Copyright 1998 IEEE. Published in the Proceedings of ISORC'98, 20-22 April 1998 in Kyoto, Japan. Personal use of this material is permitted. However, permission to reprint/republish this this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works, must be obtained from the IEEE. Contact: Manager, Copyrights and Permissions, IEEE Service Center, 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 08855-1331, USA.

BDL, a language of distributed reactive objects*†

Jean-Pierre Talpin‡, Albert Benveniste‡, Benoît Caillaud‡,
Claude Jard§, Zakaria Bouziane‡, Hubert Canon¶

IRISA, Campus de Beaulieu, F-35042 Rennes.

Abstract

*We introduce the definition of a language of distributed reactive objects, a Behaviour Description Language (*BDL*), as a unified medium for specifying, verifying, compiling and validating object-oriented, distributed reactive systems. One of the novelties in* BDL *is its seamless integration into the Unified Modeling Language approach (*UML*).* BDL *supports a description of objects interaction which respects both the functional architecture of system designs and the declarative style of diagram descriptions. This support is implemented by means of a partial-order theoretical framework. This framework allows to specify both the causality and the control models of object interactions independently of any hypothesis on the actual configuration of the system. Given the description of such a configuration, the use of* BDL *offers new perspectives for a flexible verification of systems by modeling them as an asynchronous network of synchronous components. It allows an optimized code generation by using compilation techniques developed for synchronous languages. It permits an accurate validation and test of applications by supporting the manipulation of both causal and control dependencies.* BDL *aims at maximizing the re-usability of high-level specifications while minimizing programming effort and test-case based validation of distributed systems.*

1. Introduction

The development of distributed software systems is a highly complex process. In order to manage this complexity various software engineering methods have been developed, ranging from requirements and design specification techniques to verification, validation, testing and code generation tools. We focus on those methods which have a formal foundation. They are expected to be based on formally defined languages and to rely on well-defined transformations. At present, the emerging object-oriented distributed computing technology raises new questions about formal engineering development.

At the programming level, the first point to note is that there is a need to extend the interface description languages (like IDL for CORBA) with behavioural properties. Interface description languages describe the features of an object in the computational view without presenting too much of its interior. They should define all visible aspects, namely the applicable functions and the (abstract) behaviour that is connected to the functions. This latter aspect is not yet formally defined [15] (even in the TINA-ODL proposal where there is just a placeholder for the behaviour descriptions and no syntax). In the present state of knowledge, this prevent most formal validation methods from being used. We thus argue for extending interfaces with formal specifications of behaviours: our Behavioural Description Language is called BDL. Although the behavioural aspects of interfaces are located to the architectural design phase of the development of object based distributed applications, this phase in the development process must be seen in the framework of a coherent methodology and life-cycle model [14]. The emerging object-oriented methodologies are put into the Unified Modeling Language (UML), becoming a de facto standard [17]. This leads to our second point about the need to integrate BDL into UML to continuously accompany the development from specification phases to implementations (figure 1). Since we focus on dynamical aspects, we start from the idea that the most abstract level of behaviours can be described using scenari (like Message Sequence Charts). We will enrich these scenarios by a formal semantics and facilities to describe complex programming controls using variables and conditions.

Our paper introduces the definition of BDL as a unified medium for specifying, verifying, compiling and validating distributed, object-oriented, reactive systems. One of the novelties in BDL is its seamless integration into the UML approach. BDL supports a description of object components interaction which respects both the functional archi-

*Funded by ALCATEL, project REUTEL-2000.
†Funded by the French Ministry for Research, program ARCTICA.
‡Institut National de Recherche en Informatique et Automatique.
§Centre National pour la Recherche Scientifique.
¶Délégation Générale pour l'Armement, FORMA project.

tecture of system designs and the declarative style of diagram descriptions. This support is implemented by means of a partial-order theoretical framework. Together with a logical notion of time and a notion of *reaction*, this framework allows to specify both the causality and the control models of object interactions independently of any hypothesis on the actual configuration of the system. The focus of our approach is the partial orderings of method invocations in an application consisting of distributed reactive objects. In this article, we do not consider the dynamic creation of objects nor non-functional system requirements (such as *deadlines* or other temporal requirements). We focus instead on establishing a formal framework in which both verification and optimization of large-scaled, object-oriented, distributed applications are feasible by using existing tools.

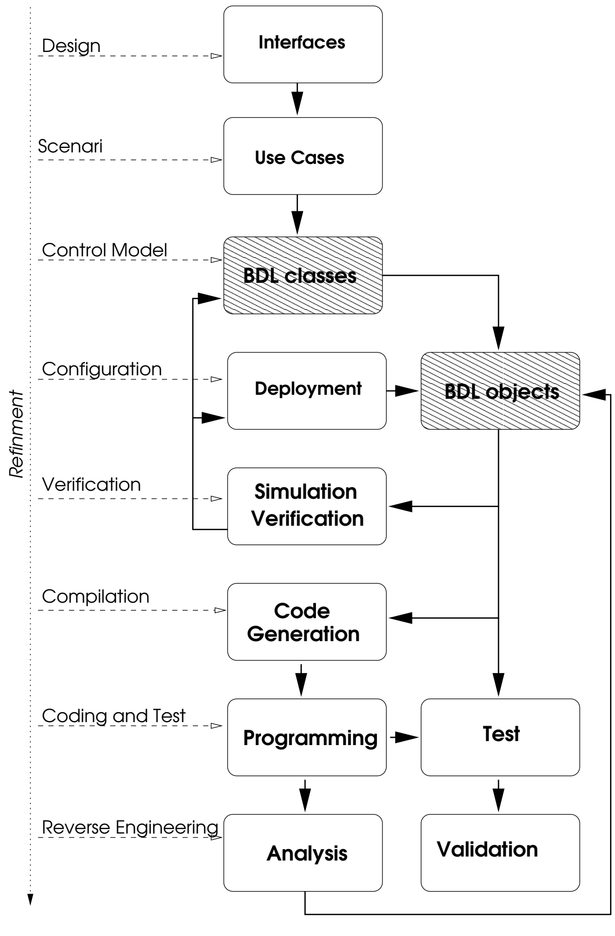


Figure 1. BDL **in its environment**

2. Presentation of BDL

In order to fulfill the requirements of supporting both a description of the functional architecture of a system and a declarative specification of the interaction between its components, the definition of BDL shall integrate the features of an object-oriented language for structuring the specification of both the causality and the control models of object components. The syntax of BDL is composed of two layers. An object layer describes the structure of a system in a way akin to an UML design. An action layer describes the interaction of each object component with its environment. The definition of BDL classes, arrays and objects allows the reuse of action specifications from the design of the system functional architecture into that of its actual deployment (dimension and distribution) in order to match the configuration of a target system.

