Validating and Dynamically Adapting and Composing Features in Concurrent Product-Lines Applications

Nasreddine Aoumeur, Kamel Barkaoui, Gunter Saake

Validating and Dynamically Adapting and Composing Features in Concurrent Product-Lines Applications

Conference Paper · August 2009

DOI: 10.1109/ECBS.2009.48 · Source: DBLP

3 authors, including:

Kamel Barkaoui
Conservatoire National des Arts et Métiers
300 PUBLICATIONS **2,164** CITATIONS

SEE PROFILE

Gunter Saake
Otto-von-Guericke-Universität Magdeburg
645 PUBLICATIONS **6,999** CITATIONS

SEE PROFILE

Validating and Dynamically Adapting and Composing Features in Concurrent Product-Lines Applications

Nasreddine Aoumeur Kamel Barkaoui[1] Gunter Saake
ITI, FIN, Otto-von-Guericke-Universität Magdeburg
Postfach 4120, D–39016 Magdeburg, GERMANY
[1]Laboratoire CEDRIC, CNAM, 292 Saint Martin, 75003 Paris - FRANCE E-mail: barkaoui@cnam.fr
E-mail: {aoumeur|saake}@iti.cs.uni-magdeburg.de

Abstract

With the pressing in-time-market towards customized services, software product lines (SPL) are increasingly characterizing most of software landscape. SPL are mainly structured through offered features, where consistent *composition and* dynamic *variability are the driving forces. We contribute to these two challenging problems when distribution and correctness are at stake. First, we soundly specify and validate any feature-oriented requirements using a component-based Petri nets framework referred to as* CO-NETS. *For rapid-prototyping, we semantically interpret in true-concurrent rewriting logic. For consistently composing features, a flexible feature-algebra is proposed. Finally, for runtime adaptability and integration of features, we leverage* CO-NETS *with an explicit aspectual-level, where features can be dynamically (un)woven on running components. The approach is thoroughly explained using a feature-intensive multi-lift system.*

Keywords*: Software product lines, Feature modelling and evolution, Component-based Petri nets, rewriting logic and* MAUDE.

1 Introduction

An SPL is a set of software-intensive systems sharing a common, managed set of features that satisfies the specific needs of a particular market segment or mission. SPL applications are thus developed from a common set of core assets in a prescribed way [10, 14, 7]. A feature is an increment in program functionality. In the Feature Oriented Domain Analysis (FODA) [5], a feature is defined as a prominent or distinctive user-visible aspect, quality or characteristic of a system. It defines a logical unit of behavior that is specified by a set of functional and quality requirements.

Several notations and formalisms such as Use Cases and scenarios are now very popular for single products development. In the SPL context, most works [5, 10] extend UML Use Cases with variability mechanisms to document PL requirements. features interactions have been also tackled using using scenarios such as UML sequence diagrams [6]. Statecharts [8] are often used for a more detailed design, as they are closer to the implementation. Features Interaction [16] has been recently coined as a new research field in software-engineering for tackling any conflict and contradiction occurring when putting features together.

Nevertheless, due to the increasing networking and volatility of applications such as the emerging service-driven systems [15], existing approaches to features interaction are becoming transcended. Among the exhibited shortcomings, we aim tackling in this contribution we point out to the followings.

- The expressiveness of the modeling framework. Indeed, most of proposals to Features Interac-

tion are based on temporal and process languages [16], known for their limited structuring.

- The capabilities of exhibiting *concurrent* and *distributed* behavior require to be promoted, as premise for service-driven applications.

- The intrinsic ability of *dynamically* adapting features. Indeed, to stay competitive, applications are adapting quickly to market changes and to satisfying very demanding customers.

We propose thus to first rigorously specifying and validating service features. Complex features are to be combined from basic ones into strategies to reflect realistic behaviour. On the other hand, capitalizing on 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 conceptual model we are proposing, referred to as CO-NETS, is based on high-level Petri nets [17], with three main distinguished capabilities:

- **Inter-component interactions**: Besides coping with intra-component computation, the model allows behaviorally interacting different components composing a complex application. The features are thus externalized, with make them very sensitive and adequate for changes.

