Contextual Coordination between Objects*

MATHIEU BUFFO [1]
DIDIER BUCHS [2]

[1] CUI, Université de Genève
24, rue du Général Dufour
1211 Genève 4, SWITZERLAND
buffo@cui.unige.ch

[2] EPFL-LGL-DI
Swiss Federal Institute of Technology
1015 Lausanne, SWITZERLAND
buchs@di.epfl.ch

Abstract

Notwithstanding the promises it provides for the future of software engineering, the object paradigm suffers from a lack of development methods focusing on the coordination between objects. The distinction of coordination and computation mechanisms is an important separation principle for increasing the management and understanding of the interactions among objects and the configurations of objects on the target systems. This paper shows that contextual coordination must be introduced during the object oriented analysis in order to early take into account the inevitable constraints of the concrete implementation. This contextual coordination leads to a hierarchical model composed of imbricated execution contexts of objects.

KEY WORDS: Coordination, Object-Oriented Systems, Software Engineering, Formal Methods, Distributed Systems.

1 Introduction

From a software engineering point of view, general purpose software development can be divided into three main phases, namely analysis, design and implementation. Object-oriented methodologies influence these three phases, leading to so-called *object-oriented analysis or specification*, *object-oriented design* and *object-oriented implementation* [1]. Moreover, in such a framework, the essence of the design phase is continuous *transformation* or *refinement* [9]. The problem of implementing the abstract specification is then to find a correct sequence of design steps that brings the specification more and more close to the constraint of the target system. This is made by the development of a global architecture of the system [10], by means of a refinement of the relationships between the specification's objects.

* This work has been sponsored partially by the Esprit Long Term Research Project 20072 «Design for Validation» (DeVa) with the financial support of the OFES (Office Fédéral de l'Education et de la Science), and by the Swiss National Science Foundation project 2000.40583.94 «Formal Methods for Concurrency».

The relationships between objects are know under the name of *coordination* in computer science. More precisely, the coordination is the process of managing dependencies among activities [13]. An interesting property of the coordination is that it can be view as an orthogonal concept to the computation; computational features and coordination features should be described independently each other [8]. The separation of computation and coordination is often concretized in the choice of different languages and in the choice of different levels of description. Moreover, it should be possible to split programs in pure computational software entities and in pure coordination entities, with the same properties of reuse, inheritance, substitution and dynamicity as the traditional objects (generally encompassing both computation and coordination) [2]. As an illustration, imagine a process farm, managed by a controller, as shown in figure 1. The workers are pure computational software entities, and the controller is a pure coordination entity. In this situation there is a natural separation between the coordination and computation principles. It must be noted that it is not always the case, and that considerable efforts can be needed in certain situations in order to clearly perform this separation. Moreover, the workers as well as the controller should be reusable in a well designed analysis method.

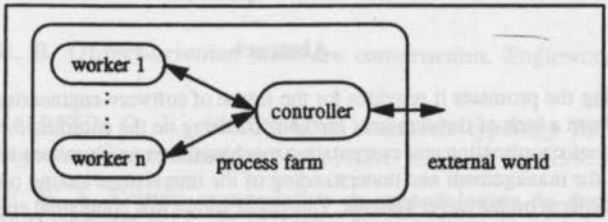

Figure 1 : a process farm

Coordination consist in interactions and configurations. Interactions, also called synchronizations, are the links between the entities composing a coordinated system. Configurations are the structural relations between the entities. For instance, consider the figure 1 again. The configuration resides in the fact that a process farm is composed of one controller and many workers, encapsulated into the farm. The interactions are the links between the components of the farm and between the farm and the external world.

From a technical point of view, general coordination models can be generally divided into three classes.

- The mathematical models describe coordination by means of abstract terms, as CCS [14], the π-calculus [15] or the so-called chemical machines (like CAM [4]). These models are generally used as an underlying layer by the operational and structural models, for the formal definition of their semantics.
- The operational models are focusing on the operations required by the coordination, like Linda or PVM. These models are often used for implementing concurrent activities in sequential programming languages.
- Structural models, as ToolBus [3] and Darwin [12], focus on the relationships between the components taking part to the coordination. A coordinated system is depicted as a set of abstract entities and by some relations between these entities. As a result, a coordination structure is mapped on the system.

Thus, from a software engineering point of view, structural coordination models seem to be well adapted for describing relationships between objects in a software system.

Unfortunately, from a software engineering point of view, current structural object coordination suffers from some weaknesses. In particular, the use of traditional object coordination leads to the following problems:

- objects reuse is hard to achieve as objects are charged of information about their actual execution context,
- formal framework for object composition and grouping is missing, preventing the structuring of the problem into sub-problems,
- synthetic genesis of software systems promoted by usual object-oriented techniques are favoured with respect to analytic decompositions, despite of the fact that analytic approaches are close to the natural way of thinking.

