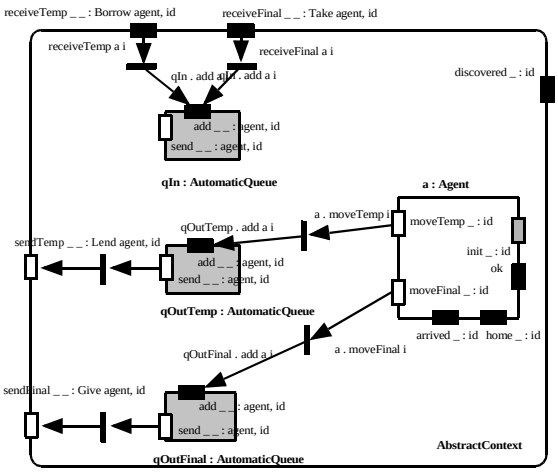

Figure 3: The abstract context

The `Bank` context

We have defined a class `Account`, and the bank naturally contains one instance of this class for all the other actors of the system (here, in this simplified model of our example, the `DrinksDispenser` and `Bank` contexts).

Apart from the `receive` and `send` methods and gates, the `Bank` context has a `createAccount` method, which, given an identity and an amount of money, creates an account for this identity that contains the specified amount of money (see figure 4).

The `Bank` context also has a `TransferManager` object, which manages the transfers between accounts, by waiting for the acknowledgement from both protagonists before actually withdrawing the money from one account to credit it to the other account. The reason for this will be detailed below in the subsection addressing the security issues.

The `MobilePhone` context

The `MobilePhone` context has, in addition to what has already been mentioned, two methods `select` (one to select agents and one to select items), and a `VirtualShop` object. When an agent arrives in the `MobilePhone` context, if it is a commercial agent, it is immediately put into the `VirtualShop` object. The mobile phone owner may then select the agents in this object by calling the `select` (agent) method. When he does this, the agent immediately dis-

```
Context Bank;
Inherit AbstractContext;
Interface
    Use
        Id;
        Money;
        Account;
        Agent;
    Gate
        send _ _ : Give agent, id;
    Methods
        createAccount _ _ : id, money;
        receive _ _ : Take agent, id;
        discovered _ : id;
Body
    Use
        BankAgent;
        BTransferManager;
        Queue;
    Method
        transfer _ from _ to _ : money, id, id;
    Objects
        BTM : bTransferManager;
        q : queue;
    Axioms
        discovered i With (ba . initHome idbank Owner
i) . . ((q . add ba i) . . (q . moveAll));
        createAccount i m With a . createAccount i m;
        receive a i With (a . arrived idbank) . . (q .
moveAll);
        ba . sendTellBank m from i to j With (BTM .
send m from i to j) / / ((a . getId i) / / (a .
withdraw m));
        ba . receiveTellBank m from i to j With (BTM .
receive m from i to j) / / ((a . getId j) / / (a .
credit m));
        q . send a i With send a i;
        a . move i With q . add a i;
    Where
        a : account;
        m : money;
        i,j : id;
        ba : bankAgent;
        a : agent;
End Bank;
```

Figure 4: The Bank context

plays its list of products and prices, and he may then select an `Item` (in our case, a drink) (see figure 5).

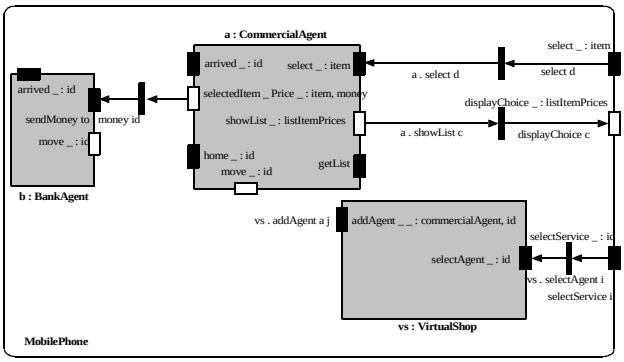

Figure 5: The MobilePhone context

The `DrinksDispenser` context

We will not detail this context very much in this paper, as it has already been treated in a previous paper [8]. The difference is that it does not have a money and coin management system anymore, as it now has an account. Let us very briefly recall its features. It has a central unit, which manages all the stock. Drinks come into drinks containers, which may be added to the stock (this is modelled by the external method `addDrink`; see figure 6, and please recall that we have not shown all the `AbstractContext` features, such as the queue). It also has a `displayId` gate, which displays the identity of the buyer when a drink is delivered.

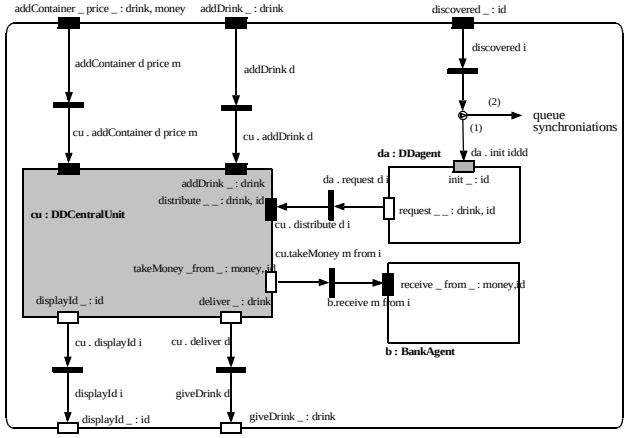

Figure 6: The DrinksDispenser context

Please note that in the picture, we have not shown the part inherited from the abstract context: So, the queue and its synchronizations have not been shown.

