BDL: A Semantics Backbone for UML Dynamic Diagrams

Benoit Caillaud, Jean-Pierre Talpin, Jean-Marc Jézéquel, Albert Benveniste, Claude Jard

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

BDL : A Semantics Backbone for UML Dynamic Diagrams

Benoît Caillaud , Jean-Pierre Talpin , Jean-Marc Jézéquel , Albert Benveniste , Claude Jard

N˚4003

September 5, 2000

THÈME 1

ISSN 0249-6399 ISRN INRIA/RR--4003--FR+ENG

INRIA
RENNES

BDL : A Semantics Backbone for UML Dynamic Diagrams *

Benoît Caillaud , Jean-Pierre Talpin , Jean-Marc Jézéquel , Albert Benveniste , Claude Jard† ‡

Thème 1 — Réseaux et systèmes
Projet Pampa, Epatr

Rapport de recherche n ° 4003 — September 5, 2000 — 46 pages

Abstract: The UML (Unified Modelling Language) comprises various types of notations, to model the functional architecture, the behaviour of its components, and its deployment. Dynamic diagrams provide descriptions of the components and system behaviour. Examples of dynamic diagrams are collaboration and sequence diagrams to specify high level abstractions for sequences of actions involving several components of the system. Activity diagrams, state diagrams, and statecharts are used to specify the detailed behaviour of a single component.

In this report we propose a new formalism, called Bdl, to serve as a semantic backbone for dynamic diagrams of UML. Bdl diagrams allow to provide a set of UML diagrams a *global* dynamic semantics. It allows to specify the behaviour of systems. It provides a common semantics to the different dynamic diagrams — this report analyses in detail sequence diagrams and statecharts.

Composing components requires different types of communication, synchronous or asynchronous. While a precise description of these choices is essential at deployment stage, it is useful not to bother with this at early design stages. To this end, Bdl supports a flexible, dual synchronous/asynchronous semantics for its communications. It provides sounded support for moving from synchronous to asynchronous communication while preserving dynamic semantics.

We illustrate the use of Bdl on a small example of service adaptation in telecommunications.

(Résumé : tsvp)

* This work was supported by Alcatel under project Reutel2000. For additional information, see `http://www.irisa.fr/pampa/bdl/`

† IRISA, Campus de Beaulieu, 35042 Rennes cedex, France, `surname.name@irisa.fr`

‡ B.C., JP.T., and A.B. are with Inria. JM.J. is with Université de Rennes 1, and C.J. is with CNRS.

BDL: un socle sémantique pour les diagrammes dynamiques d'UML

Résumé : On propose un nouveau formalisme, appelé BDL, destiné à servir de socle sémantique aux diagrammes dynamiques d'UML. BDL permet de conférer une sémantique comportementale globale à un système impliquant plusieurs composants. On peut traduire en BDL différents types de diagrammes, tels que les diagrammes de séquence ou les diagrammes d'états (statecharts), ce qui permet ensuite de combiner ces différents diagrammes.

BDL offre au choix une sémantique synchrone ou asynchrone pour la communication entre composants. Le passage de l'une à l'autre est sémantiquement contrôlé pour être garanti correct. Ceci permet plus de flexibilité au concepteur pour modifier son architecture lors du processus qui conduit de la spécification initiale au déploiement.

Nous illustrons l'utilisation de BDL sur un exercice d'adaptation de service en télécommunications.

Contents

1 Introduction

Since it has been standardized by the OMG in 1997, the UML ("Unified Modeling Language" [20]) has been on its way to become also a *de-facto* standard diffusing at high speed in industrial circles. UML comprises in particular the following types of notations, that bring various worthwhile points of view on the system:

- package and class diagrams, essence of the structuring into objects;
- deployment diagrams: they are used to specify architecture;
- collaborations and sequence diagrams (or scenarios of the type "Message Sequence Charts – MSC"): they are typically used to specify high level abstractions for sequences of actions, procedure calls, and communications instanciating some use cases;
- activity diagrams and "StateCharts": they are typically used to describe the behavior of a component, at least with regard to its control. They are not used to describe the whole behavior of the system. For each component, one can generate the code carrying out this control.

Support for UML diagrammatic notations in object-oriented modeling tools keeps growing, making it easier to use UML in a wide range of contexts, from basic applications for personal computers to large and complex software. The spectrum of UML-modelizable systems will grow up with the next projected releases of the notation involving real-time, scheduling and performance, or enterprise distributed object computing. It is likely that such additions to the UML will push designers to use it for even larger, and more critical software, e.g. in telecommunications where it might ultimately supercede SDL.

In this context, even if the static semantics of UML (relying upon a well defined *meta-model* [8]) is quite well established, the fuzzyness of its dynamic semantics is now seen as a major drawback hampering its use for critical software systems, where there is a strong emphasis on validation. This point has recently been given a lot of attention [13, 7, 17, 14, 18, 15, 2], building on many other efforts to give a formal semantics to some UML aspects (StateCharts [19], MSC etc.) taken in isolation. But there is still the challenge of giving UML a *global* dynamic semantics, fully taking into account all its dynamic diagrams combined with the composability and extensibility brought in by the static diagrams (involving composition/aggregation and inheritance). This paper specifically tries to answer the following questions:

1. How to give a set of UML diagrams a *global* dynamic semantics? How to use UML to provide *systems* with a dynamic semantics, not only objects or components?
2. How can we correctly extend the behavior of a class associated to a statechart, e.g., add an Intelligent Network (IN) service to a Plain Old Telephone System (POTS)? This problem is known as the inheritance anomaly [16], and it is present in UML because of the imperative nature of statecharts.

3. What does it mean on the global dynamic level that two or more dynamic diagrams are composed together (meaning their supporting classes are linked through composition/aggregation or even a simple relation)? Can we be less naive than simply using an asynchronous composition semantics, which is the hell for designers (combinatory explosion of range of possible behaviors) and implementors (too inefficient) alike? Is there a way to abstract the asynchronous/synchronous issues, and delay the semantics choice until the deployment diagrams are known?

To answer these questions we introduced a new formalism, we call it BDL (Behavior Description Language) — an early presentation can be found in [11]. BDL is aimed at serving as a semantics backbone for the UML dynamic aspects, in Section 2 we discuss how BDL should be. We then outline a graphical syntax for BDL (Section 3; a formal presentation of BDL is given in appendix). Although invisible to the UML modeler, this syntax will help us discuss our example (inspired from a Plain Old Telephone System —POTS) from its basic modeling in UML to its semantics translation to BDL and its extension by inheritance to provide a new IN service (Section 4). We then present how the designer can decide on asynchronous vs. synchronous at the latest stage (Section 5).

2 What BDL should be

2.1 BDL in the UML design flow

Figure 1 depicts the way we envision using BDL in the UML design flow. In this figure,

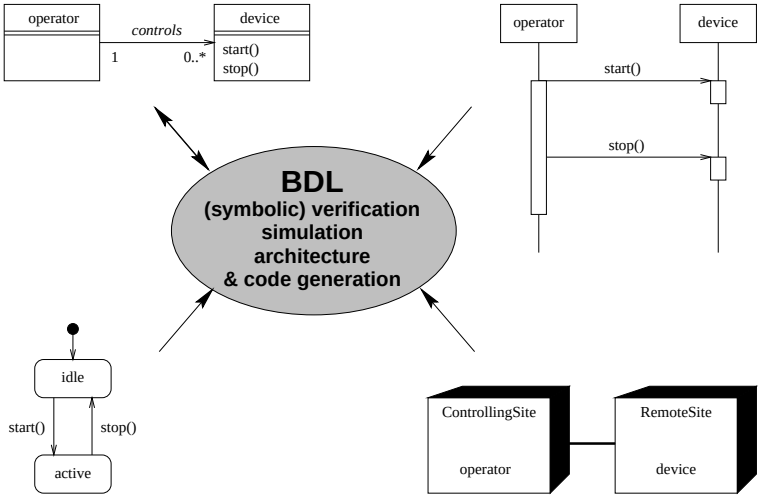

Figure 1: BDL *in* UML.

several notations typical from UML are shown, namely : a class diagram, a sequence diagram, a state diagram, and a deployment diagram. BDL aims at integrating behavioural notations from UML into a single, unified notation based on a well mathematically defined semantics. In particular, state diagrams should be translatable into BDL. Also, sequence or collaboration diagrams, which aim at expressing scenarios, should also be translatable into BDL. While not specifying behaviours, the deployment diagram has also a value for BDL, as it allows to model the deployment architecture, and in particular the type of communication used between instances : BDL will take advantage of the deployment diagram in choosing a suitable semantics for the communication (e.g., stricly synchronous, or using notifications or asynchronous messages). As for class diagrams, they are simply hosts for the BDL diagrams.

From these requirements the following difficulties emerge :

- State diagrams are an imperative style of notation. On the other hand, sequence diagrams express scenarios, they are declarative and express a property. Therefore BDL shall be a declarative notation in which executable programs (e.g., from state diagrams) can also be specified and code generated for them.

- Deployment diagrams allow to specify custom communications (e.g., stricly synchronous, or using notifications or asynchronous messages). Therefore BDL should support a *dual synchronous/asynchronous* semantics for communication.

- Dynamic instanciation should be supported while preserving a clean mathematical semantics for the resulting behaviours.

In the following subsection we discuss our basic idea for merging the styles of different behavioural notations, e.g., state, sequence, and dataflow diagrams. Then we sketch what we mean by a *dual synchronous/asynchronous* semantics for communication.

2.2 From state, sequence, and dataflow diagrams, to partial orders

Our first aim is to design a formalism which serves as a UML "semantic backbone". This means in particular that it should be possible to translate into BDL the different notations used in UML to refer to behaviours. In the figures 2, 3, 4, and 5, we explain why we have chosen BDL to be a formalism to specify transition systems, in which transition relations are specified by means of directed graphs.

Consider first figure 2. The first diagram depicts an automaton (alike a state diagram).

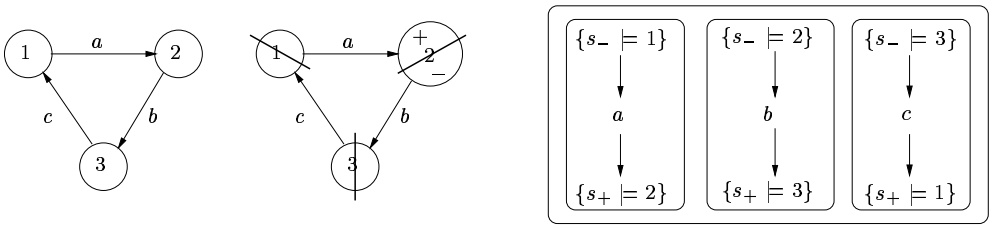

Figure 2: *From automata to* BDL.

In the second diagram we indicate that we divide the states into a sink (marked with a "+") and a source (marked with a "−"). In the third diagram, sinks are indexed by $_+$ and sources by $_-$, while the transitions are just listed; the juxtaposition of the three inner boxes indicates nondeterministic choice among the three reactions. The sources and sinks are written as $\{s_\pm \models 1, 2, 3\}$, meaning that $s_\pm$ satisfies that pre/post state equals $1, 2, 3$. Clearly, this second diagram is a rephrazing of the first one.

Consider next figure 3. We show on the left diagram a scenario with three instances

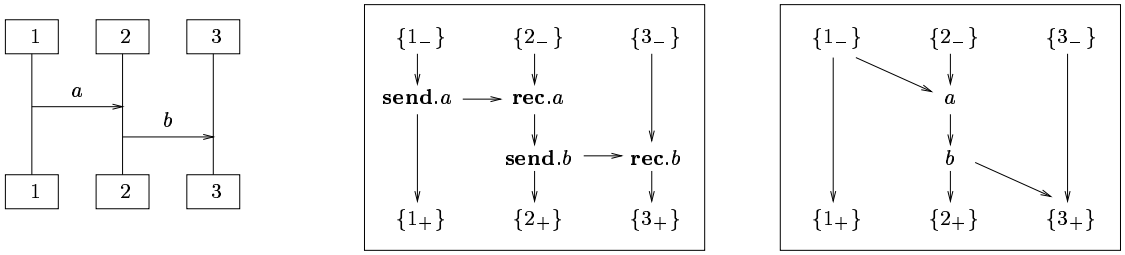

Figure 3: *From scenarios to* BDL. $\{1_-\}$, etc., is a short for $\{1_- \models \top\}$, meaning that pre- and post-conditions for pins $1, 2, 3$ are not specified (this is in accordance with the common use of scenarios which aim at describing semi-formally examples of behaviours).

and two messages (alike a sequence diagram). In the second diagram we re-express it as a partial order. Symbol a which refers, in the scenario, to the transmission of message labelled a, is decomposed into its emission and its reception, with the ordering $\mathbf{send}.a \rightarrow \mathbf{rec}.a$. Then, $\mathbf{rec}.a \rightarrow \mathbf{send}.b$ expresses the ordering between these two message transmissions as specified in the scenario. In the third diagram, we have removed the two events $\mathbf{send}.a$ and $\mathbf{rec}.b$, and called the remaining events simply a and b; this simplification is possible when $\mathbf{send}.a$ is the first event on instance 1, and $\mathbf{rec}.b$ is the last event on instance 3, it will be used in our example of section 4.

Consider figure 4. The left diagram is a dataflow operator, which combines one sample

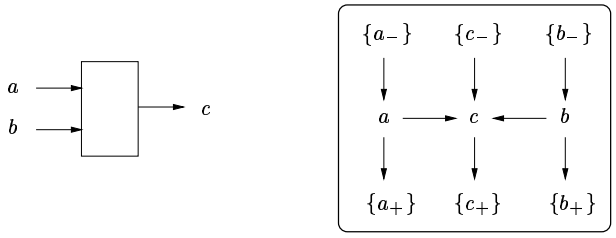

Figure 4: *From dataflow diagrams to* BDL. Note that the pre- and post-pins are in fact not needed, as they specify trivial pre- and post-conditions.

of a with one sample of b and delivers one sample of c. This is abstracted as the resulting causality constraint on the right diagram.

In figure 2, pins $\{s_{\pm} \models \ldots\}$ play an essential role in specifying the transitions of the automaton. In contrast, in the cases of figures 3 and 4 the top and bottom pins do not seem to be useful. They will however play a role in the definition of the runs of such specifications, i.e., in their operational semantics. We discuss this point now.