This paper shows that an adequate coordination model, based on a hierarchically structured coordination, allows to solve these problems in a simple and elegant way.

The next section describe structured coordination using contexts, and shows how such kind of coordination solves the traditional object coordination weaknesses. Then section 3 present a coordination language based on hierarchical coordination using contexts, namely COIL, implementing the concepts introduced in the previous section. Related works are discussed in section 4. Finally, section 5 concludes this paper.

2 Contextual Coordination

Contextual coordination leads to natural models with structured coordination. In such models, computation entities are modelled using active objects, while coordination entities are structured according to the notion of context.

2.1 What is a context

Following the Webster's New Encyclopedic Dictionary [17], a context is *the circumstances surrounding an act or event*. From a computer scientist point of view, a context is an execution environment encompassing a computing unit.
Modelling using contexts and objects tends to encompass the computing units, namely the active objects, of a software system into their execution context. Therefore, this technique provides context-abstraction to the object, making them pure computational units, and likewise it provides computation-abstraction to the contexts, making them pure coordination units.

2.2 Contexts and objects

The main difference between contexts and objects is that a context is a complex software entity that encaspulates actions while an object is a basic software entity which models actions.
Objects are able to perform actions and computations as they are active entities. Objects can manipulate other entities belonging to the software system, as well as they can be manipulated by other entities. Objects can establish dynamic interaction links to other entities, by means of traditional message sending. Therefore, objects are bound with dynamic and reflexive elements of a software system.

components and connections are drawn using dotted lines.

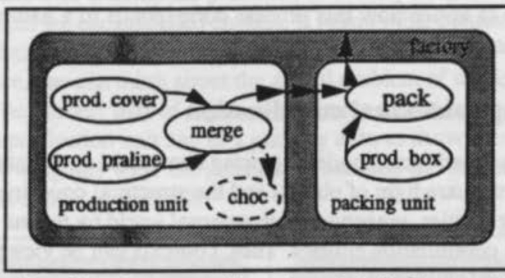

Figure 3 : chocolate factory with a chocolate in the production unit

An interesting fact is that static components and static connections can be considered as forming the basic structure of the context hierarchy, while dynamic components and links are evolving inside this structure.

2.5 Contexts modifications

It was mentioned in section 2.2 that contexts are passive entities not manipulating other entities. Hence contexts can not manipulate their contents. Nevertheless, contexts can be manipulated by dynamic entities as objects. This leads to the notion of context modifications.
Contexts modifications consist of the moving of the dynamic components from one context to another one. For instance, keeping the same example, it is reasonable to think that the component "choc" will move to the context "packing unit" in order to be packed, as shown in figure 4.

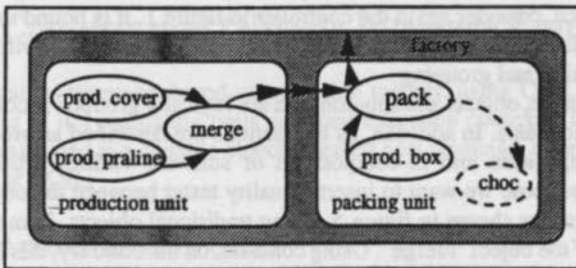

Figure 4 : chocolate factory with a chocolate in the packing unit

2.6 Contexts encapsulation

Contexts do not only coordinate their components, they also hide them from the external world. Indeed, contexts are encaspulated entities, and their components are protected from the outside world. In other words, dynamic components and links are local to their context, and external interactions can only take place through static connections.
For instance, consider the object "choc" in figure 3. This object cannot be reached by the object "pack", because both objects are not statically linked and they are not belonging to the same context. Therefore, "choc" must migrate to the packing unit in order to be handled by "pack", as shown

Conversely, contexts are passive entities. Contexts are not modelling actions, but are modelling the coordination of their encompassed actions. Thus, objects can be manipulated, but can not manipulate other software entities, being totally passive. Contexts hold static interaction links, linking static encompassed entities.
In fact, objects and context are complementary to each other. Using contexts, the external world is encapsulated for objects, just as -following a traditional point of view- objects are encapsulated for the external world.