As in UML, a set of identifiers I is assumed to name classes C, objects O, types T, ports X, symbolic constants V. Compound identifiers consisting of class names C, object names O, array references $C[V]$, prepended to identifiers I are assumed as well.

$S ::=$	class C is D	class
$\mid$	class C of C' is D	typed
$\mid$	class $C[V:T]$ is D	indexed
$\mid$	class $C(I:T)$ is D	template
$\mid$	object O of C	instance
$D ::=$	S	component
$\mid$	$C\langle I\rangle$	bound class
$\mid$	port $X:T$	port
$\mid$	A	action

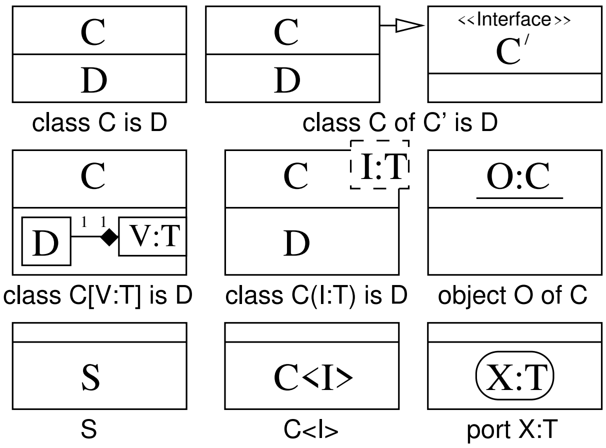


Figure 2. Syntax of BDL

The formal syntax of BDL is defined in the figure 2 together with its UML graphical representation. The description of a system consists of structures S: classes and objects. A class C consists of a declaration D possibly restricted to an interface of type C', or parameterized by a template identifier I of type T or indexed by an identifier I of type T. An object O is an element of a class C. A declaration D is either the definition of a component S, a bound class $C\langle I\rangle$ of a template class C with the parameter I, a port X receiving messages of type T or an action A. Both notions of attribute and method in UML correspond to the notion of port in BDL.

$$
\begin{array}{rll}
A ::= & X(V) & \text{valuation} \\
| & X(V) \to Y(W) & \text{causality} \\
| & !A & \text{activation} \\
| & A \text{ or } A' & \text{disjunction} \\
| & A \parallel A' & \text{composition} \\
| & \text{any } V\!:\!T\ A & \text{disjunction} \\
| & \text{all } V\!:\!T\ A & \text{composition}
\end{array}
$$

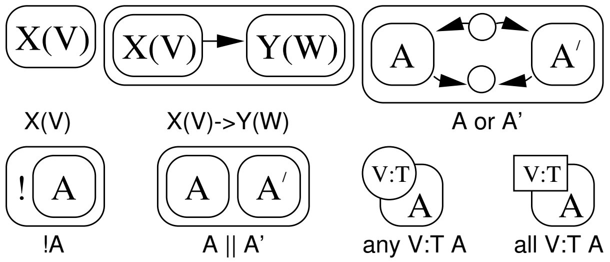


Figure 3. BDL **actions** A

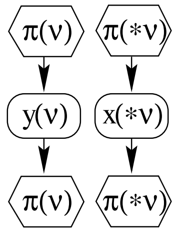
An action A is either a valuation $X(V)$ (i.e. ≪value V is present at port X≫), the causality specification ≪$X(V) \to Y(W)$≫ (i.e. ≪$X(V)$ is the cause of $Y(W)$≫) the disjunction ≪A or A'≫ of actions (non-deterministic choice), the activation ≪$!A$≫ of an action (i.e. ≪of course A≫), the synchronous composition ≪$A \parallel A'$≫ of actions. Enumerated composition and disjunction are written all $V : T\ A$ and any $V\!:\!T\ A$.

3. Semantics of BDL

The meaning of a BDL program is defined thanks to a true-concurrency semantics in which families of directed labeled graphs describe the causal dependencies (pre-order relations) between events for each possible behaviour of the program (section 3.1). There are three kinds of vertices : association of values to ports, activations/terminations and value passing through pins. Composition operators (section 3.2) on families of directed graphs are then introduced to reflect iteration and synchronous/asynchronous parallel composition of BDL terms: *Graph concatenation* (via pins) corresponds to repetitive activations of objects (iteration). If objects are interpreted as sets of constraints, the *synchronous composition* of two objects is the conjunction of the corresponding sets of constraints. *Asynchronous composition* is a weaker parallel composition in which it is not possible to test for the absence of port occurrence. It reflects asynchronous communication in a declarative setting. Finally, a discussion on *endochrony* (section 3.4) provides some intuitive insight on the design of BDL.

3.1. Semantic Domains

We are given a finite set of *names* x denoting *ports* as defined in the section 2. In addition, we consider a finite set of *pins* π with values in non-empty domains (written $\mathcal{D}_\pi$). Pins π will be used to model ≪state-transitions≫ and carry values from a ≪reaction≫ to another. Pins are denoted by the graphical notations of the figure 4 (minimal and maximal pins start_π and exit_π holding given values v or arbitrary values, written $*v$).



$$\begin{pmatrix}\mathsf{start}_\pi(\nu) \to y(\nu), \\ y(\nu) \to \mathsf{exit}_\pi(\nu)\end{pmatrix}$$

$$\begin{pmatrix}\{\mathsf{start}_\pi(*\nu) \to y(*\nu), \\ y(*\nu) \to \mathsf{exit}_\pi(*\nu)\} \mid \\ \forall *\nu \in \mathcal{D}_\pi \cap \mathcal{D}_x\end{pmatrix}$$

Figure 4. Pins

An object O is represented by a quadruple $(\Sigma, \Pi, \Gamma, \gamma_0)$ where Σ denotes its ports x, Π its pins π, Γ a family of well-formed labeled directed graphs γ and γ_0 an initial graph which only contains pins of the form $\mathsf{exit}_\pi(v)$ with $\pi \in \Pi$ and $v \in \mathcal{D}_\pi$.

Family of Graphs A term $\gamma \in \Gamma$ is a directed, possibly cyclic, graph, with vertices taken in some infinite ground set of events and labels of one of the forms: a pair $x(v)$ of (port, value) such that $x \in \Sigma$ and $v \in \mathcal{D}_x$, or a sorted pair $\mathsf{start}_\pi(v)$ or $\mathsf{exit}_\pi(v)$ of (sort, pin, value) where $\pi \in \Pi$ is a pin and $v \in \mathcal{D}_\pi$.