Abstract Data Types (ADT)

CO-OPN ADT modules define data types by means of algebraic specifications: They specify one or more sorts (names of data types) with generators and operations. The properties of the operations are given as positive conditional equational axioms in the body of the module. For instance figure 7 describes the ADT Drink which is just an enumeration of values.

```
ADT Drink;
Inherit Item;
    Rename
        item -> drink;
Interface
    Use  Item;
    Sort drink;
    Subsort drink -> item;
    Generators
 Ice Tea,Soda,Beer,Whisky: -> drink;
End Drink;
```

Figure 7: The Drink ADT;

Classes

CO-OPN classes are described by means of modular algebraic Petri nets with particular parameterized external transitions which are the *methods* of the class. The behavior of transitions are defined by so-called *behavioral axioms*, similar to the axioms in an ADT. A method call is achieved by synchronizing external transitions, according to the fusion of transitions technique. The axioms have the following shape:

`Cond => eventname With synchro : pre -> post`

in which the terms have the following meaning:

- `Cond` is a set of equational conditions, similar to a guard in Petri nets;
- `eventname` is the name of a method with the algebraic term parameters;
- `synchro` is the synchronization expression defining the policy of transactional interaction of this event with other events; the dot notation is used to express events of specific objects and the synchronization operators are sequence, simultaneity and non-determinism.
- `Pre` and `Post` are the usual Petri net flow relation determining what it is consumed from and what it is produced in the object state places.

CO-OPN provides tools for the management of graphical and textual representations [9].

As an example, let us detail the agents class a little more. We have two kinds of agents: The commercial agents and the bank agents. In the system modelled here, the only commercial agent is the drinks dispenser agent (`DDagent`) representing the drinks dispenser in the mobile phone context; this agent may go to the "discovered" mobile phone, display the list of drinks and prices, record an order, and migrate back to the dispenser. We also have a bank agent (`Bagent`) in both the mobile phone context and the drinks dispenser context. This bank agent records money transfer orders, and migrates back to the bank. It also helps deal with some security issues, as detailed below.

3.2 Transactions

CO-OPN synchronization have transactional semantics. It means that a synchronization succeeds if and only if all of its sub-synchronizations succeed (cf subsection on classes). Otherwise, the synchronization fails and, if there is no other way to fulfil the request, the state of the system does not change. This property of CO-OPN remains true with mobility.

Take for example the case where the `DrinksDispenser` context cannot deliver the drink chosen (and already paid for) by the customer. The whole "select drink" transaction will be aborted, and the customer will in fact not pay for the undeliverable drink: the whole system, including the mobile objects and the accounts will remain unchanged. In particular, objects that were moved during the failed synchronization, will be replaced in their original locations. This happens even if the failure is due to the non-accessibility of interacting components for a short time. In this context it is not crucial to not succeed because the transaction process is under user guidance (the customer) and can be redone.

3.3 Mobility

In CO-OPN, mobility is the movement of an object `o` from a context `C1` to another context `C2`.

As soon as `o` has moved, `C1` may not call its methods anymore (they fail); Symmetrically, before the move, `C2` may not call its methods (nor can it see its gate calls). Syntactically, mobility is managed by the four key words: **`give`**, **`take`**, **`lend`**, and **`borrow`**.

A declaration of a parameter of a context method can be decorated with one of these four mobility keywords. When a method that has a **`give`** parameter is called, the corresponding argument is automatically exported and removed from the context. Conversely, the take keyword imports its argument into the context. The difference between **`take/give`** and **`lend/borrow`** is the duration of object migration. Keywords **`give`** and **`take`** denote final moves. Keyword **`lend`** sends an object for the duration of a transaction. When the transaction finishes (commits or fails), the object is taken back. The same goes for **`borrow`**.

An example of the use of these keywords can be found in the interface of context `AbstractContext`:

```
send_ :Give agent;
receive _: Take agent;
```

In our case, the mobile objects are the objects from class `Agent`. It is crucial that a context may not call an object's methods once it has given it to another context. Otherwise, we would have dramatic security consequences: The mobile phone could possibly order more drinks without paying them, or the bank could secretly make the mobile phone pay amounts of money.

Another issue is that we have used the **`lend/ borrow`** keywords in order the ensure that the commercial agents come back to the mobile phone after an order. We have seen that the drinks dispenser sends an agent to the mobile phone once it has been discovered. If the mobile phone decides to buy a product, the agent goes back to the dispenser, bringing the order with it. After the product has, possibly, been delivered, we want the agent to go back to the mobile phone, so it may order more products. This is handled by using the **`lend`** and **`borrow`** keywords: When the `MobilePhone` context orders a product, it lends the agent to the `DrinksDispenser` context, and once the order transaction is finished (be it committed or failed), the agent automatically comes back.

3.4 Security issues

For such a system, security has been our major concern during the whole design and modelling phases. In this subsection, we will detail all security issues we got interested in, and show how we have coped with them.

First of all, as mentioned above, we do not want a context to be able to interact with an object which it no longer contains. This is directly avoided with CO-OPN mobility semantics.

Another issue, is that we need to make sure that:

- if the mobile phone owner actually pays for a drink, he eventually receives a drink.
- if the drinks dispenser actually delivers a drink, it has already been credited the right amount of money on its account.

These expectations are handled by the transactional semantics of CO-OPN. Indeed, we basically have a transaction of the following form (figure 8 and figure 9):

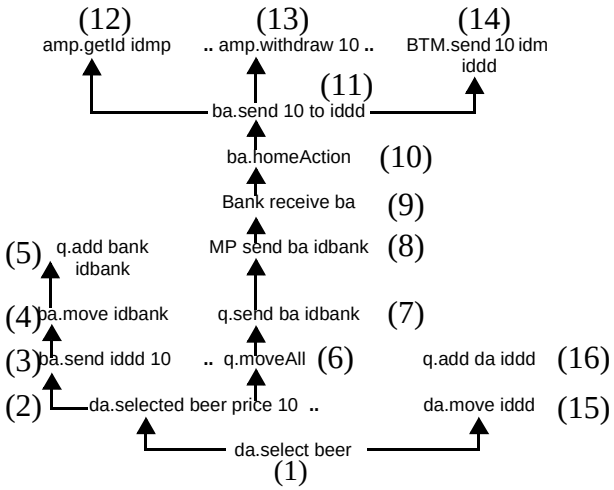

Figure 8: The buying process transaction. (Legend: 1. MP: MobilePhone, q: queue, ba: bank agent of MP, da: drink distributor agent, acc_mp: account of the mobile)

