Features Interaction in Adaptive Service Environments: A Reflective Petri Nets Approach

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

***Abstract*— With the ubiquity of service-driven applications, the challenging concern remains on keeping their respective software components dynamically *updated* and fully *consistent* as service functionalities and composition behaviours change. Features Interaction tackles this problem though only at design-time and in centralized systems and thus do not scale up to service-driven systems, which are by essence volatile, distributed and compositional. To soundly and dynamically adapting features in such environments, we propose a specification/validation framework based on a component-based Petri nets variant endowed with reflection capabilities and a true concurrent rewrite logic-based semantics. We particularly demonstrate how this framework allows specifying updating and composing features in a *runtime* and *true-concurrent* way, with the ability of visual animation and symbolic validation and verification.**

Keywords— Features and services, High-level Petri nets, rewrite logic, dynamic evolution, specification and validation

I. Introduction

The pervasiveness of the internet coupled by the standardization of plethora of Web-services languages (e.g. BPEL, WSDL, WSCI)[1] are promoting service-oriented computing (SOC) as the future software development paradigm. Main characteristics of this paradigm include the abstraction from concrete services implementation (using services interfaces), reactivity and full distribution, rapid and unanticipated service requirements evolution and dynamic composition of services.

With the inability of such XML-based languages in delivering such challenging promises on their own, both academia and industry agree on the need of leveraging this technology to cope with early phases of requirements specification validation and verification. To keep in pace with market volatility and unanticipated requirements evolution of service functionalities or *features*, designers are thus facing challenging conceptual problems to develop dynamic and consistent evolving service-driven systems. In banking systems, for instance, new packages of different advanced functionalities (e.g. withdrawals, loan/mortgage packages) are increasingly offered mostly on-the-fly (and online) to attract more new customers and reward those already in contract.

Features Interaction (FI) has been recently coined as a new research field in software-engineering for tackling these problems at different levels, with special emphasis on the conceptual level [13]. The last conference on FI [21] surveys most of these recent advances. Nevertheless, due to the mentioned complexity and volatility of the emerging service-driven systems, existing approaches remain far from tackling features interaction in such volatile distributed and compositional environments, and thus suffering from:

- The expressiveness of adopted specification frameworks, in terms of advanced structuring mechanisms (e.g. inheritance, aggregation, composition and modularity). Indeed, most of proposals are based on temporal and process languages [21], known for their limited structuring.
- The capabilities of exhibiting *concurrent* and *distributed* behaviour—as premises for service-driven applications. In this sense, *diagrammatical* specifications with graphical animation and formal reasoning capabilities seem very crucial.
- The intrinsic ability of *dynamically* adapting or

adding new system features to stay competitive, adapt quickly to market changes, satisfy very demanding customers, and keep in pace with the application / technology requirements evolution.

Besides that, new emerging formalisms to SOC (e.g. temporal-based [24], [20], rule-based ones [12], Petri-nets based [16], [28], [9]) are still far from addressing features specification, interaction and evolution, and are thus mostly focussing stable application features like the structure and some restricted behaviors in static manner.

The present paper aims at contributing to overcome some of these severe shortcomings, by proposing first milestones towards new approach to features interaction that intrinsically captures the above crucial issues and thus seems very appropriate for specifying evolving and interacting features in adaptive service-driven applications. The approach is based on a variant of component-based Petri nets endowed with reflection capabilities for dynamic evolution and true-concurrent rewrite logic-based semantics for rapid prototyping via smooth shift between the base- and the meta-level. This conceptual model, referred to as CO-NETS [3], [4], has been before mainly applied to evolving concurrent information systems.

We propose thus to rigorously specifying and validating different service functionalities or features being it simple or complex (by combining different basic features into strategies to reflect realistic behaviours). On the other hand, with the reflection capabilities, such features can be dynamically manipulated (i.e. added/removed/updated) without stoping non-concerned features or decreasing the degree of distribution of the whole running application.

The rest of this paper is organized as follows. The next section presents the informal running example using UML diagrams. In the third section, we demonstrate how to specify, compose and validate features using the CO-NETS framework and its rewrite logic semantics. In the fourth section, we address the problem of how features may be dynamically manipulated using a meta-level constructions over CO-NETS component specifications. We conclude this paper by some remarks and outlying our future work. With the endeavor to attract a wide-audience, we kept the paper presentation at a sufficient intuitive level, while pointing hints to formal concepts when required.

II. THE MULTI-LIFT SYSTEM: INFORMAL DESCRIPTION

To illustrate our approach to features specification/validation, dynamic evolution and interaction in complex adaptive distributed service-driven applications, we adopt throughout this paper a simplified version of a multi-lift system. We mention that this application has been intensively adopted in features interaction research [23], [13], [21], though only at design-time and using just one lift.

In contrast to existing proposals, that directly build on formal descriptions, we first present an informal description based on semi-formal diagrammatical UML [7] ”profiled” class-diagrams. We should point out that other diagrams such as sequence diagrams and particularly state-charts could complement this structural description, but as we are adopting CO-NETS for behavioural concerns they are not required.

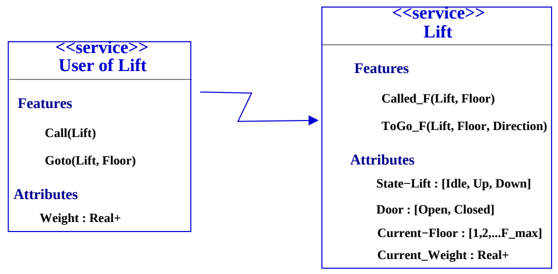

Fig. 1. The Lift and User services as profile UML Classes

As depicted in Figure 1, each lift is structurally characterized by the following attributes: *Identity* (as name or number), for uniquely identifying each lift; *Current floor* on which a given lift may be on at a given time. We denote it by `Cur_Floor`; *Lift state* expressing whether the lift is *idle*, *going up* or *going down* (we denote such state by `State-Lift`). To keep track of directions while serving intermediate floors, we add the state value ”`Stop`”, which can ”suffixes” the other state values. For instance, ”`Up.Stop`” implies that the lift is going up and is currently stopping at an intermediate floor; *Door state* (shortly `Door`), that could be either open (`Open`) or closed (`Closed`); and *Current weight* (shortly `Wg`) is in kg for instance.

The service operations or features acting on such attributes are the following:

- The lift may be called from the outside (by a user) at any time. We denote this service functionality as `Called(LiftId, Floor)`. As depicted in Figure 1, the user initiates this functionality through a call operation we denote by `Call(LiftId)`.
- Being inside a lift, users can trigger the lift to travel to a specific floor. We denote this lift service functionality by `ToGo(LiftId, Floor, Direction)`. This triggering operation from the user ”service” is denoted by `GoTo(LiftId, Floor)` at the lift service side.
- Besides these observed functionalities, internal operations include: the opening/closing of the lift (through programmable sensors) and the update of its weight in consequence.

III. Features Specification in Co-nets

We first present how any application structural aspects are specified using Co-nets, and apply them to the running example. We then present system behavioural specification using Co-nets.