Extremal vertices For each $\gamma \in \Gamma$, the set of pins occurring in γ is denoted $\Pi_\gamma = \{\pi \mid \exists v \in \mathcal{D}_\pi \text{ s.t. } \mathsf{start}_\pi(v) \in \gamma \text{ or } \mathsf{exit}_\pi(v) \in \gamma\}$ By definition, $\Pi_\gamma \subseteq \Pi$. Vertices that are associated with pins are denoted by $\mathsf{start}_\Gamma = \{\mathsf{start}_\pi(v) \mid \exists \gamma \in \Gamma \text{ s.t. } \pi \in \Pi_\gamma\}$ and $\mathsf{exit}_\Gamma = \{\mathsf{exit}_\pi(v) \mid \exists \gamma \in \Gamma \text{ s.t.} \pi \in \Pi_\gamma\}$.

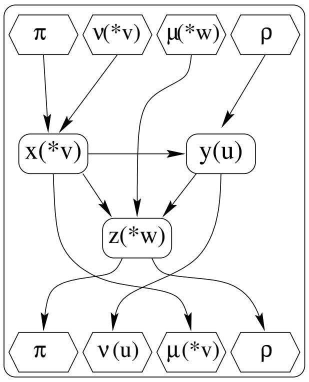
Well-formed graphs A labeled directed graph is well-formed (e.g. figures 4 and 5) if and only if four conditions are satisfied:

1. For all $x \in \Sigma$, γ contains at most one vertex of the form $x(v)$ s.t. $v \in \mathcal{D}_x$.
2. For all $\pi \in \Pi$, γ contains at most one $\mathsf{start}_\pi(v)$ (resp. $\mathsf{exit}_\pi(v)$) s.t. $v \in \mathcal{D}_\pi$.
3. Minimal elements of γ are of the form $\mathsf{start}_\pi(v)$ s.t. $\pi \in \Pi$ and $v \in \mathcal{D}_\pi$.
4. Maximal elements of γ are of the form $\mathsf{exit}_\pi(v)$ s.t. $\pi \in \Pi$ and $v \in \mathcal{D}_\pi$.

Silent and active graphs We assume that every family Γ contains a family of silent graphs $\tau_\Gamma = \{\vee_{v \in \mathcal{D}_\pi}\{\mathsf{start}_\pi(v) \to \mathsf{exit}_\pi(v)\} \mid \pi \in \Pi\}$. This models the fact that Γ may do nothing within a considered reaction.

This guarantees a stuttering robustness property [13]. We write $!\Gamma = \Gamma \setminus \tau_\Gamma$ for the family of active graphs of Γ.

Graphical notations An extension of the graphical syntax of figure 3 will be useful to support our discussion on the mathematical semantics of BDL. In the figure 5, the formula on the right defines the family obtained from the graph on the left. It is a set of well-formed graphs with universal quantification of variables $*v$ and $*w$.



$$\left(\begin{array}{l} \{\mathsf{start}_\pi \to x(*v), \\ \mathsf{start}_\nu(*v) \to x(*v), \\ \mathsf{start}_\mu(*w) \to z(*w), \\ \mathsf{start}_\rho \to y(u), \\ x(*v) \to y(u), \\ x(*v) \to z(*w), \\ x(*v) \to \mathsf{exit}_\mu(*v), \\ y(u) \to z(*w), \\ y(u) \to \mathsf{exit}_\nu(u), \\ z(*w) \to \mathsf{exit}_\pi, \\ z(*w) \to \mathsf{exit}_\rho\} \\ \forall *v \in \mathcal{D}_\nu \cap \mathcal{D}_x \cap \mathcal{D}_\mu, \\ \forall *w \in \mathcal{D}_\mu \cap \mathcal{D}_z \end{array}\right)$$

Figure 5. Graph with quantifications

3.2. Semantic Operators

Concatenation Consider two objects with identical Σ and Π and respective graph families Γ and Γ'. Pick $\gamma \in \Gamma$ and $\gamma' \in \Gamma'$.

• Graphs γ and γ' are said to be *concatenables* (denoted $\gamma \odot \gamma'$) if and only if the following condition is satisfied: $(1) \forall \pi \in \Pi, \mathsf{exit}_\pi(v) \in \max(\gamma) \wedge \mathsf{start}_\pi(v') \in \min(\gamma') \Rightarrow v = v'$.

• If condition (1) is satisfied, then the concatenation $\gamma \circ \gamma'$ is obtained by

1. Identifying $\mathsf{exit}_\pi(v) \in \max(\gamma)$ and $\mathsf{start}_\pi(v) \in \min(\gamma')$
2. Taking the disjoint union of γ and γ'
3. Retaining only those vertices $\mathsf{exit}_\pi(v)$ and $\mathsf{start}_\pi(v)$ which remain extremal after step 3, and discarding the other ones associated with pins
4. Whenever a vertex is discarded, edges are inserted so that the transitive closure of the graph, restricted to vertices that are not discarded is unchanged.

• The family $\Gamma \circ \Gamma'$ of graphs is composed of those $\gamma \circ \gamma'$ s.t. γ and γ' are concatenables: $\Gamma \circ \Gamma' = \{\gamma \circ \gamma' \mid \gamma \in \Gamma,\ \gamma' \in \Gamma',\ \gamma \odot \gamma'\}$.

In this way we define: $\mathcal{G} = \gamma_0 \circ (\Gamma)^\omega$, where superscript $(.)^\omega$ denotes (infinite) countable concatenation, and *this defines the semantics of an object*. Note that the concatenation $\Gamma \circ \Gamma'$ of families of graphs do not need a graphical notation, as the semantics of an object is given by $\mathcal{G}$.

Example Figure 6 depicts the concatenation of a family of graphs Γ (that of figure 5) with itself. The diagram on the left in the figure 6 shows the maximal and minimal vertices of Γ. Their concatenation is shown on the diagram in the middle where shared pins π, ρ, μ and ν are superimposed in the grey box. In the diagram on the right, they are erased and transitive edges inserted.

