A Commutative Model Composition Operator to Support Software Adaptation

Sébastien Mosser, Mireille Blay-Fornarino, Laurence Duchien

A Commutative Model Composition Operator to Support Software Adaptation*

Sébastien Mosser[1], Mireille Blay–Fornarino[2], and Laurence Duchien[3]

[1] SINTEF IKT, Oslo, Norway, sebastien.mosser@sintef.no
[2] I3S – UMR CNRS 7271 (formerly 6070), University Nice Sophia-Antipolis, France, blay@polytech.unice.fr
[3] INRIA Lille–Nord Europe, LIFL – UMR CNRS 8022, University of Lille 1, France, laurence.duchien@inria.fr

Abstract. The *adaptive software* paradigm supports the definition of software systems that are continuously adapted at run-time. An adaptation activates multiple features in the system, according to the current execution context (*e.g.*, CPU consumption, available bandwidth). However, the underlying approaches used to implement adaptation are ordered, *i.e.*, the order in which a set of features are turned on or off matters. Assuming feature definition as etched in stone, the identification of the *right* sequence is a difficult and time–consuming problem. We propose here a composition operator that intrinsically supports the commutativity of adaptations. Using this operator, one can minimize the number of ordered compositions in a system. It relies on an action–based approach, as this representation can support preexisting composition operators as well as our contribution in an uniform way. This approach is validated on the Service–Oriented Architecture domain, and is implemented using first–order logic.

1 Introduction

The *"adaptability"* of a software is defined through its capability to react to changes and consequently to adapt itself to new environments [24]. Adaptation is now considered as a first–class problem [19], and software must be developed with the ability of being adapted during their whole life–cycle, to properly support the emergence of new technologies and the obsolescence of legacy ones. Adaptation mechanisms strongly rely on composition operators to support the introduction (or removal) of new features inside adaptive systems [15]. For example, the detection of a sudden drop in network bandwidth turns on a `cache` feature, and thus triggers the composition of `cache` artifacts (*i.e.*, model elements)

* This work is partially funded by the EU Commission through the REMICS project (contract number 257793), the SINTEF strategic project MODERATES, the French Ministry of Higher Education and Research, Nord–Pas de Calais Regional Council and FEDER through the Contrat de Projets Etat Region Campus Intelligence Ambiante (CPER–CIA) 2007-2013.

with the existing system. Existing approaches used to support such compositions rely on order–dependent operators, *e.g.*, aspect weaving [12] or functional composition [1]. Thus, the order in which features are turned on or off matters.

This order–dependency triggers several issues in the context of adaptive systems, as the designer has to explicitly control this order. The model elements associated to a feature F are composed with the existing system s as soon as the adaptation engine identifies a situation that requires F to be present in s. An immediate problem is the adaptation of unforeseen elements introduced by other features which lead to unexpected results (so-called *fragile point-cut* problem in the aspect-oriented literature [9]). Thus, the implementation of such feature assets is difficult: it must take into account the implementation of all the other feature assets to be sure that its composition produces the expected system.

In this paper, **we propose a new composition operator (called *parallel*, and denoted as $||$) that allows designers to minimise the number of ordered compositions** (and the associated issues, *e.g.*, non–deterministic result if two compositions cannot commute). Using this operator, it is possible to reify that several features are turned on independently of each others, ensuring commutativity at the composition level, by construction. Such a property helps to tame the complexity of feature definition, guarantees the determinism of the computed result and also ensures the consistency of the composed system, whatever the order of composition used at the implementation level is.

2 Motivations & Challenges

Motivations. The starting point of this study is the modelling of a *Car Crash Crisis Management System* (CCCMS), started in 2010. This case study was designed as a prototypical usage of aspect–oriented modelling techniques [13], involving multiple concerns that had to be composed in a non–trivial way with respect to the requirements document. During the elaboration of our response to this case study [21], we encountered several situation where multiple and different concerns had to be composed on the same element in the original model. Actually, this situation happened 40 times in this case study, and up to 5 concerns were composed on these *shared join points* (SJP). Thus, up to $5! = 120$ combinations can be used if we consider these compositions as sequential. More critically, these sequences do not lead to the same result, as some of them cannot commute safely! The designer has to identify which order has to be used for each SJP.

The second step that triggers our research of a new operator to support composition is the study of dynamic adaptation in the context of business processes. Where the models handled by the CCCMS were "simply" design models (*i.e.*, static), we describe in [22] a process used to support the dynamic adaptation and un-adaptation of running business processes. According to a "*models@run.time*" point of view, the adaptation of a running system is assimilated to the composition of new model elements with the model associated to the running system, and the propagation of the adapted model at the run-time level. But contrarily to the CCCMS, in this case, there is no human-in-the-loop to control the