2.3 Contexts hierarchy

Contexts are encompassing actions, and an action can be described by an active object. However, an action can also be expressed as a context encapsulating sub-actions. Thus, contexts contain objects and sub-contexts, and this leads to the notion of contexts hierarchy. Moreover, a context must contain the connections between its components and itself.
In other words, contexts are composed of components and connections. *Components* are either objects or contexts. *Connections* are linking components and their context. Finally, a software system is a hierarchy of contexts.
Figure 2 shows a system modelling a chocolate factory. The factory consists in two contexts, each of them modelling a sub-factory. The first context is the producing unit, which care about the production of both covers and pralines, and which merge them together to produce a chocolate. The second context is the packing unit, which packs the chocolates into suitable boxes. This example illustrates the hierarchy of the contexts; "factory" encompass "production unit" and "packing unit", both of them encompassing some objects. Moreover, one can see that connections are respecting the hierarchy.

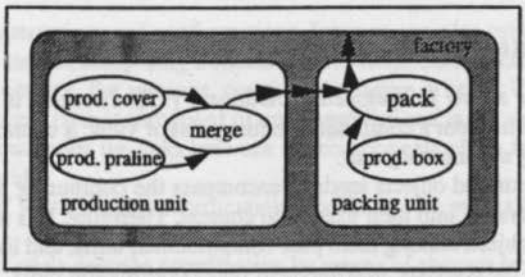

Figure 2 : chocolate factory

2.4 Static and dynamic components

Both components and connections can be static or dynamic. Static elements are described as a property of their encompassing context, and they exist as long as the context exists. For example, in figure 2, "prod. praline" and "merge" are static components. On the contrary, dynamic elements are created and manipulated by the objects of the context; dynamic elements are described as properties of the encompassed objects, using standard object-oriented techniques.
For instance, a chocolate can be viewed as an object dynamically created and managed by the component "merge", as shown in figure 3. In this case, the context "production unit" contains three static components and one dynamic component (namely the object "choc"). By convention, dynamic

in figure 4. During the moving, the chocolate pass through the super-context, actually through the factory. It is remarkable to notice how this process corresponds to a natural intuition of the functioning of a real chocolate factory.

2.7 Modelling using contextual coordination

To summarize the above stated facts, modelling using contextual coordination is a general modelling technique, merging the paradigm of objects and the structural coordination. Objects are active fully dynamic computing entities, preserved from external world by means of encapsulation, while contexts are hierarchical coordination entities. Thus, contexts can be viewed as hierarchical interfaces around the computing entities.

From a software engineering point of view, this modelling technique brings many advantages in addition to its simplicity. In particular, the facilities for object reuse, object composition and system analysis.

- Objects reuse.
  Actually, the major obstacle in object reuse reside in the fact that traditional objects are interconnections laden. Despite of the fact that traditional objects are reached only through well-defined entry-points, namely the methods, which can be considered as input ports abstractions, there is a lack of symmetric output ports abstractions. Therefore objects usually code the synchronizations they require into their core, which obviously prevent object reuse [16]. Conversely, using contexts, objects are able perform abstract synchronizations through output ports, whose connections are depicted by the context; object reuse is facilitated and in many cases just consists in changing the context.
  For instance, consider again the controller in figure 1. It is bound to the workers by its context. Placing it into another context will allow it to be reused without modifications.
- Object composition and grouping.
  Using contexts, objects and sub-contexts are logically grouped according to their structural relationships. In addition, as interactions are described as properties of the contexts, composition and re-composition of software entities arise naturally. As an example, suppose we want to insert a quality tester between the objects "prod. cover" and "merge", as shown in figure 5. Using traditional objects, it implies a rewriting of the core of the object "merge". Using contexts, on the contrary, this adding just implies a modification of the context "production unit"

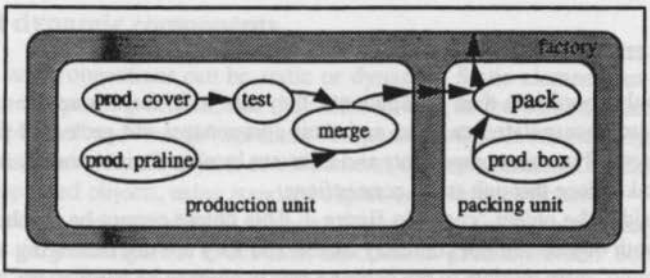

Figure 5 : chocolate factory with quality test

- System analysis.

  Contexts allow to decompose problems in loosely-coupled sub-problems. Moreover, sub-problems are encapsulated entities. It follows that contextual coordination provide a natural framework for system analysis.

  For instance, one can think about the global problem of modelling a chocolate factory, as shown on the left part of figure 6. A simple analysis shows that a factory is composed of a production unit and of a packing unit, as shown on the right part of figure 6. Then, following this way, the production unit and the packing unit are analysed, and the result is shown in figure 2.