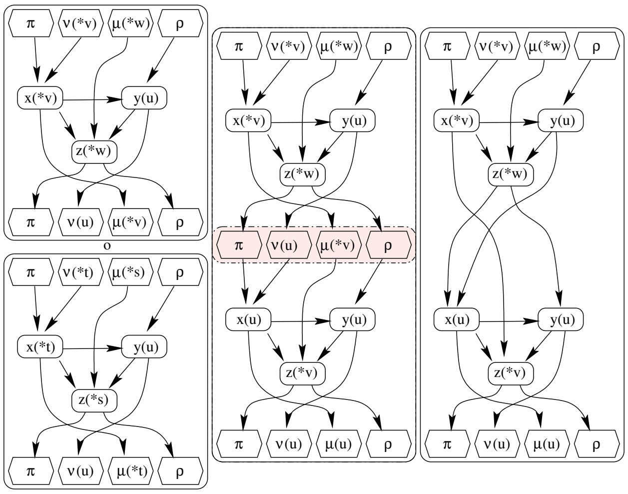


Figure 6. Graph concatenation

Synchronous composition Consider again two objects $(\Sigma, \Pi, \Gamma, \gamma_0)$ and $(\Sigma', \Pi', \Gamma', \gamma_0')$. Two graphs $\gamma \in \Gamma$ and $\gamma' \in \Gamma'$ are said *synchronously composable* (denoted $\gamma \bowtie \gamma'$) if and only if any vertex $x(v)$, $\mathsf{start}_\pi(v)$ or $\mathsf{exit}_\pi(v)$, which satisfies (2) : $x \in \Sigma \cap \Sigma'$, or $\pi \in \Pi \cap \Pi'$ respectively, occurs in γ if and only if it occurs in γ'. If the condition (2) is satisfied, then the *synchronous composition* $\gamma \| \gamma'$ is defined as the union of the graphs γ and γ' (where identical vertices are superimposed). The family $\Gamma \| \Gamma'$ is composed of those graphs $\gamma \| \gamma'$ such that $\gamma \in \Gamma$ and $\gamma' \in \Gamma'$ are composable: $\Gamma \| \Gamma' = \{\gamma \| \gamma' \mid \gamma \in \Gamma, \gamma' \in \Gamma', \gamma \bowtie \gamma'\}$. This defines the composition of objects as $(\Sigma \cup \Sigma', \Pi \cup \Pi', \Gamma \| \Gamma', \gamma_0 \| \gamma_0')$, assuming that $\gamma_0 \bowtie \gamma_0'$. Hence, if objects are considered as sets of constraints, synchronous composition corresponds the conjunction of the corresponding constraints.

Example The left part of the figure 7 depicts $\gamma \| \gamma'$, where γ and γ' are the two graphs inside the rectangles. Pin π and port y are shared: they are superimposed via composition, and the mechanism of port/pin unification results in fixing value v for ports x and y. The resulting graph is shown on the right hand side.

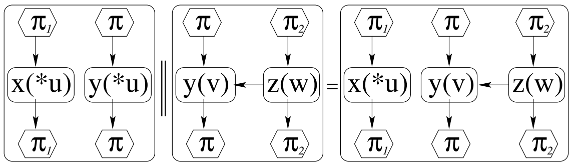


Figure 7. Graph composition

Disjunction The disjunction $\Gamma \bigvee \Gamma'$ of two families Γ and Γ' is the set theoretic union of the families Γ and Γ'. The disjunction of two objects $O = (\Sigma, \Pi, \Gamma, \gamma_0)$ and $O' = (\Sigma', \Pi', \Gamma', \gamma'_0)$ with composable initial graphs ($\gamma_0 \bowtie \gamma'_0$) is defined as follows: $O \bigvee O' = (\Sigma \cup \Sigma', \Pi \cup \Pi', \Gamma \cup \Gamma', \gamma_0 \| \gamma'_0)$.

Asynchronous composition Consider two objects such that every $\gamma \in \Gamma$ and $\gamma' \in \Gamma'$ are circuit-free. We will also need to consider their asynchronous composition, which, roughly speaking, consists in 1/ constructing Γ^ω and $(\Gamma')^\omega$ (note that pins and memories are erased while building the ω-concatenation), and then 2/ taking the composition via intersection, exactly as before. Formally speaking:
1. Let $\gamma_\omega \in \Gamma^\omega$ be a possibly infinite labeled graph. Pick all vertices that are labeled with the same port name, say x: they are organized into a linear order, we call it a *flow*. Thus a flow has the form $x(v_1) \rightarrow x(v_2) \rightarrow x(v_3) \rightarrow \ldots$ and carries a (finite or infinite) sequence $v_1, v_2, v_3, \ldots$ of values.
2. Two graphs $\gamma_\omega \in \Gamma^\omega$ and $\gamma'_\omega \in (\Gamma')^\omega$ are said *asynchronously composable* if, for any shared port $x \in \Sigma \cap \Sigma'$, the sequences of values $v_1, v_2, v_3, \ldots$ and $v'_1, v'_2, v'_3, \ldots$ of the two flows are identical.
3. If the condition above is satisfied, then the *asynchronous composition* $\gamma_\omega \|_a \gamma'_\omega$ is defined as the union of the graphs γ and γ' (where identical vertices are superimposed).
4. The family $\gamma_0 \circ (\Gamma)^\omega \|_a \gamma'_0 \circ (\Gamma')^\omega$ is composed of those graphs $\gamma_\omega \|_a \gamma'_\omega$ such that $\gamma_\omega \in \gamma_0 \circ (\Gamma)^\omega$ and $\gamma'_\omega \in \gamma'_0 \circ (\Gamma')^\omega$ are composable: this defines $\mathcal{G} \|_a \mathcal{G}'$.

Note that this again corresponds to taking the conjunction of corresponding constraints, but without the possibility to test for the absence of ports in an activated object.

3.3. Discussion

We see that we have defined two *different* notions of composition.

Synchronous composition is compliant with the synchronous model, in which programs progress according to a possibly infinite sequence of *reactions* $\ll P \equiv R^\omega \gg$. This model of abstract machine allows us to decide upon presence/absence of a given port within a considered reaction, a significant advantage for BDL specification. In addition, synchronous composition is defined at the level of a single reaction, which makes it simple and easy to use for system specification. This model of abstract machine is perfectly suitable when the target architecture consists of a single processor, or any other architecture in which a global notion of program state is easily available. In turn, synchronous composition is not compliant with target architectures involving nondeterministic multi-threading, and definitely not compliant with distributed architectures involving asynchronous communication media.

Asynchronous composition as we have defined it uses a minimally demanding model of communication: any communication medium which is compliant with that of a partial ordering would do. This amounts to requiring a send-receive type of communication in which messages are not lost and not shuffled — yet a significant requirement, but still much more realistic than asking for implementing perfect synchrony in distributed architectures. Practically, this means that our abstract machine model of asynchronous communication does not handle fault-tolerance by itself. In turn, it is recognized that handling such type of module composition at specification stage makes it difficult for the designer to keep track of deadlock freedom and correctness, unless she/he would stick on very strict and restrictive programming disciplines (such as client-server). This is why we have decided to base specification on a synchronous model of communication, and, to base actual distributed implementation on an asynchronous model of communication, and, finally, to thoroughly study how one can safely move from the former model to the latter one.