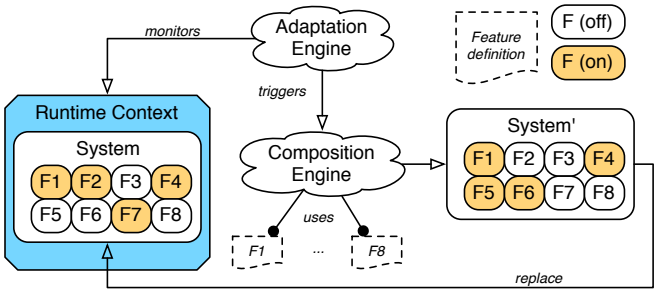


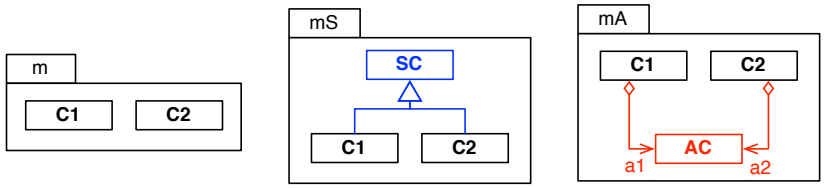
Fig. 1. Intrinsic relation between *adaptation* and *composition*

order of compositions. Based on *Complex–Event Processing* (CEP) techniques, the adaptation engine automatically triggers the needed composition, without any human intervention, as depicted in FIG. 1. We consider here a system s as the result of the composition of a set of features (here $s = \{F_1, F_2, F_4, F_7\}$). The adaptation engine monitors at run-time the execution context, and according to changes in this context, identify the new set of feature needed in the system ($s' = \{F_1, F_4, F_5, F_6\}$). It triggers the composition engine to properly compose all these features in order to build the adapted system s'. Then, the old system is replaced by the newly composed one, and the adaptation loop continue. As a consequence, the potential non-determinism of the composition process is a critical issue. The adaptation engine identifies a set of features needed in the system (*e.g.*, a cache, a local database, a low-energy consumption wifi driver) according to the current context, and if the composed system depends on the way these features are composed (*i.e.*, their order), the result of the adaptation process in not predictable. As a matter of fact, additional knowledge has to be *a-priori* stored in the adaptation engine to patch the composition directive generated by the CEP engine. This knowledge introduces ordering constraints needed to enact a correct sequence of compositions.

Running example. To illustrate our proposition, and for the sake of concision, we restrict the problem to its essence and use a simple model m to represent the system to be adapted. The associated class–diagram is depicted in FIG. 2(a). This model initially contains two classes C_1 and C_2. We also consider the two following adaptations:

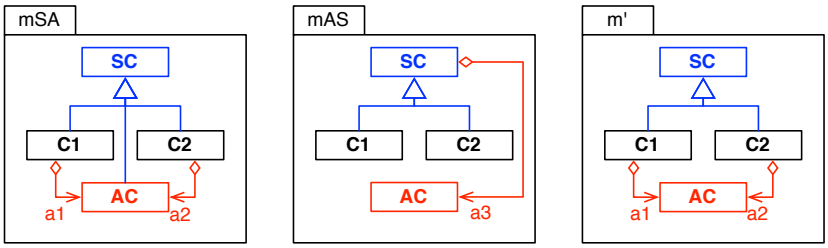
- S: This adaptation introduces a class SC in the given model, and adds an inheritance relation between all the top-level[4] classes and SC. It is a simplification of the modifications needed to introduce an *Observable* pattern into a model.
- A: This adaptation introduces a class AC in the given model, and adds an aggregation relation between all the top-level classes and AC. This adaptation

[4] A *top–level* class is defined here as a class that does not inherits from another one, *i.e.*, at the top-level of the inheritance hierarchy.



(a) m (b) $m_S = S(m)$ (c) $m_A = A(m)$

Fig. 2. Feature composition: $F(\mu)$



(a) $m_{SA} = S(A(m))$ (b) $m_{AS} = A(S(m))$ (c) $m' = ?$

Fig. 3. Sequential composition ($S \bullet A \neq A \bullet S$)

can be used to add a persistence manager at the higher level of abstraction and then supports instance persistence through polymorphism.

Such adaptations represent *de facto* new features to be introduced (*i.e.*, composed) in the model. Batory *et al.* [2] use modern mathematics to model features and their introduction: *(i)* programs are constants, and *(ii)* features are functions that produce a program when applied to a program. Thus, we consider here a program tantamount a model, and we denote as $F(\mu)$ the fact that the model elements associated to the feature F are composed with the model μ (*i.e.*, F is a model transformation). We depict in Fig. 2(b) and Fig. 2(c) the two previously described features, separately composed with m.

According to this representation, the explicit ordering of feature introduction is modelled through functional composition: $(F \bullet G)(\mu) = F(G(\mu))$. In sequential composition, the commutativity of features depends on their internal definitions. As A and S both *(i)* modify all the available top-level classes and *(ii)* introduce a new one, their sequential composition cannot commute, as represented in Fig. 3(a) and Fig. 3(b). This issue highlights the fact that $\bullet$ is not commutative by essence: the order of the composition impacts the obtained model. As a consequence, this compositional model cannot be used to produce the model depicted in Fig. 3(c) without changing the implementation of A or S.

Challenges. An obvious solution is to consider that the features should not use quantified definitions (*i.e.*, avoid constructions like "for all top level classes do ...") but instantiated definitions instead (*i.e.*, use only constructions like "for

C_1 and C_2, do ..."). Unfortunately, this approach scale with difficulty (in the CCCMS case study, a feature had to be applied at 27 different places), and does not allow one to reuse a feature from a system to another one (usually, these selectors are implemented as XPath expression to dynamically identify model elements). Thus, to produce the model m', where A and S are introduced independently of each others, we need to define an explicitly unordered composition operator, denoted as $||$. This operator is complementary to the $\bullet$ one, as it ensures commutativity of features composition when such a property is needed. Several challenges need to be faced to define $||$:

- $\mathcal{C}_1$: *Adaptation re-usability.* To support the reuse of an adaptation, a designer must be able to define an adaptation independently of the concrete models element defined in the targeted system (*e.g.*, using quantifiers, "for all ...").
- $\mathcal{C}_2$: *Adequacy with "usual" composition operator such as aspects weaving or features composition.* The key idea is not to reinvent the wheel. We aim to propose a new operator that complements the others when an ordering is not explicitly needed.
- $\mathcal{C}_3$: *Adaptation Isolation & Determinism.* If at the requirements level two features are not expressed as joint (*i.e.*, there is no explicit ordering dependencies between them), the composition operator must be able to reflect this decision and consequently ensure a deterministic result.
- $\mathcal{C}_4$: *Inconsistency detection.* Through usual composition operator, both $S \bullet A$ and $A \bullet S$ lead to consistent (but different) models after composition. The composition operator must be able to identify inconsistencies that can be introduced during the process, if any.

