Distributed Systems Specification and Runtime Manipulation of their Features using Co-nets: Application to a System with Several Lifts

Nasreddine Aoumeur[1] Gunter Saake

ITI, FIN, Otto-von-Guericke-Universität Magdeburg
Postfach 4120, 39016 Magdeburg, Germany
E-mail: {aoumeur|saake}@iti.cs.uni-magdeburg.de

June 2001

[1]This work is supported by a DAAD scholarship.

Abstract

Most of present-day software systems are data-intensive and fully distributed. Moreover, such systems are often layered into several towers depending on the features they may offer to their users. This latter characterization induce challenging problems to designers of real-world distributed systems, including how should given features be dynamically introduced, modified, interact with existing ones? and so on.

In this paper we propose a true-concurrent object Petri-net-based framework for addressing such challenges. The model, referred to as CO-NETS, integrates object-orientation concepts with modularity aspects into an appropriate algebraic Petri nets variant, and it is semantically interpreted in rewrite logic. This integration suffices for adequately addressing the data-intensive and full distribution. For dynamically introducing and modifying features, we soundly extend this integration with some reflection capabilities. Moreover, we present how to interact different features using strategies on the rewriting logic reflection level. All the proposed ideas are illustrated using a system with several lifts.

Contents

Chapter 1

Introduction

Most of present-day software systems are characterized as reactive systems with large amount of data and complex and fully distributed behaviour. Moreover, besides these key aspects, system designers are also facing another crucial challengings, namely the dynamic evolution of such systems in terms of different features they may offer to their users. In fact in most cases, real-world complex systems, such as telecommunication applications, are structured into a hierarchy of layers, each one is mainly distinguished by the features that may offer. Among the conceptual challenges involving features are particularly: how to dynamically express them, to introduce them, to modify them and how to coherently integrating them with each others ?.

Although the discipline for coping with feature aspects is quite new—often referred to as feature interaction— there have been severals interesting proposals, in terms of appropriate formal languages [GRS96][Rya97][Hei98]. However, due to the complexity of the problem we are far from a widely accepted framework. Moreover, most of existing proposals have concentrated more on the features themselves, while ignoring or disadvantaging the structural and behavioural aspects of the system. With the aim to overcome such limitations, we propose in this paper an appropriate framework that aims to cover most of the mentioned aspects. The model, referred to as CO-NETS [AS99b], consists in a sound and complete integration of object oriented constructions with some modularity concepts into a well-suited variety of algebraic Petri nets. The model is semantically interpreted in rewriting logic. Moreover, recently we have enhanced this proposal with some reflection capabilities allowing runtime modification of existing behaviour [Aou00]. In some detail, the suitability of this framework for covering most of the above challenging problems, could be highlighted by the following:

- As we pointed out, the CO-NETS approach is based on integrating object-oriented structuring mechanisms with high level Petri nets. Thus, while the object-oriented concepts (i.e. classification, object composition, etc) are the best abstraction mechanisms for coping with any a huge amount of data and knowledge, the Petri nets are the leading for expressing distribution and concurrency. Moreover, for allowing incremental constructions of complex systems as interacting components—where each component is regarded as a hierarchy of classes— we enrich each class with explicit interface (including structural as well as dynamic aspects).

- Because any suitable specification framework should be endowed with deduction rules for rapid-prototyping purpose, the proposed integration is soundly interpreted in true concurrency way using rewriting logic [Mes92]. This allow us to execute our specification using concurrent rewriting techniques and, in particular, current implementation of the MAUDE language [CDE+99].

- For dynamically introducing, modifying or deleting new features, as an appropriate behaviour, we have extended this integration by some reflection capabilities. The main ideas here are, first, the distinction between a fixed 'forever' features and features which may be subject to evolutions. Second, for this latter category, we have introduced a meta-level composed of meta-places that contain a behaviour as tokens, and three associated transitions for dynamically adding, deleting or modifying such behaviour. Third, we relate the usual object-level with this meta-level using syntactically appropriate read-arcs and semantically suitable inference rules: two constructions that suffice for propagating a given behaviour from the meta-level to the object-level.

- Besides these CO-NETS capabilities for dealing with complex distributed systems and their features, we introduce in this paper another necessary extension for managing features interaction and composition. This sound extension consists in using the reflection capabilities of rewriting logic [CM96]. More precisely, given rewrite rules governing different transitions behaviour of a given CO-NETS, we can formulate expressions over these behaviour. Each expression reflects the way on which different transitions (in this expression) should be performed. In this way we free the CO-NETS from controlling strategies for performing different transition, but also we offer a flexible way for dynamically selecting or modifying any strategy.

The rest of this paper is organized as follows. The next section presents an overview of some basic CO-NETS concepts. Also, we introduce and specify a simplified version of the lift system using this CO-NETS framework. In the third section, we addresses the problem of how features may be dynamically manipulated. The fourth section focuses on strategies as a way for composing and interacting system features. We conclude this paper with some remarks and outlying our future work.

Chapter 2

CO-NETS : A short Overview

We recall in this section some basic aspects of the CO-NETS approach, and apply them to the running example. In this sense, first, we present how template signatures, describing structural and functional aspects of a given distributed system, are specified. Second, we comment on the construction of object Petri nets from such templates. Finally, we present the true concurrency semantics associated with the behaviour of such nets. More detail about different forms of inheritance and interaction among components may be found in [AS00a] [AS99b].

2.1 Template Signature

A template signature defines the structure of object states and the form of operations which have to be accepted by such states. Basically, we follow the general object signature proposed for MAUDE [Mes93]. That is, object states are regarded as algebraic terms — precisely as tuples— and messages as operations sent or received by objects. More precisely, we adopt the following:

- The object states are algebraic terms of the form

  $$\langle Id|atr_1 : val_1, ..., atr_k : val_k, at\_bs_1 : val'_1, ..., at\_bs_{k'} : val'_{k'}\rangle$$

  - Id is an observed object identity taking its values from an appropriate abstract data type defining the sort OId;
  - $atr_1, .., atr_k$ are the local (i.e. hidden from the outside) attribute identifiers having as current values respectively $val_1, .., val_k$.
  - The observed part of an object state is identified by $at\_bs_1, ..., at\_bs_{k'}$, while their associated current values are $val'_1, ..val'_{k'}$.
  - Also, we assume that all attribute identifiers (local or observed) range their values over a suitable sort denoted AId, while their associated values are ranged over the sort $Value$ with $OId < Value$ (i.e. OId is a subsort of $Value$) in order to allow object valued attributes.

- In contrast to the indivisible object state proposed in MAUDE which avoids any form of intra-object concurrency, we introduce a simple deduction rule, we called 'object-state splitting / merging' rule, that permits to split (resp. recombine) the object state as needed. Moreover, it provides a meaning to our notion of observed attributes by allowing separation between intra- and inter-component states. This deduction rule may be described as follows: $\langle Id|attrs_1, attrs_2\rangle = \langle Id|attrs_1\rangle \oplus \langle Id|attrs_2\rangle$; with $attr_i$ as an abbreviation of $atr_{i1} : val_{i1}, ..., atr_{ik} : val_{ik}$.

- In addition of conceiving messages as terms —that consist of message name, object identifiers to which the message is addressed, and, possibly, parameters— we make a clear distinction between internal, local messages and external as imported or exported messages. Local messages allow to evolve object states of a given component, while external ones allow communication between different components by exclusively using their observed attributes.

Noting that this distinction between local and observed features in a given template is mainly dictated by the aforementioned requirements, and it represents the kernel towards the notion of a component. Such distinction was already explicitly realized, for instance in the ALBERT language [DB95] [AS99a].