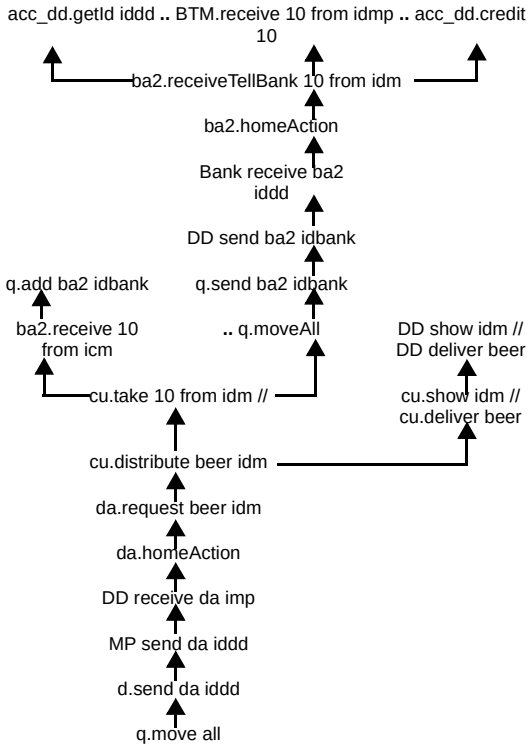

Figure 9: The buying process transaction. (Legend: 2. DD: drink distributor context, ba2: bank agent of DD, cu: central unit of DD, acc_dd: bank account of DD, BTM: bank transfer manager)

The execution of method `select_: item` in context `MobilePhone` is composed of two sub-synchronizations which are executed sequentially: `da.select_:item` (figure 8) and `q.moveAll` (figure 9). To understand these pictures, the reader should remember that in CO-OPN, a synchronization occurs simultaneously with all its sub-synchronizations (vertical arrows in figure 8 and figure 9). It is similar to the merging of transitions in classical Petri Nets. The sequential synchronizations is explicitly specified using sequence operator `_.. .` Therefore, the definition of `select` is written as:

```
select i With da.select i .. q.moveAll
```

This way, we ensure that the execution of `q.moveAll` occurs after the execution of `da.select i` and the results (in terms of resources, i.e. tokens in algebraic Petri nets) of the former are visible to the latter.

Let us present the beginning of the synchronization `select beer` in detail. The invocation of `da.select beer` ((1) on figure 8) results in sequential synchronization with two gates of `da: selected beer 10` (2) and `move iddd` (15). The first gate (2) notifies the environment of the purchase, while the second gate (15) asks to send the commercial agent back home. The `MobilePhone` context routes (figure 5) the service request `selected beer 10` to the method `send 10 to iddd` (3) of the bank agent and, in sequence, to `q.moveAll` (6).

The bank agent records this request and asks to be moved at the `Bank` context (4). As a result, it is added (5) to the queue, awaiting to be sent.

The execution of `q.moveAll` (6) that follows accomplishes the "send" operation. The top-level context `World` resolves the `send ba idbank` request (8) emitted by `MobilePhone` by calling the `receive ba idmp` method (number 9) of `Bank`, that, in its turn, invokes `ba.homeAction` (10). The execution of the former results in recording the "send money" order. This terminates the execution of the request `da.selected beer 10` (2).

The execution of (1) continues with `da.move iddd` (15) and then the rest of the axiom (figure 9). Basically, commercial agent `da` will return to its home host and ask to distribute the drink. Among other things, the bank agent (`ba2`) of the `DrinksDispenser` context will move to the `Bank` context to receive the money sent by the mobile phone.

Something else we thought was important (for security reasons) is that the mobile phone (owner) should not ask the bank agent for a transfer of money from an account `i` to an account `j`. Instead, we want the mobile to ask for a transfer of money to account *j*, and the bank agent coming back to the bank will automatically ask for a transfer from its origin's account (which is the entity which has initially ordered the transfer).

Finally, a mobile owner should not be able to order a drink from the dispenser if he is not in the dispenser zone, i.e. if he is not discovered. This is done the following way: The dispenser only sends one of its agents to the contexts which have called its `discovered` method. Conversely, once it has ordered a drink, it must be able to order as many drinks as it wants: We need the dispenser agent to come back to the `MobilePhone` context once a first buying transaction is finished. This is handled by the **`lend/borrow`** mechanism.

3.5 Subtyping/Inheritance and extensibility

Let us briefly recall the meaning of subtyping and inheritance in CO-OPN. **`Subtype`**, in CO-OPN is a keyword indicating strong subtyping (like in Liskov [10]) of class w.r.t. another class. **`Inherits`** is the keyword indicating we are *inheriting* from another class or context. Inheritance in CO-OPN is purely syntactical. It corresponds to a simple copy and paste of the superclass description. The technical details may be found in [1].

During the design/modelling phases, we have often chosen to use subtyping and inheritance mechanisms; The reason for this may be unclear to the reader, as we only have, for example, one kind of item (drink), or one kind of commercial agent (drinks dispenser agent). The motivation is extensibility.

In deed, with the way we have modelled our system, we may add as many product dispensers as we like, without the need to change anything. The only constraints are that a dispenser must inherit from the `AbstractContext` context, and send a commercial agent upon any call of its `discovered` method. This agent must allow a customer to select a product with a given price. The products need only be a subsort of `Item`.

We can also add any number of mobile phones in our system, and as long as they are connected to the bank and have an account there, they will be able to order products from the different dispensers. We then

reach the original complexity of our example as described in section 2.