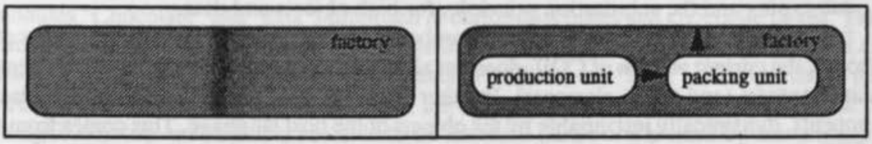

Figure 6 : different points of view on the chocolate factory

Last but not least, from a methodological point of view, this technique allows to unify the concepts of object-oriented system and distributed system. Indeed, the nature of object-oriented systems is similar to that of distributed systems; objects can be viewed as loosely coupled software entities similar to distributed processes, but object granularity is different from process granularity. This prevent a direct mapping between objects and processes. The use of contexts helps to find granularity and localization concepts close both to the nature of the problem and to the target system. But this is out of the scope of this paper [7].

3 COIL (Contexts and Objects Interface Language)

COIL is a coordination language, based on structural models; using COIL, a software system is described by means of it's computing units and their execution contexts. Thus, around computational entities programmed in a suitable object-oriented host language, coordination layers are described by means of COIL.

3.1 The object model used by COIL

COIL was developed around the SANDS system [5] [6]; its syntax is coming from the syntax of the CO-OPN$_{/2}$ (Concurrent Object Oriented Petri Nets) specification language used in SANDS and its semantics is given by translation to CO-OPN$_{/2}$. Despite of this fact, COIL is general enough to be used in many different situations, provided the use of suitable object-oriented host language and model.

Actually, COIL assumes the following requirements about the object model :

- Data are defined by means of abstract data types.
- Objects are *typed* active computing entities.
- Objects have input ports (usually called *methods*) and output ports (usually called *gates*).
- Message passing is realized using *parametrized synchronous* method calls. Despite of the fact that method calls are directional, parameters are exchanged in both direction, through

the use of an *unification* mechanism. Moreover, methods can refuse calls; in this case, the synchronization fails.

This semantics is operationally more complex than usual object oriented models (it is based on nested transaction models) but it seems that we increase considerably the generality of concurrency management that can be found in usual object oriented languages.

Any language which can fulfil these requirements should be able to be a host language for COIL. In particular, the following notions are compatible with COIL:

- active objects (i.e. spontaneous actions of objects),
- dynamic objects (i.e. dynamic creation and management of objects),
- sub-typing and the substitution principle, for both objects and data,
- full concurrency (i.e. at any time, any number of methods can be called any number of time).

Moreover, the current version of COIL does not allow objects (written in the host language) to manipulate contexts (and COIL elements). In other words, there is no notion of typed "classes" of components, dynamically instantiable by the objects of the host language. This comes from the fact that COIL is really a layer encompassing the host language, while the contrary is not true.

3.2 COIL language

The COIL language with it's static semantics is introduced below through the formal modelling of the chocolate factory shown in figure 2. Objects and classes used by this example will be sketched using the CO-OPN$_{/2}$ specification language. The reader is supposed to be familiar with the notion of signature and with the differences between classes and types [5]. Source code of COIL is fragmented into modules, each module corresponding to a context.

Although COIL specifications are written in plain text, an equivalent graphical notation is provided for the convenience of the user.

3.2.1 Basic concepts

Using COIL, a context is described by a module. The module starts with the keyword "Context" followed by its name, and ends with the keyword "End" followed by its name too. Inside the module, two parts are defined, namely the interface part and the body part. The interface part contains actually the interface of the context, and the body part encapsulates the private components of the context. The content of the interface part defines the *exported signature* of the context. The content of both interface and body are defining the *internal signature* of the context. Of course, the interface part must be correctly typed with respect to the exported signature, and the body part must be correctly typed with respect to the internal signature.

As an example, figure 7 sketches the context "PackingUnit". The left part contains the textual notation while the right part depicts the graphical form. Graphically, contexts are represented by smooth rectangles, while objects are represented by ellipses.