Remark 2.1.1 *For object state operations, we adopt a mix-fix notation, where as mentioned each object state is described as a tuple of the form $\langle Id|atr_1 : val_1, .., at\_bs_1 : val'_1, ...\rangle$. More precisely, by adopting the OBJ language for describing the data level as well as template signatures, an object state definition can be described as follows— where the operator $\_,\_$ is defined in a generic way, and the Id-attributes sort allows for capturing a part (i.e. external or local attributes) of an object state.*

```
obj object-state is
  sort AId .
  subsort OId < Value .
  subsort Attribute < Attributes .
  subsort Id_Attributes < object .
  subsort Local_attributes External_attributes < Id_Attributes .
  protecting Value OId AId .
  op _:_ :  AId Value -> Attribute .
  op _,_ :  Attribute Attributes -> Attributes [associ. commu. Id:nil] .
  op ⟨_|_⟩ :  OId Attributes -> Id_Attributes .
endo.
```

In addition to this notion of object state signature, a template signature is obtained by defining different messages (i.e. method invocations) which may be received or send by object(s). Also, for modularity purpose we distinguish between local messages and external messages as mentioned above. All in all the general form in specifying a given template respects the following:

```
obj Template-Signature is
  protecting object-state, s-atr$_1$,...,s-atr$_n$, s-arg$_{11,1}$,.., s-arg$_{l1,l1}$,
            ...,s-arg$_{i1,1}$,...,s-arg$_{i1,i1}$ ...
  subsort Mes$_{l1}$, Mes$_{l2}$,...,Mes$_{ll}$ < Local_Messages .
  subsort Mes$_{e1}$, Mes$_{e2}$,...,Mes$_{ee}$ < Exported_Messages .
  subsort Mes$_{i1}$, Mes$_{i2}$,...,Mes$_{ii}$ < Imported_Messages .
  subsort local-attributes obs-attributes < Id-attributes .
  (* local attributes *)
    op ⟨_|atr$_1$ : _,...,atr$_k$ : _⟩ :  OId s-atr$_1$ ... s-atr$_k$ -> Local-Attributes.
  (* observed attributes *)
```

op $\langle \_ \mid atrbs_1 :, ..., atrbs_{k'} : \_ \rangle$: `OId` $\texttt{s-atbs}_1$ ... $\texttt{s-atbs}_{k'}$ $\rightarrow$ `obs-Attributes.`
(* local messages *)
op $\texttt{ms}_{l1}$: `OId` ... $\texttt{s-arg}_{l1,1}$ ... $\texttt{s-arg}_{l1,l1}$ $\rightarrow$ $\texttt{Mes}_{l1}$ . ...
(* export messages *)
op $\texttt{ms}_{e1}$: `OId` ... `OId` ... $\texttt{s-arg}_{e1,1}$ ... $\texttt{s-arg}_{e1,e1}$ $\rightarrow$ $\texttt{Mes}_{e1}$ . ...
(* import messages *)
op $\texttt{ms}_{i1}$:`OId` ... `OId` ... $\texttt{s-arg}_{i1,1}$ ... $\texttt{s-arg}_{i1,i1}$ $\rightarrow$ $\texttt{Mes}_{ip}$ . ...
endo.

2.1.1 Application to the lift system

In the following, first, we present informally the running case study, afterwards we present its data level and template signature.

Informal description of the case study

As we pointed out the case study we use throughout this paper for illustrating the developed ideas consists in a distributed system with several lifts. Thinking in an object oriented way [Weg90] [RBP$^+$91], the simplified (and initial) version we treat in this paper may be informally described as follows:

- Each lift is structurally characterized by its identity (name or number), by the current floor it is on (shortly `Cur_F`), by its state expressing if it is idle, going up or going down (shortly `St`), by the current state of its door (`St`), and by its current weight (`Wg`).

- The operations or methods that may offer each lift are the following. The lift can open or close its door, go to any called floor with respect to a given direction (up or down) and stop at intermediate floors if there have been a corresponding call meanwhile. Moreover, each lift is able to cancel calls from the same floor. Also, to make a distinction between a request inside the lift (i.e. a goto) or a call from the outside, we use for both a method we denote by `Goto_F(Id-lift, dest-floor, call-or-goto)`; where the value of the parameter `call-or-goto` allows to distinguish between the two kinds of request. Finally, for technically keeping trace of a given call while serving its intermediate floors, we introduce another variant of goto, that is `Goto_F(Id-lift, dest-floor, direction)`, where `direction` stands for the direction (i.e. Up or Dw) and `dest-floor` is the selected floor (which should be the max (resp. min) when the direction is Up (resp. Dw) from current calls).

Data specification for the lift signature

The different data sorts we need for the template signature are the following. For describing different floor levels, we use the sort `Floors` whose elements are natural constants (0,1, 2, .., k). For capturing different states a given lift may be in, we use the `StateF` whose elements are also constants, namely (the self-explained ones) `idle` or `Up` or `Dw`. The doors state is captured by a data sort `Door`, with two values: `op` for open and `cl` for closed. Also, we denote by `Wmx`, the maximal weight tolerated by each lift, which has to be a positive real constant. For distinguishing between a call from outside the lift and goto from the inside, we use a data sort denoted `Call-or-Goto` with two values: `out` for a call from the outside and `in` for a goto from the inside of the lift. Besides these data sorts, for capturing correctly the behaviour of a given lift we present in the next subsection, we need a function we called `less(K1, K2)`, where `K1` denote final destination of the lift and `K2` the current floor of the lift. This function has to return the first minimal next floor number between $K1$ and $K2$ which is currently requested. The two equations corresponds to the two cases : the lift is going up (i.e. $K1 > K2$) or the lift is going down (i.e. $K1 < K2$). All in all this level is:

```
obj Lift-Data is
  protecting Real+ nat .
  sort  Door  StateF  Call-or-Goto.
  op 0, 1, 2, 3, 4, 5, ..  , k :  → Floors .
  op idle, Up, Dw :  → StateF .
  op op, cl :  → Door .
  op in, out :  → Call-or-Goto .
  opopbesidle, Up, Dw :  → StateF .
  op Wmx :  → Real+ .
  vars O : Call-or-Goto .
  vars K1, K2 :  Floors .
  eq less(K1, K2) == if K1 > K2 then K2 + 1 or
      K2+2 or ...  or K1 - 1 endif .
  eq less(K1, K2) == if K1 < K2 then K2 - 1 or
      K2 - 2 or ...  or K1 - 1 endif .
endo.
```

The lift template signature

Using this data level and the informal OO description, the template signature corresponding to the lift can be given as follows.

```
obj Lift is
  extending Object-State .
  protecting Lift-Data .
  sort Id.Lift < OId .
  sort GOTO  TEST  LIFT .
  (* the Lift object state declaration *)
   op ⟨_|Cur_F : _, St : _, Dr : _, Wg : _⟩:  Id.Lift
       Floors  StateF  DoorSt  Real+ → Lift .
(* Messages declaration *)
   op Goto_F : Id.Lift  Floors  Call-or-Goto → GOTO .
   op Goto_F : Id.Lift  Floors  StateF  → GOTO .
   op TestF : Id.Lift  Floors  Floors  nat → TEST .
  vars L : Id.Lift .
  vars S, D : StateF .
  vars W, W' : Real+ .
  vars K, K1, K2, K' : Floors .
endo.
```

We note that all the specified variables will be used in the corresponding nets later.

2.2 Template specification

Given a template signature denoted by TS that captures the structural aspects of a distributed system, its behaviour is constructed by associating a CO-NET with this signature—leading to the notion of template specification that we denote by $SP =\prec TS, Net \succ$. Informally speaking, the net associated with a given template signature is constructed as follows.

- The places of the net are precisely defined by associating with each message generator one 'message' place. Also, with each object state sort an 'object' place is associated. We denote the set of all places by P.
- Transitions, which may include conditions, reflect the effect of messages on object states (i.e. method body) to which they are addressed. Note that, for distinguishing between local messages and external ones we draw the later with bold lines.

2.2.1 The object net of the lift system

Following these very simplified rules for deducing an object net from a given template, in Figure 2.1 we depict the corresponding net modeling the dynamic of a system with several lifts. First with the object sort `Lift` we have associated a corresponding place which contains the current component-state of each lift. Also, with the message sort `GOTO` we have associated a corresponding place.

The behaviour of this variant of systems with several lifts is captured by different transitions. The transitions `Go-in-out`, `Close-Dr` and `Open` are self-explained, they respectively reflect: the change of the weight (by going in and out) when the door is open; the close of the door unless the maximal weight is exceeded, and the openness of the door. Note that in these transitions, we have selected just the necessary attributes; this is possible due to the splitting / recombining deduction rule. Also, in these transitions we did not care about the state, because if these parts of state (in these transitions) are selected then automatically the lift is a in particular floor and not in between; otherwise these parts would be under processing by other transitions.