- **Prototyping and validation**: The model is semantically governing by distributed rewriting-logic [13] and its MAUDE language [4]. Using current MAUDE implementation, prototypes can be derived using symbolic concurrent rewriting techniques. This with the usual graphical animation permit validating the system against its requirements.

- **Runtime adaptability**: As we demonstrate in the paper, recapitulating on reflection [3] and aspect-oriented techniques [11], we present how to leverage the model to cope with dynamic-adaptability in transparent and separate manner. In this manner, features are dynamically woven on running components.

The rest of this paper is organized as follows. The next section presents the informal running example using UML-class diagrams. In the third section, we present how to model, compose and validate features with CO-NETS. In the fourth section, we address the problem dynamically adapting features using an aspectual-level over CO-NETS components. We conclude this paper by some remarks.

2 The Multi-lift System: Informal Description

To illustrate our approach to features modelling, validation and dynamic evolution, we adopt a simplified version of a multi-lift system. This application has been used in several approaches to FI (e.g. [18, 9, 16]). Nevertheless, they consider only a single lift and the interactions are checked only statically, that is at design-time.

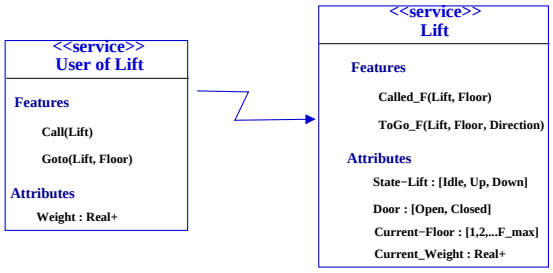

Figure 1. The Lift and User services as profile UML Classes

As depicted in Figure 1, each lift is characterized by: the *Identity*, *Current floor* (shortly `Cur_Floor`) and *Lift-state* expressing whether the lift is *idle*, *going up* or *going down*. To keep track of directions while serving intermediate floors, we add states such "`Stop`" as suffixes for other states. For instance, "`Up.Stop`" implies that the lift is going up and is intermediately stopping. The *Door-state* (shortly `Door`), that could

be either open (`Open`) or closed (`Closed`). Finally, *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) through `Called(LiftId, Floor)`. This can be initiated through `Call(LiftId)`.
- We denote a travel to a given floor by `ToGo(LiftId, Floor, Direction)`. It is initiated (by the user) through `GoTo(LiftId, Floor)`.
- Internal features to any lift include: Opening/closing (using sensors) and weight-update.

3 Features Specification 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. In CO-NETS, we define them 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; $at_1, .., at_k$ are the local attributes whereas $bs_1, ..., bs_{k'}$ represent the observed ones.
- Similarly, we distinguish between imported / exported and internal messages.

3.1 Lift-services structure modelling

First, we precisely define the data required for expressed a given lift component. These data types include the lift-states (e.g. idle, up, dw), the floors, the door-date and the max. allowed for the weight. Algebraically, these basic required data takes this form:

```
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.
```

For the lift itself is to be precisely defined with respect to the CO-NETS structure as follows. First, its state is to be defined with its attributes. Then, the messages to be received as features are defined.

```
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.
```

3.2 Features behaviour modelling in CO-NETS

The CO-NETS model is incrementally constructed from its structure as follows.. The net Places are constructed by associating with each message generator one 'message' place. With each state sort we also associate one 'state' place. The net transitions, which may include conditions, reflect the intended effect of each feature on service states.

3.2.1 The Lift-service behaviour

Figure 2 depicts the corresponding user multi-lifts CO-NETS model. This behavior is incrementally conceived 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. The transition `Tcalled`