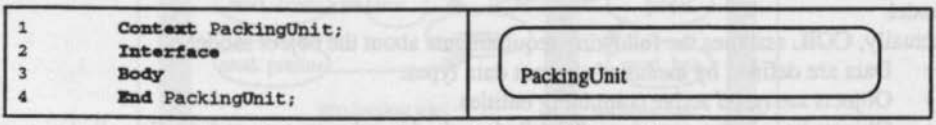

Figure 7 : sketch of the packing unit

3.2.2 The imports

COIL contexts can import the definitions made in another component, through the use sections. A use section in composed of the keyword "Use", followed by the name of the modules to import. Actually, an import is a signature increment. Therefore, there is two types of use sections: the use section of the interface and the use section of the body. The interface's use section augment both exported and internal signatures of the context, by adding the exported signature of the used modules. Likewise, the use section of the body increment the internal signature by the exported signature of the used modules. Imports of contexts are allowed only in the body use section.
For instance, figure 8 sketches the packing unit with its imports. This context uses in its interface the modules "Chocolate" and "Box", supposed to define chocolates and chocolate boxes. Thus every items exported by "Box" are not only accessible inside the context "PackingUnit", but also exported by the context. On the contrary, the classes "ProdBox" and "Pack" are imported in the body part. As a consequence, their exported items are accessible only to the body part of the context "PackingUnit"; they are neither accessible to its interface part, nor exported by the context.

```
1       Context PackingUnit;
2       Interface
3           Use
4               Chocolate, Box;
5       Body
6           Use
7               ProdBox, Pack;
8       End PackingUnit;
```

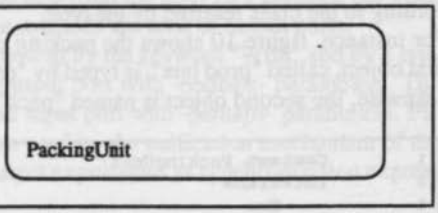

Figure 8 : sketch of the packing unit with the imports

Graphically, imports are usually omitted for sake of clarity.

3.2.3 Ports definitions

The next step in the specification of a context is the definition of its ports. The ports are used by the context to access the external world. Therefore, ports definitions are part of the interface. There is two kinds of port sections; the input ports are defined in a method section and the output ports are defined in a gate section.
Going on with the packing unit, figure 9 adds the ports definitions

```
1       Context PackingUnit;
2       Interface
3           Use
4               Chocolate, Box;
5           Method
6               receive _ : chocolate;
7           Gate
8               deliver _ : box;
9       Body
10          Use
11              ProdBox, Pack;
12      End PackingUnit;
```

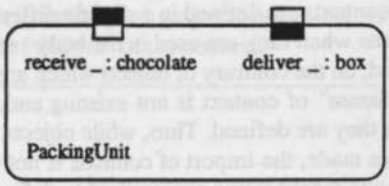

Figure 9 : sketch of the packing unit with the ports

For the packing unit, one method is defined for the input of chocolates, and one for the output of the boxes supposed to contain the chocolates. Reader can remark that the interface is correct with respect to the static semantics, as the two types exploited (namely "chocolate" and "box") are known through imports. Graphically, context ports are described as squares, with one half coloured; for gates, the coloured half is the internal one, and the reverse is true for methods. On the contrary, object ports are described with plain squares; for gates, the colour is white and for methods the colour is black.
Graphically, typing information are usually omitted for sake of clarity.

3.2.4 Sub-components

As mentioned in the previous section, sub-components are either objects or sub-context. COIL handles differently these two kind of sub-components.
Objects are defined in the object section. This section starts with the keyword "Object" (or "Objects"), followed by so-called object blocs terminated by semicolons. An object bloc is a list of names followed by a type. For each name in the list, an object with this name is instantiated, according to the class referred by the type.
For instance, figure 10 shows the packing units with it's components, actually two objects. The first object, called "prod box", is typed by "prodbox" and is an instantiation of the class "ProdBox". Likewise, the second object is named "pack" and comes from the class "Pack".

```
1     Context PackingUnit;
2     Interface
3         Use
4             Chocolate, Box;
5         Method
6             receive _ : chocolate;
7         Gate
8             deliver _ : box;
9     Body
10        Use
11            ProdBox, Pack;
12        Objects
13            prod box : prodbox;
14            pack : pack;
15    End PackingUnit;
```

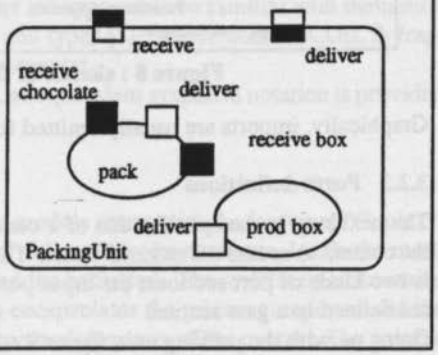

Figure 10 : sketch of the packing unit with its components

Sub-contexts are defined in a slightly different way. Actually, contexts are considered as sub-components when they are used in the body section of the encompassing context.
Indeed, on the contrary of objects which are defined by classes and must be instantiated, the notion of "classes" of context is not existing and, as a consequence, contexts are statically instantiated when they are defined. Thus, while objects must be explicitly instantiated once the import of their classes made, the import of contexts is not followed by any instantiation mechanism and no other syntactic mechanisms are required to define sub-contexts.
As an example of sub-contexts, we can consider a sketch of the context "factory" depicted in figure 2. The body of "factory" imports two modules, namely the context "ProducingUnit" and the context "PackingUnit", as shown in figure 11. Therefore, both modules are sub-components of the factory.

```
1       Context Factory;
2       Interface
3           Use
4               Box;
5           Gate
6               deliver _ : box;
7       Body
8           Use
9               ProductionUnit,
                PackingUnit;
10      End Factory;
```