For the move of the lift we have associated five transitions. The transition `Tidle` simply drops any call form the same floor. The transition `Tnxt` corresponding to a receive of a `Goto` (inside or outside) for a next or a precedent floor provided that the door is close and the state of the lift is idle. It is possible to associate a time value to a given transition [AS00b], representing here by `Tmf` and corresponding to a time-stamp for going to a next or a precedent floor. The transition `Tfar` has to capture the going to floor different form the precedent or the next ones. This case is more complex due to the fact that we have to 'serve' all current requests between the current floor and this final floor. The solution we propose consists, first, in detecting the direction for such a final request—using a comparison between the current floor and this called floor in the condition of transition `Tfar`. Second, depending on this comparison we save this goto message; where the information about the source of the call (i.e. `in` or `out`), which became irrelevant here, is replaced by the direction of the lift (i.e. `Up` or `Dw`). We have also associated a time, `Tmf`-β, to this transition, where β represent a short time (seconds) before arriving to a given floor in which the lift tests, through the next following transitions, whether there are intermediate requests or not. These transitions that are *systematically* performed after the transition `Tfar` are either `Tint` or `Tfinal`—this is because the state of this lift is now Up or Dw, whereas the transitions `Tnxt` and `Tfar` require an idle state.

Of course the transitions `Go-in-out`, `Close-Dr`, `Open-Dr` or `Tidle` may be performed when the lift is in an intermediate floor. The transition `Tint`(ermediate) is first performed as many time as there are requests between the current floor and the final floor (i.e. $K1$). This is ensured using the function `less(final-floor, current-floor)` which allow for traversing all requests in the place `GOTO`, and selecting each time the minimal floor in-between, if any there is one. In this case, the current floor will be this minimal floor as illustrated in the lower part of Figure 2.1. If there is no (or no-more) such an intermediate floor then the transition `Tfinal` is performed, and thereby the state of the lift is again set to idle and henceforth all transitions may be performed.

2.3 CO-NETS: Semantical Aspects

Given a OB-NET associated with a template specification, its semantics should provide us the *permissible* states which a marked CO-NET may be in. On the other hand, it should allow us for formally deducing from an initial state, in a true concurrency way, any any other permissible reachable state. By permissible, we mainly understand the respect of the uniqueness of object identities and the respects of the encapsulation property during state evolution.

2.3.1 Objects creation and deletion

Regarding a marked CO-NET as a society of objects and messages imply that each object must have its proper identity that persists through a change. In order to ensure the uniqueness of object identities and their dynamic creation / deletion, we propose the following conceptualization:

1. We associate with each a marked CO-NET modeling a component denoted as Cp a new place of sort $Id.obj$ ($< OId$). Such place contains then the *current object identifiers* of objects in Cp.
2. For the creation of new objects, we introduce a new message sort denoted Ad_{Cp} and an associated symbol operation (i.e. creation message) ad_{Cp} indexed by $Id.obj \times Ad_{Cp}$.
3. Each object state creation should be performed through the net depicted in the left hand side of Figure 2.2. The intended semantics (we present later) for the notation $\sim$ is that for firing the transition NEW the identifier Id should not already exist in the place $Id.obj$ (i.e. the notation $\sim$ captures the notion of inhibitor arc). After firing this transition, there is an addition of the new identifier Id to the place $Id.obj$ and creation of a new object namely $\langle Id \mid atr_1 : in_1, ..., atr_k : in_k \rangle$, where $in_1, ..., in_k$ are optional initial attribute values.

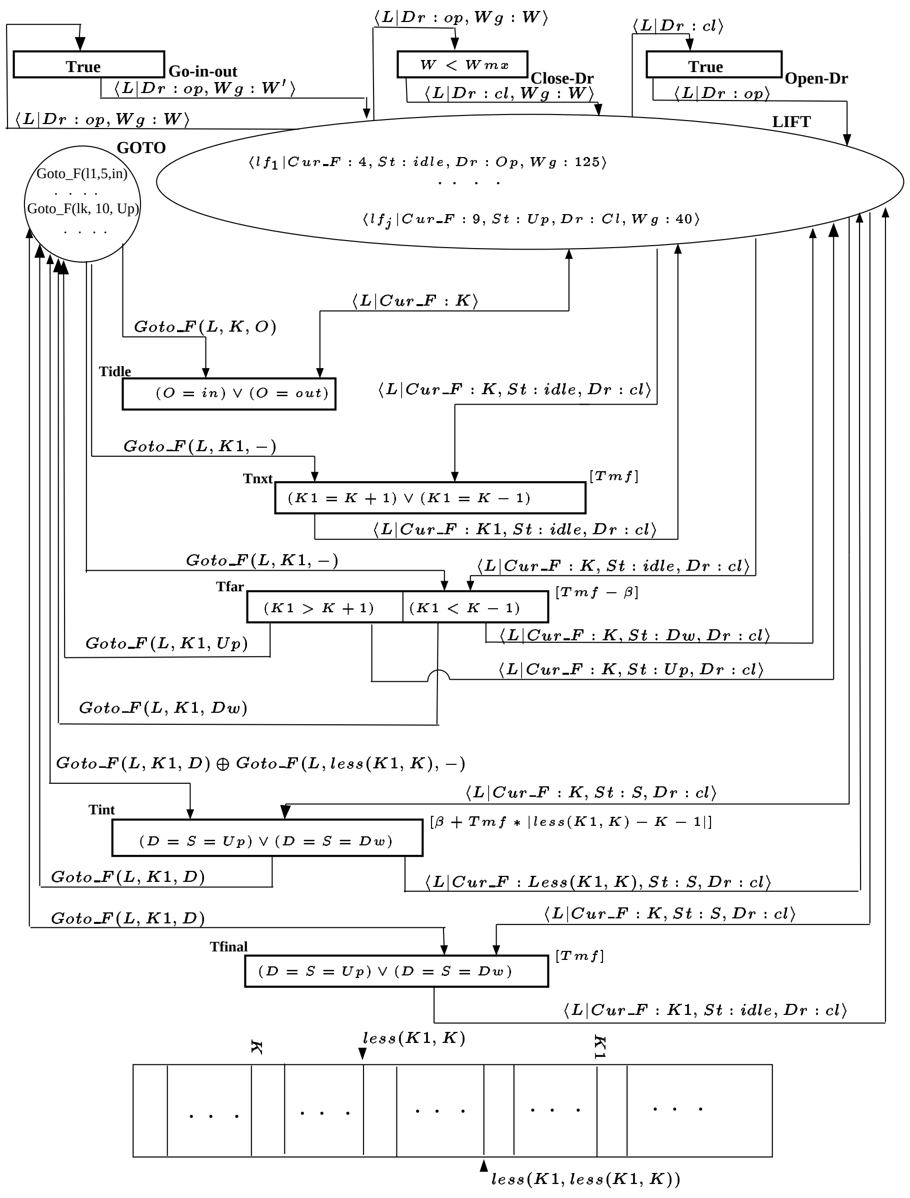
$\langle L|Dr : op, Wg : W\rangle$
$\langle L|Dr : cl\rangle$
True
Go-in-out
$W < Wmx$
Close-Dr
True
Open-Dr
$\langle L|Dr : op, Wg : W'\rangle$
$\langle L|Dr : cl, Wg : W\rangle$
$\langle L|Dr : op\rangle$
$\langle L|Dr : op, Wg : W\rangle$
LIFT
GOTO
Goto_F(l1,5,in)
. . . .
Goto_F(lk, 10, Up)
. . . .
$\langle lf_1|Cur\_F : 4, St : idle, Dr : Op, Wg : 125\rangle$
. . . .
$\langle lf_j|Cur\_F : 9, St : Up, Dr : Cl, Wg : 40\rangle$
$\langle L|Cur\_F : K\rangle$
$Goto\_F(L, K, O)$
Tidle
$(O = in) \vee (O = out)$
$\langle L|Cur\_F : K, St : idle, Dr : cl\rangle$
$Goto\_F(L, K1, -)$
Tnxt
$[Tmf]$
$(K1 = K + 1) \vee (K1 = K - 1)$
$\langle L|Cur\_F : K1, St : idle, Dr : cl\rangle$
$Goto\_F(L, K1, -)$
$\langle L|Cur\_F : K, St : idle, Dr : cl\rangle$
Tfar
$[Tmf - \beta]$
$(K1 > K + 1)$
$(K1 < K - 1)$
$Goto\_F(L, K1, Up)$
$\langle L|Cur\_F : K, St : Dw, Dr : cl\rangle$
$\langle L|Cur\_F : K, St : Up, Dr : cl\rangle$
$Goto\_F(L, K1, Dw)$
$Goto\_F(L, K1, D) \oplus Goto\_F(L, less(K1, K), -)$
$\langle L|Cur\_F : K, St : S, Dr : cl\rangle$
Tint
$[\beta + Tmf * |less(K1, K) - K - 1|]$
$(D = S = Up) \vee (D = S = Dw)$
$Goto\_F(L, K1, D)$
$\langle L|Cur\_F : Less(K1, K), St : S, Dr : cl\rangle$
$Goto\_F(L, K1, D)$
$\langle L|Cur\_F : K, St : S, Dr : cl\rangle$
Tfinal
$[Tmf]$
$(D = S = Up) \vee (D = S = Dw)$
$\langle L|Cur\_F : K1, St : idle, Dr : cl\rangle$
K
$less(K1, K)$
K1
. . .
. . .
. . .
. . .
. . .
$less(K1, less(K1, K))$

Figure 2.1: The Lift Component as a CO-NET.