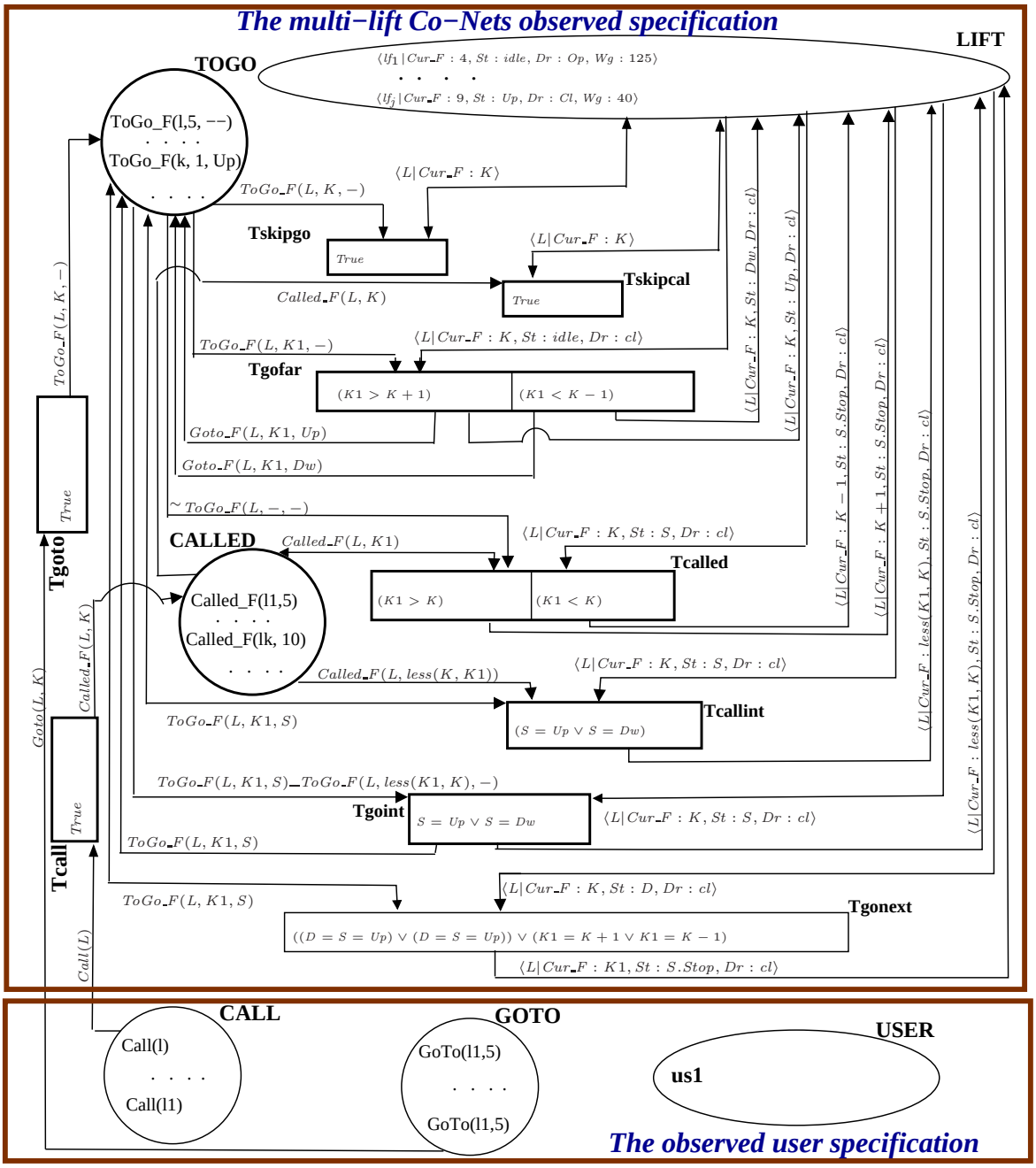

Figure 2. The Multi-Lift-User Components observed specification

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 $^{\sim}$ 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`.

3.3 Features validation with MAUDE

The main ideas of interpreting CO-NETS in rewrite-logic are as follows. We bind each place-marking mt with its place p using the pair (p, mt). Tokens within mt are gathered as a multiset with the union operator __. CO-NETS states are multisets over different the pairs (p_i, mt_i), using a union operator denoted $\otimes$. 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.

Example: By applying these guidelines, the CO-NETS lift transitions are governed with the following rules.

- **[Tskipgo]**: $(TOGO, ToGo\_F(L, K, \_))\otimes$
  $(Lift, \langle L|Cur\_F : K\rangle) \Rightarrow (Lift, \langle L|Cur\_F : K\rangle)$
- **[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 are concurrently applied to any given (initial) *state* marking. 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 [1].