A. Specification of structural aspects in Co-nets

A service signature defines the structure of service states and the form of (received/invoked) messages which have to be accepted by such component states. We regard service states as algebraic terms —precisely as tuples— and messages as operations sent or received by services. More precisely, Co-nets service signatures are specified as follows.

- States are algebraic tuples of the form $\langle Id|at_1 : vl_1, ..., at_k : vl_k, bs_1 : vl'_1, ..., bs_{k'} : vl'_{k'}\rangle$
  Id is an observed identity taking its values from an appropriate abstract data type (ADT); $at_1, .., at_k$ are the internal (i.e. hidden from the outside) attribute identifiers with values respectively $vl_1, .., vl_k$; and The observed part of a state is identified by $bs_1, ..., bs_{k'}$, with associate values as $vl'_1, ..vl'_{k'}$.
- We distinguish between imported / exported and internal messages. Observed messages permit interacting different services using observed attributes, while internal messages allow computation within components.
- Further, to exhibit intra-concurrency, we propose to split/recombine the above state at need. This process is axiomatized by an inference rule (see [3]) taking the form of an equation : $\langle Id|attrs_1, attrs_2\rangle = \langle Id|attrs_1\rangle\langle Id|attrs_2\rangle$. In this equation the abbreviation $attr_i$ corresponds to a state part $at_{i1} : vl_{i1}, ..., at_{ik} : vl_{ik}$, while $\_\_$ a multiset union operation (see later).

We specify such service structures in a formal way using OBJ- and Maude-like notations [10], [19]. The corresponding OBJ-specification of this general service signature is depicted in the appendix.

1) Specification of the lift-service interface structure: To apply this Co-nets structural specifications to the informal description of the multi-lift system, we first require some abstract data types.

Multi-lift Data signatures. The different abstract data sorts we require for the services signature are the following. To capture different floor levels, we use the sort `Floors` whose elements are natural constants (0,1, 2, .., k). To express the states the lift may be in, we use the sort `StateF` whose elements are constants, namely `idle` or `Up`, `Dw` or `Stop`.The doors state is captured by a data sort `Door`, with two values: `op` for opened and `cl` for closed. We use a positive real constant `Wmx`, to define the maximal weight allowed by each lift.

The corresponding data specification following the adopted OBJ-like notations is given below under the name `Lift-data`. Using this data level and the above informal UML-based description, the service signature corresponding to the lift can then be described. In this signature, with each service operation we associated a ”message” sort and message operation. Please note that all the declared variables will be subsequently used in the corresponding nets.

```
obj Lift-Data is
  protecting Real+  nat .
  sort  Door  StateF.
  op 0, 1, 2,.., k : → Floors.
  op idle, Up, Dw, Stop : → StateF.
  op_._ : StateF  "Stop" → StateF.
  op op, cl : → Door.
  op idle, Up, Dw : → StateF.
  op Wmx : → Real+ .
endo.

service Lift is
  extending Service-State .
  protecting Lift-Data .
  sort Id.Lift < OId .
  sort TOGO  CALLED  LIFT .
  (* the Lift service state declaration *)
  op ⟨_|Cur_F : _, St : _, Dr : _, Wg : _⟩: Id.Lift
      Floors  StateF  DoorSt  Real+ → LIFT .
```

```
(* Features declaration *)
   op ToGo_F : Id.Lift Floors StateF → GOTO .
   op Called_F : Id.Lift Floors → CALLED .
(* Variable to use in the corresponding net *)
  vars L : Id.Lift .
  vars S, D : StateF .
  vars W, W′ : Real+ .
  vars K, K1, K2, K′ : Floors .
endsrv.
```

B. Specification of features behaviour in CO-NETS

From a given service signature we incrementally construct its corresponding behaviour by associating it a CO-NET, as follows. The net Places are constructed by associating with each message generator one 'message' place. Such places thus hold message instances acting on service states. With each state sort we also associate one 'state' place. Such places hold current service states. The net transitions, which may include conditions, reflect the intended effect of each message on service states. To distinguish between local messages and external ones, we draw the later with bold lines.

We mention on the passage, that similar other high-level Petri nets variants also integrate structured (objects-)data and behavior. They include, among others, [22], [11], [14], [6], [5] and [25]. Nevetheless, to the best of our knowledge no variant have been applied to features specification, let alone feature dynamic evolution and composition as we present in this paper. Further, a comparative studies of CO-NETS capabilities with respect to these similar proposals have been investigated in [3].

1) Specification of the lift-service interface behaviour: By applying these simple rules to the informal description of the case study, Figure 2 depicts the corresponding user-multi-lifts CO-NET-based observed and concurrent behaviour (i.e. triggered by the users).

This CO-NET-driven behavior is thus incrementally constructed by first associating with the state sort `Lift` a place containing the different lifts states. Second, with the two message sorts `TOGO` and `CALLED` we associated two corresponding places. The corresponding behaviour is captured in terms of transitions associated with these messages. For instance, the transitions `Tskipgo` and `Tskipcal` permit to skip (i.e. consume) any called/goto messages from/to the same floor where the lift car is stationing. This transition allows also to delete already served called/goto orders (performed by other transitions). The transition `Tcalled` corresponds to the case where a called order (from the outside lift) is directly served; That is, a called order is served when at that moment no goto order (from inside the lift) exists. For this purpose, the symbol ~ reflects the inhibitor arc from the place `GOTO`. The transition `Tgoint` corresponds to the existence of intermediate requested goto orders (from inside) while performing the transition `Tgofar`. That is, besides the token $ToGo\_F(L, K1, S)$ there should be another token $ToGo\_F(L, less(K, K1), -)$ in the place `TOGO`. The transition `Tcallint` is probably the most complex one as it corresponds to the case where intermediate calls are being requested from outside while performing the transition `Tgofar` (i.e. at least a message token as $ToGo\_F(L, K1, S)$ from the place `TOGO` exists[1]). The lift will serves such intermediate stops, but with still the direction ($S = Up$ or Dw) as prefix (keeping track of the final destination). As several intermediate calls may be simultaneously requested, we must serve the *nearest* one first, that we denote by a function $less(K, K1)$ we assume defined at the data-level. For instance, when going `Up`, $less(K, K1)$ first tests whether $K + 1$ is requested, if not it then tests $K + 2$ and so on till $K1 - 1$. The transition `Tgoint` corresponds to the existence of intermediate requested GoTo orders (from inside) after performing the transition `Tgofar`. That is, besides the token $ToGo\_F(L, K1, S)$ there should be other tokens as $ToGo\_F(L, less(K, K1), -)$ in the place `TOGO`. Finally, the transition `Tgonxt` concerns the two following cases of a goto message. First, it concerns performing a non-next goto message (i.e. $S = D = (Up \vee Dw)$) after serving all intermediate requested floors (using `Tgointr` and `Tcallint`) if any. Second, it concerns performing a ToGo for a next floor (i.e. $K1 = K + 1 \vee K1 = K - 1$) if any. Withe the aim to have a complete specification, we also specified intended internal behaviour of this multi-lift case study as depicted in Figure 3. This behaviour reflects the opening and closing of doors, where the lift car should be either in an idle or (intermediate) stop state. Second, while the door is open, users can go-in and go-out which means the weight may change. The transition `Tstop`

[1]Used here as read-arc with S be either Up or Dw.