2.3 Operational semantics : an informal discussion

In BDL diagrams, the squares sitting as minimal and maximal nodes of BDL graphs should be interpreted as states. More precisely, a BDL graph specifies all possible transitions relating previous states (labelled by $-$) to current states (labelled by $+$).

Keeping this in mind, the figure 5 discusses the different ways traces can be built. The top row depicts the synchronous product of two automata, together with the corresponding BDL diagram shown on the right hand side.

The mid row shows a *synchronous* trace, in which states are used as "pins", meaning that, in building the trace, a $s_- \models 2$ is superimposed to the previously written $s_+ \models 2$, etc. (this corresponds in fact to reverting the splitting of state 2, which was performed in figure 2). In doing so, pins are left visible, and depict the successive "reactions" of the system. Then, a run is just the unfolding of these reactions.

The bottom row shows the corresponding *asynchronous* trace, in which pins have been erased, so that only partial order of messages or events remain.

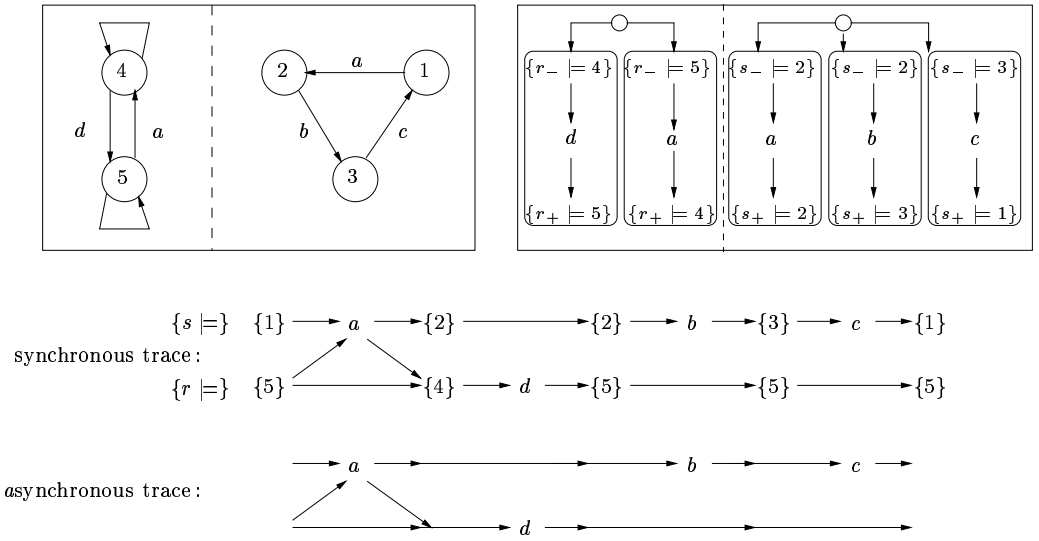

Figure 5: *Synchronous and asynchronous traces.*

3 BDL and its Visual Syntax

In this section, we present a visual syntax for BDL.

3.1 Graphs and their representations

Labelled directed graphs are the primitive visual elements in BDL. There are two kinds of vertices in these graphs: *pins*, which must be extremal vertices (either minimal or maximal) and *messages*. Messages have the form $x(e)$, where $x \in X$ is an operation (or port) of type $t \in T$ with an expression e of the same type t. Pins have the form $\{k_{-/+} \models p\}$ where $k \in K$ is an attribute (or state variable) and $p \in P_k$ is a predicate. Sign $-$ means that the pin is a precondition while sign $+$ is used for postconditions. Preconditions are used to express that some conditions on the attribute must be satisfied before a reaction is executed. All postconditions of a reaction should hold after completion of the reaction. We have chosen to keep close to a scenario type of notation, although any way of drawing such graphs can be used.

In the sequel, $k_- \models p$ and $k.\text{pre} \models p$ are equivalent notations for precondtions and similarly for $k_+ \models p$ and $k.\text{post} \models p$ which both denote postconditions. The set of predicates P_k of attribute k is a boolean algebra where $p \wedge q$ is the conjunction of predicates p and q, and $p \vee q$ is the disjunction of the two same predicates. Valid runs should be such that every precondition of each reaction is a weakening of the most recent postcondition over the same attribute, if it exists, and otherwise of the initial condition.

Here is an example of a basic graph :

$$x.\text{pre} \models v \quad \longrightarrow \quad b(u+v) \quad \longleftarrow \quad a(u) \tag{1}$$

meaning that the value carried by port a at the current reaction, is added to the value carried by attribute x at the previous reaction, and delivered through port b at the current reaction.

3.2 Dealing with synchronization

As seen in figure 5, pins are always involved in each transition (even silent ones). They provide pre- and post-conditions for the reaction in consideration.

On the other hand, referring again to figure 5, a message can be either present or absent in a given reaction. As the present/absent status for messages in a given reaction is important for synchronisation and control, we will regard it as a particular type attached to each message, we call it the *clock* of the message. Messages having the same clock shall be either all present or all absent in the same reaction.

As explained later, we have developed a corresponding typing system for BDL. Hence clocks provide a support for handling synchronisation. As clocks are symbolically manipulated by our typing system, BDL offers an elegant and sophisticated support to handle

synchronisation via syntax. Each BDL term involving messages will have a set of *clock equations* relating the clocks of the different messages. The composition operators presented next reveal and illustrate these mechanisms.

3.3 The Visual Syntax for the BDL Composition Operators

The visual syntax of BDL reuses several composition operators of UML statecharts [10], with enrichments borrowed from SyncCharts [1], see figure 6.

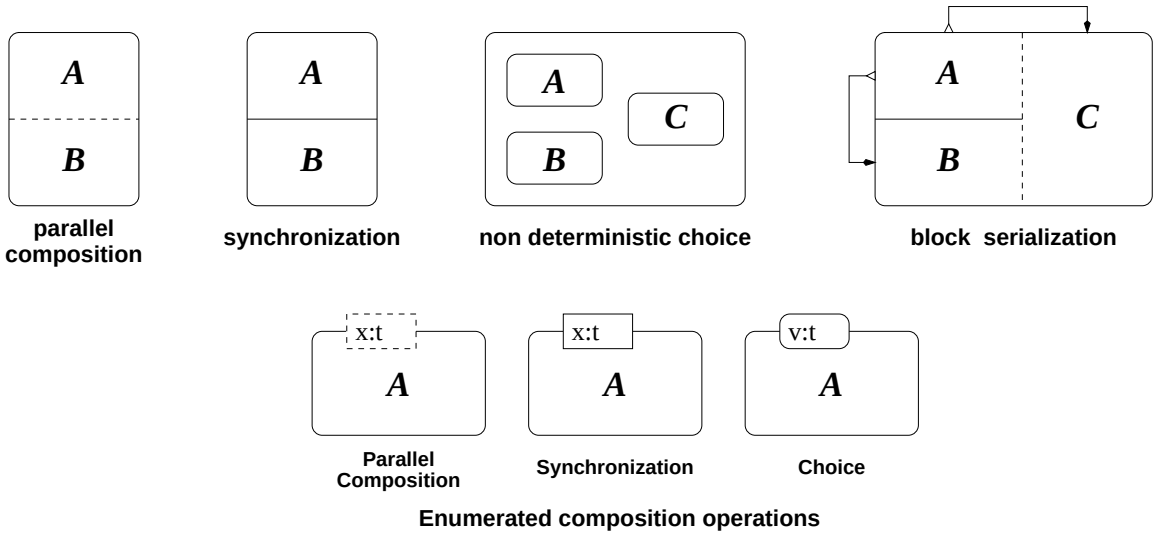

Figure 6: *Visual Syntax for the* BDL *composition operators.*

In this figure :

- Basic elements are *activities.* They are named ***A,B,C***, etc. Activities are, either basic graphs such as in (1), or, recursively, obtained by the composition operators depicted in the figure 6. Activities that involve messages are clocked : we defined the *activation clock* of the activity as being the supremum of the clocks of the messages involved in it. In other words, the considered activity is activated if and only if at least one of its messages is present in the current reaction. Activities with no messages (i.e., having only pre– or post–conditions) are remanent, we call them *unclocked.*

- The *non deterministic choice* between activities is visually represented as *or-states.* In *or-states,* at most one component can be active within a reaction. In textual syntax, the non deterministic choice is denoted by $\vee$. Two or-states have disjunctive activation clocks (this is a first example of a clock equation).

- *Parallel composition* is represented as *and-states* with *dashed* lines partitioning a box. In parallel composition, the components synchronize on their shared messages.

They unify the carried values of present shared messages, as well as of shared pins [1]. If there are no shared messages, then each component can be active in a reaction, independently from the other one. In textual syntax, the parallel composition is denoted by $\|$.

This operator results in the union of the sets of clock equations associated with each component. For $\boldsymbol{A}$ and $\boldsymbol{B}$ two clocked activities, the activation clock of their composition $\boldsymbol{A} \parallel \boldsymbol{B}$ equals the supremum of the activation clocks of the two components $\boldsymbol{A}$ and $\boldsymbol{B}$. If $\boldsymbol{A}$ is clocked but $\boldsymbol{B}$ is not, then the activation clock of $\boldsymbol{A} \parallel \boldsymbol{B}$ equals the activation clock of $\boldsymbol{A}$. If neither $\boldsymbol{A}$ nor $\boldsymbol{B}$ are clocked, then $\boldsymbol{A} \parallel \boldsymbol{B}$ is not clocked either.

- *Synchronization* is represented as *and-states* with *solid* lines partitioning a box. Synchronization is a parallel composition in which all components are forced to activate *simultaneously* (either they are all silent in the considered reaction, or they are all active, this is yet another contribution to the set of clock equations). In textual syntax, the synchronization is denoted by $\wedge$.

  For $\boldsymbol{A}$ and $\boldsymbol{B}$ two clocked activities, the activation clock of their synchronization $\boldsymbol{A} \wedge \boldsymbol{B}$ equals the activation clocks of the two components $\boldsymbol{A}$ and $\boldsymbol{B}$, which in turn must be equal. If $\boldsymbol{A}$ is clocked but $\boldsymbol{B}$ is not, then the activation clock of $\boldsymbol{A} \wedge \boldsymbol{B}$ equals the activation clock of $\boldsymbol{A}$. If neither $\boldsymbol{A}$ nor $\boldsymbol{B}$ are clocked, then $\boldsymbol{A} \wedge \boldsymbol{B}$ is not clocked either (actually using $\wedge$ in this case is allowed but not very meaningful).

- *Serialization* is represented by *arrows* joining sub-boxes of an *and-state* box (synchronization or parallel composition). Referring to figure 6, it cannot be the case that some message from activity $\boldsymbol{B}$ precedes some message from activity $\boldsymbol{A}$.

- An enumerated form for the above operators is also provided. For parallel composition and synchronization we convey that enumeration ranges over a set of ports, as this models instances running in parallel. For non deterministic choice we convey that enumeration ranges over the domain of values of some variable.

Again, the declarative nature of BDL is revealed by its parallel composition : when two instances communicate, they must agree on the message they exchange as well as on the value carried. Let us discuss a few typical examples of this unification mechanism :

- $\boldsymbol{A} \ni x(v) \parallel \boldsymbol{B} \ni x(u) : u = v$, and u is broadcast via input x to $\boldsymbol{A}$ and $\boldsymbol{B}$.

- $\boldsymbol{A} \ni x(e) \parallel \boldsymbol{B} \ni x(u) : u := e$, i.e., expression e is emitted by $\boldsymbol{A}$ via port x and is received by $\boldsymbol{B}$ at port x.

- In $\boldsymbol{A} \parallel \boldsymbol{B}$, and for x a shared port, the absence of x in a given reaction of $\boldsymbol{A}$ blocks the occurrence of any message with port x in $\boldsymbol{B}$.

[1] Recall pins are present in each reaction.

Note that variables are *local* to activities, shared variables do not exist in BDL, and communication occurs only via messages of common port. In turn, messages can be synchronous or asynchronous.

3.4 State transitions

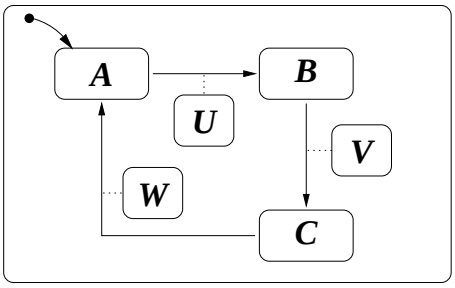

Figure 7: *High level representation of state-transition control structures*

State transitions are a useful construct in state transition diagrams. We propose a graphical notation for in BDL and show that it expands as a macro using the constructs which we already introduced. Figure 7 shows an example of the proposed state transition notation. In this figure, ***A,B,C,U,V,W*** are activities. The three activities ***A,B,C*** form an *or-state,* i.e., they are exclusive. When ***A*** is active and ***U*** is activated, then transition ***A***→***B*** occurs.

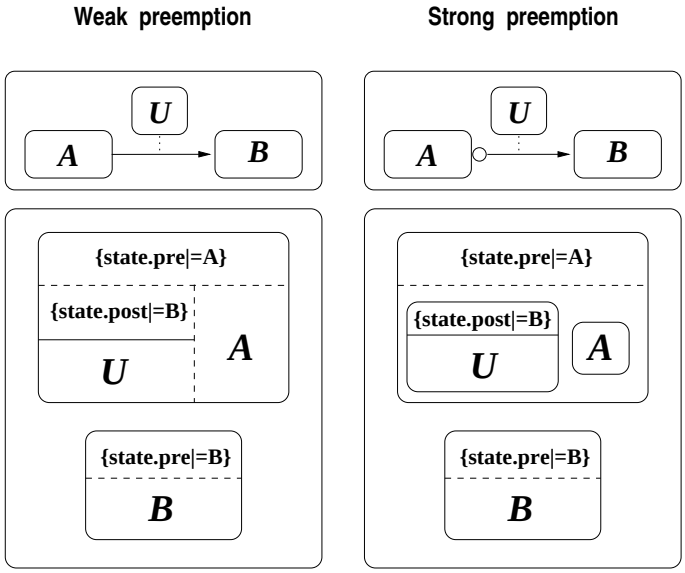

Figure 8: *Expansion of high level representation of state-transition control structures.* The left diagrams correspond to a weak preemption, whereas the right diagrams correspond to a strong preemption. These visual notations will be used in the translation of state diagrams into BDL.