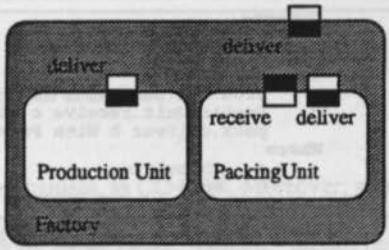

Figure 11 : sketch of the factory

3.2.5 Connections and variables

Once sub-components defined, connections between their ports and with the ports of the context must be described. This is the role of the connection section. Connections are static links between a output and input ports. However, it should be noticed that from the body of a context, the gates of the context are obviously seen as input ports and the methods are seen as output ports.
Each connection is defined as an output expression, followed by the keyword "With" and by a synchronization expression. An output expression is an output port with -perhaps- parameters. The most simple form of a synchronization expression is an input port with -perhaps- parameters. Parameters are in fact typed variables or constants and are used by the unification mechanism of the method calls of the host language. Of course, as well output expressions as synchronization expressions must be correctly typed.
Free variables are declared in an apposite section, immediately following the connection section, and is composed by the keyword "Where" followed by so-called variable blocs. A variable bloc, terminated by a semicolon, is composed by a list of names followed by a type. Each names define a variable with the declared type.
As an example, consider again the packing unit, as shown in listing 1 and figure 12. Two variables are defined, namely "c" with type "chocolate" and b with type "box". Three connections are defined. The first one links the gate "deliver" from the sub-component "prod box" with the method "receive box" of the sub-component "pack"; this connection means that every method calls issued by "deliver" must be sent to "receive box". The second connection links the method "receive" of the context (this method is considered as an output port) with the method "receive chocolate" of the sub-component "pack"; this connection means that every method calls issued through "receive" must be forwarded to "receive chocolate". The third connection link the gate "deliver" of the object "pack" with the gate "deliver" of the context (this gate is considered as an input port); thus every methods calls issued by "deliver" is forwarded to the gate "deliver" of the context.

```
1       Context PackingUnit;
2       Interface
3           Use
4               Chocolate, Box;
5           Method
6               receive _ : chocolate;
7           Gate
8               deliver _ : box;
9       Body
10          Use
11              ProdBox, Pack;
```

Listing 1 : the PackingUnit module

```
12          Objects
13              prod box : prodbox;
14              pack : pack;
15          Connections
16              prod box.deliver b With pack.receive box b;
17              PackingUnit.receive c With pack.receive chocolate c;
18              pack.deliver b With PackingUnit.deliver b;
19          Where
20              c : chocolate;
21              b : box;
22      End PackingUnit;
```

Listing 1 : the PackingUnit module

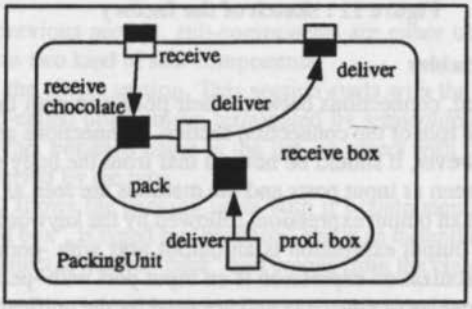

Figure 12 : the PackingUnit module

Synchronization expression can be built using three 2-arguments operations, namely "//", ".." and "+". The operation "//" is the simultaneous operation; the synchronization must occurs simultaneously on two input ports. Likewise, the operation ".." is the sequential operation, and "+" is the alternative operation.