We may also want to replace the bank agent by a new one which knows if an account is empty -and thus blocked- and tells the mobile phone, for instance, that a product request transaction fails without going back to the bank. The only requirement then is to have the new agent fulfill all the old agent's roles.

Going further, we may want to replace a component *A* with a new component *B* which behaves identically «under certain circumstances». By this, we mean that given a property *p*, the old system (with *A*) satisfies *p* if and only if the new system (with *B*) satisfies *p*.

Let us give a little more detail on this. We feel that many of the existing subtype relations are either too strong and difficult to apply and prove (such as bisimulation and strong subtyping a la Liskov [14]) or too weak and not semantically meaningful, with no guarantee (such as syntactical relations closer to inheritance). Our idea is to adapt the notion of subtyping to each situation where we need to replace a component by another one, and guarantee *something*. So we want to check whether the new component still satisfies a property under certain restrictions of use.

Our notion of subtyping is based on what we call *observation*. The idea is that we «observe» a component *o* through another predefined component: This predefined component, called the «observer» component, is really a way for us to examine only a subset of component *o*'s transition system. Indeed, we look at the composition of the observer and *o* (see figure 10).

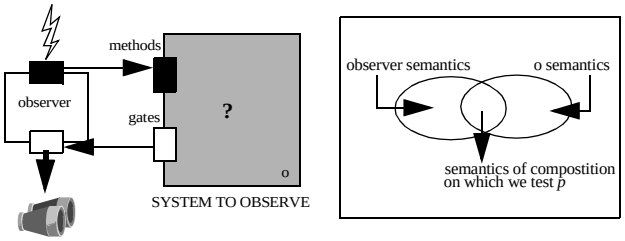

Figure 10: The concept of observation

We believe the observation process is valid because the observation system's semantics is included in the original system's semantics: the observer component does not add or modify behaviours of the system. From the latter, we design an observation system which is a pure abstraction of the original system. We are interested in proving a property *p* (for example, «if money is taken from my account, then I eventually get a drink») included in a set of properties *P*. So we first give a formal abstraction of this property *Abs(p)* (for example "*t* not fireable") and prove that the observation system satisfies *Abs(p)*. Our claim is that the system then satisfies *p* (Please note that in practice, we do not formally verify the abstraction correctness, but try to find several arguments in its favour during the verification process design.)

So basically, we say that component *s* is a subtype of component *t* w.r.t. observer *obs* and property *p* iff:

$$(obs \oplus s) \models p \Leftrightarrow (obs \oplus t) \models p ,$$

where $\models$ denotes satisfaction and $\oplus$ denotes composition.

This is all very informal, and our intent here is only to give the reader an intuition of the concepts we manipulate; The technical details may be found in [11].

Let us give an example: Suppose we want to replace the drinks dispenser by a new one which keeps track of what each customer has been buying it. All of its agents have that list with them, and will automatically deliver a particular drink upon selection of the dispenser once this drink has been ordered ten times in a row by the mobile phone owner.

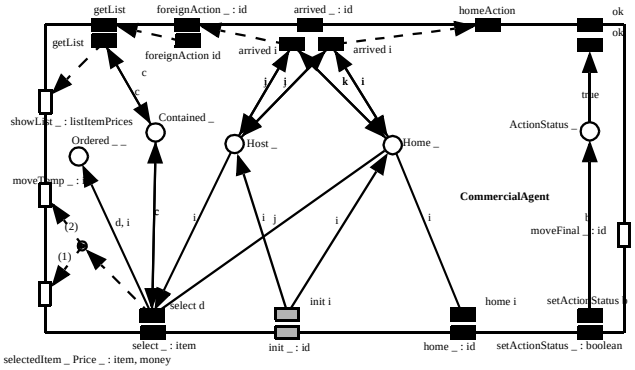

Figure 11: The CommercialAgent class

We can see the classes on figure 11 and figure 12 (the new agents class which «remember» the buying history is called `CommercialAgent2`).

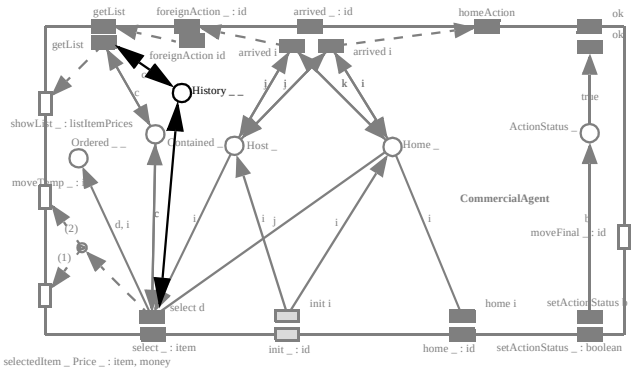

Figure 12: The CommercialAgent2 class

As we can see, the main structural difference is the place `History _ _` which contains the identities of the buyers and the products bought.

We can now look at two subtyping relations, with two properties.

Let us look at the property «if a mobile phone owner actually pays some money, he eventually receives a drink». The observer component we use might be something like the one in figure 13, because we are only interested in the behaviours following a select drink (we do not need the methods and gates used to show the list). This is observer is fairly simple because the only behavioural restriction we are looking at is a signature restriction.

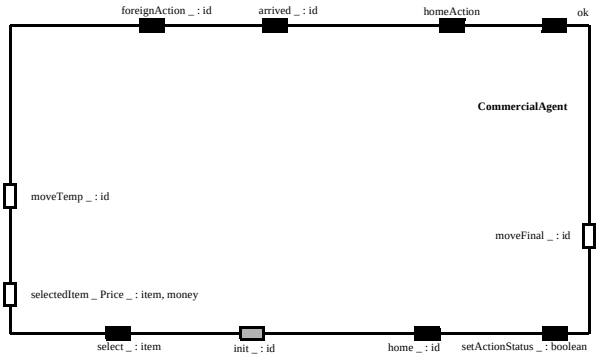

Figure 13: Observer for subtype relation 1

In this case the second agent is a subtype of the first one. Indeed, the history process does not affect the security issue presented in the previous section.