Let us expand this construct as a BDL macro. Two different expansions are proposed in figure 8, with two slightly different graphical notations. The first one corresponds to a weak preemption, and the second one to a strong preemption [2]. Let us comment these expansions :

- For the two cases, states $\boldsymbol{A}$ and $\boldsymbol{B}$ are exclusive. To indicate which state is active in the current reaction, we introduce the port **state**, of enumerated type with values in set $\{\boldsymbol{A},\boldsymbol{B}\}$. A pin $\{\mathbf{state.pre} \models \boldsymbol{A}\}$ indicates that state $\boldsymbol{A}$ was the last active state (note that neither $\boldsymbol{A}$ nor $\boldsymbol{B}$ may have been active at the previous reaction). Similarly, a pin $\{\mathbf{state.post} \models \boldsymbol{B}\}$ indicates that state $\boldsymbol{B}$ will be the next active state.

- The visual organization of solid and dashed lines (resp. depicting the $\wedge$ and $\|$ composition operators) indicates parenthesizing. In particular, the 4-*and-state* on the bottom left means

$$\{\mathbf{state.pre} \models \boldsymbol{A}\} \quad \| \quad (\,(\,\{\mathbf{state.post} \models \boldsymbol{B}\} \wedge \boldsymbol{U}\,) \,\|\, \boldsymbol{A}\,) \tag{2}$$

  The corresponding activity in the diagram of the right hand side means:

$$\{\mathbf{state.pre} \models \boldsymbol{A}\} \quad \| \quad (\,(\,\{\mathbf{state.post} \models \boldsymbol{B}\} \wedge \boldsymbol{U}\,) \vee \boldsymbol{A}\,) \tag{3}$$

- In both cases of strong and weak preemption, the term $(\,\{\mathbf{state.post} \models \boldsymbol{B}\} \wedge \boldsymbol{U}\,)$ occurs. The rules of synchronization for the $\wedge$ composition operator require that the two activities $\boldsymbol{U}$ and $(\,\{\mathbf{state.post} \models \boldsymbol{B}\} \wedge \boldsymbol{U}\,)$ have identical activation clocks. Hence, whenever $\boldsymbol{U}$ is active, then postcondition $\{\mathbf{state.post} \models \boldsymbol{B}\}$ must result. In other words, $\boldsymbol{B}$ will be the next active state when $\boldsymbol{U}$ occurs.

  Also, in both cases, $\boldsymbol{U}$ and $(\{\mathbf{state.pre} \models \boldsymbol{B}\} \,\|\, \boldsymbol{B})$ are exclusive, meaning that, when $\boldsymbol{U}$ occurs, it is not the case that $\boldsymbol{B}$ is either active or was the last active state. To summarize, $\boldsymbol{U}$ triggers the transition and can occur only when $\boldsymbol{A}$ is, or was just, active.

- The two cases differ in the way $\boldsymbol{U}$ is composed with $\boldsymbol{A}$. In the case of weak preemption $\boldsymbol{U}$ and $\boldsymbol{A}$ are in parallel composition, i.e., they can be both activated in the considered reaction. Therefore the occurrence of $\boldsymbol{U}$ when $\boldsymbol{A}$ is active does not kill $\boldsymbol{A}$ immediately, but only at the next reaction. In the case of strong preemption, $\boldsymbol{U}$ and $\boldsymbol{A}$ are exclusive, and therefore the occurrence of $\boldsymbol{U}$ prevents $\boldsymbol{A}$ from occuring.

For a formal definition of BDL and its mathematical semantics, see appendix A.

[2] We have used the SyncCharts notation to distinguish between these two different types of preemption.

4 An example: the plain old telephone service (POTS)

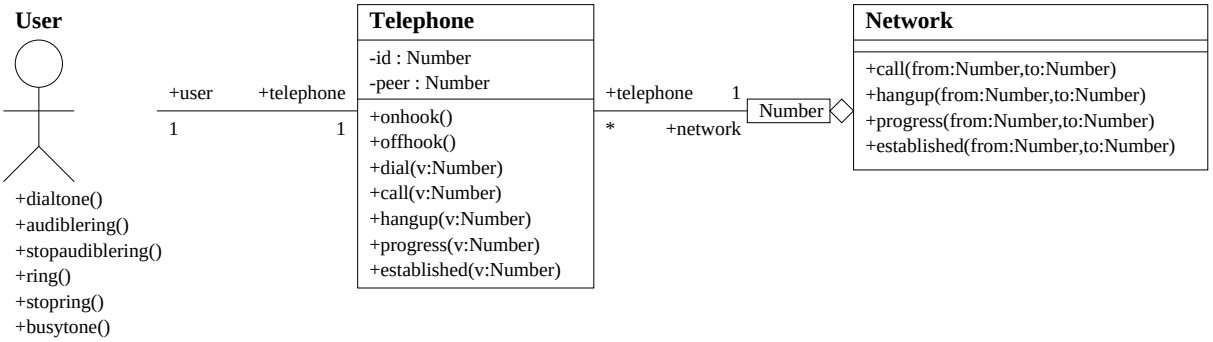

Figure 9: *The plain old telephone service:* class diagram

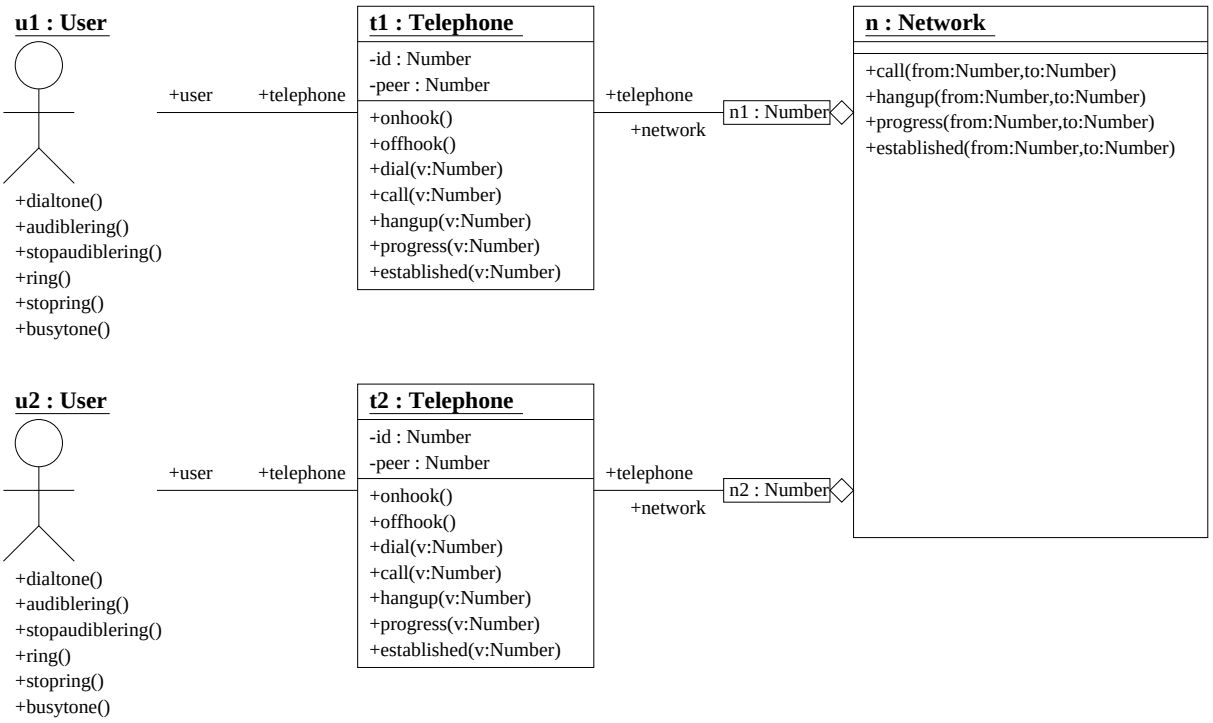

Figure 10: *The plain old telephone service:* object diagram

This section is dedicated to an in-depth application of BDL on a typical case-study: the *plain old telephone service* (POTS).It is a simplified telephony system comprizing terminals, suscriber-line management and an over-simplified telecommunication network. The service is rather basic and boils down to call-ring-answer-hangup and call-busy possible behaviours.

4.1 Specification of the POTS

The architecture of the software is defined in the class diagram of figure 9 and the object diagram of figure 10 gives a typical instantiation of the classes: terminals are modeled by class **User**, suscriber-line management is performed by class **Telephone** (its behaviour is defined by the state diagram of figure 11) and message transmission is achieved by class **Network** (figure 12).

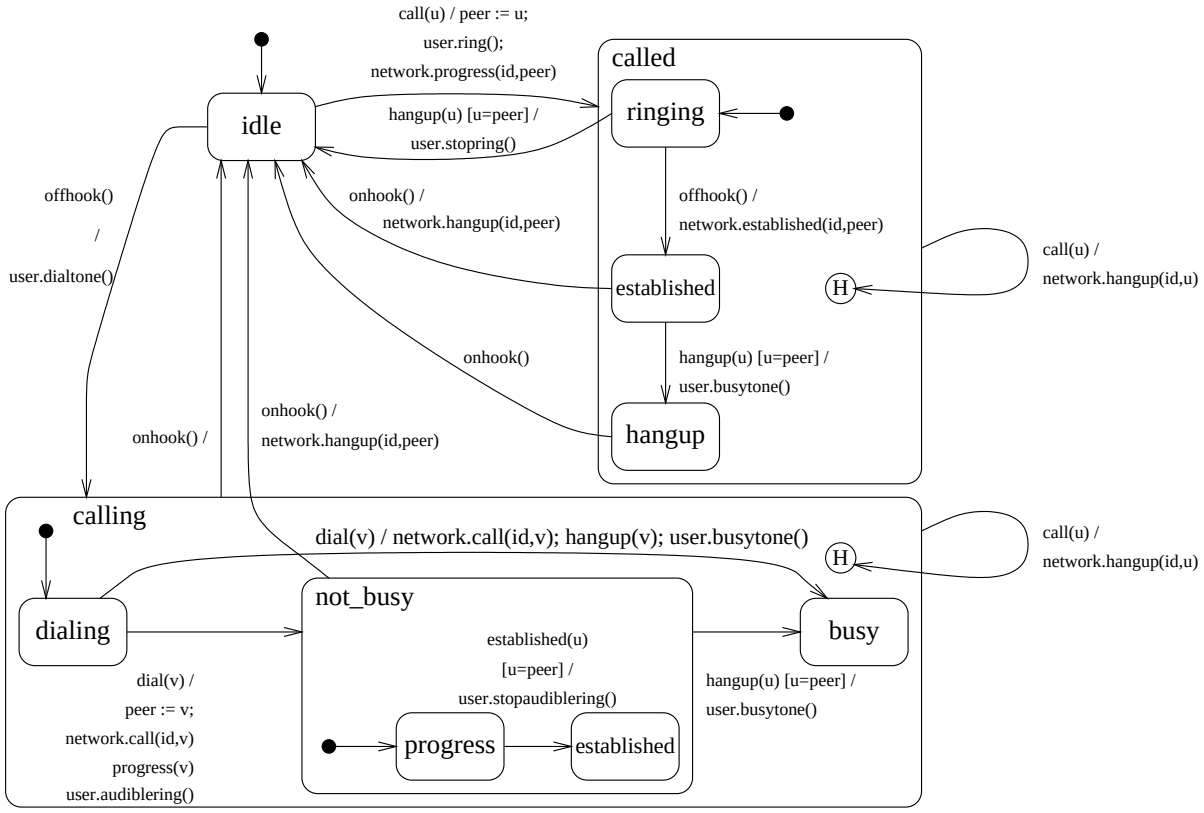

Figure 11: *The plain old telephone service:* state diagram of class **Telephone**. Attributes for the local and distant telephones are id and peer, respectively.

The semantics of interaction between objects is synchronous and in this respect differs from the standard UML semantics. This leads to much simpler state-charts in which transitions may be labeled by sets of events and actions — not just an event followed by a sequence of actions, see figure 11.

4.2 Service adaptation : call forward on busy

4.2.1 Specifying the additional feature using scenarios

We aim at upgrading the POTS system with an additional *feature* (also called "service" in the sequel): the *call_forward_on_busy* service. The principle of this service is to redirect

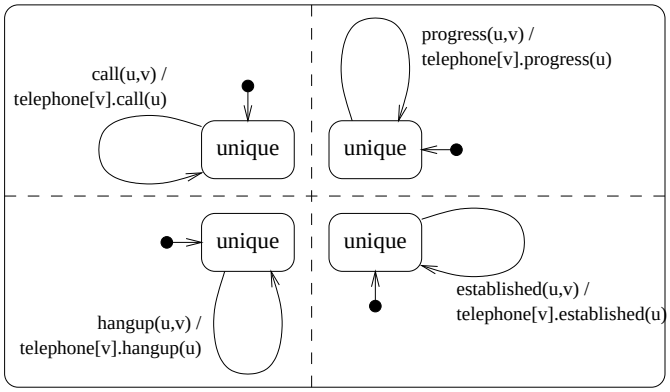

Figure 12: *The plain old telephone service:* state diagram of class **Network**. Note that, in reaction call(u,v)/telephone[v].call(u) , the two referred messages belong to *different* telephones. The same holds for each reaction.

calls aimed at suscriber y to a predefined suscriber z whenever y is not ready to accept a new call — either that y is calling or that it is being called. A typical scenario of this *call_forward* service is given in figure 13. The scenario of figure 13 terminates with a

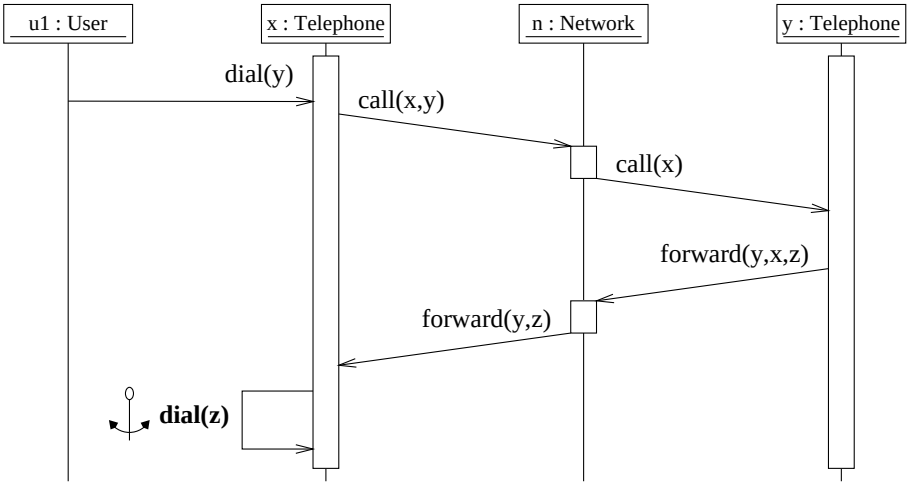

Figure 13: *The plain old telephone service:* sequence diagram of the *call_forward_on_busy* feature, note the *anchor*.

marked event **dial(z)**, we call it an *anchor*. The intuitive interpretation of an anchor is that of a "call back" :

once the anchor is emitted, then things happen as in the original system.