For example, consider a modification of the context "PackingUnit" as shown in listing 2, where we want to count every boxes output by the unit. With respect to the original module, a static sub-component is added (the counter) and the synchronization of the line 19 is modified. As a result, every time the object "pack" issues a method call on its gate "deliver", this message is forwarded to gate "deliver" of the context and, simultaneously, a method call without parameters is issued to the method "notify" of the object "counter".

```
1       Context PackingUnit;
2       Interface
3           Use
4               Chocolate, Box;
5           Method
6               receive _ : chocolate;
7           Gate
8               deliver _ : box;
9       Body
10          Use
11              ProdBox, Pack;
12          Objects
13              prod box : prodbox;
14              pack : pack;
15              counter : counter;
16          Connections
17              prod box.deliver b With pack.receive box b;
18              PackingUnit.receive c With pack.receive chocolate c;
19              pack.deliver b With counter.notify // PackingUnit.deliver b;
```

Listing 2 : packing unit with counter

```
        Where
            c : chocolate;
            b : box;
    End PackingUnit;
```

Listing 2 : packing unit with counter

3.2.6 Genericity and reuse

COIL allows genericity and reuse through a syntactic mechanism, as CO-OPN. Moreover, generic modules can have parameters.

A generic context is distinguished form normal modules by the keyword "Generic" placed before the keyword "Context". A generic context serves as template for other modules and is not auto-instantiated.

Any normal context can instantiate at most one context. The resulting context is then the syntactic merge of the instantiated module and of the normal module. The instantiated module is either a generic module or another normal module (in which case the instantiation is actually a copy).

Parametrized instantiation is instantiation with replacement of the formal parameters (which are parameters modules) by effective parameters (which are normal modules). Finally, parameter modules are automatically imported.

Listing 3 shows a generic packing unit, and its instantiation as a packing unit for chocolates. The contents of "GenericPackingUnit" is instantiated into the core of "PackingUnit", with "pack item" renamed in "pack". "Chocolate" replace "Item" thus the interface of "PackingUnit" imports actually "Chocolate".

```
    Parameter Object Item;
    Interface
        Type item;
    End Item;

    Context GenericPackUnit(Item);
    Interface
        Use
            Box;
        Method
            receive _ : item;
        Gate
            deliver _ : box;
    Body
        Use
            ProdBox, Pack(Item);
        Objects
            prod box : prodbox;
            pack item : pack;
        Connections
            prod box.deliver b With pack item.receive box b;
            PackingUnit.receive i With pack item.receive item i;
            pack item.deliver b With PackingUnit.deliver b;
        Where
            i : item;
            b : box;
    End GenericPackUnit;

    Context PackingUnit As GenericPackUnit(Chocolate);
    Rename
        pack item -> pack;
    End PackingUnit
```

Listing 3 : generic packing unit and its instantiation

4 Related Works

Many researches are conducted in the domain of structured coordination since few years. Among other, COIL is inspired by the Kristensen's complex associations, by the Bergstra's ToolBus and by the Kramer's coordination language Darwin. With regard to these researches, COIL is clearly focusing on concurrent object-oriented software engineering.

4.1 Complex associations

Complex associations [11] come from object-oriented software engineering and are structured associations between objects. Like the contexts in COIL, complex associations describe a hierarchical coordination structure. But while contexts can be viewed as boxes encompassing components, complex associations are elaborated links connecting objects and encompassing objects. In other words, COIL's contexts are structures containing links, and complex associations are links containing structures. Thus if seems to be difficult to adapt complex associations to contextual modelling techniques.

Additionally, because of the fact that complex associations are described by means of standard objects, computation is not really separated from coordination, but in the other hand reuse, inheritance and dynamicity of associations are enabled through the standard object oriented techniques.

4.2 ToolBus

The ToolBus [3] is a coordination model for describing the coordination of open distributed systems. A ToolBus architecture is composed of a communication structure, namely the bus, linking loosely-coupled software systems.

The main difference between ToolBus and COIL is that while COIL promotes pure hierarchical coordination, the bus of ToolBus forms a linear structure. For this reason, it is not possible to describe layered coordination, as used for instance in the chocolate factory.

Moreover, the bus of ToolBus is described by means of a collections of so-called scripts, taking care about the routing of the messages. Thus coordination is not fully separated from coordination, as the coordination structure comports computational parts. However, the use of scripts allows full dynamicity of the architecture, by simply adding a new software system to the bus and connecting them by new routing scripts.

Finally, from an implementation point of view, an interesting feature of ToolBus is the development of a syntactic representation for message exchange. This allows to distribute the bus among different architectures, without data compatibility problems.

4.3 Darwin

Darwin is a coordination language for distributed system [12]. COIL is very similar to Darwin, but as Darwin is adapted for the description of software systems composed of distributed processes, COIL is primarily suited for the description of object-oriented systems composed of loosely-coupled objects.

Darwin model software systems by means of a hierarchical coordination using contexts, composed of processes and components (similar to the objects and contexts in COIL). However, Darwin was not primarily designed as an object-oriented language and seems poorly evolved with features as

reuse, inheritance and sub-typing.
Moreover, it seems that for historical reasons, dynamicity of configuration are made in Darwin at the expense of merging computations and coordination, on the contrary of COIL.