However, if the subtyping relation we look at is something like bisimulation (i.e. the property is the conjunction of all CTL properties satisafied by the first commercial agent), and the observer is one which does not filter any behaviour (a «trivial» observer, if you wish), then the new agent is not a subtype to the first one, because in one case the mobile phone owner needs to select a drink after choosing the dispenser, and in the other case, he might not need choose any, if he has been buying a same drink for a long time: A difference in trace appears after 10 successive «select drinks dispenser - select beer»; in one case, «select drinks dispenser» automatically starts a beer order, and in the other, the agent still waits for a «select drink» call.

So we see that we have defined a very flexible definition of subtyping: depending on our needs, we may find that a component is suitable for substitution or not. We can define a subtype relation for every situation where we might be interested in replacing a component.

4. Validation by prototyping

Prototyping (i.e. generation of executable code, see [8][12][13]) is used to validate our specification. The generated code is a tool to either simulate the specification or prototypically implement a modeled system. The former is achieved using the interpreter tool while the latter needs drivers or user interfaces in order to manage human communication, third-party component interaction or hardware control.

4.1 The prototype interpreter tool

The interpreter tool executes CO-OPN synchronizations using generated code and translates responses back to CO-OPN notation. In the case of our example, it is possible to synchronize with the methods of the `DrinksDispenser`, `MobilePhone`, and `Bank` contexts. The execution of the synchronization results in success or failure. In case of success, the events emitted (via gates) during the synchronization will be exhibited. The Interpreter Tool also features a built-in debugger which enables step-by-step execution of queries, and exhibits the derivation tree during the query.

4.2 Modularity and configurability of the code generator

One of the cases where the code generator's flexibility is crucial is optimization. By default, the generated code uses rewriting techniques to evaluate terms. But this might be costly in terms of time and space. Therefore, we have made the code generator flexible enough to allow users to improve the efficiency of

evaluation by re-configuring the default code-generation algorithm.

For example, the evaluation of a specification often makes intensive use of numerical calculations. In this case, term representation of numbers with the `zero` and `successor` operations is not suitable for efficient evaluation. It is more interesting to represent numerical values occurring in specifications by numerical types of the target language - for example, the *int* type in Java. Our code generator can do that by allowing the choice of the appropriate specific code-generation strategy: the user may choose which representation of numbers he wants it to use.

Our generator allows the user to choose the strategy of code-generation for an individual module, a group of modules or an entire specification. Of course, the choice of code-generation strategies not only addresses data representation alternatives. More generally, this technique allows various kinds of optimizations and code instrumentations (for example, to allow debugging).

For ease-of-use purposes, a choice of pre-defined strategies for different kinds of modules (including standard library modules) is featured in the tool.

4.3 Integration of generated code

As stated above, another possible use for the generated code is integration into an application. Generated code can be integrated both as a server (you can call it), or as a client (it will call your code). The specifics of the CO-OPN specification language imply that the generated code has to implement the non-determinism and transactional semantics of specifications. The user has the choice to either hide those aspects inside the generated code or use them for finer integration. For more details on how to handle non-determinism and transactional failures in non-reversible libraries, see [12].

5. Implementation of migration

In order to implement the migration of objects, we have to carefully manage references, and differentiate references of local and non-local objects (see figure 14). We use the classical Proxy mechanism to obtain an homogeneous access to objects. This indirect reference will present local and non-local objects to clients in a similar way. Usually, the Proxy just forwards synchronizations to the real object. In the case of an already moved object, the Proxy will always respond to inquiries by a failure.

The list of objects known by a context is managed in the Known Objects Table (KOT). In the case where an object returns back to a context, no new entries will be added to the KOT. Instead, the existing Proxy will be found and linked to the returned object. This general mechanism satisfies both centralized and distributed implementations.

The described features are already supported in our current centralized implementation. One of our present research goals it to generate distributed code for such systems. We plan to implement it using Java Remote Method Invocation mechanism.

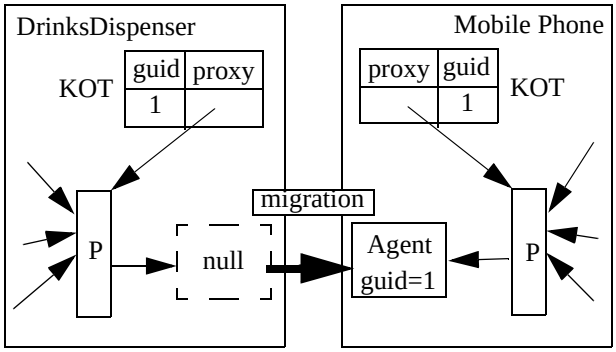

Figure 14: Implementation of object migration. (P: proxy, O: object, KOT: known objects table of the context, GUID: globally unique id associated to object)

6. Future work and conclusions

In this paper, we have presented a formal framework which allows to model mobile distributed systems based on a transactional and concurrent semantics. We have also shown how we may extend our model, by using subtyping and inheritance mechanisms. We have explained how to automatically generate the code from the resulting specification through the use of proxies, and how we may configure this code generation. Finally, we have given some detail on how our code can either incorporate some existing libraries or be incorporated in an application.

We have oriented our future work in many directions: first of all, we are working on yet another step towards ubiquitous programming, which would be to also allow CO-OPN contexts to move, instead of just objects. We believe that this way we will be able to model any mobile system. Along with this, we would like to generate Java code to cope with such migration.

Another of our goals is to extend our notion of subtyping, making it more flexible. Some work has already been conducted on this [11], and we now need to include it in the CO-OPN semantics. We are also working on the verification of such relations, and would like to include a type-checking tool to Coopn-Builder, our tool which can be downloaded at [9].