Note that the anchor is a "copy" of the "dial(z)" message from the original specification, but the difference is that it is emitted by the telephone itself, not by the user — it implements a call-back. To construct the adapted class **Telephone'** (see figure 14), we need to adapt the class **Telephone** by performing the following :

1. The scenario of figure 13 involves two telephones, the calling one, and the forwarding_on_busy one. We fold the corresponding two parts of the scenario into two alternative behaviours for a single telephone, for the two respective cases of calling and forwarding_on_busy.

2. We need to construct a ***Glue*** that will implement the informal requirement that

   > on "forwarded_on_busy", call back the new number and then behave as already specified.

3. Finally we need to combine the behaviour of the **Telephone** class (specified using a *state diagram*) with that of the **call_forward_on_busy** *scenario* by using the ***Glue***.

Due to the need for combining state diagrams and sequence diagrams (cf. item 3 above), we think this service adaptation is best implemented using BDL. Therefore we first proceed on translating the whole POTS behaviour into BDL.

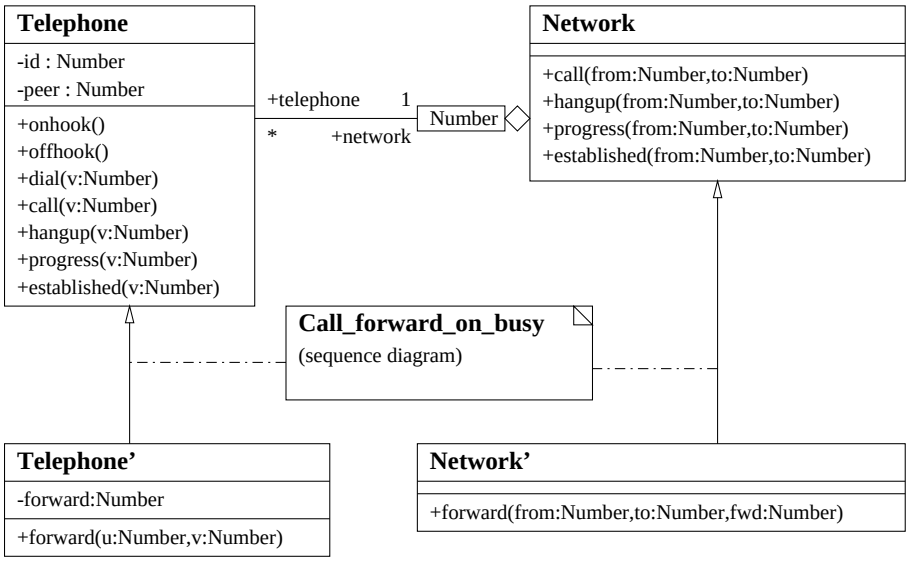

Figure 14: *The plain old telephone service:* class diagram of the *call_forward_on_busy* feature. Note the inheritance via sequence diagram of the *call_forward_on_busy* feature.

4.2.2 System behavioural specification using BDL diagrams

First, state diagrams of classes **Telephone** and **Network** are translated into BDL diagrams (figures 15 and 16). Note the following about this translation :

- The BDL diagrams are just the collection of the different possible transitions of the corresponding state diagrams, organized into nested ***or***-states.

- The state is encoded using an additional attribute "state", of enumerated type. This attribute has the set of possible state diagram configurations as its domain. The different primitive BDL diagrams have preconditions involving attribute "state". The nesting of the different states in the state diagrams is encoded, e.g., using notation "called.ringing" to refer to "ringing" substate of state "called". Moves to next state are indicated by using postconditions.

- Similarly, the peer telephone is encoded using attribute "peer", which occurs in the preconditions.

Note the following about behavioural system specification using BDL. Consider the two BDL diagrams of figures 15 and 16, and call them respectively ***telephone*** and ***network*** or ***t*** and ***n*** for short. Interpret the structure of the object diagram of figure 10 as specifying a $\|$ parallel composition. With this in mind, we get that the whole system of figure 10 has

$$\boldsymbol{u1} \parallel \boldsymbol{t1} \parallel \boldsymbol{n} \parallel \boldsymbol{t2} \parallel \boldsymbol{u2} \tag{4}$$

as its BDL diagram, where the BDL diagram ***u*** for the user accepts any behaviour involving its possible messages.

4.2.3 Preparing the scenario for subsequent integration using BDL diagrams

Now we return to our main purpose, namely the translation of the sequence diagram of figure 13 into a suitable BDL diagram. This scenario is first globally translated into BDL, see figure 17. Three problems are revealed while inspecting the original scenario of figure 13 :

1. This scenario does not specify *when* it applies, i.e., what its activation conditions are. In fact, figure 13 is an *incomplete* specification of the "call_forward_on_busy" feature, and its activation conditions need to be provided by the user. Extensions of the MSC notations would be helpful for this purpose, for instance the LSC proposal by Harel and Damm [6]. Here we shall directly instead resource to BDL preconditions.

2. In figure 13, there are two occurrences of a message of the form "dial(.)". Therefore, when translating this scenario into BDL, these two occurences cannot belong to the same reaction.

3. In addition, these two occurences are both received by telephone "x", but they are emitted by different objects. This mechanism needs to be made explicit when translating the scenario into BDL.

Let us first focus on the first diagram of the figure 17. It is just a slight modification of the scenario of figure 13. First, we have changed the name of the call-back message, which now stands **selfdial**. Then, we have split the scenario, just before the anchor, into two successive reactions.

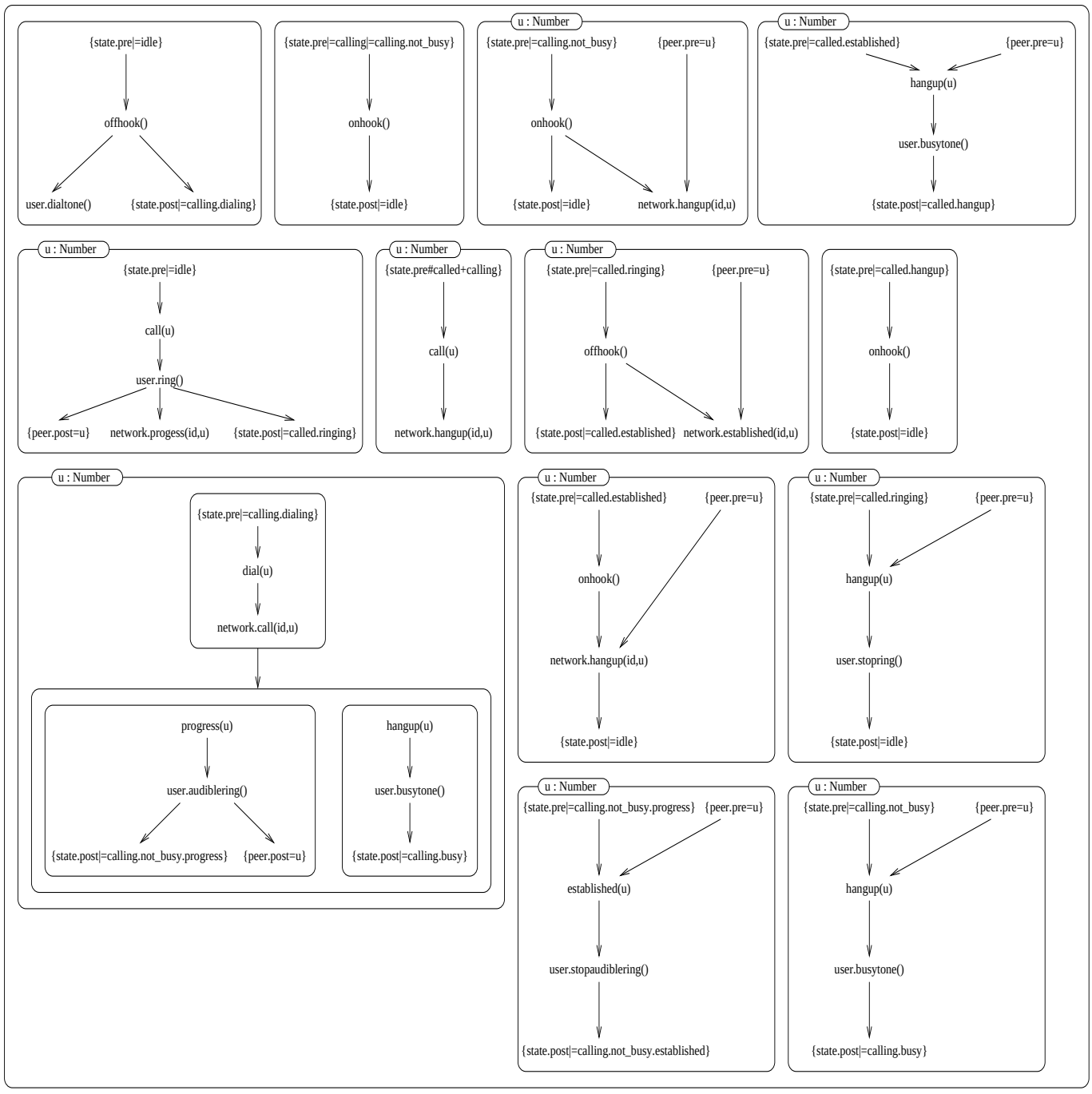

Figure 15: *The plain old telephone service:* BDL diagram of class **Telephone**

Next we move on the second diagram, which is a BDL diagram. First, we have manually added the three preconditions marked in dark grey, these are selfexplanatory. Second, we have translated the split into two successive reactions in the form of an ***or***-state in BDL.

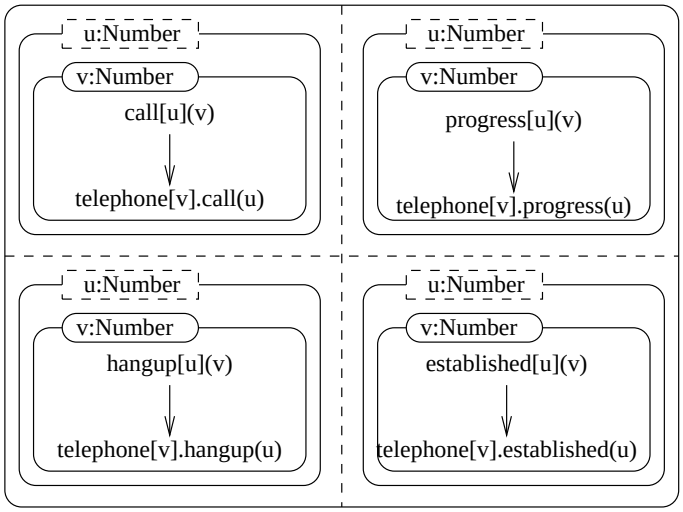

Figure 16: *The plain old telephone service:* BDL diagram of class **Network**. The bracket "[u]" indicates that "u" is a parameter indexing the different telephones, and therefore "call[u]" is an event and "call[u](v)" is the whole message (event plus carried value). The use of the ***and***-state indexed by u indicates that the network is used concurrently by the different telephones, as specified in class diagram of the figure 9.

Note the introduction of the new value "forwarded" in the domain of the "state" attribute — this modification is also seen in the class diagram of the class **Telephone'** in figure 14.

Finally, to indicate that the two "dial" and "selfdial" messages merge into a single equivalent dial message, we introduce a new "mergedial" message, which is an exclusive merge of "dial" and "selfdial", this formalizes the mechanism referred to in item 3 above.

4.2.4 Folding the sequence diagram of the *call_forward_on_busy* feature into a single BDL diagram for each class

At this stage, we need to adapt the **Telephone** class and its behaviour in such a way that system (4) behaves as the BDL diagram of figure 17 when its activation conditions hold.

This BDL diagram involves three objects, namely the two telephones "x" and "y", and the network "n". For each object we consider the restriction of BDL diagram of figure 17 to its messages, then we fold the two BDL diagrams of the two telephones "x" and "y" into a single BDL diagram for the **Telephone** class, this is shown in figure 18. The same is performed for the **Network** class, this yields a trivial BDL diagram in this case, as it sets no constraint on the network behaviour.