3.4. Endochrony and Isochrony

We seek for a class of BDL objects for which both synchronous and asynchronous types of composition are equivalent. Suppose we can exhibit such a class of so-called *endochronous* and *isochronous* objects in the sequel. Then we design systems according to the following methodology:
1. Design the system using the synchronous paradigm, by freely assembling objects and classes using synchronous composition.
2. Configurate the design, i.e., decide upon the actual architecture for execution. This results in distributing the system on a set of ≪processors≫ (they may just correspond to threads), having the form of BDL objects obtained by synchronous composition of finer grain modules from system specification design stage.
3. Check if the so obtained processors are endochronous and isochronous. If so, then we know that their communication can actually be interpreted as an *asynchronous* one, hence subsequent distribution would become much easier. If this property is not satisfied, then some additional protocols need to be included to the processors, but this topic is beyond the scope of this paper.

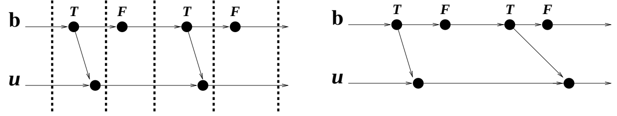


Figure 8. Endochrony

Thus these questions of endochrony and isochrony are central to our approach and make it attractive. How endochrony is feasible is intuitively illustrated in the figure 8, which shows the graph associated with the statement: $\ll$Guard$\langle$b, u$\rangle\gg$, defined in section 4. The diagram on the left depicts the history or trace of this statement, showing the successive instants or reactions separated by thick dashed lines. Pins have been erased when performing concatenation. In the diagram on the right, an instant has been twisted and thick dashed lines have been removed, i.e., the notion of a reaction or instant has been lost. However, no information has been lost: we know that u should be got exactly when port b holds the value *true*, and thus it is only needed to wait for b in order to know whether u is to be waited for also. This suggest the way endochrony is formally defined (see [19]). An object O considered with a given environment O' is isochronous if and only if object O has the same reactions (in terms of presence/absence of ports, disregarding values of ports) whether it is considered on its own or placed in the environment $(O \| O')$.

3.5. Meaning of BDL

Thanks to the mathematical framework introduced in sections 3.1 and 3.2, we can give a graphical denotation of the BDL actions A (figure 3) embedded in classes and objects. The mathematical denotation $[\![A]\!]$ of an action A is defined in the figure 9 by induction on the structure of A. To any term A recursively corresponds a graph Γ whose ports Σ and pins Π are both $\{x \mid x \text{ occurs in } A\}$. The function $[\![\_]\!]$ considers actions A resulting of the expansion of well-typed BDL structures S (as defined in the figure 16 of the appendix). In the figure 9, we write $[\![A]\!][*V/V]$ for the substitution of V by $*V$ in $[\![A]\!]$ and $\forall W : T[\![A]\!][W/V]$ for the composition of all $[\![A]\!][W/V]$ s.t. W is of type T.

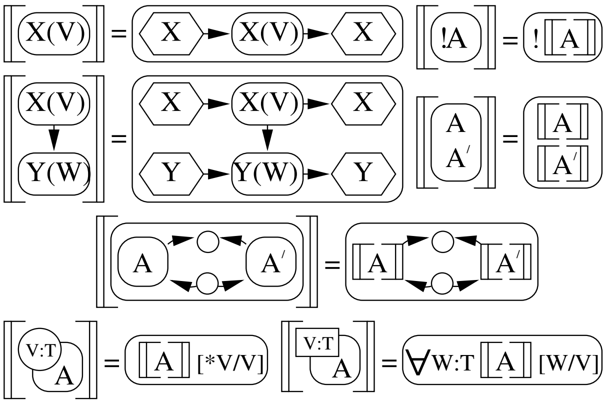


Figure 9. Denotation $[\![A]\!]$ of an action A

4. Using BDL

The complete definition of the BDL language consists of the core BDL syntax of section 2 enriched with a predefined class (Cell, figure 10) and a set of pervasive classes (such as Guard or Sync, figures 12 and 13) which can be defined in BDL with the only help of class Cell. The class Cell (defined figure 10) is a generic class of two parameters (the first one, T, is a data-type and the second one , V, is a value of that data-type). Class Cell acts like a memory in which values ranging over data-type T can be stored (using port set) and retrieved (using port get). Value are transported from a transition to another by resourcing to pins. Parameter V is the initial value of the memory m.

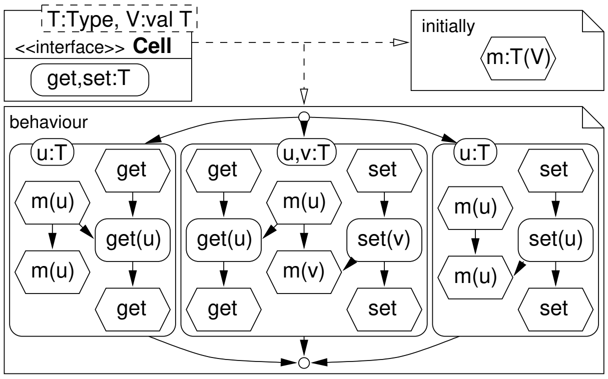


Figure 10. The pervasive class Cell

A typical usage of class Cell is given in figure 11. The class Counter (on the left in the figure) emits on the port count the number of tick messages it receives (a trace of the counter, consisting of the concatenation $\gamma_0 \circ \gamma_1 \circ \gamma_2$, is depicted on the right in the figure). Figure 12 gives the definition of class Sync, which can be used to enforce sequentiality between occurrences of groups of ports or objects.