5 Conclusions

Management of complex systems, and consequently coordination of activities, are becoming one of the major research domain in computer science. Structured coordination models using contexts are the most promising of coordination models, as they allow a natural description of the coordination problems.
COIL is a new general language for coordination of concurrent active objects with structured coordination using contexts. COIL is based on the formal language CO-OPN$_{/2}$ and exploit its homogeneous ability to manage concurrent modelling.

From a software engineering point of view, COIL offers a natural model, close both to the user and to the computer. This model, with the usual encapsulation of its computing units, is well-suited for bottom-up synthetic constructions of software systems. Moreover, thanks to the reverse encapsulation provided by contexts, this model is also particularly adapted for top-down analytic approaches. Thus, software systems specified using COIL are both *decomposable* and *re-composable*. Moreover, on the contrary of standard object techniques, COIL promotes full reusability of its components, reusability of computing entities as well as reusability of coordination entities.

The main part of our current research is focusing on the application of COIL to distributed systems. Finally, development of COIL having fully dynamic features, i.e. allowing objects of the host language to instantiate and manipulate contexts, would be of great interest.

Acknowledgments

We would like to thank Erik Urland for his helpful comments on earlier drafts of this paper.

Bibliography

[1] M. Aksit and L. Bergmans. Obstacles in object oriented software development. In *Proceedings of OOPSLA'92*, 1992.

[2] F. Arbab. The iwim model for the coordination of concurrent activities. In P. Ciancarini and C. Hankin, editors, *Coordination Languages and Models*, volume 1061 of *LNCS*, pages 34–56. Springer-Verlag, 1996.

[3] J. Bergstra and P. Klint. The toolbus coordination architecture. In P. Ciancarini and C. Hankin, editors, *Coordination Languages and Models*, volume 1061 of *LNCS*, pages 75–88. Springer-Verlag, 1996.

[4] G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96:217–248, 1992.

[5] O. Biberstein and D. Buchs. Structured Algebraic Nets with Object-Orientation. In *Proceedings of the "Application and Theory of Petri Nets 1995" workshop on "Object-Oriented Programming and Models of Concurrency"*, Torino, Italy, June 1995.

[6] D. Buchs, P. Racloz, M. Buffo, J. Flumet, and E. Urland. Deriving parallel programs using sands tools. *Transputer Communications*, 3(1):23–32, Jan. 1996.

[7] M. Buffo and D. Buchs. From object-oriented specifications to distributed systems using hierarchical contexts. Submitted for the HICSS-30 conference, 1997.

[8] P. Ciancarini, K. K. Jensen, and D. Yankelevich. On the operational semantics of a coordination language. In O. Nierstrasz, P. Ciancarini, and A. Yonezawa, editors, *Object-Based Models and Languages for Concurrent Systems, Proceedings of the ECOOP '94 Workshop on Models and Languages for Coordination of Parallelism and Distribution*, LNCS 924, pages 77–106. Springer-Verlag, 1995.

[9] D. de Champeaux, D. Lea, and P. Faure. The process of object-oriented design. In *Proceedings of OOPSLA'92*, 1992.

[10] D. Garlan and D. Perry. Software architecture: Practice, potential and pitfalls. In *Proceedings of the 16th International Conference on Software Engineering*. IEEE Computer Society Press, May 1994.

[11] B. B. Kristensen. Complex associations: Abstractions in object-oriented modeling. In *Proceedings of OOPSLA'94*, pages 272–283, Portland, Oregon, USA, 1994.

[12] J. Magee, N. Dulay, and J. Kramer. Structuring parallel and distributed programs. *Software Engineering Journal*, pages 73–82, Mar. 1993.

[13] T. W. Malone and K. Crowston. The interdisciplinary study of coordination. *ACM Computing Surveys*, 26(1):87–119, Mar. 1994.

[14] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *LNCS*. Springer-Verlag, 1980.

[15] R. Milner. The polyadic pi-calculus: a tutorial. Technical Report ECS–LFCS–91–180, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, UK, Oct. 1991. Proceedings of the International Summer School on Logic and Algebra of Specification, Marktoberdorf, August 1991. Reprinted in *Logic and Algebra of Specification*, ed. F. L. Bauer, W. Brauer, and H. Schwichtenberg, Springer-Verlag, 1993.

[16] J. M. Purtilo. The polilith software bus. *ACM Transcations on Programming Languages and Systems*, 16(1):151–174, Jan. 1994.

[17] Webster. *Webster's New Encyclopedic Dictionary*. Black Dog & Leventhal, New York, 1994 edition, 1994.