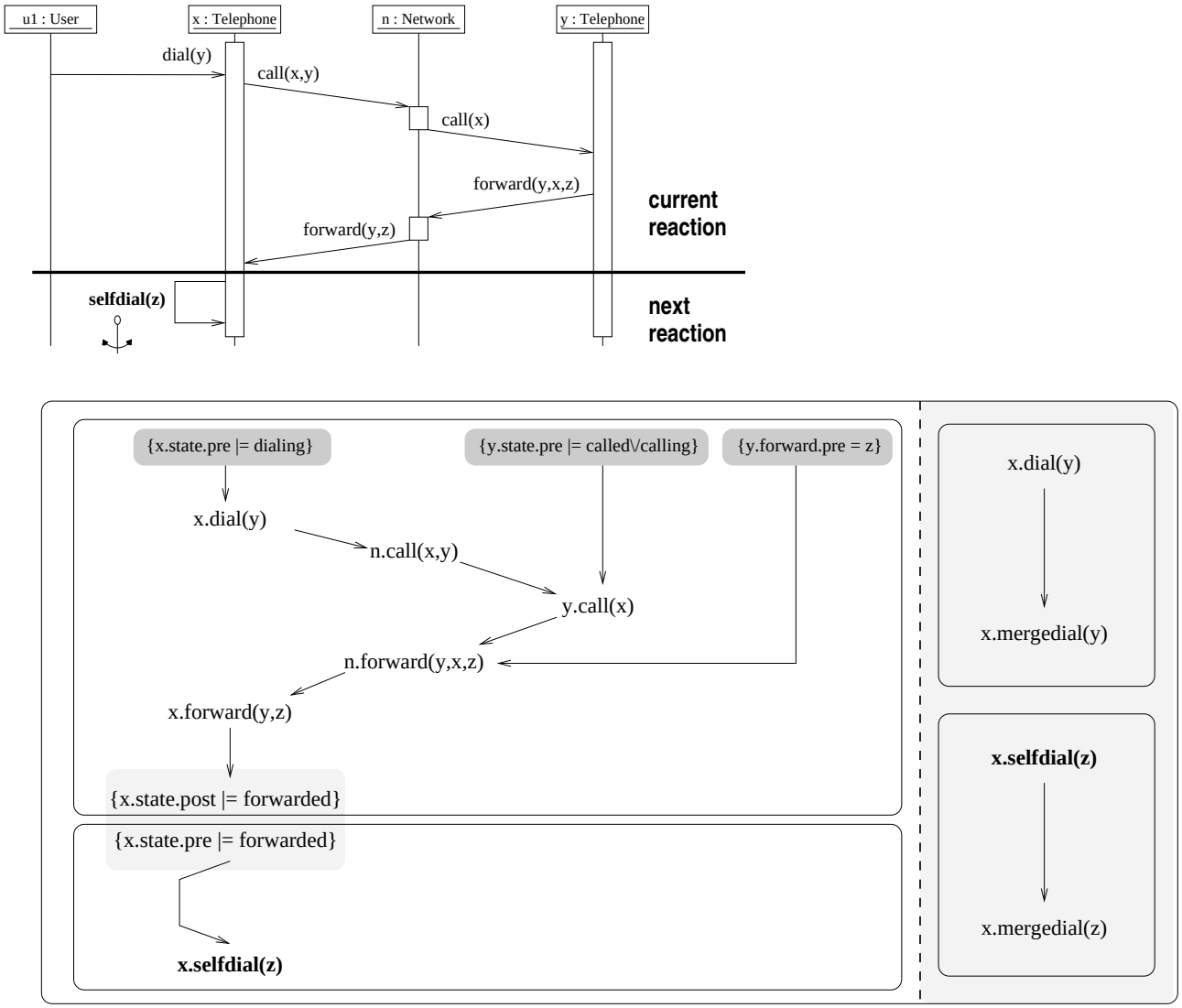

Figure 17: *The plain old telephone service:* split sequence diagram of the *call_forward_on_busy* feature, and derived BDL diagram. We have emphasized in light grey the new post/pre-condition on attribute **state**, as well as the "merge" glue. Dark grey indicates add-ons requested from the *user*, whereas light grey indicates automatic add-ons.

4.2.5 Integrating the BDL feature specification with the original BDL specification to derive service adaptation

Behaviour ***telephone'*** of class **Telephone'** is defined from behaviour ***telephone*** of class **Telephone** in the following way :

$$\textbf{\textit{telephone'}} = \left(\begin{array}{l} \textbf{\textit{call\_forward\_on\_busy}} \\ \| \\ \textbf{\textit{telephone}}[\text{dial/mergedial}] \\ \| \\ \textbf{\textit{glue}} \end{array} \right) \quad (5)$$

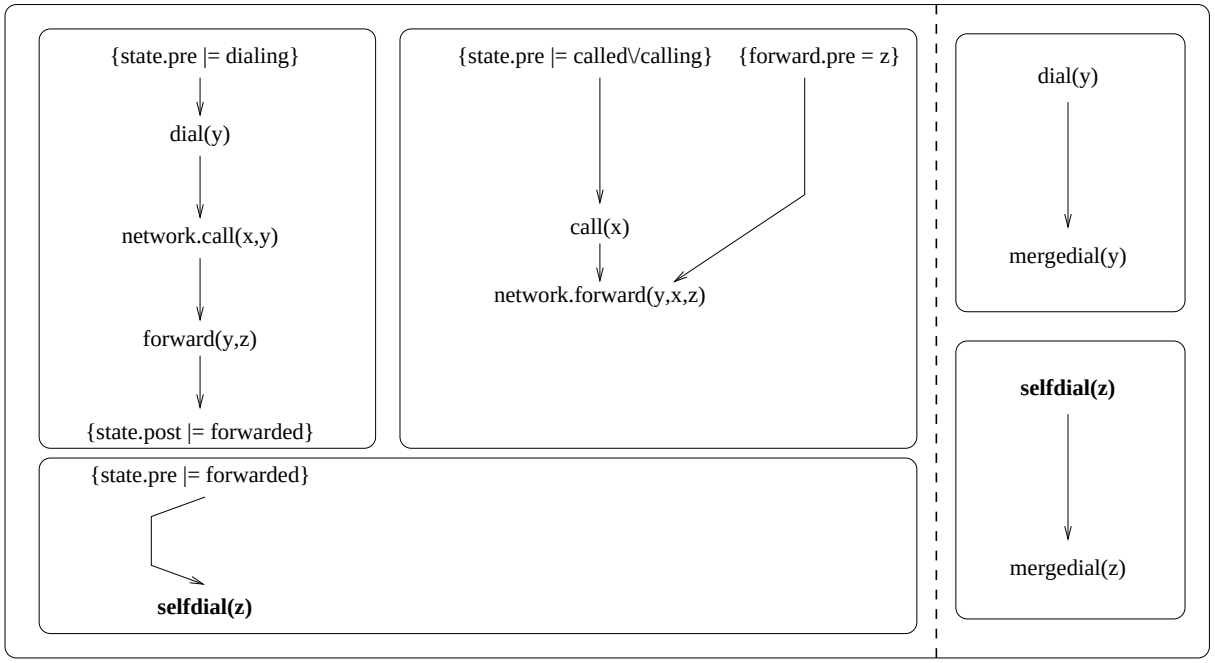

Figure 18: *The plain old telephone service:* BDL diagram of the *call_forward_on_busy* part of class **Telephone'**. This diagram is obtained as follows. We start from the BDL diagram of figure 17. Then we split it into what is relevant to telephone **x** and what is relevant to **y**. Finally we fold the two corresponding diagrams into a single one, for each case of initial calling telephone, and forwarding telephone.

where "[dial/mergedial]" denotes renaming of "dial" by "mergedial", **Glue** is an additional BDL term intended to activate the new feature on due conditions, and inhibit the original behaviour in the same case. The BDL diagram ***glue*** of class **Glue** is shown in figure 19 (behaviour of class **Network'** is enhanced in much the same way, the only difference being that no glue BDL term is required). The following comments are in order about figure 19:

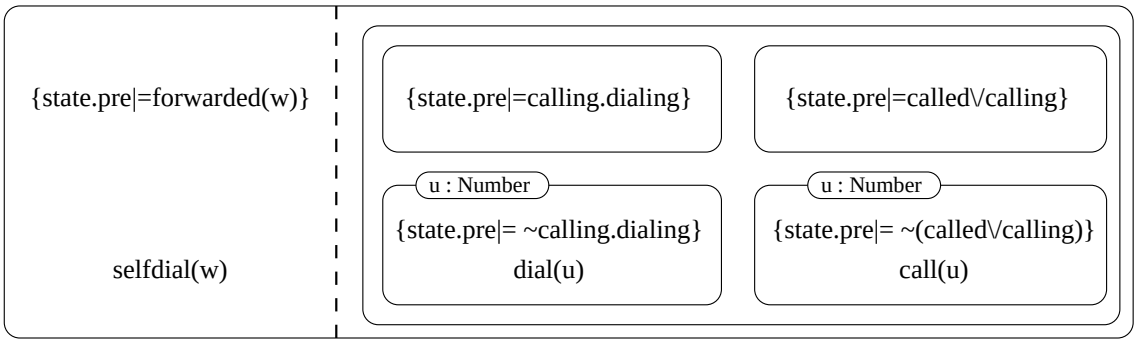

Figure 19: *The plain old telephone service:* glue BDL term for class **Telephone'**

- The highest level of the ***glue*** diagram is an ***and***-state, with two components. The first component requires that, whenever precondition " {state.pre⊨forwarded(w)} " holds, then "selfdial(w)" must be emitted immediately, i.e., in the current reaction. Then we know from figure 18 that, once "selfdial(w)" is emitted, then "mergedial(w)" is also emitted. This implements the call back mechanism : call back occurs on the anchor.

- The second component of this ***and***-state has "dial" and "call" as its declared events. Therefore the *absence* of one of these two events in the considered reaction is synchronizing. In particular, precondition " {state.pre⊨calling.dialing} " forbids the occurrence of message "dial(u)". On the other hand, the same precondition activates the BDL diagram of figure 18. Hence the ***glue*** implements the informal requirement that the considered precondition 1/ disables the normal behaviour of class **Telephone**, and 2/ enables the alternative behaviour implementing the added feature.

4.3 Service adaptation, a systematic approach using BDL

In this subsection we summarize the approach of the POTS example and explain how it can be made systematic.

4.3.1 The specification

It consists of the following items.

The basic service. It is specified using :

- a class diagram for its structure,
- an object diagram for its instanciation, and
- a state diagram for each class to specify its behaviour.

In the class and object diagrams, we interpret the relation between classes as behaving like the parallel composition $\|$ of its components. Using the translation of state diagrams into BDL for each class[3], we derive a specification for the behaviour of the whole system, very much like in subsection 4.2.2. Assume that the object diagram comprises objects $\mathbf{O}_1, \ldots, \mathbf{O}_n$, and denote by $\boldsymbol{O}_1, \ldots, \boldsymbol{O}_n$ their corresponding BDL diagrams, the system BDL diagram $\boldsymbol{S}$ is given by

$$\boldsymbol{S} \;=\; \boldsymbol{O}_1 \,\|\, \ldots \,\|\, \boldsymbol{O}_n \tag{6}$$

[3] We have built a draft prototype of this translation [21].

The additional feature for service adaptation. It is specified using a finite set of sequence diagrams and associated *activation conditions*[4]. Each sequence diagram shall be a basic MSC, and the set of its maximal messages constitutes its *anchor,* and we assume that messages from the anchor are also messages from the basic service. The intention is that, once the anchor has been emitted, the basic service is resumed.

4.3.2 Translating the additional feature into BDL

First, we need to split each sequence diagram into successive synchronous reactions. A requirement for a set of messages to belong to the same reaction is that

two different messages shall not be labelled with the same event. **(React)**

As this requirement is not satisfied for general sequence diagrams, we need an algorithm for automatically splitting a sequence diagram into successive reactions. The following backward greedy algorithm is a convenient solution:

initialization : last reaction = *anchor*;
loop : (1) backward augment the current reaction by adding messages not violating condition **(React)**;
(2) insert a new pre/postcondition to mark the change of reaction.

The algorithm is correct since by construction it is not possible, for an anchor, to violate condition **(React)**.

Second, in comparing the sequence diagram of the additional feature and the state diagrams associated with the basic service, it may happen that the same event is emitted by *different* objects (this was for instance the case of the "dial" event in the POTS example). The solution presented in figure 17 can be generalized and made automatic in a trivial way.

When these two transformations are done, translating each sequence diagram into a BDL diagram is straightforward. The collection of sequence diagrams is then translated as the $\|$ parallel composition of their corresponding BDL diagrams, we denote it (symbol $\boldsymbol{F}$ is reminiscent of "feature")

$$\boldsymbol{F} \;=\; \boldsymbol{F}_1 \parallel \ldots \parallel \boldsymbol{F}_m \tag{7}$$

Our requirement is that, when proper preconditions are activated, the adapted system $\boldsymbol{S}$' should inhibit its behaviour (6) and behave like (7). This is *not* to be achieved by a global adaptation of the overall system, but rather by a local adaptation of each class. We discuss this next.

[4] As already said, such activation conditions could be specified using enhanced versions of sequence diagrams, in this work we simply use BDL itself for this purpose.

4.3.3 Folding the BDL feature diagram into each class

In general the BDL diagram of the feature for adding traverses several objects. To derive the service adaptation for each object, we need to fold this BDL diagram, this is simply achieved by means of computing a restriction to the set of messages known by each class. Computing restrictions is a standard operation in BDL and does not raise any particular problem. At this stage we have, for each class,

1. its original behaviour in the form of a state diagram, and
2. its restricted service adaptation.

Formally, for $\boldsymbol{F}_i$ a BDL feature diagram and $\mathbf{O}_j$ an object, we denote by $\boldsymbol{F}_{ij}$ the restriction of $\boldsymbol{F}_i$ to the set of events relevant to object $\mathbf{O}_j$, and we set

$$\boldsymbol{F}_{\bullet j} \;=\; \boldsymbol{F}_{1j} \parallel \ldots \parallel \boldsymbol{F}_{mj}$$

Since operator $\parallel$ is commutative, associative, and idempotent ($P \parallel P = P$), we get

$$\boldsymbol{F} \;=\; \boldsymbol{F}_{\bullet 1} \parallel \ldots \parallel \boldsymbol{F}_{\bullet n} \tag{8}$$

Now we need to recombine $\boldsymbol{F}_{\bullet j}$ with the BDL behaviour $\boldsymbol{O}_j$ in such a way that the combination disables the normal behaviour $\boldsymbol{O}_j$ and activates the feature $\boldsymbol{F}_{\bullet j}$ instead. This is analyzed in the next subsection.

4.3.4 Gluing original objects with the new service

We need to introduce the following notations regarding the preconditions and messages of diagram $\boldsymbol{F}_{\bullet j}$:

α : a generic anchor of $\boldsymbol{F}_{\bullet j}$;
$pre(\alpha)$: the precondition preceding anchor α ;
$pre(\boldsymbol{F}_{\bullet j})$: a generic activation precondition for $\boldsymbol{F}_{\bullet j}$;
$m(pre(\boldsymbol{F}_{\bullet j}))$: the set of messages following precondition $pre(\boldsymbol{F}_{\bullet j})$ in BDL diagram $\boldsymbol{O}_j$.