3.4 Features Composition using Strategies

We argue that imposing careful-control strategies for firing transitions represents a crucial step towards decreasing undesirable interactions of service features. Further, it 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*.

We thus adopt operators like `sequence` (";"), `choice` ("+"), `parallelism` ("|") on transitions. We give below, for illustration, the corresponding inference rules for imposing a choice ('+') while performing transitions. 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).

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`
$\mathbf{Trl_1} + \mathbf{Trl_2} \Leftrightarrow (p_1, sm_1) \otimes (p_2, sm_2) \otimes S_l \Rightarrow S_r$

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

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.**
$(sm_1 = \sigma_1(m_1) \ \wedge \ \sigma_1(L_1) \in S_l \ \wedge \ (p_1, sm_1) \otimes S_l \Rightarrow S_r) \vee$
$(sm_2 = \sigma_2(m_2) \ \wedge \ \sigma_2(L_2) \in S_l \ \wedge \ (p_2, sm_2) \otimes S_l \Rightarrow S_r)$
endo .

3.4.1 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).

4 Features Runtime-Adaptability

As we emphasized, existing FI approaches lack *runtime* manipulation of features. This prevents adjusting such features to avoid undesirable interactions and/or to timely respond to requirements features change.

The main ideas for building a meta-level from a given CO-NETS component, may be intuitively summarized in the following:

Meta-tokens as transition behavior: As any CO-NETS transition is composed of an label, input/output arc inscriptions with corresponding input/output places, and a condition inscription, we first propose to *gather* them as a tuple :

$\langle$`trans_id: version | in-inscript., out-inscript., cond.` $\rangle$

Where `version` as natural number captures *different* behavior 'versions'. With respect to the inter-component transition general pattern depicted in Figure **??**, this tuple takes a more precise form:

$$\langle t : i\,|(obj, IC_{obj}) \overset{i_p}{\underset{p=i_1}{\otimes}} (Mes_p, IC_p), (obj, CT_{obj}) \overset{h_r}{\underset{q=h_1}{\otimes}} (Mes_q, IC_q), TC(t)\rangle$$

Aspectual-level for Meta-tokens: To allow manipulating—namely modifying, adding and/or deleting—such tuples (i.e. transitions' behavior), we propose an appealing Petri-net-based proposal that consists in: (1) gathering such tuples into a corresponding place that we refer to as a *meta-place*; (2) associating with this meta-place three main message operations—namely addition of new behavior, modification of an existing behavior, and deletion of a given behavior; and (3) as for usual CO-NETS components conceive for each of these three message types three places and three respective `meta-transitions` for effectively and concurrently adapting any meta-transition as tuple.

Relating the two levels with read-arcs: Once building such aspectual-level, to dynamically manipulating any transition dynamics the next important steps are twofold. First, we slightly enrich (selected) CO-NETS component transitions by just *adding* (meta-) variables (we denote by $IC\_$, $CT\_$, $TC\_$) using a disjunction operator (e.g $\vee$) to each of their input/output and condition inscriptions. In term of aspect-oriented concepts, these variables play the *jointpoints* for dynamically capturing the advices [**?**]. Transitions with these (meta-)variables are referred to as *evolving* ones. Secondly, in order to permit *weaving* any new behavior (as meta-token) on the component-level "hooked" transitions, we propose to *relate* through *read-arcs* the meta-place with such respective non-instantiated transition.

Weaving meta-tokens as usual transitions: The dynamic weaving consists in *selecting* from the meta-place a given meta-token as *advice* and *transforming* it to a usual (instantiated) transition rule. Given such a non-instantiated meta-rewrite rule, we can then *dynamically select* any particular tuple-as-behavior from the meta-place and derive a usual transition rule. This process is captured by the following inference rule.