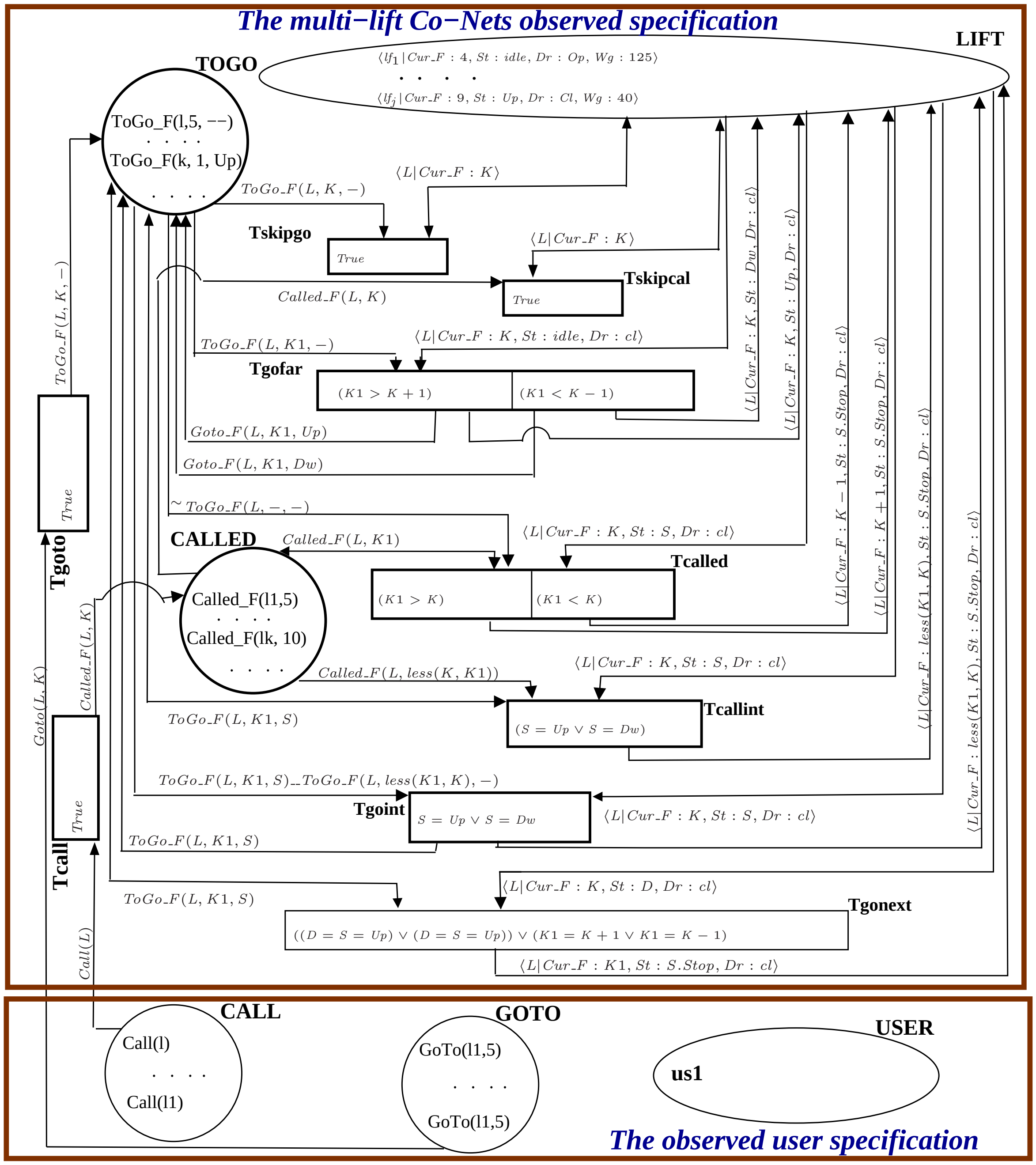

Fig. 2. The Multi-Lift-User Components observed specification

allows the lift to stop at intermediate floors, which allows the possibility of performing `Tgo-io`, `Topen` and `Tclose`. After that the lift continues its way by performing the transition `Tcont`. We should note that detection of reaching any floor is governed by suitable sensors (component) we are simply omitted here for sake of simplicity.

C. *Features validation with* CO-NETS *semantics*

One of the main advantage of the CO-NETS approach is its operational semantics expressed in rewrite logic [18], where each transition is governed by a corresponding rewrite rule interpreted in a suitable instantiation of this logic we referred to as CO-NETS rewrite theory. The main ideas of this interpretation can be sketched as follows. To bind each place marking mt with its

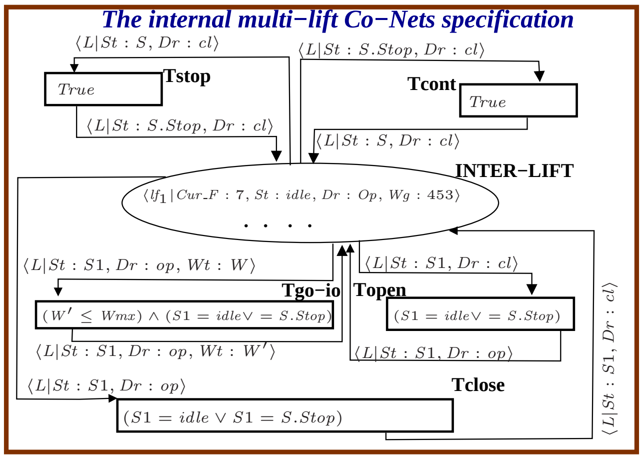

Fig. 3. The Internal Multi-lift Specification

corresponding place p we capture them as a pair (p, mt). Different tokens within mt are gathered using a multiset union operator we denote by $\_\_$. To represent CO-NETS states as multisets over different the pairs (p_i, mt_i), we introduce another multiset generated by a union operator we denote by $\otimes$. That is, a CO-NETS state is described as a multiset of the form: $(p_1, mt_1) \otimes (p_1, mt_2) \otimes ....$ To exhibit a maximum of concurrency, we allow distributing $\otimes$ over $\_\_$. That is, if mt_1 and mt_2 are two marking parts in a given place p as $(p, mt_1\_\_mt_2)$, then we can always split it to $(p, mt_1) \otimes (p, mt_2)$. To exhibit intra-state concurrency, we permit the splitting and recombining of such state tuple at a need.

Figure 4 presents a general pattern for CO-NETS intra-component transitions. It expresses that when messages enter in contact with some (attributes of) states, under eventual constraints on such messages and attributes, the resulting is the consumption of such messages, the creation of new messages and the change of states.