Instanciation of these notations for the POTS example are :

α : **selfdial(z)**
$pre(\alpha)$: {state.pre|=forwarded}
$pre(\boldsymbol{F}_{\bullet j})$: {state.pre|=dialing}, resp. {state.pre|=called∨calling}
$m(pre(\boldsymbol{F}_{\bullet j}))$: dial(y), resp. call(x)

The needed glue is given by :

$$\boldsymbol{glue}_j \;=\; \left(\begin{array}{l} \|_{\alpha}\ (\, pre(\alpha)\, \alpha\,) \\ \| \\ \vee_{pre(\boldsymbol{F}_{\bullet j})}\ (\, pre(\boldsymbol{F}_{\bullet j})\,) \vee (\, \sim pre(\boldsymbol{F}_{\bullet j})\, m(pre(\boldsymbol{F}_{\bullet j}))\,) \end{array} \right) \tag{9}$$

and, finally,

$$\boldsymbol{O}_j{}' \;=\; \boldsymbol{O}_j \parallel \boldsymbol{glue}_j \parallel \boldsymbol{F}_{\bullet j} \tag{10}$$

5 Deploying BDL diagrams

5.1 Motivations and objectives

Current practice

Deploying UML diagrams onto a distributed architecture possibly raises serious difficulties. UML and object oriented modelling provides assistance for inheritance, instanciation, interfacing, and associated typing. So far this was the good side of the picture, now come the bad news.

Inter objectcommunication mechanisms are hidden behind higher level notions of message passing or remote function call. Side effects with access to shared variables can also be hidden behind black-box actions. The actual implementation of the associated communication mechanisms by the underlying executives are diverse and not semantically formalised. While keeping informal is within the philosophy of UML modelling at early stages of the design, this is no longer acceptable when one comes to the deployment phase. There, designs are assumed clean enough to warrant effective deployment, and uncertainties in behaviour are certainly undesirable at this stage. On the other hand, everyone knows that one gets very easily trapped into subtle sources of bugs when *distributed* deployment is being performed. This is why strong programming disciplines are frequently advocated, by relying on simple but rigid architectures (e.g., client-server).

One of the major objectives of BDL is to provide strong assistance for flexible but correct *distributed* deployment. Here correctness refers to behaviours.

Requirements

We have identified the following needs when deployment comes to consideration :

(a) We need a formal, but maximally permissive model of the communication infrastructure. Formality is needed for supporting the mathematical reasoning needed to secure "correct deployment". But a precise model can also be unacceptable from the practical point of view : the assumed formal model may not be satisfied by the dominant communication infrastructures. Therefore our approach is to still keep formal, while reducing our requirements to the very minimum and keeping them easy to grasp, even from an intuitive point of view.

(b) The functional architecture of an application is attached to the application, not necessarily to the implementation infrastructure. Indeed, modern development techniques ask for reusing the same application on different implementation infrastructures. Hence we see the need for keeping specification and implementation architectures mostly independent from each other.

For instance, one frequently groups several objects onto the same capsule at deployment. Assume that the architecture of the application is rigid, this means that message passing must keep the only way inter object communication occurs, be it

between different capsules or within a single capsule. Therefore we need to resource to the implementation of message passing, both for communication between capsules, and within the same capsule. Using message passing within the same capsule is clearly causing overhead, producing a scheduling of the different object executions at compile time would be more efficient.

These remarks lead to a very simple idea : if we know how to handle architectures in a flexible manner, then the above difficulties would be overcome. We like to group several objects into a cluster that can again be regarded as an object, and can be considered as such when deployment comes to consideration. Instead of handling flat object diagrams as usual in UML, we prefer to have hierarchical BDL diagrams, in which objects can be grouped recursively, in a full fledged hierarchical manner.

Our approach

To satisfy requirement (b) we have chosen to resource the the so-called *synchronous programming* approach [4] [9]. Let us recall the main features of this approaches, as borrowed from [5, 3]. The synchronous programming paradigm is essentially characterized as follows :

1. Behaviours progress via an infinite sequence of *reactions*: $P = R^\omega$, where R denotes the family of possible reactions.

2. Within a reaction, decisions can be taken on the basis of the *absence* of some events.

3. Parallel composition is given by taking the pairwise conjunction of associated reactions, whenever they are composable: $P_1 \| P_2 = (R_1 \wedge R_2)^\omega$.

Feature 3 provides us with a satisfactory answer to our request (b) : the above notion of parallel composition is simple, and is both associative and commutative. Behaviours can be grouped recursively to form a new behaviour. Grouping is revertible and de-grouping just amounts to removing the parentheses in a parallel composition $\|$ expression.

Also, the rich technology developed for the compilation of synchronous programs provides us with an efficient implementation of inter-object communications, for objects sitting on the same capsule. Within the same capsule, the synchrony hypothesis is acceptable. And the compilation techniques from synchronous languages transform the inter-object communication into a precomputed scheduling of their behaviours, in which atomic actions for different objects can interleave and use bidirectional communications. Still, compiled code is correct by construction. So far for request (b).

However, request (a) calls for considering asynchronous communications too, as this is the type of service provided by distributed infrastructures. The following can be stated about the kind of asynchrony we consider in this work :

1. Reactions cannot be observed any more: since no global clock exists, global synchronization barriers which indicate the transition from one reaction to the next one are no more observable. Instead,

(A) a reliable communication medium composed of point-to-point channels is assumed, in which messages are not lost, and, for each individual channel, messages are sent and received in the same order.

We call a flow the totally ordered sequence of values sent or received on a given communication channel.

2. Absence cannot be detected, and thus cannot be used to exercise control.

3. Composition occurs by means of unifying each individual flow shared between two capsules.

The theory of desynchronisation developed in [5, 3, 12] provides us with the link between the two different, synchronous and asynchronous, models. In these references, we show the following :

1. One can characterize those synchronous designs $\|_i P_i$ in which exchanging the synchronous communication infrastructure for an asynchronous one satisfying hypothesis (A) 1/ preserves the set of behaviours for each individual synchronous component P_i, and 2/ preserves the global asynchronous semantics of the whole system $\|_i P_i$. By this we mean that the history seen by each individual communication channel remains unchanged. Of course, as no global state nor time for the overall system is available, no guarantee is provided about global behaviours and states.

2. Additional signalling and schedulers just tailored to the considered application can be automatically generated, in order to make the design compliant with the above stated requirements for correct desynchronised deployment.

3. This technology can support dynamical instanciation in the following way. Assume the above mentioned additional signalling and schedulers have been generated on the basis of the generic class diagram. Then any object diagram (possibly involving dynamic instanciation) will still satisfy the conditions of point 1.

Besides dynamic instanciation, this whole approach for architecture and code generation is supported by the SIGNAL tool, and our objective is to transport it to the BDL formalism. In the sequel we illustrate this approach with our example.

5.2 The POTS example and its deployment

Here we consider the case in which the whole system of figure 10 is deployed onto a single capsule, and then several capsules, one per each object. This is illustrated in Figure 20, where we show these two cases. For the deployment on a single capsule, we interpret it by using the *synchronous* parallel composition operator $\|$, this is depicted in the first diagram. For the distributed deployment, we reinterpret the communication as being represented by the *asynchronous* parallel composition operator $\|_a$.

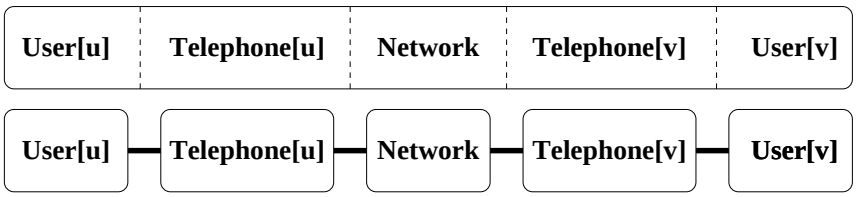

Figure 20: *The plain old telephone service:* overall BDL diagram, with one network, two telephones, and two users. Two diagrams are depicted. The first one shows **(User[u]** $\|$ **Telephone[u]** $\|$ **Network** $\|$ **Telephone[v]** $\|$ **User[v])**, where **Network** denotes the BDL diagram of figure 12, and **Telephone[u]** denotes the BDL diagram of the figure 11 in which Id=u. As **User[u]** accepts any behaviour, its BDL diagram is trivial and is not shown. The second diagram specifies deployment, it indicates that asynchronous communication is available between **Telephone[u,v]** and **Network**.

Preserving the semantics while performing asynchronous deployment amounts to checking whether $\| \equiv \|_a$ for the considered parallel compositions. We show that this is not true.

CASE 1: $\|$ **is used.** Since $\|$ synchronizes objects on their shared ports, messages

$$\begin{aligned} \text{network.progress(u,v)} \quad & \text{in} \quad \textbf{Telephone[u]}, \\ \text{progress(u,v)} \quad & \text{in} \quad \textbf{Network}, \\ \text{progress(u)} \quad & \text{in} \quad \textbf{Telephone[v]}, \end{aligned}$$

must belong to the same reaction. Analyzing this and related situations shows that using the synchronous $\|$ composition guarantees the correctness of the proposed protocol.

CASE 2: $\|_a$ **is used.** In this case, we only know that objects willing to participate to the current reaction by involving some common port must agree on their carried message, but composition does not enforce synchronization via shared ports. Hence it is possible that object **Telephone[u]** emits message "network.progress(u,v)", whereas object **Network** does not see message "progress(u,v)" in the same reaction.

This counterexample shows that $\| \not\equiv \|_a$ in this case. This formalizes the intuition that messages "progress" and "hangup" sent in this order by **Telephone[u]** may reach their destination in a reversed order, therefore confusing **Network** and also **Telephone[v]**.

Now, the nice thing about our approach is that we have an algebraic, formal, characterization of what condition should be enforced in order to preserve the semantics while performing asynchronous deployment. The condition for enforcing is the following:

$$\begin{array}{lcl} & \textbf{Telephone[u]} \quad \| \quad \textbf{Network} \\ \equiv & & \\ & \textbf{Telephone[u]} \quad \|_a \quad \textbf{Network} \end{array} \tag{11}$$

Indeed, it is shown in [3] that it is enough to have isochrony locally, for each pairwise interaction, in order to guarantee correct desynchronisation. For reasons of symmetry, enforcing (11) can be immediately mirrored for the alternative pair (**Network,Telephone[v]**). On the other hand, as **User[u]** accepts any behaviour, its interaction with any other component is isochronous. Hence enforcing (11) is sufficient.

The interaction between the two considered components is detailed in figure 21. Analysing

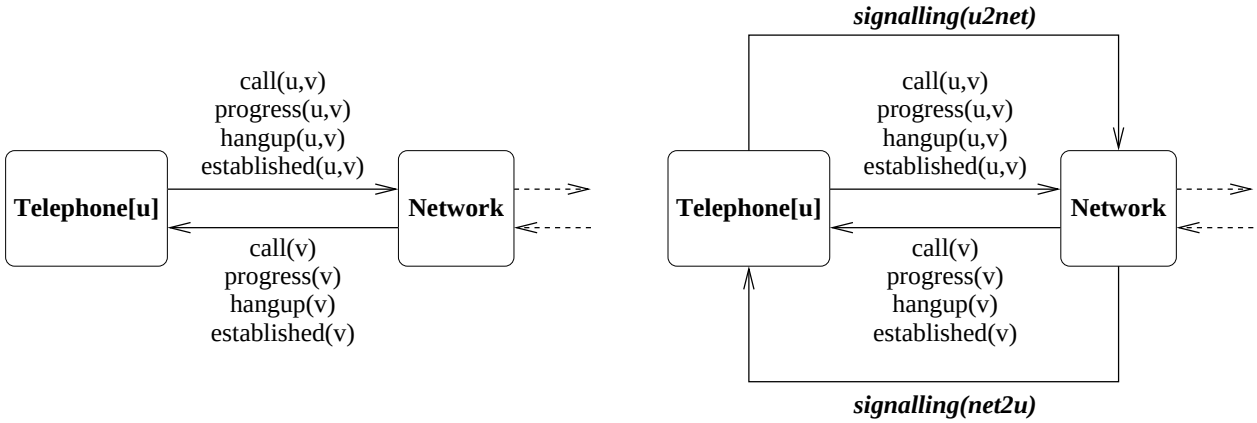

Figure 21: *Enforcing isochrony*. The first diagram depicts the communication flow between these two components. The second diagram shows the additional signalling to enforce isochrony.

the BDL diagram of figure 16, we see that the four components involved in **Network** are not synchronised. Hence, looking at figure 21, we see that we need to add proper signalling, for each directed communication. Therefore, for each direction, we add a signal of enumerated type. This signal is emitted at every reaction in which communication occurs in the considered direction. This signal indicates which port is involved in the considered message. Roughly speaking, we emulate the multiplexing of the four channels into a single one, for each direction.

This technique can be generalized and made systematic, using so-called clock hierarchies, a special data structure analysed in details in [12] and already used in the SIGNAL compiler. Note that it is different from the usual solution, namely protecting communications by appropriate protocols that are independent from the deployed application. Our solution consists in adding signalling and associated schedulers *tailored* to the deployed application, this will in general cause much less overhead.

6 Discussion and perspectives

We have proposed BDL as a semantic backbone for dynamic diagrams of UML. Statecharts, sequence diagrams, and other dynamic diagrams can be translated into BDL. We have developed draft prototypes for the translation of UML-statecharts and high level message sequence charts (HMSC) into BDL. In addition to classical control structures, BDL offers means to synchronize diagrams, and to schedule actions and diagrams (using the mechanism of causality relations), it can be used as a coordination language.

We have illustrated the use of BDL with an example of service adaptation. We believe service adaptation is an important challenge for software development in the near future. Deploying new services will proceed by reusing an existing service basis and patching the new features. In doing so, it is important to keep clear both what the original service was, and what the added feature is, and at the same time to make sure that adding the new feature does not create spurious behaviours. We have proposed a new methodology in which the above conditions are built-in : the user specifies the original service, the new feature, and the process of integrating them is then systematic and correct by construction. One may regard this case study as an example of handling inheritance in its dynamical aspects, something considered difficult.

It is advisable to make the design of the conceptual architecture mostly independent from the design of the deployment architecture, this allows the designer to upgrade these two architectures independently. We have made a proposal to make this possible by resourcing to BDL. We have equipped BDL diagrams with a dual synchronous/asynchronous communication. Synchronous communication enjoys excellent properties, making tractable the formal exploration of designs based on this semantics. However it requires the broadcast of a global step to synchronize the transitions of all components, and therefore it is suited to implement communication between objects sitting on the same capsule. Asynchronous communication better matches actual communication infrastructures, or even multithreading systems. It should be preferred for the deployment of different diagrams on different capsules. We think it is an important contribution that we offer a theory and associated algorithms to allow a correct migration between these two types of communication.