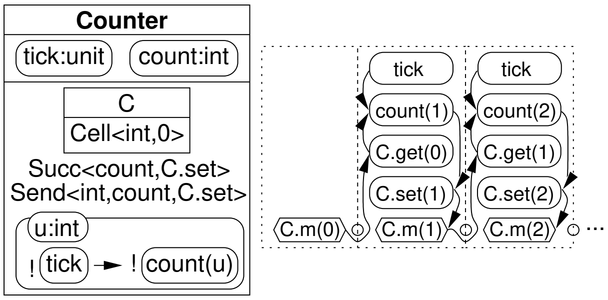


Figure 11. Using Cell to define a Counter

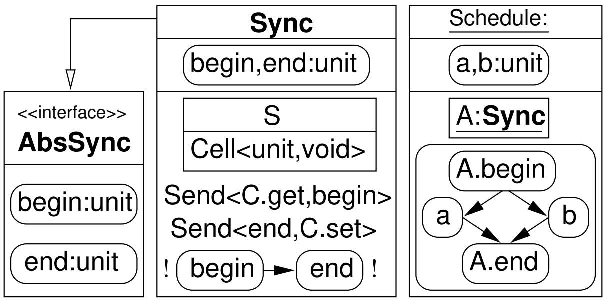


Figure 12. Using Cell to define Sync

A few other pervasive, generic, classes are detailed in figure 13. The class Guard allows to condition the activation of a message Y by a boolean port X. The class Send models the synchronous transmission of a value V from a port X to a port Y. The interfaces And and Plus show that operations on booleans or integers ports X, Y, Z can be regarded as generic classes. A more significant example of BDL specification is detailed in figure 20 of the appendix. This example is self-contained and self-explanatory.

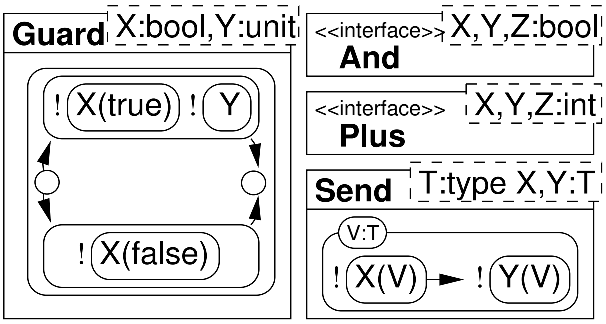


Figure 13. Other BDL pervasives

5. Prospective Applications of BDL

The use of BDL integrates naturally in an UML object-oriented development process (figure 1). It allows to define a logical interaction model between the components described in the functional architecture of a system, with the help of a preliminary UML design and elementary scenari. BDL descriptions can be added to the preliminary UML design by means of stereotypes [2] in order to enrich the model with a behavioural view of the system.

At this stage of the development, object-oriented methodologies such as OMT or UML usually recommend the refinement of declarative diagram descriptions (architecture, scenari) using imperative automata specifications for describing the behavior of system components (e.g. using SDL [20], STATECHARTS [10] or [5][1]). All these approaches offer means of directly describing an ≪heterogeneous≫ architecture (i.e. an asynchronous network of synchronous, imperative, objects).

According to the above respects, our approach radically differs from previous works [20, 10, 5]. It is motivated by the fact that stepping from a declarative description of scenari to an imperative description of behaviours is difficult without making minimal hypothesis on the actual distribution of system components and on an interpretation of communications (in the scenari) and of concurrency (in the architecture). BDL solves this key issue by implementing a declarative description of objects interaction in terms of graph families without making any assumption about their distribution. This approach allows an homogeneous description of the functional architecture of distributed object-oriented applications. Mapping this architecture to a particular, heterogeneous, configuration is regarded as a separate issue of ≪deployment≫.

Given the configuration and dimensions of a target system (by means of an UML deployment of BDL diagrams), the functional architecture of the application can be instantiated to form an assembly which defines the behaviour of the target system in terms of a distribution of reactive objects. To each object corresponds a family of graphs which can be executed by iteratively choosing composable and concatenable graphs, as defined section 3. The causal relations between the components of the target system can be interpreted in several ways: synchronous communication (within a confined target object), asynchronous communication (between distributed target objects), data or control dependencies, real-time dependencies, etc.

The interpretation of distributed objects as communicating automata or symbolic transition systems allows to use efficient verification tools, such as CADP [8], in order to verify the global safety properties of the target systems (e.g., deadlock freedom, reachability of states). The interpretation of confined objects as synchronous programs allows to compile them separately in order to generate either simulation programs or intelligent program stubs implementing the scheduling of communications and concurrency of an object with its remote environment.

Just as IDL specifications describe *what* application components exchange with its environment, BDL specifications describe *how* these components interact with it. A BDL specification allows to compile its interaction pattern (i.e. communication and concurrency) as well as an IDL specification allows the automated generation of program stubs for an application component (i.e. code for marshaling and exchanging data with the environment and for plugging it with common ORB communication resources).

Thanks to its formulation using a partial-order theoretical framework, the BDL specification of a capsule (a set of application components, embedded according to a given configuration) allows an optimized compilation by resourcing to code generation techniques employed in synchronous programming languages, such as SIGNAL [6], for implementing embedded systems. More precisely, the graph family Γ of a BDL-capsule can be translated into a STS scheduling specification (Symbolic Transition Systems, [4, 16]) which describes both the control model (e.g. the valuations) and the causality model (e.g. the arrows) of the service. By determining the tasks in this graph (i.e. the control configurations or states) and by hierarchizing them (i.e. determining the logical relations between them), the program implementing each capsule of a BDL specification can be automatically generated.

[1] which, to our late knowledge, happens to be also named BDL

6. Conclusion

We have introduced the formal definition of a declarative specification language for describing distributed reactive objects, BDL, and demonstrated that BDL provides a unified medium for specifying, verifying and compiling distributed real-time applications. In contrast to the object-oriented extensions of conventional automata-based tools for specifying distributed systems [20, 10, 5], the BDL language implements a smoother phase transition between the specification of a system and its configuration. This allow to feature a better integration into UML.

BDL supports a description of object components interaction which respects both the functional architecture of UML system designs and the declarative style of UML diagram descriptions. BDL implements a both simple and expressive theoretical framework which allows to specify both the causality and the control models of object interactions independently of any hypothesis on the actual configuration of the system. The definition of BDL offers new perspectives for a flexible verification of systems, the generation of optimized code, an accurate validation of applications by supporting the manipulation of both causal and control dependencies. BDL aims at maximizing the re-usability of high-level specifications and at minimizing programming and validation efforts for the implementation of real-time, object-oriented, distributed systems.

References

[1] R. Alur, G.J. Holzmann and D. Peled. A analyzer for Message Sequence Charts. In *Tools and Algorithms for the Construction and Analysis of Systems*. Lecture Notes in Computer Science v. 1055., p. 35–48. Springer, 1996.

[2] S.W. Ambler. *The Unified modeling language and beyond: the techniques of object-oriented modeling*, (http://www.ambysoft.com), August 1997.