2.3.2 Evolution of object States

For the evolution of object states in a given component, we propose an appropriate general pattern for the form of 'local' transitions. This pattern has to be be respected in order to ensure the encapsulation property—in the sense that no object states or messages from other components participate during such state evolution— and to preserve object identities uniqueness, with exhibiting a maximal of intra- and inter-object concurrency. This evolution schema, as depicted in Figure 2.3, can be intuitively explained as follows. The contact of the only relevant parts—possible due to the object-state splitting/merging deduction rule— of some object states, namely $\langle Id_1|attrs_1\rangle, .., \langle Id_k|attrs_k\rangle$ with some messages, namely $ms_{i_1}, .., ms_{i_p}$, declared in this component, and under some conditions on the invoked attributes and message parameters, results in the following effect: (1) the messages $ms_{i_1}, .., ms_{i_p}$ disappear; (2) the states of some (parts of) objects participating in the

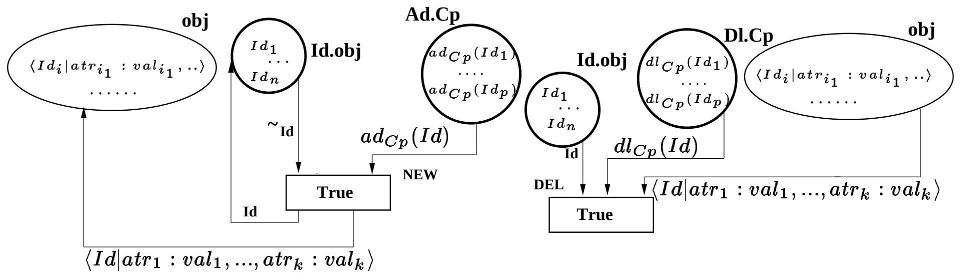
obj
$\langle Id_i|atr_{i_1} : val_{i_1}, ..\rangle$
Id.obj
Ad.Cp
$\sim$Id
$ad_{Cp}(Id)$
NEW
True
Id
$\langle Id|atr_1 : val_1, ..., atr_k : val_k\rangle$
Id.obj
Dl.Cp
obj
$dl_{Cp}(Id)$
DEL
True
$\langle Id|atr_1 : val_1, ..., atr_k : val_k\rangle$

Figure 2.2: Objects creation and deletion using OB-NETS

communication change, namely $I_{s_1}, .., I_{s_t}$. Such change is symbolized by $attrs'_{s_1}, .., attrs'_{s_t}$ instead of $attrs_{s_1}, .., attrs_{s_t}$. The other (unchanged parts of) object states are denoted by $attrs_{i_1}, .., attrs_{i_r}$ so that $\{i_1, ...i_r\} \cup \{s_1, .., s_t\} = \{1, .., k\}$[1]; (3) new messages (local or exported) are sent to object of the component Cp, namely $ms_{h_1}, .., ms_{h_r}$, which may include (explicit) deletion and/or creation of some objects.

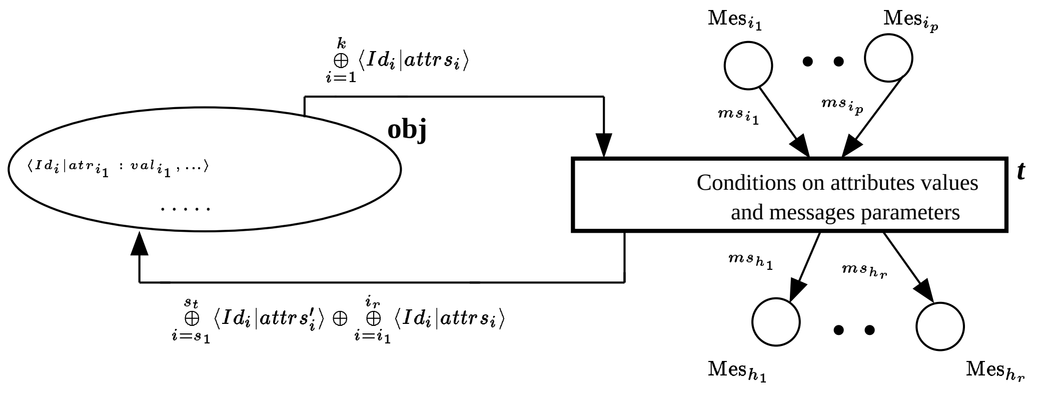
$\overset{k}{\underset{i=1}{\oplus}} \langle Id_i|attrs_i\rangle$
obj
$\langle Id_i|atr_{i_1} : val_{i_1}, ...\rangle$
Mes$_{i_1}$
Mes$_{i_p}$
ms_{i_1}
ms_{i_p}
Conditions on attributes values and messages parameters
t
ms_{h_1}
ms_{h_r}
$\overset{s_t}{\underset{i=s_1}{\oplus}} \langle Id_i|attrs'_i\rangle \oplus \overset{i_r}{\underset{i=i_1}{\oplus}} \langle Id_i|attrs_i\rangle$
Mes$_{h_1}$
Mes$_{h_r}$

Figure 2.3: A particular intra-component evolution pattern

Following our approach for generating rewrite rules gouverning a given transition (see [AS99b] [AS00a]), the corresponding rewrite rule for this general form of transition (depicted in Figure 2.3) is:

$$\textbf{t:}\ (obj, \overset{k}{\underset{i=1}{\oplus}} \langle Id_i|attrs_i\rangle) \otimes \overset{p}{\underset{k=1}{\otimes}} (Mes_{ik}, ms_{ik}) \Rightarrow (obj, \overset{t}{\underset{k=1}{\oplus}} \langle Id_{s_k}|attrs'_{s_k}\rangle \oplus \overset{r}{\underset{k=1}{\oplus}} \langle Id_{i_k}|attrs_{i_k}\rangle)$$
$$\otimes \overset{r}{\underset{k=1}{\otimes}} (Mes_{h_k}, ms_{h_k})\ \texttt{if}\ Condition \wedge M(Ad.Cp) = \emptyset \wedge M(Dl.Cp) = \emptyset.$$

In addition to this case, it is quite possible to capture the intuitive semantics of input tokens IT preceded to the special unary operator $\sim$ (used in object creation net in figure 2.2). That is, IT should not be in the corresponding marking. By assuming without loss of generality that, for instance, only the input tokens associated with the message place Mes_{i1} is preceded by the notation $\sim$, the associated rewrite rule is then:

$$\textbf{t:}\ (obj, IT_{obj}) \otimes (Mes_{i_1}, {}^{\sim}IT_{Mes_{i_1}}) \otimes \overset{p}{\underset{k=2}{\otimes}} (Mes_{i_k}, IT_{Mes_{i_k}}) \Rightarrow (obj, CT_{obj}) \otimes \otimes \overset{r}{\underset{k=1}{\otimes}} (Mes_{h_k}, CT_{Mes_{h_k}})$$
$$\texttt{if}\ Condition\ \wedge M(Ad.Cp) = \emptyset \wedge\ M(Dl.Cp) = \emptyset \wedge\ IT_{Mes_{i1}} \notin M(Mes_{i1})$$

[1]In other words, there is no *implicit* creation or deletion of (parts) of object states— that would lead to inconsistency w.r.t. the described creation/deletion schema in Figure 2.2.