References

[1] Didier Buchs and Nicolas Guelfi, ``A Formal Specification Framework for Object-Oriented Distributed Systems,'' IEEE TSE, vol. 26, no. 7, July 2000, pp. 635-652.

[2] Olivier Biberstein, Didier Buchs, and Nicolas Guelfi. Object-oriented nets with algebraic specifications: The CO-OPN/2 formalism. In G. Agha and F. De Cindio and G.Rozenberg, editors, *Advances in Petri Nets on Object-Orientation*, LNCS. Springer-Verlag, 2001.

[3] Didier Buchs and Mathieu Buffo. Rapid prototyping of formally modelled distributed systems. In Frances M. Titsworth, editor, *Proc. of the Tenth International Workshop on Rapid System Prototyping RSP'99*. IEEE, june 1999.

[4] Mathieu Buffo. Experiences in coordination programming. In *Proc. of the workshops of DEXA'98 (Int. Conf. on Database and Expert Systems Applications)*. IEEE , aug 1998.

[5] Olivier Biberstein and Didier Buchs. Structured algebraic nets with object-orientation. In *Proc.of the first int. workshop on "Object-Oriented Programming and Models of Concurrency" within the 16th Int. Conf. on Application and Theory of Petri Nets*, Torino, Italy, June 26-30 1995.

[6] Mathieu Buffo and Didier Buchs. A coordination model for distributed object systems. In *Proc. of the Second Int. Conf. on Coordination Models and Languages COORDINATION'97*, vol. 1282 of *LNCS*, pp. 410–413. Springer , 1997.

[7] Jeff Kramer, Jeff Magee, Morris Sloman, and Naranker Dulay. Configuring object-based distributed programs in rex. *IEEE Software Engineering Journal*, 7(2):139–149, 1992.

[8] Stanislav Chachkov and Didier Buchs, ``From an Abstract Object-Oriented Model to a Ready-to-Use Embedded System Controller,'' Rapid System Prototyping, Monterey, CA, IEEE Computer Society Press, June 2001, pp. 142-148.

[9] CO-OPN tools are available at http://lglwww.epfl.ch/Conform/CoopnTools/

[10] B. Liskov and J. M. Wing, A behavioral notion of subtyping, ACM Transaction on Programming Languages and Systems, 16(6):1811--1841, November 1994.

[11] S. Costa, D. Buchs, D. Hurzeler, "Observers for substitutability in CO-OPN", EPFL, technical report n°IC/2002/043, june 2002.

[12] Stanislav Chachkov and Didier Buchs, ``Interfacing Software Libraries from Non-deterministic Prototypes,'' International Workshop on Rapid System Prototyping, July 1-3, 2002, Darmstadt, Germany

[13] Stanislav Chachkov and Didier Buchs, ``From Formal Specifications to Ready-to-Use Software Components: The Concurrent Object-Oriented Petri Net Approach,'' International Conference on Application of Concurrency to System Design, Newcastle, IEEE Computer Society Press, June 2001, pp. 99-110.

[14] B. Liskov and J. M. Wing, A behavioral notion of subtyping, ACM Transaction on Programming Languages and Systems, 16(6):1811--1841, November 1994.

Full Specification

ADTs

```
Adt Id;
Interface
    Sort
        id;
    Generators
        dbuks : -> id;
        stas : -> id;
        zelele : -> id;
        iddd : -> id;
        idbank : -> id;
        idmobile : -> id;
        idabstract : -> id;
End Id;
```

```
Adt Item;
Interface
    Sort
        item;
End Item;
```

```
Adt Money;
Inherit Naturals;
    Rename
        natural -> money;
End Money;
```

In what follows, the List and the Pair packages are prespecified in the cfc CO-OPN library.

```
Adt ListDrinkPrices As List(Pair(Drink,Money));
Rename
    list -> listDrinkPrices;
End ListDrinkPrices;
```

```
Adt ListItemPrices As List(Pair(Item,Money));
Rename
    list -> listItemPrices;
Interface
    Operation
        _ _ isInside _ : item money listItemPrices -
> boolean;
Body
    Axioms
        i m isInside [ ] = false;
        i m isInside < i m > ' l = true;
        ! (< i m > = < i2 m2 >) => i m isInside < i2 m2
> ' l = i m isInside l;
    Where
        i : item;
        m : money;
        i2 : item;
        m2 : money;
        l : listItemPrices;
End ListItemPrices;
```

```
Parameter Adt AbstractId;
Interface
    Use
        Id;
    Generator
        AG : -> id;
End AbstractId;
```