The contribution of this paper aims to tackle these challenges, through the definition of a *parallel* composition operator, denoted as $||$. We assume that the features used to implement adaptations are based on property selection (*e.g.*, "for all model elements like *this*, do *that*"), as this writing style supports feature re-usability into multiple systems ($\mathcal{C}_1$). On the one hand, if a designer knows the composition order associated to a given set of features, he/she can use existing composition operators to implement the composition ($\mathcal{C}_2$). On the other hand, when such an order is not explicit, the application of a feature F must not impact the application of feature F', for all input models ($\mathcal{C}_3$). Nevertheless, as such isolated composition may lead to inconsistencies, we provide an automated mechanism to identify inconsistencies in the composed result ($\mathcal{C}_4$).

3 An Action–based Approach

Inspired by cutting–edges researches on the modelling research field (*e.g.*, PRAXIS), we use an action–based approach to represents models. According to this paradigm, *"Every model can be expressed as a sequence of elementary construction operations. The sequence of operations that produces a model is composed of the operations performed to define each model element."* [4]. This section formalises the action–orientation used to support the definition of both $||$ and $\bullet$ (formally defined in SEC. 4).

Formalising Actions. The PRAXIS method [4] defines four operations to model models, allowing one to *(i)* `create` a model element, *(ii)* `delete` a model element, *(iii)* set a property in a model element (`setProperty`) and finally *(iv)* set a reference from a model element to another one (`setReference`). We propose here a generalisation of the approach where the expressiveness is dedicated to the handling of attributed graphs. Consequently, we are not restricted to class-based models, and this definition works for any type of artifact that can be represented by a graph (the validation example relies on behavioural model initially modelled as business processes). That is, we consider a model as a set of model elements (*i.e.*, nodes), interconnected through relations (*i.e.*, edges). Sets are intrinsically unordered, and do not contain duplicates. Using first–order logic as underlying foundations, we define the following closed terms (*i.e.*, actions) to interact with a given model:

- $add_n(N, Kind)$: introduces a node N in the model. $Kind$ specializes the node (*e.g.*, a class, a UML annotation).
- $add_e(N, N', Kind)$: defines an edge from N to N' in the model. $Kind$ specializes the reified relation (*e.g.*, inheritance, aggregation).
- $del_e(N, N', Kind)$: deletes the edge from N to N', according to its $Kind$ (as several relations of different kinds may exist between two nodes).
- $del_n(N)$: deletes the node N in the model.

Models as Action Sets. We define a model as a tuple of four action sets (EQ. 1). A model $\mu \in \mathcal{M}$ is composed by *(i)* a set of node additions A_n, *(ii)* a set of edges additions A_e, *(iii)* a set of edge deletions D_e and finally *(iv)* a set of node deletions D_n. In our example, node kinds are restricted to $\{Cl\}$ (for "class"), and relation kinds are defined in $\{Ag, In\}$ (respectively "aggregation" and "inheritance"). For example, considering each class as a node, the model depicted in FIG. 2(a) is reified in EQ. 2.

$$\mu = (A_n, A_r, D_r, D_n) \in \mathcal{M} \tag{1}$$

$$m = (\{add_n(C_1, Cl), add_n(C_2, Cl)\}, \emptyset, \emptyset, \emptyset) \tag{2}$$

The union of two models is used to combine several models into a single one. It is defined as the distribution[5] of the usual set union operator into each contained set:

$$\mu = (A_n, A_e, D_e, D_n) \cup (A'_n, A'_e, D'_e, D'_n) = (A_n \cup A'_n, A_e \cup A'_e, D_e \cup D'_e, D_n \cup D'_n)$$

Relations with action sequences. We do not use plain action sequence representation to avoid permutations issues. Using such a representation, two different action sequences (s_0, s_1) that lead to the same model are considered as non–equal, where our set–driven representation (μ) ensures unicity:

$$s_0 = [add_n(a, Cl), add_n(b, class)], s_1 = [add_n(b, Cl), add_n(a, Cl)], s_0 \neq s_1$$
$$\mu = (\{add_n(a, Cl), add_n(b, Cl)\}, \emptyset, \emptyset, \emptyset)$$

[5] One can notice that such a distribution can also be used to implement others model combination operator (*e.g.*, intersection, difference).

The underlying idea is that a sequence of actions always respects the following steps: it *(i)* adds nodes, *(ii)* adds edges between these nodes, *(iii)* deletes existing edges and finally *(iv)* deletes isolated nodes. In a given step, the internal ordering does not matter (adding x before y is not relevant with regard to the final model). Thus, one can see our representation as a canonical form of an action sequence mandatory to build a given model: the division into four subsets supports this partial ordering.

Consistency. Using this representation, model consistency is ensured according to several logical rules. As the detection of inconsistencies in models is a dedicated research field [3], we always assume in this paper that the handled models are *consistent*. For a given model $\mu = (A_n, A_r, D_r, D_n)$, this property is ensured according to the following rules:

P_1: *"Related elements existence"*. An action that adds a relation between two elements (here classes) C and C' assumes that these two classes are added with the associated actions in A_n (Eq. 3).