Remark 2.3.1 *The operator $\otimes$ is defined as a multiset union and allows for relating different places identifiers with their current marking. Moreover, we assume that $\otimes$ is distributive over $\oplus$ i.e. $(p, mt_1 \oplus mt_2) = (p, mt_1) \otimes (p, mt_2)$ with mt_1, mt_2 multisets of terms over $\oplus$ and p a place identifier. The condition $(M(Ad.Cp) = \emptyset \wedge M(Dl.Cp) = \emptyset)$ ensures that any deletion and creation message should be performed at first; and this in order to avoid any form of inconsistency like the manipulation of an object already logically deleted and physically still existing (i.e. there is a sending message for deleting this object but not already performed). Note that in the last rewrite rule we have added the condition $IT_{Mes_{i1}} \notin M(Mes_{i1})$ to ensure that the input tokens preceded by $^{\sim}$ should not be in the corresponding marking. It is also worth mentioning that, due to the state splitting / merging deduction rule, we should avoid any deletion of just a part of a given object state. For this aim, we require that before performing any deletion of an object Id (i.e. firing the transition DEL) we should 'gather' all its eventually split part. We achieve that by applying this deduction rule rather as a rewrite rule: $\langle Id \mid attrs_1 \rangle \oplus \langle Id \mid attrs_2 \rangle \Rightarrow \langle I \mid attrs_1, attrs_2 \rangle$, until obtaining the corresponding normal form (which always exists in this case); this ensures that the whole object state is deleted after. Finally, we define a* CO-NETS *state with respect to a given component as the multiset of all pairs of (places, current marking). That is, a* CO-NETS *state denoted $\underline{s}$ is equal to $\underline{s} = \underset{p_i}{\otimes}(p_i, M(p_i))$, $\forall p_i \in P$.*

Example 2.3.2 *By applying this general form of rewrite rule to our case study, we result in the following rewrite rules:*

Go-in-out: $(Lift, \langle L|Dr : op, Wg : W\rangle) \Rightarrow (Lift, \langle L|Dr : op, Wg : W'\rangle)$

Close-Dr: $(Lift, \langle L|Dr : op, Wg : W\rangle) \Rightarrow (Lift, \langle L|Dr : cl, Wg : W\rangle)$ `if` $W < Wmx$

Open-Dr: $(Lift, \langle L|Dr : cl\rangle) \Rightarrow (Lift, \langle L|Dr : op\rangle)$

Tidle: $(GOTO, Goto\_F(L, K1, O) \otimes (Lift, \langle L|Cur\_F : K\rangle) \Rightarrow$
$(Lift, \langle L|Cur\_F : K\rangle)$ `if` $(O = in) \vee (O = out)$

Tnxt: $(GOTO, Goto\_F(L, K1, -) \otimes (Lift, \langle L|Cur\_F : K, St : idle, Dr : cl\rangle) \overset{[Tmf]}{\Rightarrow}$
$(Lift, \langle L|Cur\_F : K1, St : idle, Dr : cl\rangle)$ `if` $(K1 = K + 1) \vee (K1 = K - 1)$

Tfar: $(GOTO, Goto\_F(L, K1, -) \otimes (Lift, \langle L|Cur\_F : K, St : idle, Dr : cl\rangle) \overset{[Tmf-\beta]}{\Rightarrow}$
`if` $(K1 > K + 1)$ `then` $(GOTO, Goto\_F(L, K1, Up) \otimes$
$(Lift, \langle L|Cur\_F : K, St : Up, Dr : cl\rangle)$ `else if`
$(K1 < K - 1)$ `then` $(GOTO, Goto\_F(L, K1, Dw) \otimes$
$(Lift, \langle L|Cur\_F : K, St : Dw, Dr : cl\rangle)$ `endif .`

Tint: $(GOTO, Goto\_F(L, K1, D \oplus Goto\_F(L, less(K1, K, -)) \otimes (Lift, \langle L|Cur\_F : K, St : S, Dr : cl\rangle)$
$\overset{[\beta + |less(K1,K) - K - 1| * Tmf]}{\Rightarrow}$
$(GOTO, Goto\_F(L, K1, D) \otimes (Lift, \langle L|Cur\_F : less(K1, K), St : S, Dr : cl\rangle) //$ `if` $(D = S = Up) \vee (D = S = Dw)$

Tfinal: $(GOTO, Goto\_F(L, K1, D)) \otimes (Lift, \langle L|Cur\_F : K, St : S, Dr : cl\rangle)$
$\overset{[Tmf]}{\Rightarrow}$
$(Lift, \langle L|Cur\_F : K1, St : idle, Dr : cl\rangle) //$ `if` $(D = S = Up) \vee (D = S = Dw)$

Chapter 3

Runtime manipulation of features

The purpose of this section is, first, to review the main ideas and corresponding constructions for handling runtime modification that we proposed in [Aou00]. Then, we propose a more adequate inference rule for propagating a given behaviour from the meta-level to the object level.

3.1 Meta-places and non-instantiated transitions constructions

For handling runtime modification of CO-NETS component specifications, the constructions we proposed in [Aou00] may be summarized as follows:

1. In order to free some CO-NETS transitions[1] from their rigidity, we propose to replace each of their three components— namely input tokens inscribing their input arcs, output tokens inscribing their output arcs and their conditions— by appropriate variables those sorts are exactly the sorts of different inscriptions. Such transitions with only variable inscriptions, are referred to as *non-instantiated* transitions, and their general form is sketched in the lower right hand-side of Figure 3.1. In this general pattern for non-instantiated transitions, all (arc-) inscriptions, namely $IC_{obj}, IC_{i_1}, .., IC_{i_p}$ for input arcs, $CT_{obj}, CT_{h_1}, .., CT_{h_r}$ for output arcs and TC for the condition, should be regarded as appropriate variables.

2. Second, we gather all (input and output) arc inscriptions as well as condition, those variable inscriptions have taken their places, into a single *tuple*: $\langle$`transition_id, version` $|$ `(input-)multiset, (output-)multiset, condition` $\rangle$. In particular, such a tuple for (the general pattern in) transition t in Figure 2.3, takes the following form, where the index i represent a particular *version* of such transition.

$$\langle t : i \mid (obj, IC_{obj}) \bigotimes_{k=i_1}^{i_p} (Mes_k, IC_k), (obj, CT_{obj}) \bigotimes_{k=j_1}^{j_q} (Mes_k, CT_k), Cond. \rangle$$

[1]In the same spirit as in [SRSS98], we assume that some behaviour, i.e. transitions, is fixed forever reflecting minimal properties of the specified application.

3. Third, we consider such 'behaviour' tuples as tokens w.r.t. a corresponding place namely `meta-place` in Figure 3.1, that constitutes the first element of our meta-level. On the basis of this behaviour tokens, it became quite possible to delete some of them, modify some of them or introduce new ones: This corresponds respectively to the transitions `DEL`, `MODIF` and `ADD`[2] and their corresponding places `Del-Bh`, `Chg-Bh` and `Add-bh`.

4. Finally, we relate to two levels, i.e. the meta-place in the meta-level with each non-instantiated transition in the object level, using an appropriate *read-arc*.