Classes

```
Class Account;
Interface
    Use
        Money;
        Id;
    Type
        account;
    Methods
        getId _ : id;
        withdraw _ : money;
        credit _ : money;
    Creation
        createAccount _ _ : id money;
Body
    Places
        Identity _ : id;
        Savings _ : money;
    Axioms
        getId i:: Identity i : ->;
        (n > = m) = true => withdraw m:: Savings n -
> Savings (n - m);
        credit m:: Savings n -> Savings n + m;
        createAccount i m:: -> Identity i, Savings m;
    Where
        i : id;
        m : money;
        n : money;
End Account;
```

```
Class Queue;
Interface
    Use
        Agent;
        Id;
    Type
        queue;
    Gate
        send _ _ : agent id;
    Methods
        add _ _ : agent id;
        moveAll;
Body
    Use
        Naturals;
    Method
        move;
    Places
        Q _ _ : agent id;
        count _ : natural;
    Initial
        count 0;
    Axioms
        add a i:: -> Q a i;
        (this = Self) => moveAll With this . move . . th
is . moveAll:: ->;
        move With this . send a i:: Q a i, count succ n
-> count n;
        move:: count 0 -> count 0;
    Where
        a : agent;
        n : natural;
        this : queue;
        i : id;
End Queue;
```

```
Class AutomaticQueue;
Interface
    Use
        Agent;
        Id;
    Type
        automaticQueue;
    Gate
        send _ _ : agent id;
    Method
        add _ _ : agent id;
Body
    Transition
        send;
    Place
        Q _ _ : agent id;
    Axioms
        add a i:: -> Q a i;
        this = Self => send With this . send a i:: Q a i
 ->;
    Where
        a : agent;
        i : id;
        this : automaticQueue;
End AutomaticQueue;
```

```
Class Agent;
Interface
    Use
        Id;
    Type
        agent;
    Gates
        moveTemp _ : id;
        moveFinal _ : id;
    Methods
        arrived _ : id;
        home _ : id;

        ok;
    Creation
        init _ : id;
Body
    Use
        Booleans;
    Methods
        homeAction;
        foreignAction _ : id;
        setActionStatus _ : boolean;
    Places
        Host _ : id;
        Home _ : id;
        ActionStatus _ : boolean;
    Axioms
        (this = Self) => arrived i With this . homeActio
n:: Host j, Home i -> Host i, Home i;
        ! (k = i), (this = Self) => arrived i With this
. foreignAction i:: Host j, Home k -> Host i, Home k;
        init i:: -> Home i, Host i;
        home i:: Home i : ->;
        setActionStatus b:: -> ActionStatus b;
        ok:: ActionStatus true ->;
    Where
        j : id;
        i : id;
        k : id;
        this : agent;
        b : boolean;
End Agent;
```

```
Class BankAgent;
Inherit Agent;
    Rename
        agent -> bankAgent;
Interface
    Use
        Agent;
        Id;
        Money;
    Type
        bankAgent;
    Subtype
        bankAgent -> agent;
    Gates
        sendTellBank _ from _ to _ : money id id;
        receiveTellBank _ from _ to _ : money id id;
    Methods
        send _ to _ : money id;
        receive _ from _ : money id;
    Creation
        initHome _ Owner _ : id id;
Body
    Places
        OwnerId _ : id;
        SendOrder _ to _ : money id;
        ReceiveOrder _ from _ : money id;
    Axioms
        send m to j With this . moveTemp i:: Home i : -
> SendOrder m to j;
        receive m from j With this . moveTemp i:: Home i
 : -> ReceiveOrder m from j;
        this = Self => homeAction With this . sendTellBa
nk m from i to j / /
 this . setActionStatus true:: OwnerId i : SendOrder m t
o j ->;
        this = Self => homeAction With this . receiveTel
lBank m from i to j / /
 this . setActionStatus true::  OwnerId j : ReceiveOrder
 m from i ->;
    Where
        m : money;
        i : id;
        j : id;
        this : bankAgent;
End BankAgent;
```

```
Class CommercialAgent;
Inherit Agent;
    Rename
        agent -> commercialAgent;
Interface
    Use
        Agent;
        ListItemPrices;
        Item;
    Type
        commercialAgent;
    Subtype
        commercialAgent -> agent;
    Gates
        showList _ : listItemPrices;
        selectedItem _ Price _ : item money;
    Methods
        select _ : item;
        getList;
Body
    Places
        Contained _ : listItemPrices;
        Ordered _ _ : item id;
    Axioms
        this = Self => getList With this . showList c::
Contained c -> Contained c;

        this = Self, ((d m isInside c) = true) => select
 d With (this . selectedItem d Price m) . . (this . move
Temp j):: Home j : Host i, Contained c -
> Ordered d i, Contained c;
```

```
        this = Self => foreignAction id With this . getL
ist:: ->;
    Where
        c : listItemPrices;
        this : commercialAgent;
        d : item;
        m : money;
        i : item;
        j : item;
        id : id;
End CommercialAgent;
```

```
Class DrinksContainer;
Interface
    Use
        Drink;
    Type
        dc;
    Methods
        addDrink _ : drink;
        dispenseDrink _ : drink;
    Creation
        init _ : drink;
Body
    Places
        drinks _ : drink;
        kind _ : drink;
    Axioms
        addDrink d:: kind d -> kind d, drinks d;
        dispenseDrink d:: drinks d ->;
        init d:: -> kind d;
    Where
        d : drink;
End DrinksContainer;
```

```
Class DDCentralUnit;
Interface
    Use
        Drink;
        Money;
        Id;
    Type
        dDCentralUnit;
    Gates

        takeMoney _ from _ : money id;
        displayId _ : id;
        deliver _ : drink;
    Methods
        distribute _ _ : drink id;
        addContainer _ price _ : drink money;
        addDrink _ : drink;
Body
    Use
        DrinksContainer;
    Place
        container _ price _ : dc money;
    Axioms
        this = Self => distribute d i With ((this . take
Money p from i) . . (c . dispenseDrink d)) . . ((this .
displayId i) / /
 (this . deliver d)):: container c price p -
> container c price p;
        addContainer d price p With c . init d:: -
> container c price p;
        addDrink d With c . addDrink d:: container c pri
ce p -> container c price p;
    Where
        this : dDCentralUnit;
        p : money;
        c : dc;
        d : drink;
        i : id;
End DDCentralUnit;
```

```
Class VirtualShop;
Interface
    Use
        Id;
        CommercialAgent;
    Type
        virtualShop;
    Methods
        selectAgent _ : id;
        addAgent _ _ : commercialAgent id;
        removeAgent _ : id;
Body
    Use
        Naturals;
    Places
        CommercialAgents _ _ : commercialAgent id;
        SelectedAgent _ _ : commercialAgent id;
        count _ : natural;
    Initial
        count 0;
    Axioms
        addAgent ca i:: -> CommercialAgents ca i;
        removeAgent i:: CommercialAgents ca i ->;
        selectAgent i With a . getList:: CommercialAgent
s a i, count 0 -
> CommercialAgents a i, SelectedAgent a i, count 1;
        selectAgent i With a . getList:: CommercialAgent
s a i, count 1, SelectedAgent a2 i2 -
> CommercialAgents a i, SelectedAgent a i, count 1;
    Where
        ca : commercialAgent;
        a : commercialAgent;
        i : id;
        this : virtualShop;
        a2 : commercialAgent;
        i2 : id;
End VirtualShop;
```

```
Class WorldConnector;
Interface
    Use
        Agent;
        Id;
    Type

        worldConnector;
    Gates
        sendFinal _ from _ to _ : agent, id id;
        sendTemp _ from _ to _ : agent, id id;
    Methods
        receiveTemp _ from _ to _ : agent, id id;
        receiveFinal _ from _ to _ : agent, id id;
Body
    Axioms
        this = Self => receiveTemp a from i to j With th
is . sendTemp a from i to j:: ->;
        this = Self => receiveFinal a from i to j With t
his . sendFinal a from i to j:: ->;
    Where
        a : agent;
        i : id;
        j : id;
        this : worldConnector;
End WorldConnector;
```