P_2: *"Deletion of existing relations"*. An action that deletes a relation between two elements C and C' with a given $Kind$ assumes that this relation is added in A_e (Eq. 4).

P_3: *"Deletion of isolated elements"*. An action that deletes an element C assumes that all relations involving C are deleted in D_e (Eq. 5).

$$add_e(C, C', \_) \in A_e \Rightarrow add_n(C, Cl) \in A_n \wedge add_n(C', Cl) \in A_n \tag{3}$$

$$del_e(C, C', k) \in D_e \Rightarrow add_e(C, C', k) \in A_e \tag{4}$$

$$del_n(C) \in D_n \Rightarrow \begin{cases} \forall add_e(C, X, K_{cx}) \in A_e, \exists del_e(C, X, K_{cx}) \in D_e \\ \wedge\ \forall add_e(Y, C, K_{yc}) \in A_e,\ \exists del_e(Y, C, K_{yc}) \in D_e \end{cases} \tag{5}$$

4 Using Actions to support || and •

In this section, we present how the model representation described in the previous section supports feature introduction, and the definition of both • and || operators.

Using Actions to Introduce Features. Using a functional approach, base models are considered as *constants* (*e.g.*, $\mu \in \mathcal{M}$), and features are defined as *functions* that map an input model μ into an enriched model μ'. Thus, introducing a feature F into a model μ means to use the latter as the input of the former: $\mu' = F(\mu)$. In our approach, we propose to consider F as a two steps function: *(i)* the copy of the input model μ into the output one and *(ii)* the generation of the actions necessary to modify μ and then produce the expected model as output, denoted as $\Delta_F(\mu)$. Our action–based representation of models allows the designer to represent these elements in an endogenous way, as both μ and $\Delta_F(\mu)$ are modelled as sets of actions. Thus, we obtain μ' as the following: $\mu' = F(\mu) = \mu \cup \Delta_F(\mu)$.

For example, we consider here the feature S described in the previous section. Assuming a function named *top* that returns the set of top-level classes discovered in its input, one can implement Δ_S as the following: for an input model μ, it *(i)* adds the SC class and then *(ii)* generates the addition of an inheritance relation between all top-level classes and SC.

$$\begin{aligned}\Delta_S : \mathcal{M} &\to \mathcal{M} \\ \mu &\mapsto (\{add_c(SC, Cl)\}, \{add_e(X, SC, In) | X \in top(\mu)\}, \emptyset, \emptyset)\end{aligned}$$

Thus, the introduction of S in m (FIG. 2(b)) is modelled as the following:

$$\begin{aligned} m &= (\{add_n(C_1, Cl), add_n(C_2, Cl)\}, \emptyset, \emptyset, \emptyset) \\ \Delta_S(m) &= (\{add_c(SC, Cl)\}, \{add_e(C_1, SC, In), add_e(C_2, SC, In)\}, \emptyset, \emptyset) \\ m_S &= S(m) \ = \ m \cup \Delta_S(m) \\ &= (\{add_c(SC, Cl), add_n(C_1, Cl), add_n(C_2, Cl)\}, \\ &\quad \{add_e(C_1, SC, In), add_e(C_2, SC, In)\}, \emptyset, \emptyset) \end{aligned}$$

As said in the consistency paragraph, we assume to work with consistent models and features. Thus, the introduction of a feature into a model always leads to a *consistent* model: let μ a consistent model, and F a given feature. Even if $\Delta_F(\mu)$ may be inconsistent (*e.g.*, deleting a class that is added in μ and not in $\Delta_F(\mu)$, violating P_3), the result of its union with m is therefore consistent.

4.1 Sequential Composition: $\bullet$

Using F and G as two features, and μ a model, we define the functional composition operator $\bullet$ as the following:

$$\begin{aligned} G(\mu) &= \mu \cup \Delta_G(\mu) \\ (F \bullet G)(\mu) &= F(G(\mu)) = G(\mu) \cup \Delta_F(G(\mu)) = \underbrace{\mu \cup \Delta_G(\mu)}_{G(\mu)} \cup \Delta_F(\underbrace{\mu \cup \Delta_G(\mu)}_{G(\mu)}) \end{aligned}$$

The operator holds the following properties, and thus behaves as the "usual" operator:

- Identity: Let Id be the identity feature[6], and F a given feature. $F = F \bullet Id$.
- Idempotence: Let F be a feature. In the general case, $F(F(\mu)) \neq F(\mu)$
- Commutativity: this property cannot be ensured in the general case, and its implementation–dependent. It can be ensured if and only if the two functions F and G are orthogonal. In the general case, $F \bullet G(\mu) \neq G \bullet F(\mu)$.