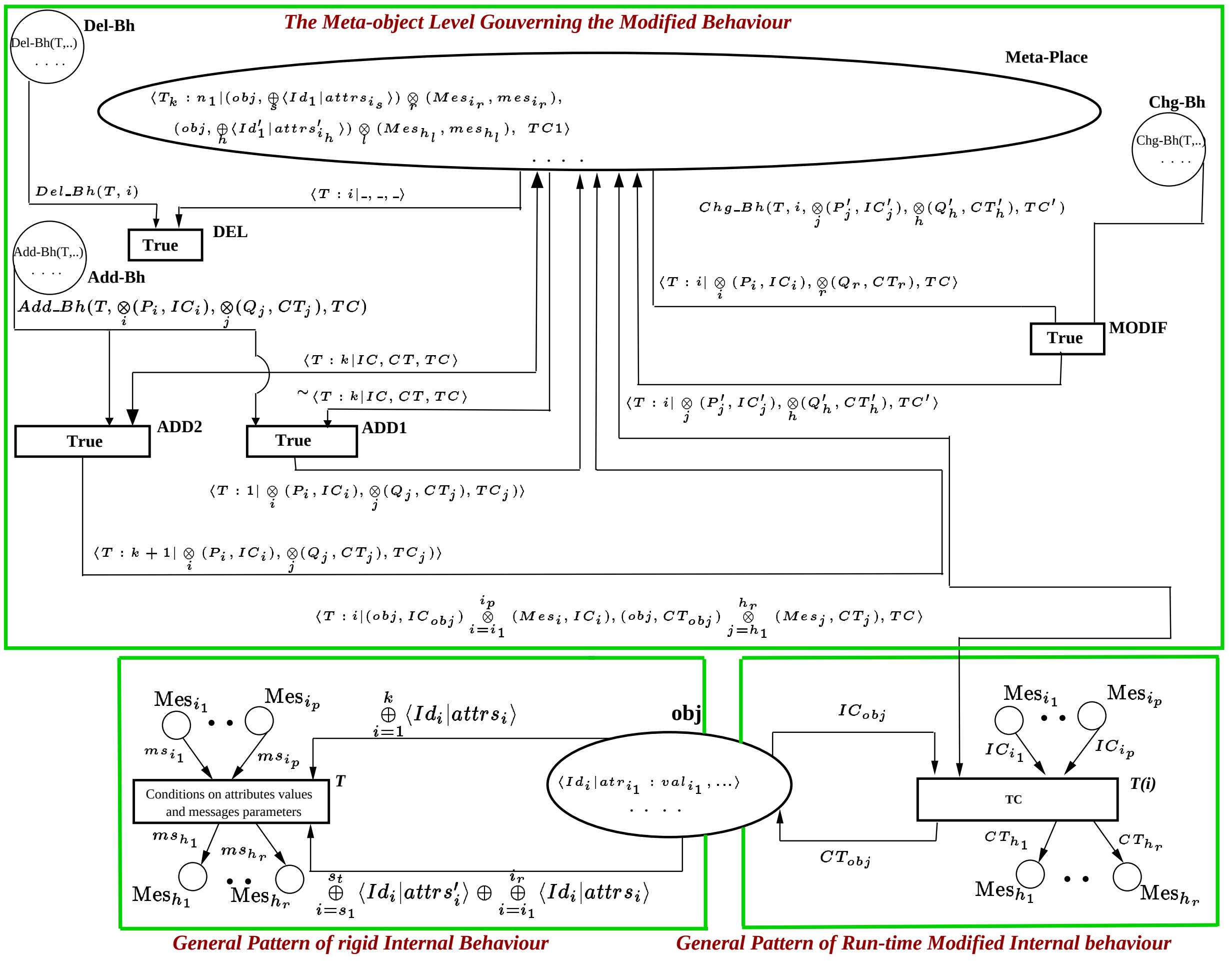
The Meta-object Level Gouverning the Modified Behaviour
Del-Bh
Del-Bh(T,..)
Meta-Place
Chg-Bh
Chg-Bh(T,..)
DEL
True
Add-Bh
Add-Bh(T,..)
MODIF
ADD2
ADD1
obj
T
Conditions on attributes values and messages parameters
TC
T(i)
General Pattern of rigid Internal Behaviour
General Pattern of Run-time Modified Internal behaviour

Figure 3.1: The general pattern for handling dynamic behaviour in Co-nets

3.1.1 Application for manipulating the lift features

Applied to the lift system, the proposed approach allows us to manipulate any feature of the system in a very flexible way, which is moreover achieved while the system is still running. This manipulation concerns different kinds of operations on features, namely the addition of new features without resorting to stopping the system, the deletion of existing

[2]In fact the transition `ADD` is composed of two transitions `ADD1` and `ADD2` corresponding to the cases of adding a new version for an existing transition behaviour or a (first version for) new transition.

features, and finally the modification, that is, the update of some outdated features. In a more detail, the features we propose to deal with to illustrate the appropriateness of this approach, are the following:

- First, we would like to introduce a feature that allows for some particular lifts (selected using their identities) to go to a 'stationary' floor when there is no call inside or outside. But, we would at the same time let this stationary floor modifiable, for instance, depending on the rush time—the underground floor in the morning and the appropriate one in the evening. This of course cannot be specified using a fixed transition, rather it should be considered as a token that can be update whenever necessary. Such a token may have the following form:
  $\langle Reset : 1|(GOTO, {}^{\sim}Goto\_F(L,-,-)) \otimes (Lift, \langle L|Cur\_F : K, Dr : cl, Wg : 0\rangle), (Lift, \langle L|Cur\_F : 0, Dr : cl, Wg : 0\rangle), K \neq 0\rangle$
  In this token, we have chosen for instance as stationary floor the ground one (i.e. F =0). We also note that the condition ${}^{\sim}Goto\_F(L,-,-)$ implies that no requests are currently existing (in the place `GOTO`). Of course, now using the transition `MODIF` at the meta-level, we can add more complex conditions, depending for instance on the time as well as changing the stationary floor in consequence.

- In the initial specification, we have included the possibility of canceling any requests from (inside or outside) the same floor where the lift is. An improvement of this feature may be the cancellation also of any request from inside the lift when it is empty (i.e. its weight is set to zero). For this, we propose to consider the transition `Tidle` as non-instantiated one, and introduce its new behaviour as a token. This behaviour takes the following form:
  $\langle Tidle : 1|(GOTO, Goto\_F(L, K1, O)) \otimes (Lift, \langle L|Cur\_F : K, Dr : cl, Wg : W\rangle), (Lift, \langle L|Cur\_F : K, Dr : cl, Wg : W\rangle), ((K1 = K) \vee (O = in \wedge W = 0))\rangle$

- The feature that we would also like for some lifts (we denote by `LsLf` as a list of identities of some selected lifts), is that, when their weights are greater than the 2/3 of the their maximal weight they skipped intermediate floors. This feature concerns directly the transition `Tnxt` that has to be now considered as a non-instantiated. In this behaviour as a token, we should add the weight attribute in input arc from the lift place and add the condition $W < 2/3Wmx$ to the transition. That is to say the taken would be now as follows:
  $\langle Tnxt : 1|(GOTO, Goto\_F(L, K1, D) \oplus Goto\_(L, less(K1, K), -))\otimes$
  $(Lift, \langle L|Cur\_F : K, St : S, Dr : cl, Wg : W\rangle), (GOTO, Goto\_F(L, K1, D) \otimes (Lift, \langle L|Cur\_F : K, Dr : cl, Wg : W\rangle), ((D = S = Up) \vee (D = S = Dw)) \wedge (W < 2/3Wmx\rangle$

All these features are illustrated in Figure 3.2, where IC_{var}, CT_{var} and TC_{var} are of course appropriate variables for capturing changing input inscriptions, output inscriptions and conditions respectively. Also, we note that all unchanged transition in the net depicted in Figure 2.1 have been ignored for getting a simple net.

3.2 Semantical part : the meta-inference rule

For theoretical underpinning of these constructions, we propose in the following, with respect to the same CO-NETS semantic rewrite logic framework an adequate inference rule that can be regarded as a more flexible formulation of the one proposed in [Aou00]. The main ideas under this reformulation as described below are the following. First, we generate the rewrite rule associated with the non-instantiated transition in Figure 3.1 in the same way as we done for usual CO-NETS transitions, except that we introduce a new binary operator denoted $\|_r$ separating the other pairs of place-tokens and the read-arc inscription. This operator is necessary because we should express the fact that tokens brought using read-arc are *not* from the object level (i.e. the CO-NETS state), but from the meta-level state. We have called this rule as non-instantiated rewrite rule, namely t^{nins} because it cannot be applied directly. From this non-instantiated transition, we can obtain a usual transition by selecting one behaviour as a token, from the meta-place, by applying different corresponding substitutions to different variables in the read-arc token. This fact is reflected by the inference rule, where $M(P_{meta})$ represents the current marking of the place `meta-place`, while the notation $|[T_{s(p_i)}]|_\oplus$ represents a class of (multiset) term (over the associativity, commutativity of $\oplus$) those sort is exactly the one of the place p_i.