Following the above guidelines, the rewrite rule corresponding to this general transition pattern, takes the following form:

Tintra: $(obj, \underset{i=1}{\overset{k}{\_\_}} \langle Id_i|attrs_i\rangle) \overset{p}{\underset{k=1}{\otimes}} (Mes_{ik}, ms_{ik})$
$\Rightarrow (obj, \underset{k=1}{\overset{t}{\_\_}} \langle Id_{s_k}|attrs'_{s_k}\rangle \_\_ \underset{k=1}{\overset{r}{\_\_}} \langle Id_{i_k}|attrs_{i_k}\rangle)$
$\overset{r}{\underset{k=1}{\otimes}} (Mes_{h_k}, ms_{h_k})$ `if` *Condition.*

Example: By applying these guidelines for generating CO-NETS transition rewrite rules to the lift component depicted in Figure 2, the rewriting rules of some selected transitions (due to space limitation) is given below:

Tskipgo: $(TOGO, ToGo\_F(L, K, \_)) \otimes$
$(Lift, \langle L|Cur\_F : K\rangle) \Rightarrow (Lift, \langle L|Cur\_F : K\rangle)$

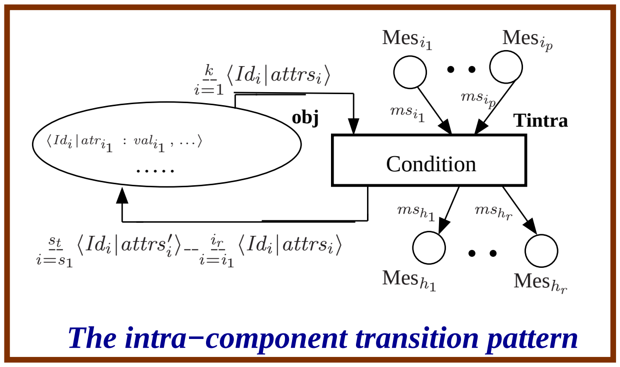
Mes$_{i_1}$
Mes$_{i_p}$
$\underset{i=1}{\overset{k}{\_\_}} \langle Id_i|attrs_i\rangle$
obj
ms_{i_1}
ms_{i_p}
Tintra
$\langle Id_i|atr_{i_1} : val_{i_1}, ...\rangle$
.....
Condition
ms_{h_1}
ms_{h_r}
$\underset{i=s_1}{\overset{s_t}{\_\_}} \langle Id_i|attrs'_i\rangle \_\_ \underset{i=i_1}{\overset{i_r}{\_\_}} \langle Id_i|attrs_i\rangle$
Mes$_{h_1}$
Mes$_{h_r}$
The intra−component transition pattern

Fig. 4. A General pattern for transitions

Tskipcal: $(CALLED, Called\_F(L, K)) \otimes$
$(Lift, \langle L|Cur\_F : K\rangle) \Rightarrow (Lift, \langle L|Cur\_F : K\rangle)$
Tgoint: $(TOGO, ToGo\_F(L, K1, S)\_\_$
$ToGo\_F(L, less(K, K1), -)) \otimes$
$(Lift, \langle L|Cur\_F : K, St : S, Dr : cl\rangle)$
$\Rightarrow ((TOGO, ToGo\_F(L, K1, S)) \otimes$
$(Lift, \langle L|Cur\_F : less(K, K1), St : S.Stop, Dr : cl\rangle)$
`if` $(S = Up \vee = Dw)$

Tgonext: $(TOGO, ToGo\_F(L, K1, D)) \otimes$
$(Lift, \langle L|Cur\_F : K, St : S, Dr : cl\rangle)$
$\Rightarrow (Lift, \langle L|Cur\_F : K1, St : S.Stop, Dr : cl\rangle)$
`if` $(D = S = Up \vee = Dw) \vee ((K1 = K \pm 1))$

These transition rewrite rules governing the behaviour of lift CO-NETS component can be concurrently applied to a given (initial) *state* marking. As we mentioned such state marking is to be described as a multiset of the form:$\underset{i}{\otimes}(p_i, M(p_i))$. Where $M(p_i)$ is the current marking of the place p_i. The corresponding inference rules of this CO-NETS rewrite theory and illustration on how to concurrently applied them can be found more in detail in [3].

D. Features Composition using Strategies in CO-NETS

In the previous section we presented how deriving rewrite rules governing different service features and how applying them to a given initial configuration. However, like all Petri nets variants we let undefined the *order* in which different transitions (rules) have to be fired (i.e transitions are randomly fireable). Although such undefined control may enhances parallelism and true concurrency, we stress its inappropriateness for features interaction. That is, we argue that imposing carefully chosen control strategies for firing transitions represents a crucial step towards decreasing undesirable interactions of service features, and also allows detecting and validating large cases of conflicting interactions (before completing such detection with some property-checking).

Considering our case study with the above rewrite rules, for instance, we have to impose that after firing the transition `Tgofar`, the transitions `Tcallint` and `Tgoint` have to be repeatedly and concurrently attempted before trying the transition `Tgonxt`. Otherwise, if we directly fire `Tgonxt` after `Tgofar` then all intermediate call or goto (from inside and outside) will be *skipped*, and thus the whole lift functioning become compromised. The same is for the transitions `Tskipcal` and `Tskipgo` which have to fired at the right time with high priority otherwise they will be interpreted as an effective call or goto (i.e. the lift travel back each time to serve such fictive calls and goto's!).

The approach we are proposing is inspired by two existing related proposals for controlling rules in rewrite logic. The first method detailed in [26], [27] puts forward a so-called message algebras inspired from process algebras operators. It adopts thus operators like `sequence` (denoted as usual by ";"), `choice` ("+"), `parallelism` ("|"), etc, on messages appearing in the left-hand sides of rewrite rules. To capture the intended meaning of message expressions based on such operators, the approach restricts rules to at most one message in the left-hand side (called "simple" MAUDE). The control itself is enforced through adequate inference rules, taking the rules and the current configuration (i.e the running program as multiset of messages and objects instances). The second approach [8] permits dropping the restriction about the form of rules, and is based on the intrinsic reflection capabilities of rewrite logic [17]. That is, to enforce a rewriting strategy using the above operators, this reflection-based approach first transforms all concerned rules to their meta-representation and then apply algebras-like meta-rules system to enforce any given strategy and then reflect it at the usual base-level.

Towards controlling CO-NETS transitions firing, we strive for benefiting as much from these two proposals while considering the specificities of CO-NETS transitions rewrite rules. More precisely, we aim to keep the simplicity of the proposed message algebras but apply it instead on transition rules (using their names as "labels") as in [8] yet without using the complexity of reflection.