[6] $\forall \mu \in \mathcal{M},\ \Delta_{Id}(\mu) = (\emptyset, \emptyset, \emptyset, \emptyset) \Rightarrow Id(\mu) = \mu$

Running example We now consider Δ_A, the function used to implements the previously defined A feature:

$$\begin{aligned}\Delta_A : \mathcal{M} &\to \mathcal{M}\\ \mu &\mapsto (\{add_n(AC, class)\}, \{add_e(X, AC, Ag(\_))|X \in top(\mu)\}, \emptyset, \emptyset)\end{aligned}$$

With this function and the previously defined one Δ_S, one can build the model m_{SA} depicted in FIG. 3(a), which represents $(S \bullet A)(m)$.

$$\begin{aligned}\Delta_A(m) &= (\{add_n(AC, Cl)\}, \{add_e(C_1, AC, Ag(a_1)), add_e(C_2, AC, Ag(a_2))\}, \emptyset, \emptyset)\\ m_A &= A(m) \;=\; m \cup \Delta_A(m)\\ &= (\{add_n(AC, Cl), add_n(C_1, Cl), add_n(C_2, Cl)\},\\ &\quad \{add_e(C_1, AC, Ag(a_1)), add_e(C_2, AC, Ag(a_2))\}, \emptyset, \emptyset)\\ top(m_A) &= \{AC, C_1, C_2\}\\ \Delta_S(m_a) &= (\{add_n(SC, Cl)\},\\ &\quad \{add_e(C_1, SC, In), add_e(C_2, SC, In), add_e(AC, SC, In)\}, \emptyset, \emptyset)\\ m_{SA} &= S(A(m)) = m \cup \Delta_A(m) \cup \Delta_S(m \cup \Delta_A(m)) = m_A \cup \Delta_S(m_A)\\ &= (\{add_n(AC, Cl), add_n(C_1, Cl), add_n(C_2, Cl), add_n(SC, Cl)\},\\ &\quad \{add_e(C_1, AC, Ag(a_1)), add_e(C_2, AC, Ag(a_2)),\\ &\quad\; add_e(C_1, SC, In), add_e(C_2, SC, In), add_e(AC, SC, In)\}, \emptyset, \emptyset)\end{aligned}$$

4.2 Parallel Composition: ||

Using F and G as two features, and μ a model, we define the parallel composition operator || as the following:

$$\begin{gathered}F(\mu) = \mu \cup \Delta_F(\mu),\;\; G(\mu) = \mu \cup \Delta_G(\mu)\\ (F||G)(\mu) = F(\mu) \cup G(\mu) = \mu \cup \Delta_F(\mu) \cup \Delta_G(\mu)\end{gathered}$$

As the || operator is defined over set union, it holds the following properties:

- Identity: considering Id as the identify feature and F as a given feature, $(F||Id)(\mu) = \mu \cup \Delta_F(\mu) \cup (\emptyset, \emptyset, \emptyset, \emptyset) = F(\mu)$.
- Idempotence: let F be a given feature. $F||F(\mu) = F(\mu) \cup F(\mu) = F(\mu)$.
- Commutativity: the operator relies on set union, which is commutative. $(F||G)(\mu) = (G||F)(\mu) = \mu \cup \Delta_F(\mu) \cup \Delta_G(\mu)$.

When applied to the previous example, one can build the model m' depicted in FIG. 3(c) as the following:

$$\begin{aligned}m' &= (A||S)(m) = m \cup \Delta_A(m) \cup \Delta_S(m)\\ &= (\{add_n(AC, Cl), add_n(C_1, Cl), add_n(C_2, Cl), add_n(SC, Cl)\},\\ &\quad \{add_e(C_1, AC, Ag(a_1)), add_e(C_2, AC, Ag(a_2)),\\ &\quad\; add_e(C_1, SC, In), add_e(C_2, SC, In)\}, \emptyset, \emptyset)\end{aligned}$$

4.3 Impact on Model Consistency

The previously described definition of feature composition assumes their consistency: for a given model μ and any feature F, the composition of F with μ always leads to a consistent model (*i.e.*, a model that respects the P_i constraints).

- $\bullet$: the sequential composition operator is consistent by construction: it simply chains the compositions.
- $||$: the parallel composition operator works on a different basis (*i.e.*, model union), and then may lead to inconsistent models.

Let F and G two consistent features. For a given model μ, we ensure by construction that both $F(\mu)$ and $G(\mu)$ are also consistent. But their parallel composition $\mu' = F(\mu) \cup G(\mu)$ may be inconsistent, according to the following rules:

- P_1: *"Related elements existence"*. This property is violated if and only if a feature adds a relation that involves an element deleted by another feature.
- P_2: *"Deletion of existing relations"*. This property can be violated if and only if a feature F deletes a relations added in another feature F'. Such a situation also implies a violation of P_1 (F' defines a relation between unknown elements).
- P_3: *"Deletion of isolated elements"*. This property is violated since a feature deletes an element used by the other one in a newly added relation (see P_1).

In fact, the computation of an inconsistent resulting model (*after* the composition) identifies an issue in the features: they cannot be composed in parallel as is, as one relies on the other. It is then a typical use case for a sequential composition ($\bullet$). It tackles challenge $\mathcal{C}_4$, as such erroneous situation can be automatically detected (*e.g.*, through the satisfaction of a logical predicate).

5 Implementation & Validation

In this section, we describe how the approach is implemented in a logical language, and emphasize the need for using the $||$ operator in the context of a complex case study.

5.1 Implementation

We provide a reference implementation of the approach[7]. This framework supports the definition of features as Prolog predicates, and includes a *Domain–Specific Language* (DSL) to express compositions. This language is domain independant, as it relies on the action sequences previously defined, reifying models as attributed graphs. The engine compiles compositions expressed through the DSL into logical predicates (using ANTLR[8]), and supports their execution in

[7] http://www.gcoke.org
[8] http://www.antlr.org/ (version 3.3)

Listing 1.1. Ordered composition ($\bullet$): $m_{sa} \neq m_{as}$

```
1 composition ordered(m) {
2   a(model: m)    => (output: m_a);
3   s(model: m_a) => (output: m_sa);
4 } => (m_sa);
```

Listing 1.2. Parallel composition ($||$): $m' = (A||S)(m) = (S||A)(m)$

```
1 composition parallel(m)  {
2   s(model: m) => (output: m_p);
3   a(model: m) => (output: m_p);
4 } => (m_p);
```

a Prolog interpreter (SWI-Prolog[9]). At run-time, SWI-Prolog provides the JPL framework, which implements a bidirectional Java/Prolog bridge. Thus, the engine can be connected to any tool reachable through the Java language, *e.g.*, the *Eclipse Modeling Framework* (EMF). The implementation of the running example used in this paper is available in the code repository[10].