All this was made possible thanks to a rigorous mathematical semantics supporting BDL. This semantics basis has a strong relation to so-called synchronous languages [4][9], and benefits from recent progresses in this area [5][3]. An important aspect is that these recent progresses make it possible to exploit this theory in the context of dynamic creation of instances [12].

Draft prototypes of translators from statecharts and message sequence charts into BDL have been developed. A translator from BDL into the SIGNAL synchronous language [4] is under development, it will provide a first route to automatic code generation from BDL. These developments are performed in the framework of the Umlaut tool for UML [13].

A Appendix : formal presentation of BDL

A.1 Syntax

The syntax of the core of BDL is given below and its complete abstract syntax is defined in figures 22 and 23 as an extension of the UML meta-model.

A.1.1 Name-spaces

Five name-spaces have been defined in BDL: ports $x \in X$, attributes $k \in K$, variables $v \in V$, types $t \in T$, constants $c \in C$ and operations $f \in F$. Pins are labelled with the subscript $()_-$ to refer to a pre-pin, or $()_+$ to refer to a post-pin. An expression can be a variable, a constant or an operation: $e ::= v|c|f(e_1, \ldots, e_n)$. Operations are typed and valid expressions must be type consistent. Expressions are provided by an assumed host language. Message $x(e)$ is the association of port $x \in X$ of type $t \in T$ with an expression e of the same type t. Pins have the form $\{k_{-/+} \models p\}$ where $k \in K$ is an attribute and $p \in P_k$ is a predicate. Sign $-$ means that the pin is a precondition while sign $+$ is used for postconditions.

A.1.2 Pre-orders and directed graphs

BDL terms are compositions of pre-ordered sets of messages (occurrences of ports associated with an expression). In BDL, pre-orders are represented via (possibly cyclic) directed graphs $\gamma = (S, \rightarrow)$, where S is a finite set S, and $\rightarrow$ is a reflexive binary relation.

A.1.3 Composition Operators

They are listed below :

$$
\begin{array}{rcll}
\Gamma & ::= & \gamma & \text{a directed graph} \\
& | & \Gamma \vee \Gamma & \text{choice} \\
& | & \Gamma \parallel \Gamma & \text{parallel composition} \\
& | & \Gamma \wedge \Gamma & \text{synchronization} \\
& | & \Gamma \rightarrow \Gamma & \text{serialization} \\
& | & \vee_{v:t}\ \Gamma & \text{enumerated choice} \\
& | & \|_{x:t}\ \Gamma & \text{enumerated parallel composition} \\
& | & \wedge_{x:t}\ \Gamma & \text{enumerated synchronization}
\end{array}
\tag{12}
$$

A.2 Semantics

We first give the semantics of a *single reaction* of a BDL term, in the form of a family of directed graphs. Then we give the *synchronous* trace semantics of a BDL term. And finally we give the ***a**synchronous* trace semantics of a BDL term.

A.2.1 BDL well-formed directed graphs

Referring to (12), we first describe what a γ is. Symbol γ denotes a labelled, possibly cyclic, directed graph, with the following types of vertices:

messages: two different messages of γ shall have different ports.

pre-/post-conditions: the two sets of pre- and post-conditions shall be non empty.

Trivial pre-/post-conditions of the form $\{k_\pm \models \mathbf{true}\}$ are added for mathematical convenience for the case where no such condition was specified. Also, we write for short $\{k_\pm\}$ instead of $\{k_\pm \models \mathbf{true}\}$.

A.2.2 BDL terms as families of directed graphs

We now give the semantics of a *single reaction* of a BDL term, in the form of a family of directed graphs. Let X and X' denote the sets of declared ports of Γ and Γ', respectively. Also, τ denotes the empty graph, with no vertex. Then

$$\Gamma \vee \Gamma' \;=\; \text{disjunction of } \Gamma \text{ and } \Gamma' \tag{13}$$

$$\Gamma \parallel \Gamma' \;=\; \bigvee_{\substack{(\gamma,\gamma') \in \Gamma \times \Gamma' \\ \gamma \bowtie \gamma'}} \gamma \sqcup \gamma' \tag{14}$$

$$\Gamma \wedge \Gamma' \;=\; [\,(\Gamma \setminus \tau) \;\parallel\; (\Gamma' \setminus \tau)\,] \;\vee\; \tau \tag{15}$$

$$\Gamma \to \Gamma' \;=\; \parallel_{\substack{(x,x') \in X \times X' \\ x \neq x'}} \;\to_{x,x'}\;, \tag{16}$$

where

$$\to_{x,x'} \;\triangleq\; \bigvee_{\substack{v \in \mathbf{dom}(x) \\ v' \in \mathbf{dom}(x')}} (\,\tau \vee x(v) \vee x'(v') \vee x(v) \to x'(v')\,)$$

The following explanations are in order:

(13) self explanatory.

(14) Expression $\gamma \bowtie \gamma'$ means that graphs γ and γ' are *unifiable,* i.e., for every shared declared port x, $x(e) \in \gamma$ iff $x(e') \in \gamma'$ (shared ports enforce synchronization).

For a unifiable pair $\gamma \bowtie \gamma'$, then $\gamma \sqcup \gamma'$ denotes the unification of γ and γ', defined as follows:

1. if x is present in γ (and therefore also in γ', by definition of $\bowtie$),replace $x(e') \in \gamma'$ by $x(e)$;
2. take the union of the graphs γ and γ' ;
3. declare a new **assertion** port α_x, with domain consisting of the singleton T (i.e., α_x is either true or absent), and add to the union of the graphs γ and γ' the message $\alpha_x(e = e')$;
4. for k an attribute of γ or γ', its associated pre-/post-condition is $\{k_\pm \models p \wedge p'\}$, where $\{k_\pm \models p\}$ and $\{k_\pm \models p'\}$ are the pre-/post-condition associated with k in γ and γ', respectively. We take the convention that $p =$ **true** if k is not an attribute of γ. Pre-/post-conditions play no role in synchronization, parallel composition combines them via conjunction.

A particular case of interest is $e' = v'$, a free variable in Γ'. Then unification is performed by performing, in γ', the substitution e/v'. Hence x is regarded as an "input port" of γ' and an "output port" of γ, and we have specified an output-to-input connection. In this case, the additional assertion $\alpha_x(e = e')$ becomes a tautology and can be omitted.

(15) $\Gamma \setminus \tau$ removes from Γ the silent reaction modelled by the empty graph τ. Therefore it represents its set of active reactions. Hence $[\,(\Gamma \setminus \tau) \parallel (\Gamma' \setminus \tau)\,]$ is the set of reactions of $\Gamma \parallel \Gamma'$ in which both components (γ, γ') are non silent. Then the silent transition (we always want it to be possible) is added to $[\,(\Gamma \setminus \tau) \parallel (\Gamma' \setminus \tau)\,]$. Therefore, the reactions of $\Gamma \wedge \Gamma'$ are those reactions of $\Gamma \parallel \Gamma'$ in which the components are either both active, or alternatively both silent.

(16) this term has $X \cup X'$ as a set of variables, where X and X' are sets of variables for Γ and Γ', respectively. For each pair of different ports (x, x'), the BDL term $\rightarrow_{x,x'}$ *forces the order* $x \rightarrow x'$ *when both* x *and* x' *are present,* but it also allows x or x' to be present alone. The quantified choice is over the pairs of values (v, v') ranging over the domain $\mathbf{dom}(x) \times \mathbf{dom}(x')$ of domains of x and x'. Then, term $\Gamma \rightarrow \Gamma'$ is the parallel composition of the $\rightarrow_{x,x'}$ statements. While referring to the pair (Γ, Γ'), it only depends on the respective sets of variables X and X'. To summarize, this statement does not raise any constraint on the values carried by the involved messages, it only constraints the ordering of messages.

This statement is particularly useful in the following contexts : $\Gamma \parallel \Gamma' \parallel (\Gamma \rightarrow \Gamma')$, or $(\Gamma \wedge \Gamma') \parallel (\Gamma \rightarrow \Gamma')$. In these case it forces, in the parallel composition of Γ and Γ', every action or message within Γ to precede every action or message within Γ'.

Using these formal definitions, the precise meaning of the diagrams of figure 6 is given next :

$$\text{parallel composition} \;:\; A \parallel B \tag{17}$$

$$\text{synchronization} \;:\; A \wedge B \tag{18}$$

$$\text{non deterministic choice} \;:\; A \vee B \vee C \tag{19}$$

$$\text{block serialization} \;:\; \begin{pmatrix} \begin{pmatrix} & (A \wedge B) \\ \| & (A \to B) \end{pmatrix} \\ \| \quad C \\ \| \quad (A \to C) \end{pmatrix} \tag{20}$$

Note that, for the graphical block-serialization, we could have chosen, instead of (20), the following alternative definition :

$$\text{block serialization} \;:\; \begin{pmatrix} \begin{pmatrix} & (A \wedge B) \\ \| & (A \to B) \end{pmatrix} \\ \| \quad C \\ \| \quad ((A \wedge B) \to C) \end{pmatrix} \tag{21}$$

Definition (21) for the serialization is structural, whereas (20) is not. On the other hand, (20) allows more flexible specifications, therefeore we preferred it.

A.2.3 The *synchronous* trace semantics of a BDL term

Let Γ be a BDL term, and X, K its declared set of ports and attributes. For $\gamma_1, \gamma_2 \in \Gamma$ we define the synchronous concatenation $\gamma_1 \bullet \gamma_2$ of γ_1 and γ_2 as follows :

1. It is defined only if, for each $k \in K$, $\{k_+ \models p_1\} \Rightarrow \{k_- \models p_2\}$ holds.
2. For each $x \in X$:
   - For $i = 1, 2$, in γ_i rename the declared port x by x_i
   - in γ_1 rename the post-pin $\{k_+ \models p_1\}$ by $\{k_1 \models p_1\}$.
   - in γ_2 rename the pre-pin $\{k_- \models p_2\}$ by $\{k_1 \models p_1\}$.
3. take the union $\gamma_1 \cup \gamma_2$.

Note that intermediate pins are kept. They indexed by a fresh index, we take it equal to 1 in the present case. Here is an example :

$$\begin{pmatrix} \{y_-\} & \to & y(e) & \to & \{y_+\} \\ & & \downarrow & & \\ \{x_-\} & \to & x(e) & \to & \{x_+ = e\} \end{pmatrix} \bullet \begin{pmatrix} \{x_- = e'\} & \to & x(e') & \to & \{x_+\} \\ & & \downarrow & & \\ \{z_-\} & \to & z(e') & \to & \{z_+\} \end{pmatrix}$$

$$= \tag{22}$$

$$\begin{pmatrix} \{y_-\} & \to & y_1(e) & \to & \{y_1\} & \to & \to & \to & \{y_+\} \\ & & \downarrow & & & & & & \\ \{x_-\} & \to & x_1(e) & \to & \{x_1 = e\} & \to & x_2(e) & \to & \{x_+\} \\ & & & & & & \downarrow & & \\ \{z_-\} & \to & \to & \to & \{z_1\} & \to & z_2(e) & \to & \{z_+\} \end{pmatrix}$$

The pins indicate the end of a reaction, this characterizes the paradigm of synchrony. Then we define

$$[\![\Gamma]\!] = \Gamma^{\bullet\omega} , \quad \text{where} \quad \Gamma \bullet \Gamma = \bigvee_{(\gamma_1,\gamma_2)\in\Gamma\times\Gamma} \gamma_1 \bullet \gamma_2 \tag{23}$$

A.2.4 The *asynchronous* trace semantics of a BDL term

Let Γ be a BDL term, and X its declared set of ports. For $\gamma_1, \gamma_2 \in \Gamma$ we define the asynchronous concatenation $\gamma_1 \circ \gamma_2$ of γ_1 and γ_2 as follows:

1. It is defined only if, for each $k \in K$, $\{k_+ \models p_1\} \Rightarrow \{k_- \models p_2\}$ holds.
2. For each $x \in X$:
   - For $i = 1, 2$, in γ_i rename the declared port x by x_i
   - in γ_1 rename the post-pin $\{k_+ \models p_1\}$ by $\{k_1 \models p_1\}$.
   - in γ_2 rename the pre-pin $\{k_- \models p_2\}$ by $\{k_1 \models p_1\}$.
3. Take the union $\gamma_1 \cup \gamma_2$.
4. Erase the intermediate pins $\{k_1 \models p_1\}$ and take the transitive closure in $\gamma_1 \cup \gamma_2$.

Note that intermediate pins are removed. Here is an example, compare with (22):

$$\begin{array}{l} \left(\begin{array}{ccccc} \{y_-\} & \to & y(e) & \to & \{y_+\} \\ & & \downarrow & & \\ \{x_-\} & \to & x(e) & \to & \{x_+ = e\} \end{array} \right) \circ \left(\begin{array}{ccccc} \{x_- = e'\} & \to & x(e') & \to & \{x_+\} \\ & & \downarrow & & \\ \{z_-\} & \to & z(e') & \to & \{z_+\} \end{array} \right) \\ = \\ \left(\begin{array}{ccccccc} \{y_-\} & \to & y_1(e) & \to & \to & \to & \{y_+\} \\ & & \downarrow & & & & \\ \{x_-\} & \to & x_1(e) & \to & x_2(e) & \to & \{x_+\} \\ & & & & \downarrow & & \\ \{z_-\} & \to & \to & \to & z_2(e) & \to & \{z_+\} \end{array} \right) \end{array} \tag{24}$$

Note that the intermediate pins, which indicate the end of a reaction in (22), are lost here. Then we define, compare with (23):

$$[\![\Gamma]\!]_a = \Gamma^{\circ\omega} , \quad \text{where} \quad \Gamma \circ \Gamma = \bigvee_{(\gamma_1,\gamma_2)\in\Gamma\times\Gamma} \gamma_1 \circ \gamma_2 \tag{25}$$

A.2.5 Synchrony vs. *a*synchrony

In this subsection we give a brief description of how the link between synchrony and asynchrony can be formalized, see [3, 5] for a detailed exposition.