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

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

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

4.1 Runtime adaptability of the multi-lift features

Different lift features can now be dynamically manipulated in an incremental way. As specific illustration of such dynamic adaptivity of different features, we restrict ourselves to the following cases:

- **Stationary floors:** We aim dynamically bringing some specific lifts to travel to a 'stationary' floor. For instance, at rush-time which may vary depending on the context. The transition-as-tuple that captures this is as follows.:
  $\langle Reset : 1 | (TOGO, \tilde{} \; ToGo\_F(L,-,-)) \otimes (CALLED, \tilde{} \; Called\_F(L,-,-)) \otimes (Lift, \langle L | Cur\_F : K, Dr : cl, Wg : 0\rangle), (Lift, \langle L | Cur\_F : 0, Dr : cl, Wg : 0\rangle), (K \neq 0) \wedge (L \in List(Lifts))\rangle$

- **Avoid unnecessary travel:** In our original CO-NETS specification we allowed canceling any request from a same floor (see transitions `Tskipgo` and `Tskipcal`). Nevertheless, to completely protect the lift from (kids abuse!) unnecessary travel, we have to further 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 when the weight is zero. 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*, for instance its weight is more than `2 / 3 Wmax`, it is more practical to skip intermediate calls. The transition `Tcallint` should then be adapted to.
  $\langle Tcallint : 1 | (CALLED, Called\_F(L, less(K, K1))) (TOGO, ToGo\_F(L, K1, S)) \otimes (Lift, \langle L | Cur\_F : K, St : S, Wg : W\rangle), (TOGO, ToGo\_F(L, K1, S)) \otimes (Lift, \langle L | Cur\_F : less(K, K1), St : S.Stop, Wg : W\rangle), ((S = Up) \vee (S = Dw)) \wedge (W < 2/3\, Wmx)\rangle$

All these features are illustrated in Figure 3 with IC_{var}, CT_{var} and TC_{var} as appropriate variables for capturing adaptive input inscriptions, output inscriptions and conditions respectively.

5 Conclusions

As product-line applications become largely dominating, challenging problems such as dynamic variability and features interaction require special emphasis. We proposed therefore an approach for formally specifying, validating, composing and dynamically evolving features in distributed dynamic service-driven environment. The proposed approach is based on a tailored integration of component concepts with high-level Petri nets, we endowed with an adaptive aspectual-level. The approach governed by true-concurrency rewrite logic, which permits symbolic validation besides animation. A variant of a multi-lift system was taken as proof-of-concept. Software tools supporting are been implemented to automate the approach. For formal verification of features interaction, we are recapitulating on our previous [2], that allows for shifting from the MAUDE language to Lamport's temporal logic of actions TLA [12]. Such verification phase is crucial for logically detecting inconsistencies non-detected unwished interactions of different features.

References

[1] 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.

[2] N. Aoumeur and G. Saake. Modelling and Certifying Concurrent Systems: a MAUDE-TLA Driven Architectural Approach. In *In Proc. of of 5th International Conference on Information Technology : New Generations (ITNG'08)*. IEEE CS, 2008.

[3] W. Cazzola, R. Stroud, and F. Tisato, editors. *Reflection and Software Engineering*. Lecture Notes in Computer Science Vol. 1826, Springer, 1996.

[4] M. Clavel, F. Duran, S. Eker, P. Lincoln, N. Marti-Oliet, J. Meseguer, and C.L: Talcott. All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic. *Lecture Notes in Computer Science (springer)*, 4350, 2007.