Using the engine, one can express compositions using the DSL. We represent in Lst. 1.1 how the framework supports ordered compositions ($\bullet$). A composition is named (here `ordered`), and consumes models (here `m`) to produces new ones (here `m_sa`). The way such models are produced is represented as a set of composition directives: line 2 implements the application of the feature `a` using `m` as input `model`, and storing its `output` into `m_a`. A directive is triggered as soon as all its input artifacts are available (*i.e.*, existing or computed by another directive). The parallel composition operator is implemented as the absence of order between directives (Lst. 1.2, next page). In a composition named `parallel`, we only declare that `s` and `a` use the model `m` as input, and store their result in `m_prime` (lines 2 and 3). In front of such a declaration, the engine computes each set of actions independently, and will perform the union of the generated actions sequences before executing it. If an inconsistency is detected (which is not the case here), an error is raised to the designer.

5.2 Validation

We focus here on the Cccms case study, as it is the largest one we used to validate the $||$ operator. The case study was designed by Kienzle *et al.* [13] as a reference framework to compare different aspect–oriented modelling approaches. This example is a *real-life* example, involving thousands of model elements according to real-life business processes. In this context, the considered models

[9] http://www.swi-prolog.org/ (version 5.10.4)
[10] http://code.google.com/p/gcoke/source/browse/trunk/lines/ase_xp/

to be composed reify business processes, *i.e.*, behavioural models. The business processes involved in the CCCMS[11] are modelled as graphs, where nodes are activities and relations implement a partial order between these activities. The case study defines hundreds of activities scheduled by thousands of relations, which makes the example suitable for "real–life" complexity. We instantiated two variants of the requirements: *(i)* a system that only fits the *business* requirements and *(ii)* a system that includes several *non–functional* (NF) concerns. The final system (including NF concerns) defines 146 compositions. As stated in the motivations of this paper, we identified up to 40 shared join points in this study ($\sim 27\%$). On these points, up to 5 concerns had to be composed, leading to 120 potential sequences of composition. This situation triggers a humongous amount of verifications to be checked on the composed system, which is modelled as a set of dense graphs (hundreds of nodes, thousands of relations). Thus, the execution of checkers to verify the consistency of the composed system costs a lot of resource and CPU–time, as the verifications rely on the systematic check of each path defined in a given graph (subject to combinatorial explosion).

While designing the CCCMS, instead of systematically using the $\bullet$ operator and manage all the complexity by hand, we used the $||$ operator to support the compositional approach. The requirement document stated that the features were supposed to be orthogonal, and as a consequence the $||$ operator perfectly implements this intention. The inconsistency detection mechanism (applied on action sequence instead of large graphs) was then used to identify the situations where an order should be defined. Results are summarised in TAB. 5.2.

- The business version uses 24 features and defines 28 composition directives to build the complete system. It can then be considered as a large simplification of the expected system. In this version, only 2 composition directives where identified as conflicting, and actually had to be explicitly ordered (*i.e.*, implements a $\bullet$ composition). All the other compositions can be computed independently. This point illustrates that from a business point of view, the absence of ordering is really important. Applying these features as an ordered sequence can produce unexpected results, like the ones shown in FIG. 3. Through sequential composition, designers would had to *(i)* check the composed system to verify that the obtained result does not contain such feature interactions and/or *(ii)* avoid the usage of quantifiers to anticipate such situations.
- The introduction of NF concerns includes in our case five additional features, dealing with security, persistence and statistical logging. In this configuration, we use 146 composition directives to build the complete system (business + NF). Up to 73% of these directives were unordered in this case study. The others requires an order to meet the requirement specifications. For example, we had to introduce security features after all the others to secure the complete process. It is important to notice that this need was not explicitly documented in the requirements, but accurately detected by the inconsis-

[11] http://www.adore-design.org/doku/examples/cccms/start

Table 1. Composition directives used in the CCCMS

System	#Composition	#Ordered	#Parallel
Business	28	2 (7%)	26 (93%)
Business + NF	146	39 (27%)	107 (73%)

tency detection mechanism. This point highlights the complementarity of the sequential and parallel composition operators.

6 Related Works

Modern mathematics were proposed as a support of feature–oriented software development [2]. This algebraic representation allows the usage of equation optimisers to rewrite the compositions in an efficient way [16]. It is possible to reify the interaction of a feature and another one [17] using mathematical derivative function. Another lead is to use commuting diagrams [14] to explore the different composition orders. The way features are composed together can be constrained through the usage of *design constraint rules*, expressed as attributed grammars [18]. A "valid" composition is consequently identified as a word recognised by the design constraint grammar (identifying conflicting features upstream). The contribution of this paper complements these works, as it also reify compositions as mathematical expressions. The major difference with these works is the definition of a commutative and idempotent composition operator.