[3] C. André, F. Boulanger, M.-A. Péraldi, J.-P. Rigault, and G. Vidal-Naquet. Objects and synchronous programming. In *European Journal on Automated Systems*, v. 31(3), p. 417–432. Hermes, 1997.

[4] A. Benveniste, P. Le Guernic and P. Aubry. Compositionality in dataflow synchronous languages: specification and code generation. In *Malente Workshop on Compositionality*. W.P. de Roever, A. Pnueli Eds, 1997.

[5] F. Bertrand and M. Augeraud. BDL, a language to control the behavior of concurrent objects. In *Conference on Domain-Specific Languages*, p. 133-144. USENIX, 1997.

[6] A. Benveniste, P. Le Guernic and C. Jacquemot. Synchronous programming with events and relations: the SIGNAL language and its semantics. In *Science of Computer Programming*, v. 16, 1991.

[7] R. Breu, U. Hinkel, C. Hofmann, C. Klein, B. Paech, B. Rumpe and V. Thurner. Towards a formalization of the unified modeling language. In *Proceedings of the 11th European Conference on Object-oriented programming*. Lecture Notes in Computer Science v. 1241. Springer, 1997.

[8] J.-C. Fernandez, H. Garavel, A. Kerbrat, L. Mounier, R. Mateescu and M. Sighineanu. Computer Aided Verification. In *CADP: a protocol validation and verification toolbox*, 1996.

[9] J. Grabowski, E. Rudolph and P. Graubman. Message Sequence Charts: composition techniques versus OO-techniques. In *Proceedings of the 7th.* SDL *forum*, 1995.

[10] D. Harel and E. Gery. Executable Object Modeling with Statecharts. In *Proceedings of the International Conference on Software Engineering*, p. 246-257. IEEE, 1996.

[11] I. Jacobson, M. Christerson, P. Jonsson and G. Overgaard. *Object-Oriented Software Engineering – A Use Case Driven Approach*. ACM, 1992.

[12] C. Jard, J.-M. Jézéquel and L. Nedelka. *An Approach to Integrate Formal Validation in an OO Life-cycle of Protocols*. In *Proceedings of* FMOODS'96. Chapman & Hall, March 1996.

[13] L. Lamport. Specifying concurrent program modules. In *ACM Transactions on Programming Languages and Systems*, v. 5(2), p. 190-222. ACM, 1983.

[14] S. Pickin, C. Sanchez, J.-C. Yelmo, J.-J. Gil and E. Rodriguez. Introducing formal notations in the development of object-based distributed applications. In *Proceedings of* FMOODS'96, p. 87–102. Chapman & Hall, March 1996.

[15] A. Prinz. Including behaviour into interfaces. In *Proceedings of* FMOODS'96, p. 37–49. Chapman & Hall, March 1996.

[16] A. Pnueli, et al. Symbolic Transition Systems. In *Malente Workshop on Compositionality*. W.P. de Roever, A. Pnueli Eds. September 1997.

[17] J. Rumbaugh, I. Jacobson and G. Booch. *Unified Modeling Language Reference Manual*. ISBN n. 0-201-30998-X. Addison-Wesley, 1997.

[18] B. Selic, G. Gullekson and P. T. Ward. *Real-Time Object-Oriented Modeling*. John Wiley & Sons, 1994.

[19] J.-P. Talpin, A. Benveniste, C. Jard, B. Caillaud, Z. Bouziane and H. Canon. BDL, a specification language for distributed object-oriented real-time systems. *Technical Report n. 1145*. IRISA, 1998.

[20] *Recommendation Z.100 - CCITT specification and description language (*SDL*)*. International Telecommunication Union, March 1993.

Appendix

We formally define well-typed BDL phrases by the relation $E \vdash S:t$ (figure 16) which associates a structure S to its type or signature t (figure 14) given an environment E. An environment E is a finite relation between identifiers I and types t (i.e. $E \ni (I:t)$). A type is either a data-type d or an object type t. The relation $E \vdash S:t$ is defined by induction on the syntax of BDL (figures 2 and 3). Each rule in the definition has the form $\frac{P_{1..n}}{C}$ and defines the relation C as the conclusion of the conjunction of smaller premises P_1 to P_n. Each rule is universally quantified over all its meta-names.

$$d ::= \mathsf{bool} \mid \mathsf{int} \mid \mathsf{union}\ V_{1..n}$$
$$t ::= \mathsf{type}\ d \mid \mathsf{port}\ d \mid \mathsf{val}\ d \mid \mathsf{class}\ E \mid (I:t).t' \mid [V:d].t$$

Figure 14. Data-types d and object-types t

The relation $E \vdash S:t$ implements a contra-variant sub-typing relation $t \le t'$ between objects (figure 15). We identify the types $(I:t).t'$ and $(J:t).(t'[I/J])$ for all J not occurring in t, t'.

$$\mathsf{port}\ d \le \mathsf{port}\ d \qquad \mathsf{val}\ d \le \mathsf{val}\ d$$
$$\frac{t \le t' \quad E \le E'}{E,(I:t) \le E',(I:t')} \qquad \frac{E \le E'}{\mathsf{class}\ E \le \mathsf{class}\ E'}$$
$$\frac{t' \le t''' \quad t'' \le t}{(I:t).t' \le (I:t'').t'''} \qquad \frac{t \le t'}{[V:d].t \le [V:d].t'}$$

Figure 15. Sub-typing relation $t \le t'$

The relation of well-typing $E \vdash S:t$ is structurally decomposed into a relation $E \vdash D:t$ on declarations D, a relation $E \vdash I:t$ on identifiers and a relation $E \vdash A$ on actions A. Notice that the type associated with a structure S is exactly its ≪interface≫. Also notice that declaring this interface (i.e. ≪class $C:C'$ is D≫) prevents other identifiers (i.e. in D but not in C') to be visible from the environment.

Similarly, every declaration D introduces a binding between an identifier and a structure. Therefore, the relation $E \vdash D:t$ also associates declarations D to object types. The type-checking of the selection of an element in a compound class or in an array class is implemented by the relation $E \vdash I:t$ between compound identifiers I and types t as follows.