For each (meta-)rewrite rule :

$$t^{nins} : |[\overset{k}{\underset{i=1}{\otimes}}(p_i, IC_i)]| \,\|_r (P_{meta}, \langle t : i \mid [\overset{k}{\underset{i=1}{\otimes}}(p_i, IC_i)]|, |[\overset{l}{\underset{j=1}{\otimes}}(q_j, CT_i)]|, TC_i\rangle)$$
$$\Rightarrow [\overset{l}{\underset{j=1}{\otimes}}(q_j, CT_i)]| \quad \texttt{if} \quad TC_i$$

We have:

$$\frac{\begin{array}{c}\exists \sigma_i \in [T_{s(p_i)}]_\oplus, .., \exists \sigma_j \in [T_{s(q_j)}]_\oplus, \exists \sigma \in [T_{bool}] \wedge \\ \langle t : k \,|[\overset{k}{\underset{i=1}{\otimes}}(p_i, \sigma_i(IC_i))]|, |[\overset{l}{\underset{j=1}{\otimes}}(q_j, \sigma_j(CT_i))]|, \sigma(TC_i)\rangle \in M(P_{meta})\end{array}}{t^{ins} : |[\overset{k}{\underset{i=1}{\otimes}}(p_i, \sigma_i(IC_i))]| \Rightarrow [\overset{l}{\underset{j=1}{\otimes}}(q_j, \sigma_j(CT_i))]| \quad \texttt{if} \quad \sigma(TC_i)}$$

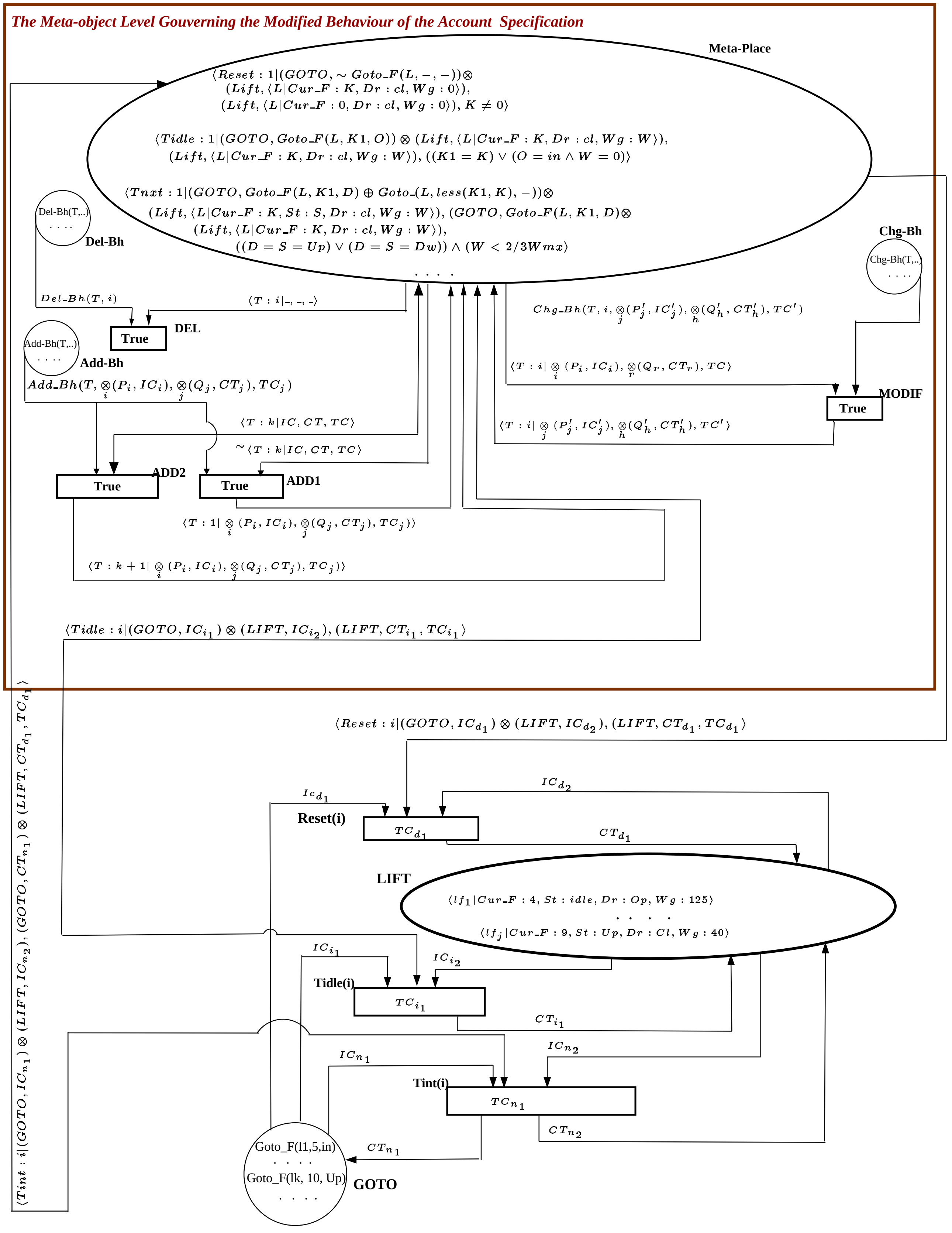

Figure 3.2: Dynamic manipulation of the lift system features

Chapter 4

Strategies in CO-NETS as features interaction

Besides its true concurrency nature and its operationality allowing generation of rapid-prototypes, the key feature of rewriting logic is also its *intrinsic* reflection capabilities. For our CO-NETS framework, the crucial advantage of this second meta-level is the possibility of introducing appropriate strategies for *controlling* the way in which different transitions should be fired. This is of great benefit for reflecting a current functioning of a given system, without resorting to fix forever a particular way of functioning. Moreover, this allows to free as much as possible the net from such control—this of course led to a very simple and flexible net specification. In the following, we give more detail about this level, then we apply it to the case study.

4.1 Strategies using reflection in rewrite logic

Rewriting logic is reflective [MOM96], that is, there is a finitely presented rewrite theory $\mathcal{U}$ that is *universal* in the sense that we can represent in $\mathcal{U}$ any finitely presented rewrite theory $\mathcal{R}$ (including $\mathcal{U}$ itself) as a term $\overline{\mathcal{R}}$, any terms t, t' in $\mathcal{R}$ as terms $\overline{t}$, $\overline{t'}$, and any pair $(\mathcal{R}, \sqcup)$ as a term $(\overline{\mathcal{R}}, \overline{t})$, in such a way that we have the following equivalence

$$(\dagger)\ \mathcal{R} \vdash t \longrightarrow t' \Longleftrightarrow \mathcal{U} \vdash \langle \overline{\mathcal{R}}, \overline{t} \rangle \longrightarrow \langle \overline{\mathcal{R}}, \overline{t'} \rangle$$

In the same way this equivalence may be applied between the meta-level and the meta-meta.level, and so on, leading to a reflective that may be illustrated as follows:

$$(\dagger)\ \mathcal{R} \vdash t \longrightarrow t' \Longleftrightarrow \mathcal{U} \vdash \langle \overline{\mathcal{R}}, \overline{t} \rangle \longrightarrow \langle \overline{\mathcal{R}}, \overline{t'} \rangle \Longleftrightarrow \mathcal{U} \vdash \langle \overline{\mathcal{U}}, \overline{\langle \overline{\mathcal{R}}, \overline{t} \rangle} \rangle \longrightarrow \langle \overline{U}, \overline{\overline{\mathcal{R}}, \overline{t'}} \rangle \rangle$$

For a system of rewrite rules, gouverning transitions behaviour as in our case, it is now possible to represent it as (a pair of) datatypes, and the rewriting of any CO-NETS state representing the current marking can be now completely controlled. In this sense, an expressive language for composing different rewrite rules has be developed for the MAUDE language; that we adopt here. First a kernel is defined stating how rewriting in the object level is accomplished at the metalevel. In particular, Maude supports a strategy language kernel which defines the operation:

op meta-apply : Term Label Nat − > Term.
A term **meta-apply(t,l,n)** is evaluated by converting the metaterm t to the term it represents[1] and matching the resulting term against all rules with the given label l. The first n successful matches are discarded, and if there is an $(n+1)th$ successful match its rule is applied, and the resulting term is converted to a metaterm and returned; otherwise, $error*$ is returned.