We propose thus a "`Transitions_Algebra`" instead of the above message algebra. We give below the corresponding inference rules for the choice ('+') and the sequence (';') operators; the other could operators such as parallelism, loop, ect are to be captured in the same manner that we skipping as simple exercise!. As each CO-NETS transition is usually triggered by at least one message instance, the associated inference rule consists in ensuring the existence of a matching of this transition rule (via a substitution) in the current CO-NETS-state while enforcing the meaning of each operator.

obj `TRANSITIONS_Algebra` **is**
protecting CO-NETS`-State .`
op $\_+\_ \;\; \_;\_ \;\; \_;;\_ \;\; \_|\_ \;\; \_||\_ \; : \; T\_labels \; T\_labels \rightarrow T\_labels$
vars $m_1, m_2, sm_1, sm_2 \; : \; Msg$.
vars $S_l \; , \; S_r \; , \; S_{l_1} \; , \; S_{l_2} \; , \; S_{r_1} \; , \; S_{r_2} \; , \; S_h$ `:` CO-NETS`-State`
vars $p_1 \; , \; p_2$ `: Places`
vars $Trl_1 \; , \; Trl_2$ `: Transitions_Rules`
$\mathbf{Trl_1} \; + \; \mathbf{Trl_2} \; \Leftrightarrow \; (p_1, sm_1) \otimes (p_2, sm_2) \otimes S_l \Rightarrow S_r$
`with`
$Trl_1 \; : \; (p_1, m_1) \otimes L_1 \Rightarrow R_1$
$Trl_2 \; : \; (p_2, m_2) \otimes L_2 \Rightarrow R_2$
$\exists \; \sigma_1, \sigma_2 \; : \; X \rightarrow T_{s(p_i)}$ **s.t.**
$(sm_1 = \sigma_1(m_1) \; \wedge \; \sigma_1(L_1) \in S_l \; \wedge \; (p_1, sm_1) \otimes S_l \Rightarrow S_r) \vee$
$(sm_2 = \sigma_2(m_2) \; \wedge \; \sigma_2(L_2) \in S_l \; \wedge \; (p_2, sm_2) \otimes S_l \Rightarrow S_r)$

$\mathbf{Trl_1} \; ; \; \mathbf{Trl_2} \; \Leftrightarrow \; (p_1, sm_1) \otimes (p_2, sm_2) \otimes S_{l_1} \otimes S_{l_2} \Rightarrow S_{r_1} \otimes S_{r_2}$
`with`
$Trl_1 \; : \; (p_1, m_1) \otimes L_1 \Rightarrow R_1$
$Trl_2 \; : \; (p_2, m_2) \otimes L_2 \Rightarrow R_2$
$\exists \; \sigma_1, \sigma_2 \; : \; X \rightarrow T_{s(p_i)}$ **s.t.**
$(sm_1 = \sigma_1(m_1) \; \wedge \; \sigma_1(L_1) \in S_{l_1} \; \wedge \; \sigma_1(R_1) \in S_{r_1} \otimes S_h \; \wedge$
$(p_1, sm_1) \otimes S_{l_1} \Rightarrow S_{r_1} \otimes S_h) \; \wedge$
$(sm_2 = \sigma_2(m_2) \; \wedge \; \sigma_2(L_2) \in S_{l_2} \otimes S_h \; \wedge \; \sigma_2(R_2) \in S_{r_2} \; \wedge$
$(p_2, sm_2) \otimes S_{l_2} \otimes S_h \Rightarrow S_{r_2})$
endo .

Informally speaking, the choice strategy ('$+$') allows applying an eligible transition (i.e. with a matching to a part of the CO-NETS-marking or state) among at least two transition rules (in our case Trl_1 and Trl_2). For the sequence strategy (';') a two transition rules have to be sequentially applied, that is, after applying the first rule, a part of the resulting CO-NETS-marking Trl_1 (represented by the sub-state S_h) has to rewritten while applying the second transition rule.

Application to the multi-lift system. One possible logical strategy consists of repeating the following: (1) eliminate any redundant request from outside and inside, that is, first each time perform the transitions `Tskipgo` and `Tskipcal`; (2) check for the farthest requested floor from inside, that is, try performing the transition `Tgofar`. When all requested floors are just next ones (i.e. Up or Down) perform the transition `Tgonxt` ; (3) serve any (intermediate) requested floors (both from inside and outside) while travelling to the farthest floor selected from 2. That is, perform the transitions `Tintcall` and `Tgoint`; and (4) serve this final destination by performing the transition `Tgonxt`. With the above notations, this strategy corresponds to the following algebra:

$$[(\texttt{Tskipgo} \parallel \texttt{Tskipcall})\ ;\ ((\texttt{Tgofar}\ ;\ (\texttt{Tintgo} \parallel \texttt{Tintcall})\ ;\ \texttt{Tnxtgo})\ +\ (\texttt{Tnxtgo}))]^*$$

The notation $[...]^*$ perform this process repeatedly yet with the interference of any local behaviour (i.e. the application of local transitions at anytime and at need).

IV. Features Runtime Evolution in Co-nets

As we emphasized, existing FI approaches lack *run-time* manipulation of features, which prevents adjusting such features to avoid undesirable interactions and/or to timely respond to requirements features change. In this section, first we present how to incrementally extend the Co-nets framework with a Petri nets meta-level, where service functionalities can be dynamically adapted to requirements change and thus circumvent the just mentioned shortcomings in FI. We then apply this ”evolving” Co-nets to the running example.

A. A Petri nets-based Meta-level for Co-nets

The main ideas for building a meta-level from a given Co-nets component, we detailed in [4], may be intuitively summarized in the following:

1) **Meta-data for transition dynamics:** As any reflection technique, the first step consists in representing base-level entities as (meta-)data. For the Co-nets case, we should recall that each transition is composed of: (1) an identifier or label, (2) input inscriptions with their respective places (as multiset), (3) output inscriptions with their respective output places (as multiset), and (4) the transition condition. So it is intuitive to represent such transition dynamics as a *tuple* of the form:
   $\langle$`transition_id:version` $|$ `(input-)multiset, (output-)multiset, condition` $\rangle$
   With respect to the intra-component transition pattern proposed in Figure 4, such tuple takes the following precise form:
   $\langle Trl\ :\ i \mid (obj, IC_{obj}) \bigotimes_{k=i_1}^{i_p} (Mes_k, IC_k),$
   $(obj, CT_{obj}) \bigotimes_{k=j_1}^{j_q} (Mes_k, CT_k), Cond.\rangle$
2) **Dynamically manipulating such tuples:** To dynamically manipulating such transitions behaviour as tokens, we first propose to conceive a (meta-)*place* to gather their instantiated forms (i.e. with concrete places, inscriptions and conditions) as (meta-)*tokens*. Besides that, we propose three places with corresponding transitions to permit adding, removing or updating any transition behaviour as token. We denote such places with `Add-Bh`, `Chg-Bh` and `Del-bh` as the places for holding messages for adding/updating and deleting such meta-tokens with three respective transitions TADD, TCHG and TDEL for effectively adding, modifying or deleting explicit (meta-)tokens.
3) **Relating the two levels with read-arcs:** As next step in this reflection reasoning is to allow, on the one hand, *reifying* (bringing up) any Co-nets components transitions behaviour (from the base-level) to this meta-level to manipulate it, and on the other hand dynamically *reflecting* (bringing down) any transition behaviour-as-meta-tokens to a normal Co-nets transition. With the aim to keep the base-level of the Co-nets components specification unchanged, we propose just to add read-arcs from the meta-place to some transitions subject to change and evolution. For such subject-to-change transitions, we also propose to enrich their input/output inscriptions and conditions with corresponding variables through a disjunction operator (we denote by $\vee$) as shown in the right-hand side of the lower part of Figure 5. These variables with associated (input/output) places are accordingly put into a transition tuple as inscription for the read-arc relating the meta-place to such non-instantiated transition.
4) **Propagating meta-tokens to transitions behav-

iour: The propagation or the reflection consists in *selecting* from the meta-place a given meta-token—which we refer to as a *non-instantiated* and non-existing transition at the base level—and *transforming* it to a usual (instantiated) transition rule which can be used as and with the other transition rules. Given such a non-instantiated meta-rewrite rule, we can then *dynamically select* any particular tuple as a behaviour from the meta-place and derive a usual transition rule.