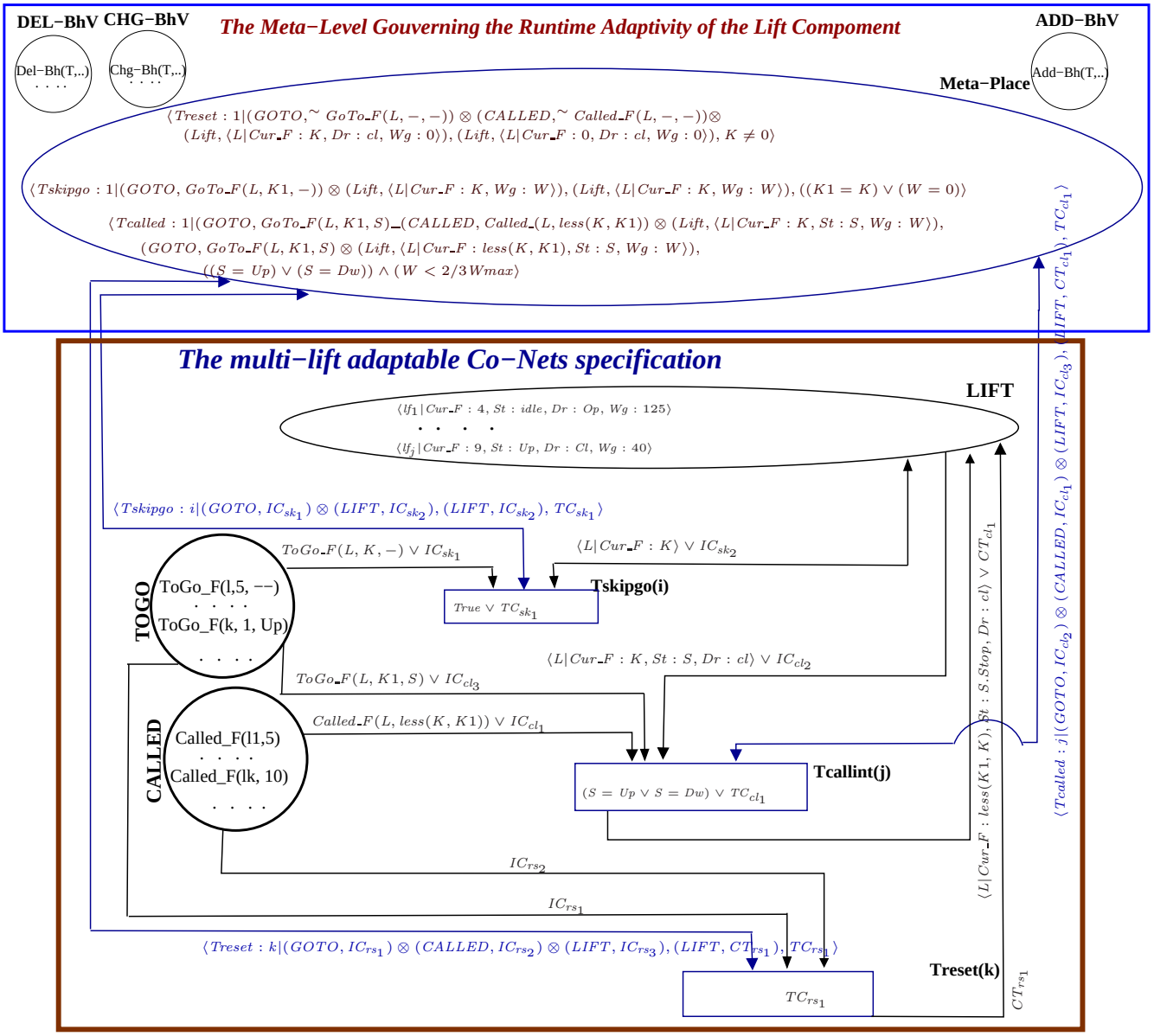

Figure 3. Dynamic manipulation of the lift system features

[5] P. Clements and L. Northrop. *Software Product Lines: Patterns and Practice*. Reading, MA: Addison Wesley, 2001.

[6] H. Gomaa and M Saleh. Feature-driven dynamic customization of software product lines. In *In Proc. Reuse of Off-the-Shelf Components*, volume 4039 of *Lecture Notes in Computer Science*, pages 58–72, 2006.

[7] G. Halmans and K. Pohl. Communicating the Variability of a Software-product Family. *Software System Modelling*, 3:15–36, 2003.

[8] D. Harel. On Visual Formalisms. *Communication of the ACM*, 31(5):514–530, 1988.

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

[10] K. Kang, J. Lee, and P. Donohoe. Feature-oriented product line engineering. *IEEE Software*, 19(4):58–65, 2002.

[11] G. et al. Kiczales. An Overview of AspectJ. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'01)*, pages 327–353. LNCS 2072, 2001.

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

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

[14] M. Mezini and K. Ostermann. Variability Management with Feature-Oriented Programming and Aspects. In *In Proc. SIGSOFT04/FSE12*, 2004.

[15] M.P. Papazoglou. *Web Service: Principles and Technology*. Prentice-Hall, Englewood Cliffs, 2007.

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

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

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