The opposite approach of the one described in this paper is to analyse the set of available features and to automatically identify the needed composition order, as implemented by the CAPUCINE framework [25]. Using CAPUCINE, a *Feature Diagram* (FD [8]) is used to express the business variability of a given system. Using an aspect–oriented modelling approach, features are bound to assets that implement aspect models: a fragment of model to be added (*i.e.*, advice) and a selector used to identify where this fragment should be added (*i.e.*, point-cut). CAPUCINE analyses the given elements according to two directions: from the FD to the models and *(i)* from the models to the FD. On the one hand, the latter analyses the set of selectors against the set of model fragments, and identifies hidden dependencies between features that were not expressed in the FD. On the other hand, the former verifies for each constraints expressed in the FD (*e.g.*, "F requires F'") if the implementation follows it (*i.e.*, the selector defined in F matches elements defined in the fragment of model associated to F'). These analysis are complementary to the parallel composition operator, as one can use it to automatically discriminate the features that requires a sequential ($\bullet$) composition and the features that rely on the parallel operator ($||$). Thus, it is possible to *(i) detect* hidden ordering with CAPUCINE and *(ii) ensure* that others features are composed in isolation.

Another lead to ensure commutative composition is followed by the model transformation community [5]. In this work, the key idea is to analyse the set of

model elements impacted (*e.g.*, read, modified, deleted) by a given transformation τ, and then reason about these different sets to check if two transformations may commute. This reasoning capabilities are formalised using set theory, and dedicated to model–transformation. Such an analysis ensures the consistency of models after a parallel composition. Thus, this approach complements ours: *a posteriori* inconsistency detection can be avoided at run-time if commutativity safety can be proved. However, our composition operator *ensures* the parallel application of a set of features, by construction, whatever their definition.

Model weaving can also be considered as a way to support adaptation. This paradigm relies on aspect weaving at the model level. In this context, it is possible to use optimisation techniques to select the best model to be woven with the current one [26]. But intrinsically, these approaches implements an aspect weaving algorithm, which is by nature not commutative. Our approach is thus complementary to these ones, as one can implement our || operator in such a framework and then support unordered composition.

More specifically, the Mata approach [27] supports the weaving of models aspects using a graph–based approach. This approach supports powerful conflict detection mechanisms, used to support the "safe" composition of models [23]. The underlying formal model associated to this detection is based on critical pair analysis [7]. Initially defined for term rewriting systems and then generalised to graph rewriting systems, critical pairs formalise the identification of a minimal example associated to a potentially conflicting situation. This notion supports the development of rule–based systems, identifying conflicting situations such as "the rule r will delete an element matched by the rule r'" or "the rule r generates a structure which is prohibited according to the existing preconditions". This work is complementary with the one presented in this paper, as it can be used to handle inconsistencies in a more detailed way.

7 Conclusions & Perspectives

In this paper, we introduced a new composition operator (denoted as ||), that enables the parallel composition of existing features. Using an action–based approach, we formally defined this new operator and the existing ones (*e.g.*, sequential), as well as its prototypical implementation using a logical language. We identified four challenges, accurately tackled by the approach. The operator supports feature re-usability ($\mathcal{C}_1$), and complements the existing ones (to be used when an order is needed, $\mathcal{C}_2$). It also ensures determinism in the composition ($\mathcal{C}_3$), as the composition order does not matter when || is used. Finally, inconsistency detection mechanisms are provided to ensure the safety of the parallel composition ($\mathcal{C}_4$). The operator was validated in the context of SOA business processes, illustrating how it scales in front of large systems.

Immediate perspectives of this work are to apply the operator to multiple application domains. We plan to focus on the two following research fields, which highly rely on compositions to support their adaptation: *(i)* Cloud–computing and *(ii)* Internet of Things. For the former, it is known that the design of effi-

cient distributed systems is a tedious task. The use of composition algorithms to support their adaptation according to a step-wise approach tames such a complexity, and ensure properties in the composed result (difficult to be checked by humans). In the context of the REMICS[12] project, we are dealing with the migration of legacy systems into cloud-based application. In this context, the need for adaptation is double: *(i)* models of legacy applications have to be adapted *w.r.t* models of clouds to enact a cloud version of the application, and *(ii)* at run-time, run-time models have to be adapted to accurately use the power of the cloud (*e.g.*, "elasticity", which refers to an unlimited resource provisioning capability). The Internet of Things domains is driven by the multiplication of embedded devices (*e.g.*, sensors, smartphone, PDA, tablet PC). Intrinsically, the Internet of Things aims to compose multiple devices into an autonomic entity, able to reconfigure itself at run-time [6], according to changes in its environment (*e.g.*, a more accurate display device is discovered, and the application is reconfigured to broadcast the main content to this new device) [20]. These two application domains will support large-scale experimentation of the || operator, based on real case studies provided by industrial partners.