The general form of rewrite rules for non-instantiated transitions is as such:

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

Note the operator $\|_r$ separating the read-arc inscription from the other inscriptions allows explicitly distinguishing between the input/output arcs and this read-arc inscriptions.

Given such a non-instantiated meta-rewrite rule, we can then select any particular tuple from the meta-place and derive a usual rule. This process is captured by the following inference rule.

With the existence of the following substitutions: $\exists\sigma_i \in [T_{s(p_i)}]_{--}, .., \exists\sigma_j \in [T_{s(q_j)}]_{--}, \exists\sigma \in [T_{bool}]$

The following usual rewrite rule as the new (k*th* behaviour for the transition $t(-)$ is obtained.

$$\frac{\langle t^{nis} : k\ |[\bigotimes_{i=1}^{k}(p_i,\sigma_i(IC_i))]|,|[\bigotimes_{j=1}^{l}(q_j,\sigma_j(CT_i))]|,\sigma(TC_i)\rangle \in M(P_{meta})}{t(k) : |[\bigotimes_{i=1}^{k}(p_i,\sigma_i(IC_i))]| \Rightarrow [\bigotimes_{j=1}^{l}(q_j,\sigma_j(CT_i))]| \quad \texttt{if} \quad \sigma(TC_i)}$$

B. Dynamically manipulating the multi-lift features

With respect to the multi-lift CO-NETS components we proposed in previous sections, different lift features can now be dynamically manipulated in an incremental way, namely runtime addition of new features without resorting to stopping the running specification, the deletion of existing features, and finally the modification, that is, the update of some outdated features.

As specific illustration of such dynamic adaptivity of different features, we restrict ourselves to the following cases:

- **Stationary floors:** We would like to introduce a feature that allows for some particular lifts to travel to a 'stationary' floor when there is no call inside or from outside. However, we would like at the same time to let this stationary floor variable, for instance, depending on rush hours. This of course cannot be specified using a fixed transition, rather it should be considered as a token that can be *dynamically* updated whenever necessary. For the simple case, where the stationary floor is to be the underground (floor zero(0)), such a meta-token takes the form:
$\langle Reset : 1|(TOGO, ^{\sim} ToGo\_F(L,-,-)) \otimes (CALLED, ^{\sim} Called\_F(L,-,-)) \otimes (Lift, \langle L|Cur\_F : K, Dr : cl, Wg : 0\rangle), (Lift, \langle L|Cur\_F : 0, Dr : cl, Wg : 0\rangle), (K \neq 0) \wedge (L \in List(Lifts))\rangle$
- **Avoid unnecessary travel:** In our original CO-NETS specification we allowed canceling any request from (inside or outside) a same floor (see transitions `Tskipgo` and `Tskipcal`). Nevertheless, to completely protect the lift from (kids abuse!) unnecessary travel, we have to consider the case of requesting (from inside) for floors without being in the lift car (i.e. the *weight* in zero(0)). To do so, we have to consider the transition `Tskipgo` as an evolving one, and introduce its new behaviour as a meta-token. This behaviour takes the following form:
$\langle Tskipgo : 1|(TOGO, ToGo\_F(L, K1, -)) \otimes (Lift, \langle L|Cur\_F : K, Wg : W\rangle), (Lift, \langle L|Cur\_F : K, Wg : W\rangle), ((K1 = K) \vee (W = 0))\rangle$
- **Serving "onboard" first:** When a given lift is *nearly full*, that is its weight is for instance more that 2/3 of the `Wmax`, and is traveling far (more than next floor), it is more practical to skip intermediate calls from outside. This means that the firing of the transition `Tcallint` has now to be subject to this weight condition change. The corresponding transition tuple (as new version) takes thus the form:
$\langle Tcallint : 1|(CALLED, Called\_F(L, less(K, K1))) (TOGO, ToGo\_F(L, K1, S)) \otimes (Lift, \langle L|Cur\_F : K, St : S, Wg : W\rangle), (TOGO, ToGo\_F(L, K1, S)) \otimes (Lift, \langle L|Cur\_F : less(K, K1), St : S.Stop, Wg : W\rangle), ((S = Up) \vee (S = Dw)) \wedge (W < 2/3\, Wmx)\rangle$

All these features are illustrated in Figure 6 with IC_{var}, CT_{var} and TC_{var} as appropriate variables for capturing adaptive input inscriptions, output inscriptions

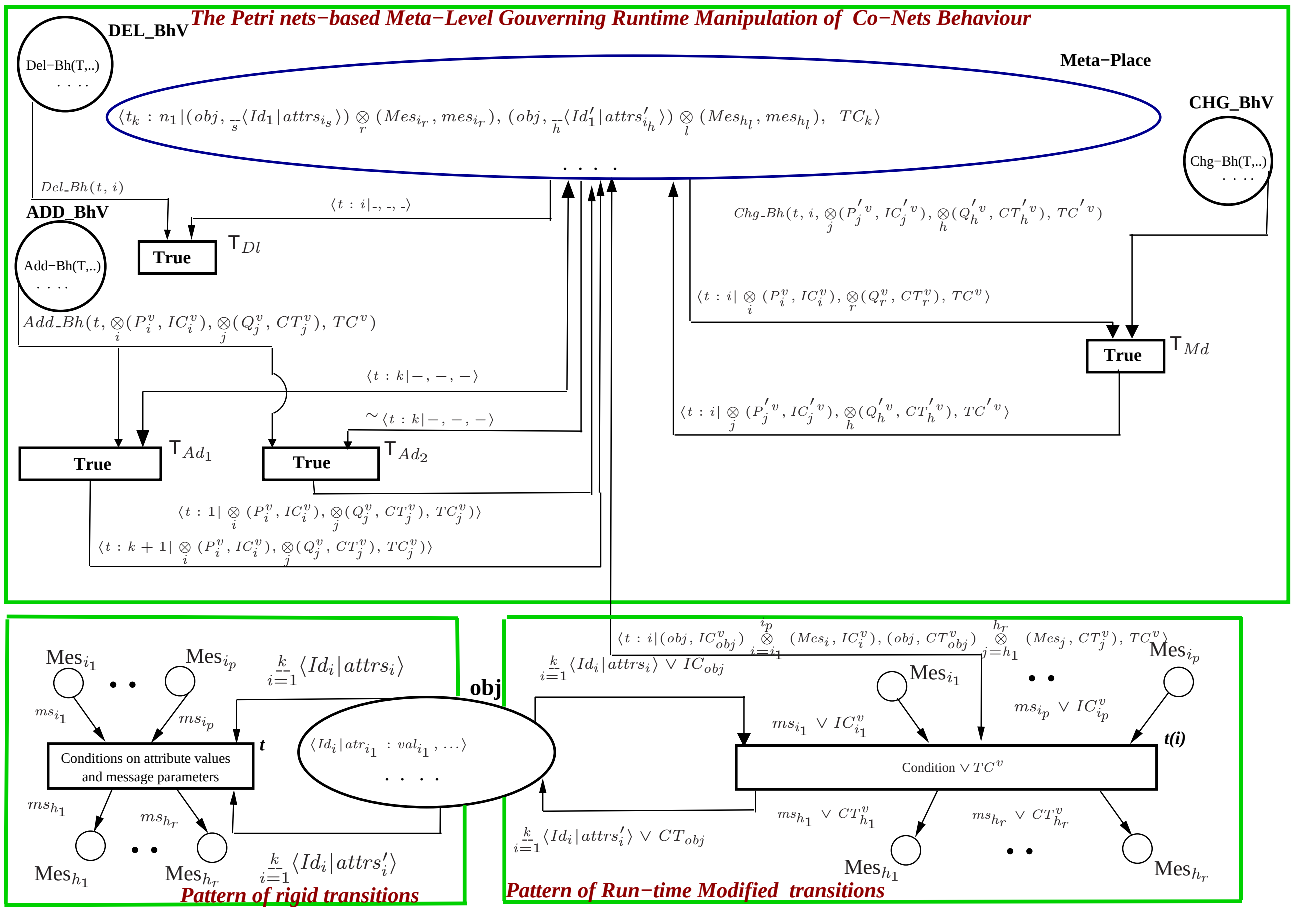
The Petri nets−based Meta−Level Gouverning Runtime Manipulation of Co−Nets Behaviour
DEL_BhV
Del−Bh(T,..)
Meta−Place
CHG_BhV
Chg−Bh(T,..)
ADD_BhV
Add−Bh(T,..)
True
T_Dl
T_Md
T_Ad1
T_Ad2
obj
t
t(i)
Conditions on attribute values and message parameters
Condition ∨ TC^v
Pattern of rigid transitions
Pattern of Run−time Modified transitions