Contexts

```
Parameter Abstract Context AbstractContext;
Interface
    Use
        Agent;
        Id;
    Gates
        sendFinal _ _ : Give agent, id;
        sendTemp _ _ : Lend agent, id;
    Methods
        receiveFinal _ _ : Take agent, id;
        receiveTemp _ _ : Borrow agent, id;
        discovered _ : id;
Body
    Use
        AutomaticQueue;
    Objects
        qIn : automaticQueue;
        qOutFinal : automaticQueue;
        qOutTemp : automaticQueue;
    Axioms
        a . moveFinal i With qOutFinal . add a i;
        a . moveTemp i With qOutTemp . add a i;
        qOutFinal . send a i With sendFinal a i;
        qOutTemp . send a i With sendTemp a i;
        receiveFinal a i With qIn . add a i;
        receiveTemp a i With qIn . add a i;
    Where
        a : agent;
        i : id;
End AbstractContext;
```

```
Context Bank;
Inherit AbstractContext;
Interface
    Use
        Id;
        Money;
        Account;
    Method
        createAccount _ _ : id money;
Body
    Use
        BankAgent;
        BTransferManager;
    Method
        transfer _ from _ to _ : money id id;
    Object
        BTM : bTransferManager;
    Axioms

        discovered i With (ba . initHome idbank Owner i)
 . . (qOutFinal . add ba i);
        createAccount i m With a . createAccount i m;

        qIn . send a i With a . arrived idbank;
        ba . sendTellBank m from i to j With (BTM . send
 m from i to j) / / ((a . getId i) / /
 (a . withdraw m));

        ba . receiveTellBank m from i to j With (BTM . r
eceive m from i to j) / / ((a . getId j) / /
 (a . credit m));
    Where
        a : account;
        m : money;
        i : id;
        ba : bankAgent;
        j : id;
End Bank;
```

```
Context MobilePhone;
Inherit AbstractContext;
Interface
    Use
        Id;
        Item;
        Money;
        ListItemPrices;
        VirtualShop;
    Gate
        displayChoice _ : listItemPrices;
    Methods
        select _ : item;
        selectService _ : id;
Body
    Use
        CommercialAgent;
        BankAgent;
    Object
        vs : virtualShop;
    Axioms
        select d With ca . select d . . ca . ok;
        ca . showList c With displayChoice c;
        selectService i With vs . selectAgent i;
        qIn . send b i With b . arrived idmobile;
        qIn . send ca i With vs . addAgent ca i / /
 ca . arrived idmobile;
        ca . selectedItem d Price m With ca . home i . .
 (b . send m to i . . b . ok);
    Where
        d : item;
        m : money;
        a : agent;
        b : bankAgent;
        ca : commercialAgent;
        c : listItemPrices;
        i : id;
        j : id;
End MobilePhone;
```

```
Context DrinksDispenser;
Inherit AbstractContext;
Interface
    Use
        Id;
        Drink;
        Money;
    Gates
        displayId _ : id;
        giveDrink _ : drink;
    Methods
        addContainer _ price _ : drink money;
        addDrink _ : drink;
Body
    Use
        DDCentralUnit;
        DDagent;
        BankAgent;
    Object
        cu : dDCentralUnit;
    Axioms
        addContainer d price m With cu . addContainer d
price m;
        addDrink d With cu . addDrink d;
        cu . displayId i With displayId i;
        qIn . send a i With a . arrived iddd;
        discovered i With (da . init iddd) . . (qOutFina
l . add da i);
        cu . deliver d With giveDrink d;
        cu . takeMoney m from i With b . receive m from
i;
        da . distribute d i With cu . distribute d i;
    Where
        d : drink;
        m : money;
        i : id;
        a : agent;
        da : dDagent;
```

```
        b : bankAgent;
End DrinksDispenser;
```

```
Context World;
Inherit WorldSegment(DrinksDispenser,Id):MorfDD;
Inherit WorldSegment(Bank,Id):MorfBank;
Inherit WorldSegment(MobilePhone,Id):MorfMobile;
End World;
```

```
Generic Context WorldSegment (AbstractContext,AbstractId
);
Body
    Use
        WorldConnector;
        Agent;
        Id;
        AbstractId;
    Use Context
        AbstractContext;
    Object
        wc : worldConnector;
    Axioms
        sendTemp In AbstractContext a i With wc . receiv
eTemp a from AG to i;
        wc . sendTemp a from i to AG With receiveTemp In
 AbstractContext a i;
    Where
        i : id;
        j : id;
        a : agent;
End WorldSegment;
```

```
Context BankSegment As WorldSegmentId(Bank,Id);
Morphism
    AG -> idbank;
End BankSegment;
```

Morphisms

```
Morphism MorfBank;
Morphism
    AG -> idbank;
End MorfBank;
```

```
Morphism MorfDD;
Morphism
    AG -> iddd;
End MorfDD;
```

```
Morphism MorfMobile;
Morphism
    AG -> idmobile;
End MorfMobile;
```