References

1. Batory, D., Sarvela, J.N., Rauschmayer, A.: Scaling Step-Wise Refinement. IEEE Transactions on Software Engineering 30, 2004 (2004)
2. Batory, D.S.: Using Modern Mathematics as an FOSD Modeling Language. In: Smaragdakis, Y., Siek, J.G. (eds.) GPCE. pp. 35–44. ACM (2008)
3. Blanc, X., Mougenot, A., Mounier, I., Mens, T.: Incremental Detection of Model Inconsistencies Based on Model Operations. In: van Eck, P., Gordijn, J., Wieringa, R. (eds.) CAiSE. LNCS, vol. 5565, pp. 32–46. Springer (2009)
4. Blanc, X., Mounier, I., Mougenot, A., Mens, T.: Detecting Model Inconsistency through Operation-Based Model Construction. In: Schäfer, W., Dwyer, M.B., Gruhn, V. (eds.) ICSE. pp. 511–520. ACM (2008)
5. Etien, A., Muller, A., Legrand, T., Blanc, X.: Combining Independent Model Transformations. In: Shin, S.Y., Ossowski, S., Schumacher, M., Palakal, M.J., Hung, C.C. (eds.) SAC. pp. 2237–2243. ACM (2010)
6. Fleurey, F., Morin, B., Solberg, A.: A Model-driven Approach to Develop Adaptive Firmwares. In: Giese, H., Cheng, B.H.C. (eds.) SEAMS. pp. 168–177. ACM (2011)
7. Heckel, R., Küster, J.M., Taentzer, G.: Confluence of Typed Attributed Graph Transformation Systems. In: ICGT '02: Proceedings of the First Int. Conference on Graph Transformation. pp. 161–176. Springer-Verlag, London, UK (2002)
8. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-Oriented Domain Analysis (FODA) - Feasibility Study. Tech. rep., The Software Engineering Institute (1990), http://www.sei.cmu.edu/reports/90tr021.pdf
9. Kastner, C., Apel, S., Batory, D.: A Case Study Implementing Features Using AspectJ. In: Proceedings of the 11th Int. Software Product Line Conference. pp. 223–232. IEEE Computer Society, Washington, DC, USA (2007)
10. Katz, S., Mezini, M., Kienzle, J. (eds.): Transactions on Aspect-Oriented Software Development VII - A Common Case Study for Aspect-Oriented Modeling, Lecture Notes in Computer Science, vol. 6210. Springer (2010)

[12] http://remics.eu/, EU FP7, STREP.

11. Katz, S., Ossher, H., France, R., Jézéquel, J.M. (eds.): Transactions on Aspect-Oriented Software Development VI, Special Issue on Aspects and Model-Driven Engineering, Lecture Notes in Computer Science, vol. 5560. Springer (2009)
12. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An Overview of AspectJ. In: ECOOP '01: Procs of the 15th European Conference on Object-Oriented Programming. pp. 327–353. Springer-Verlag, London, UK (2001)
13. Kienzle, J., Guelfi, N., Mustafiz, S.: Crisis Management Systems: A Case Study for Aspect-Oriented Modeling. In: T. Aspect-Oriented Soft. Dev. [10], pp. 1–22
14. Kim, C.H.P., Kästner, C., Batory, D.: On the Modularity of Feature Interactions. In: Procs of the 7th int. conf. on Generative programming and Component Engineering. pp. 23–34. ACM, New York, NY, USA (2008)
15. Kniesel, G.: Type-Safe Delegation for Run-Time Component Adaptation. In: Guerraoui, R. (ed.) ECOOP 99 Object-Oriented Programming, Lecture Notes in Computer Science, vol. 1628, pp. 668–668. Springer Berlin / Heidelberg (1999), 10.1007/3-540-48743-3_16
16. Liu, J., Batory, D.: Automatic Remodularization and Optimized Synthesis of Product-Families. In: Proceedings of the Third International Conference on Generative Programming and Component Engineering (GPCE 2004. pp. 379–395. Springer-Verlag (2004)
17. Liu, J., Batory, D.S., Nedunuri, S.: Modeling Interactions in Feature Oriented Software Designs. In: Reiff-Marganiec, S., Ryan, M. (eds.) FIW. pp. 178–197. IOS Press (2005)
18. McAllester, D.: Variational Attribute Grammars for Computer Aided Design (Release 3.0). Tech. rep., MIT (1994)
19. McKinley, P.K., Sadjadi, S.M., Kasten, E.P., Cheng, B.H.C.: Composing Adaptive Software. Computer 37, 56–64 (July 2004)
20. Morin, B., Barais, O., Jezequel, J.M., Fleurey, F., Solberg, A.: Models@ Run.time to Support Dynamic Adaptation. Computer 42(10), 44 –51 (oct 2009)
21. Mosser, S., Blay-Fornarino, M., France, R.: Workflow Design Using Fragment Composition – Crisis Management System Design through ADORE. In: T. Aspect-Oriented Software Development [10], pp. 200–233
22. Mosser, S., Hermosillo, G., Le Meur, A.F., Seinturier, L., Duchien, L.: Undoing Event-Driven Adaptation of Business Processes. In: 8th International Conference on Services Computing (SCC'11). pp. 1–8. , IEEE, Washington DC (Jul 2011)
23. Mussbacher, G., Whittle, J., Amyot, D.: Semantic-Based Interaction Detection in Aspect-Oriented Scenarios. In: RE. pp. 203–212. IEEE Computer Society (2009)
24. Oreizy, P., Medvidovic, N., Taylor, R.N.: Runtime Software Adaptation: Framework, Approaches, and Styles. In: Companion of the 30th int. conf. on Software engineering. pp. 899–910. ICSE Companion '08, ACM, New York, NY, USA (2008)
25. Parra, C., Cleve, A., Blanc, X., Duchien, L.: Feature-based Composition of Software Architectures. In: Babar, M.A., Gorton, I. (eds.) 4th European Conference on Software Architecture. pp. 230–245. Lecture Notes in Computer Science 6285, Copenhagen, Denmark (Aug 2010)
26. White, J., Gray, J., Schmidt, D.C.: Constraint-Based Model Weaving. In: T. Aspect-Oriented Software Development VI [11], pp. 153–190
27. Whittle, J., Jayaraman, P.K., Elkhodary, A.M., Moreira, A., Araújo, J.: MATA: A Unified Approach for Composing UML Aspect Models Based on Graph Transformation. In: T. Aspect-Oriented Software Development VI [11], pp. 191–237