We define the following desynchronized version of our parallel composition operator $\|$ between BDL terms, we denote it $\|_a$.

$$\Gamma \|_a \Gamma' = \bigvee_{\substack{(\gamma,\gamma') \in \Gamma \times \Gamma' \\ \gamma \bowtie_a \gamma'}} \gamma \sqcup_a \gamma' \tag{26}$$

Expression $\gamma \bowtie_a \gamma'$ means that graphs γ and γ' are *weakly unifiable,* i.e., for every shared declared port x, $x(e) \in \gamma$ and $x(e') \in \gamma'$ implies $e = e'$. This means that if shared port x is present in both graphs, then it must hold the same value. Note that this allows, for x a shared port, that x is present in γ but absent from γ', hence, in contrast to the $\|$ synchronous composition operator, asynchronous composition $\|_a$ does not enforce synchronization via shared ports.

For a unifiable pair $\gamma \bowtie_a \gamma'$, then $\gamma \sqcup_a \gamma'$ denotes the weak unification of γ and γ', defined as follows:

1. if x is present in both γ and γ', replace $x(e') \in \gamma'$ by $x(e)$;
2. take the union of the graphs γ and γ';
3. declare a new **assertion** port α_x, with domain consisting of the singleton T (i.e., α_x is either true or absent), and add to the union of the graphs γ and γ' the message $\alpha_x(e = e')$;
4. for k an attribute of γ or γ', its associated pre-/post-condition is $\{k_\pm \models p \wedge p'\}$, where $\{k_\pm \models p\}$ and $\{k_\pm \models p'\}$ are the pre-/post-condition associated with k in γ and γ', respectively. We take the convention that $p =$ **true** if k is not an attribute of γ.

A particular case of interest is $e' = v'$, a free variable in Γ'. Then unification is performed by performing, in γ', the substitution e/v'. Hence x is regarded as an "input port" of γ' and an "output port" of γ, and we have specified an output-to-input connection. In this case, the additional assertion $\alpha_x(e = e')$ becomes a tautology and can be omitted.

Using this $\|_a$ composition operator, we can compose asynchronous trace semantics of terms Γ and Γ' by considering:

$$[\![\Gamma \|_a \Gamma']\!]_a . \tag{27}$$

The following result is shown in [3, 5]. Assume the following condition is satisfied by the considered pair (Γ, Γ'):

$$\Gamma \| \Gamma' = \Gamma \|_a \Gamma' . \tag{28}$$

Then (27) coincides with the per flow unification of the asynchronous trace semantics of the terms Γ and Γ' (this corresponds to a model of communication in which port to port channels are used, and each individual channel is loss less and maintains the ordering of messages). Condition (28) is referred to as *isochrony* and was introduced in [3]. Informally speaking, if the considered pair is isochronous, then it is equivalent

- either to 1/ consider the asynchronous trace semantics of Γ and Γ', and then 2/ compose them, asynchronously,
- or to 1/ compose synchronously the terms Γ and Γ', and then 2/ take the asynchronous trace semantics of their composition $\Gamma \parallel \Gamma'$.

We do not provide a visual notation for the asynchronous parallel composition $\|_a$. Our idea is that $\|$ is the default interpretation of our visual parallel composition notation, in the form of "and-states", and we resource to the deployment diagram to specify if asynchronous communication between distributed components is to be used.

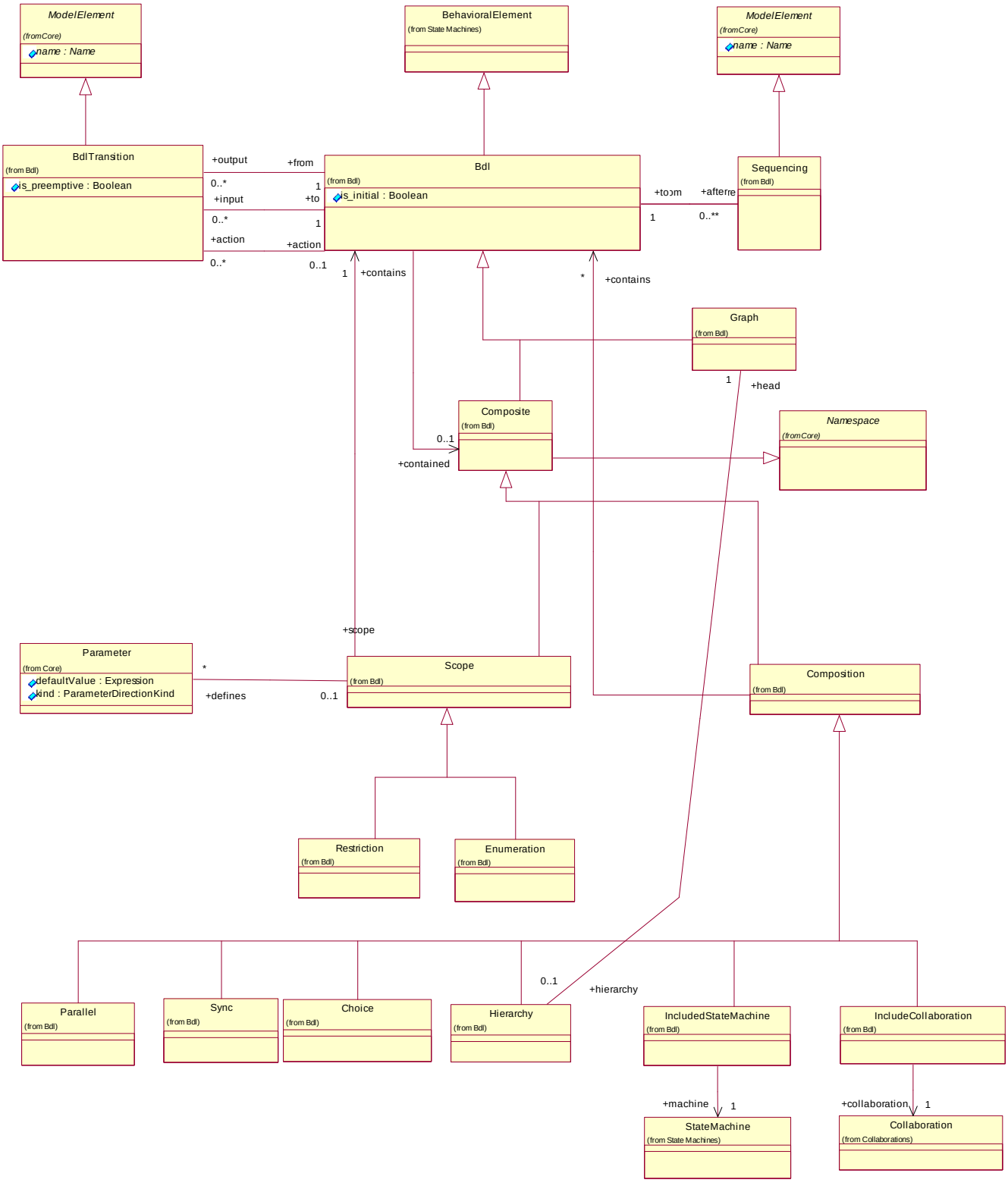

Figure 22: *The* BDL *meta-model,* part 1

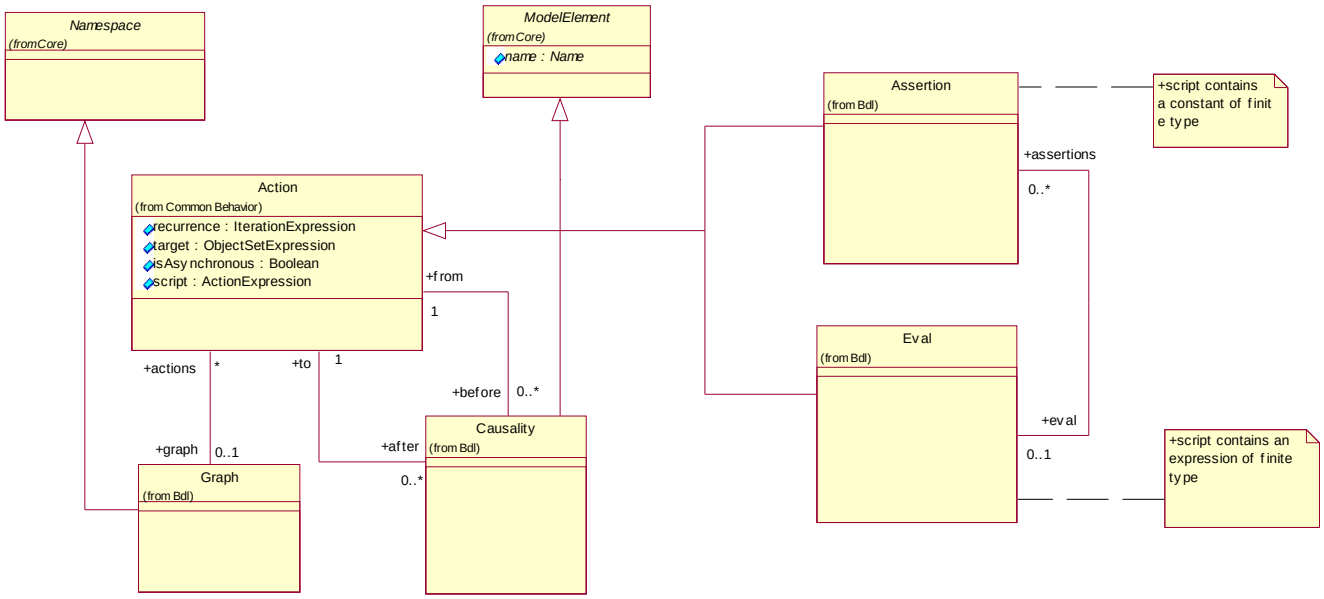

Figure 23: *The* BDL *meta-model*, part 2

References

[1] C. André. Representation and analysis of reactive behaviors: A synchronous approach. In *CESA'96, IEEE-SMC*, Lille, France, July 1996. http://www-sop.inria.fr/meije/esterel/syncCharts/TR96-28.pdf.

[2] João Araújo. Formalizing sequence diagrams. In Luis Andrade, Ana Moreira, Akash Deshpande, and Stuart Kent, editors, *Proceedings of the OOPSLA'98 Workshop on Formalizing UML. Why? How?*, 1998.

[3] A. Benveniste, Caillaud B., and P. Le Guernic. Compositionality in dataflow synchronous languages: specification & distributed code generation. *Information and Computation*, to appear. http://www.irisa.fr/sigma2/benveniste/pub/BCLg99a.html.

[4] A. Benveniste and G. Berry, editors. *Proceedings of the IEEE*, volume 79, chapter The special section on another look at real-time programming, pages 1268–1336. IEEE, September 1991.

[5] A. Benveniste, B. Caillaud, and P. Le Guernic. From synchrony to asynchrony. In J.C.M. Baeten and S. Mauw, editors, *CONCUR'99, Concurrency Theory, 10th International Conference*, volume 1664 of *Lectures Notes in Computer Science*, pages 162–177. Springer Verlag, 1999. http://www.irisa.fr/sigma2/benveniste/pub/BCLg99b.html.

[6] W. Damm and D. Harel. Lscs: breathing life into message sequence charts. In *3rd IFIP Int. Conf. on Formal Methods for Open Object-based Distributed Systems (FMOODS'99)*, pages 293–312. Kluwer, 1999.

[7] Andy Evans, Robert France, Kevin Lano, and Bernhard Rumpe. Developing the UML as a formal modelling notation. In Jean Bézivin and Pierre-Alain Muller, editors, *The Unified Modeling Language, UML'98 - Beyond the Notation. First International Workshop, Mulhouse, France, June 1998*, volume 1618 of *LNCS*, pages 297–307. Springer, 1998.

[8] Andy Evans and Stuart Kent. Core meta-modelling semantics of UML: The pUML approach. In Robert France and Bernhard Rumpe, editors, *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, volume 1723 of *LNCS*. Springer, 1999.

[9] N. Halbwachs. *Synchronous programming of reactive systems*. Kluwer Academic Pub., 1993.

[10] D. Harel and A. Naamad. The STATEMATE semantics of STATECHARTS. Technical report, I-Logix, 1995.

[11] Talpin J.-P., Benveniste A., Caillaud B., Jard C., Bouziane Z., and Canon H. Bdl, a language of distributed reactive objects. In *International Symposium on Object-Oriented Real-Time Distributed Computing*. IEEE Press, 1998.

[12] Talpin J.-P., Benveniste A., Caillaud B., and Le Guernic P. Hierarchic normal forms for desynchronization. Research report 3822, INRIA, 1999. http://www.irisa.fr/sigma2/benveniste/pub/TBCLg99.html.

[13] Jean-Marc Jézéquel, Alain Le Guennec, and François Pennaneac'h. Validating distributed software modeled with UML. In Jean Bézivin and Pierre-Alain Muller, editors, *The Unified Modeling Language, UML'98 - Beyond the Notation. First International Workshop, Mulhouse, France, June 1998*, volume 1618 of *LNCS*, pages 331–340. Springer, 1998.

[14] Laila Kabous and Wolfgang Nebel. Modeling hard real time systems with UML. In Robert France and Bernhard Rumpe, editors, *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, volume 1723 of *LNCS*. Springer, 1999.

[15] Alexander Knapp. A formal semantics for uml interactions. In Robert France and Bernhard Rumpe, editors, *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, volume 1723 of *LNCS*. Springer, 1999.

[16] S. Matsuoka and A. Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object Oriented Programming*. MIT Press, 1993.

[17] Stephen J. Mellor, Steve Tockey, Rodolphe Arthaud, and Philippe LeBlanc. Software-platform-independent, precise action specifications for UML. In Jean Bézivin and Pierre-Alain Muller, editors, *The Unified Modeling Language, UML'98 - Beyond the Notation. First International Workshop, Mulhouse, France, June 1998*, volume 1618 of *LNCS*, pages 281–286. Springer, 1998.

[18] Jeff Offutt and Aynur Abdurazik. Generating tests from UML specifications. In Robert France and Bernhard Rumpe, editors, *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, volume 1723 of *LNCS*. Springer, 1999.

[19] Ivan Paltor and Johan Lilius. Formalising UML state machines for model checking. In Robert France and Bernhard Rumpe, editors, *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, volume 1723 of *LNCS*. Springer, 1999.

[20] J. Rumbaugh, I. Jacobson, and G. Booch. *The unified modeling language reference manual*. object technology series. Addison-Wesley, 1999.

[21] Y. Wang, J-P. Talpin, A. Benveniste, and P. Le Guernic. Compilation and sitribution of state-machines using SPOTS. In *IFIP World congress, ICS'2000*, Beijing, China, August 2000.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399