Finally, the type-checking relation between environment E and actions A checks the conformance between data-types and object-types already declared and visible (in E) and the way they are used (in A).

$$\frac{E \vdash C:t}{E \vdash \mathsf{object}\ O\ \mathsf{of}\ C:\mathsf{class}\ (O:t)}$$
$$\frac{E \vdash D:t}{E \vdash \mathsf{class}\ C\ \mathsf{is}\ D:\mathsf{class}\ (C:t)}$$
$$\frac{E \vdash D:t \quad E \vdash C':t' \quad t \le t'}{E \vdash \mathsf{class}\ C:C'\ \mathsf{is}\ D:\mathsf{class}\ (C:t')}$$
$$\frac{E \vdash T:\mathsf{type}\ d \quad E,(V:\mathsf{val}\ d) \vdash D:t}{E \vdash \mathsf{class}\ C[V:T]\ \mathsf{is}\ D:\mathsf{class}\ (C:[V:d].t)}$$
$$\frac{E \vdash T:t \quad E,(I:t) \vdash D:t'}{E \vdash \mathsf{class}\ C(I:T)\ \mathsf{is}\ D:\mathsf{class}\ (C:(I:t).t')}$$

Figure 16. Relation $E \vdash S:t$

$$\frac{(E, E_{1..i-1} \vdash D_i:\mathsf{class}\ E_i)_{i=1..n}}{E \vdash D_{1..n}:\mathsf{class}\ E_{1..n}}$$
$$\frac{E \vdash T:\mathsf{type}\ d}{E \vdash \mathsf{port}\ X:T:\mathsf{class}\ (X:\mathsf{port}\ d)}$$
$$\frac{E \vdash C:(J:t').t \quad E \vdash I:t'' \quad t' \le t''}{E \vdash C\langle I\rangle:t[J/I]}$$

Figure 17. Relation $E \vdash D:t$

$$E,(I:t) \vdash I:t$$
$$\frac{E \vdash C:(V:d).t \quad E \vdash W:\mathsf{val}\ d}{E \vdash C[W]:t[W/V]}$$
$$\frac{E \vdash I:\mathsf{class}\ (E',(J:t))}{E \vdash I.J:t}$$

Figure 18. Relation $E \vdash I:t$

$$\frac{E \vdash X:\mathsf{port}\ d \quad E \vdash V:\mathsf{val}\ d}{E \vdash X(V)}$$
$$\frac{E \vdash X(V), Y(W)}{E \vdash X(V) \to Y(W)}$$
$$\frac{E \vdash A}{E \vdash !A} \qquad \frac{E \vdash A, A'}{E \vdash A\ \mathsf{or}\ A'} \qquad \frac{E \vdash A, A'}{E \vdash A \parallel A'}$$
$$\frac{E \vdash T:\mathsf{type}\ d \quad E,(V:\mathsf{val}\ d) \vdash A}{E \vdash \mathsf{all}\ V:T\ A}$$
$$\frac{E \vdash T:\mathsf{type}\ d \quad E,(V:\mathsf{val}\ d) \vdash A}{E \vdash \mathsf{any}\ V:T\ A}$$

Figure 19. Relation $E \vdash A$

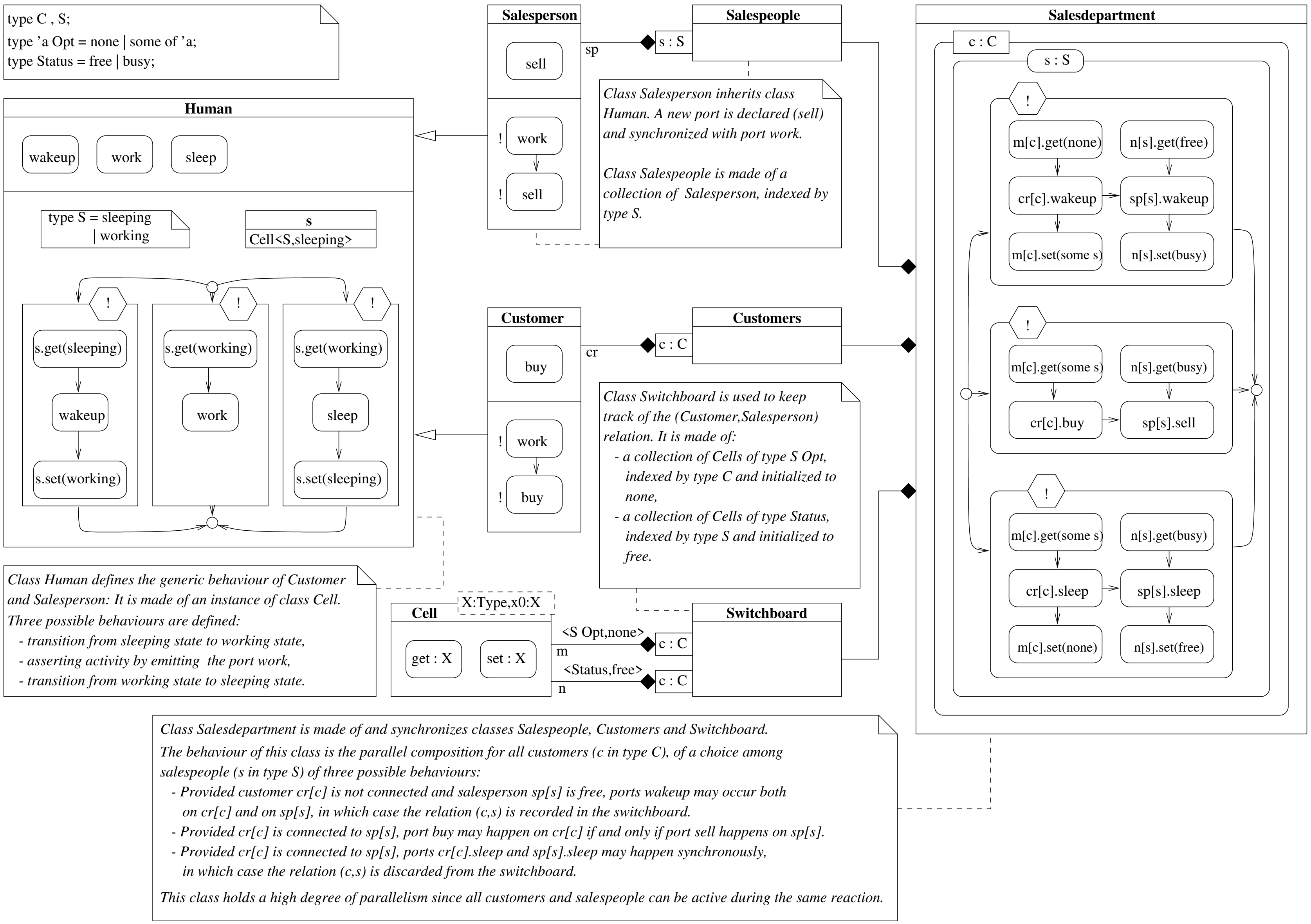


Figure 20. BDL **specification of the salesdepartment**