Fig. 5. The general pattern for handling dynamic behaviour in CO-NETS

and conditions respectively. Note that we are concentrating only on evolving transitions, with all other unchanged transitions (from Figure 2) being skipped.

V. CONCLUSIONS

We addressed the challenging yet very practical problems of formally specifying animating, validating, composing and dynamically evolving interacting features in distributed dynamic service-driven environment. The proposed approach is based on a tailored integration of component concepts with high-level Petri nets endowed with an adaptive meta-level and interpreted in true-concurrency rewrite logic. The approach has been illustrated using a variant of a multi-lift system with an informal description using a profiled UML class-diagrams.

We are developing a tool supporting the proposed framework and this feature-oriented approach. Further for properties verification purpose, we are recapitulating from previous work on relating the semantics of this framework namely rewrite logic with Lamport's temporal logic of actions TLA [15], [2]. Such verification phase is crucial for logically detecting different inconsistencies and unwished interactions of different features.

REFERENCES

[1] *Business Process Execution Language for Web Services*. IBM, 2003. version 1.1.

[2] N. Aoumeur and G. Saake. Concurrent Object Systems Modelling and Verification on the Basis of Maude and TLA+. In M. Wirsing, M. Gogolla, H. Kreowski, T. Nipkow, and W. Reif., editors, *Proc. of 1st Workshop on Rigorous development of software-intensive systems, Berlin, Germany*, pages 43–56, 2000.

[3] N. Aoumeur and G. Saake. A Component-Based Petri Net Model for Specifying and Validating Cooperative Information Systems. *Data and Knowledge Engineering*, 42(2):143–187, August 2002.

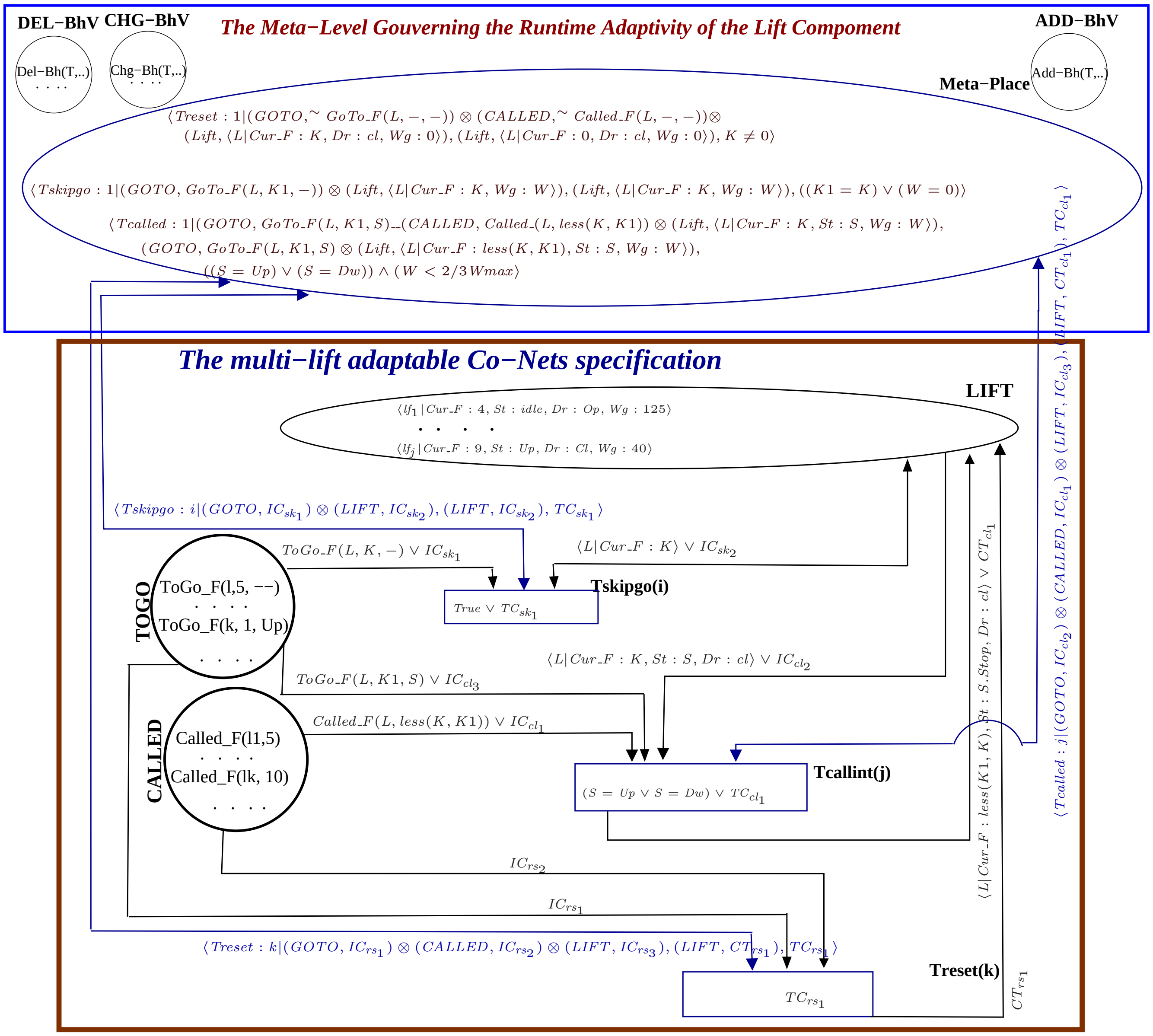
DEL−BhV
CHG−BhV
The Meta−Level Gouverning the Runtime Adaptivity of the Lift Compoment
ADD−BhV
Del−Bh(T,..)
Chg−Bh(T,..)
Add−Bh(T,..)
Meta−Place
The multi−lift adaptable Co−Nets specification
LIFT
TOGO
CALLED
Tskipgo(i)
Tcallint(j)
Treset(k)