The strategy language **STRAT** defined in [CDE+99] extends the kernel with operations to compose strategies, and also with operations to create and manipulate a solution tree obtained by the application of a strategy. It defines sorts **Strategy** and **StrategyExp** for strategies, and sorts **SolTree** and **SolTreeExp** for the solution tree. The main operations defined on strategies are:

- operations defining basic strategies:

  op idle : − > Strategy . *//* idle is an empty strategy *//*
  op apply : Label − > Strategy *// application fo a given rule //* .
  op rew_ =>_with_ : Term SolTreeExp Strategy − > StrategyExp .
  failure : − > StrategyExp .

- operations defining solution trees :

  op ? : − > SolTreeExp .
  op apply : Label − > Strategy .
  op rew_ =>_with_ : Term SolTreeExp Strategy − > StrategyExp .
  failure : − > StrategyExp .

- operations that compose strategies:
  op _; _ : Strategy Strategy − > Strategy *// application of two strategies in sequence //* .
  op _; ; _orelse_ : Strategy Strategy Strategy − > Strategy *// a choice between two strategies //.*
  op iterate : Strategy − > Strategy *// application repetitive of a given strategy until is nomore applied //.*

For a more detail about the semantics of these operations, the reader may particularly consult [CDE+99].

4.2 Application to the lift system

As we pointed out the meta-level we propose for dynamically evolving CO-NETS specification, is in itself not sufficient for expressing particularly how different system features interact with each other. In this paragraph, using the running case study, we show how the above meta-level for reasoning about rewrite rules gouverning different transitions behaviour, when applied to a given CO-NETS state, is of great help. Indeed, first recalling

[1]The datatype meta-representation of an object term is a list based one; for instance, a natural term $s(s(0)) + s(0)$ is represented by $'\_+\_['s\_['s\_['0]],'s\_['0]]$.

that usually in Petri nets a transition may be fired as soon as it become fireable, and there is no way for controlling the order in which transitions are fired. This difficulties induce that designer of the net should decided whether (s)he let irrelevant such order or opt for a default order and integrate it directly in the model—leading in most of real-case to very complex net which works only for this default strategy.

To be more explicit, in the CO-NETS specification of the lift system in Figure 2.1 obviously we have decided for the first solution, that is, no control at all of different system features. In fact, we did not state, for instance, when should the door be opened, closed, or should the lift first 'serves' the next / precedent floors or those which are far at first. Imagine that we have decided for a particular strategy; for instance, first serve the next / precedent floors, then open the floor then close the door (after some time), and then if any serve more far floors, and make that repeatedly. It is very hard to imagine how complex and artificial would be the resulting net for such a default strategy. For instance, for firing the transition `Door-Op` we have to be sure that the transition `Tnxt` has be fired. This means that we should add an artificial place and two transitions and relate them to the transition `Tnxt` as input and to the place `Door-Op` as output, and so on for the remaining.

Fortunately thanks to this rewrite logic meta-reflection, the lift specification in Figure 2.1 remains unchanged while we can formulate any strategy we would like to have. For instance, taking into account as label the name of different transitions, the above 'default' strategy we would like to have may be simply expressed by the following:

$$iterate(Tnxt; Door - Op; Door - Cl; (Tfar; Tint; Tfinal)*)$$

The meta-level of rewriting logic would respects this strategy. Moreover, we can associate complex conditions on applying such strategies.

Chapter 5

Conclusions

We presented in this paper a general-purpose framework, referred to as Co-nets, particularly suited for specifying complex distributed systems, and for dynamically manipulating their features. Methodological this framework may be regarded as three layer-based one. The first layer is a sound integration object oriented abstractions mechanism with modularity constructs into an appropriate variety of algebraic Petri nets. The second layer is meta-level one, and it allows for dynamically creating, modifying and/or deleting features while the system is still running. The third layer takes profit of the reflection capabilities of rewrite logic—as semantics for Co-nets behaviour—for dynamically composing and interacting strategies, as appropriate and complex features.

We have illustrated this framework using a non trivial specification of a variant of a system with several lifts. This case study shown us in particular the suitability of this framework for dealing complex real-world distributed systems and their features. However, after achieving this first step we are conscious that much more work remains ahead. Particularly, we are planning to extend this case study to more complex one. Also, we are focusing on the integration of temporal aspects for verifying, and not only validating, system properties.

Bibliography

[Aou00] N. Aoumeur. Specifying Distributed and Dynamically Evolving Information Systems Using an Extended CO-NETS Approach. In G. Saake, K. Schwarz, and C Türker, editors, *Transactions and Database Dynamics*, pages 91–111, Lecture Notes in Computer Science, Berlin, Vol. 1773, Springer-Verlag, 2000. *Selected papers from the 8th International Workshop on Foundations of Models and Languages for Data and Objects, Sep. 1999, Germany.*

[AS99a] N. Aoumeur and G. Saake. Operational Interpretation of the Requirements Specification Language ALBERT Using Timed Rewriting Logic. In *Proc. of 5th Int. Workshop on Requirements Engineering: Foundation for Software Quality (REFSQ'99), Heidelberg, Germany*, Presses Universitaires de Namur, 1999.

[AS99b] N. Aoumeur and G. Saake. Towards an Object Petri Nets Model for Specifying and Validating Distributed Information Systems. In M. Jarke and A. Oberweis, editors, *Proc. of the 11th Int. Conf. on Advanced Information Systems Engineering, CAiSE'99*, Lecture Notes in Computer Science, Vol. 1626, pages 381–395, Springer-Verlag, 1999.

[AS00a] N. Aoumeur and G. Saake. CO-NETS: A Formal OO Framework for Specifying and Validating Distributed Information Systems. Preprint Nr. No. 2, Fakultät für Informatik, Universität Magdeburg, 2000.

[AS00b] N. Aoumeur and G. Saake. Specifying and Validating Train Control Systems Using an Appropriate Component-Based Petri Nets Model. In M. Huba S. Kozak, editor, *Proc. of the Petri Nets in Design, Modelling and Simulation of Control Systems Special session at IFAC Conference CSD2000, Bratislava, Slovakia*, Elsevier Science, 2000. ISBN: 0-08-043546-7.

[CDE$^+$99] M. Clavel, F. Duran, S. Eker, J. Meseguer, and M. Stehr. Maude : Specification and Programming in Rewriting Logic. Technical report, SRI, Computer Science Laboratory, March 1999. URL : http://maude.csl.sri.com.

[CM96] M. Clavel and J. Meseguer. Reflection and Strategies in rewriting logic. In G. Kiczales, editor, *Proc. of Reflection'96*, pages 263–288, Xerox PARC, 1996.

[DB95] P. Du Bois. *The Albert II Language: On the Design and the Use of a Formal Specification Language for Requirements Analysis.* PhD thesis, Computer Department, University of Namur, Namur(Belgique), September 1995.

[GRS96] A. S. Guerra, M. Ryan, and A. Sernadas. Features-oriented Specifications. Technical report, School of Computer Science, University of Birmingham, 1996.

[Hei98] M. Heissel. Detecting Features Interactions—A Heuristic. In *Proc. of the 1st FIREworks Worshop*, pages 30–48. Magdeburg, Germany, 1998. Technical Report, Preprint Nr. 10, Computer Science Department, University of Magdeburg.

[Mes92] J. Meseguer. Conditional rewriting logic as a unified model for concurrency. *Theoretical Computer Science*, 96:73–155, 1992.

[Mes93] J. Meseguer. A Logical Theory of Concurrent Objects and its Realization in the Maude Language. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Object-Based Concurrency*, pages 314–390, The MIT Press, 1993.

[MOM96] N. Marti-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. In J. Meseguer, editor, *Proc. of First International Workshop on Rewriting Logic*, Electronic Notes in Theoretical Computer Science, Vol. 4, pages 189–224, 1996.

[RBP$^+$91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object Oriented Modeling and Design.* Prentice Hall, Englewood Cliffs, 1991.

[Rya97] M. Ryan. Features-oriented programming : A case study using the SMV language. Technical report, School of Computer Science, University of Birmingham, 1997.

[SRSS98] Conrad S, J. Ramos, G. Saake, and C. Sernadas. Evolving Logical Specification in Information Systems. In J. Chomicki and G. Saake, editors, *Logics for Databases and Information Systems*, chapter 7, pages 167–198, Kluwer Academic Publishers, Boston, 1998.

[Weg90] P. Wegner. Concepts and paradigms of Object-Oriented Programming. *OOPS Messenger*, 1:7–87, 1990.