Fig. 6. Dynamic manipulation of the lift system features

[4] N. Aoumeur and G. Saake. Dynamically Evolving Concurrent Information Systems :A Component-Based Petri Net Proposal. *Data and Knowledge Engineering*, 50(2):117–173, 2004.

[5] E. Battiston, A. Chizzone, and F. de Cindio. Inheritance and Concurrency in CLOWN. In *Proc. of the "Application and Theory Petri Nets" Workshop on Object-Oriented Programming and Models of Concurrency*, 1995.

[6] O. Biberstein, D. Buchs, and N. Guelfi. CO-OPN/2: A Concurrent Object-Oriented Formalism. In *Proc. of Second IFIP Conf. on Formal Methods for Open Object-Based Distributed Systems(FMOODS)*, pages 57–72. Chapman and Hall, March 1997.

[7] G. Booch, I. Jacobson, and J. Rumbaugh, editors. *Unified Modeling Language, Notation Guide, Version 1.0*. Addison-Wesley, 1998.

[8] 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.

[9] Y. Fu, Z. Dong, and H. He. An Approach to Web Services Oriented Modeling and Validation. In *of the 28th ICSE workshop on Service Oriented Software Engineering (SOSE2006)*. ACM Press, 2006.

[10] J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.P. Jouannaud. Introducing OBJ. Technical Report SRI-CSL-92-03, Computer Science Laboratory, SRI International, 1992.

[11] J. Padberg H. Ehrig and L. Ribeiro. Algebraic High-Level Nets - Petri Nets Revisited. In *Proc. Joint ADT-COMPASS Workshop*, volume 785 of *Lecture Notes in Computer Science*, pages 188–

206. Springer, 1994.

[12] R. Heckel and L. Mariani. Automatic Conformance Testing of Web Services. In *Proceedings FASE 2005*, volume 3442, pages 34–48. lncs, 2005.

[13] M. Heissel. Detecting Features Interactions—A Heuristic. In *Proc. of the 1st FIREworks Worshop*, pages 30–48, Magdeburg, Germany, 1998.

[14] K. Jensen. Coloured Petri Nets: Basic Concepts, Analysis Methods and practical Use - Volume 1 : Basic Concepts. *EATCS Monographs in Computer Science*, 26, 1992.

[15] L. Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[16] A. Martens. Analyzing Web Service Based Business Processes. In *Proceedings FASE 2005*, volume 3442, pages 19–33. lncs, 2005.

[17] 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*, volume 4 of *Electronic Notes in Theoretical Computer Science*, pages 189–224, 1996.

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

[19] J. Meseguer. Solving the Inheritance Anomaly in Concurrent Object-Oriented Programming. In *ECOOP'93 - Object-Oriented Programming*, volume 707 of *Lecture Notes in Computer Science*, pages 220–246. Springer, 1993.

[20] J. Rao, P. Kuengas, and M. Matskin. Comnposition of Semantic Web Services Using Linear Logic Theorem Proving. *Information Systems*, 31:340–360, 2006.

[21] S. Reiff-Marganiec and M.D. Ryan. *Feature Interactions in Telecommunications and Software Systems*. IOS Press—ISBN 1-58603-524-X, 2005.

[22] W. Reisig. Petri Nets and Abstract Data Types. *Theoretical Computer Science*, 80:1–30, 1991.

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

[24] M. Solanki, A. Cau, and H. Zedan. Introducing Compositionality in Web Service Descriptions. In *Proceedings of the International Conference on World Wide Web*. IEEE Computer Society Press, 2004.

[25] Ruediger Valk. Concurrency in communicating object petri nets. In G. Agha, F. de Cindio, and G. Rozenberg, editors, *Concurrent Object Oriented Petri Nets*, volume 2001 of *Lecture Notes in Computer Science*, pages 164–165. Springer, 2001.

[26] M. Wirsing and A. Knapp. A Formal Approach to Object-Oriented Software Engineering. *ENTCS*, 4, 1996.

[27] M. Wirsing, F. Nickel, and U. Lechner. Concurrent Object-Oriented Specification in Spectrum. In Y. Inagaki, editor, *Workshop on Algebraic and Object-Oriented Approaches to Software Science, Nagoya/Japan*, Electronic Notes in Theoretical Computer Science, pages 39–70. Nagoya University, 1995.

[28] X. Yi and K.J. Kochut. A CPNets-based Framework for Design and Analysis for Service Oriented Distributed Systems. *ACM Transaction on Internet Technology*, ??, 2005.

APPENDIX

Templates for service states and signatures

```
obj Service-State is
  sort AId .
  subsort OId < Value .
  subsort Attr < Attrs .
  subsort Id_Attrs < object .
  subsort Local_attrs External_attrs < Id_Attrs .
  protecting Value OId AId .
  op _ : _ : AId Value → Attribute .
  op _,_ : Attr Attr → Attrs [ass. com. nil] .
  op ⟨_ | _⟩ : OId Attrs → Id_Attrs .
endo.

obj Template-Signature is
  protecting service-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_Messs .
  subsort Mes$_{e1}$, Mes$_{e2}$,..,Mes$_{ee}$ < Export_Messs .
  subsort Mes$_{i1}$, Mes$_{i2}$,..,Mes$_{ii}$ < Import_Messs .
  subsort local-attrs obs-attrs < Id-attrs .
  (* local attributes *)
    op ⟨_ | $at_1$ : _,..,$at_k$ : _⟩ : OId s-atr$_1$ ...
       s-atr$_k$ → Local-Attrs.
  (* observed attributes *)
    op ⟨_ | $bs_1$ : ,..,$bs_{k'}$ : _⟩ : OId s-atbs$_1$ ...
       s-atbs$_{k'}$ → obs-Attrs.
  (* local messages *)
    op ms$_{l1}$: OId ...s-arg$_{l1,1}$...s-arg$_{l1,l1}$
       → Mes$_{l1}$ . ...
  (* export messages *)
    op ms$_{e1}$: OId ...OId ...s-arg$_{e1,1}$ ...s-arg$_{e1,e1}$
       → Mes$_{e1}$ . ...
  (* import messages *)
    op ms$_{i1}$:OId ...OId ...s-arg$_{i1,1}$ ...s-arg$_{i1,i1}$
       → Mes$_{ip}$ . ...
endo.
```