Using Feature Model to Build Model Transformation Chains

Vincent Aranega, Anne Etien, Sébastien Mosser

Using Feature Model to build Model Transformation Chains

Vincent Aranega[1], Anne Etien[1], and Sebastien Mosser[2]

[1] LIFL CNRS UMR 8022 Université Lille 1 - France, `firstname.lastname@lifl.fr`
[2] SINTEF IKT, Norway `sebastien.mosser@sintef.no`

Abstract. Model transformations are intrinsically related to model-driven engineering. According to the increasing size of standardised meta-model, large transformations need to be developed to cover them. Several approaches promote separation of concerns in this context, that is, the definition of small transformations in order to master the overall complexity. Unfortunately, the decomposition of transformations into smaller ones raises new issues: organising the increasing number of transformations and ensuring their composition (*i.e.* the *chaining*). In this paper, we propose to use *feature models* to classify model transformations dedicated to a given business domain. Based on this feature models, automated techniques are used to support the designer, according to two axis: *(i)* the definition of a valid set of model transformations and *(ii)* the generation of an executable chain of model transformation that accurately implement designer's intention. This approach is validated on Gaspard2, a tool dedicated to the design of embedded system.

1 Introduction

Model-Driven Engineering (MDE) advocates the principle of *separation of concerns*, through the extensive use of models in all the steps of the software development cycle [12, 18]. In this context, model transformations are used to achieve *integration of concerns* [14, 17, 3]. Considering the intrinsic complexity of the meta-models in use (*e.g.*, UML 2.x and its profiles), large model transformations (up to ten thousands lines of code) are developed. Such transformations have substantial drawbacks [15], including limited reusability, reduced scalability, poor separation of concerns, limited learnability, and undesirable sensitivity to changes. The separation of concerns paradigm advocates the decomposition of a complex system (*e.g.*, architectures, object-oriented models) into smaller artefacts. Thus, exactly as other artefacts, it is desirable to *decompose* transformations [20]. Other researches have also argued that focusing on such an engineering of transformations improves the uptake of MDE [22]. It is then essential to support the systematic definition of small model transformations with a unique intention [5], to improve scalability, maintainability and reusability of transformations. Such an approach leads to the definition of a family of transformations associated to a given domain that jointly enable to generate systems from a business domain.

The existence of small transformations raises two new issues. First, the chain designer (called *end user* in the remainder of the paper) is in presence of a family of model transformations, which needs to be organised. Secondly, the reification of the dependencies that exist between elements of this family becomes critical. As model transformations cannot be chained anyhow, dependencies that lead to *valid* transformation chains must be captured. One way to automate this development process is to use a *Software Product Line* (SPL) approach. In a SPL, multiple products are *derived* by combining a set of different core assets. One of the most important challenges of SPL engineering concerns variability management, *i.e.*, how to describe, manage and implement the commonalities and variabilities existing among the members of the same family of products. A well-known approach to variability modelling is by means of *Feature Diagrams* (FD) introduced as part of *Feature Oriented Domain Analysis* [9] back in 1990.

Our contribution is to accurately combine model transformations and SPL to support the *end user* while developing transformation-based applications. Business experts' knowledge is reified in a FD to accurately organise the different transformations according to their intentions. Then, automated code analysis techniques are used to accurately generate constraints between these transformations[1], reified in the feature model as *requirements* between features. Thus, it is possible for *end users* to use the FD to accurately define their own products, that is, a valid subset of transformations that matches their intentions. Product derivation mechanisms are then used to automatically generate the model transformation chain that implements what the *end user* asked for. The approach is validated using Gaspard2, a transformation-based tool that supports the modelling of embedded systems.

The remainder of this paper is organised as follows. In Section 2, we motivate this work by exposing the different challenges that need to be addressed in this domain. Then, Section 3 describes the approach we propose to tackle these challenges. Section 4 validates the approach by applying it to the Gaspard2 case study. Finally, Section 5 discusses the related works and Section 6 concludes this paper by exposing some research perspectives.

2 Motivation

In order to enhance reusability, variability, flexibility and verifications, Gaspard2 [8], a co-design environment dedicated to high performance embedded systems based on massively regular parallelism has been designed using Model Driven Engineering (MDE) technologies. Thus it enables the generation of VHDL, SystemC, OpenMP or Lustre code from a UML model enhanced with the *Modelling and Analysis of Real Time Embedded systems* (MARTE) profile. Each language is targeted using a chain composed of three to five dedicated transformations. These large transformations (up to 1500 lines of codes) were not reusable and hardly maintainable even by their own developers.

[1] Informally, a transformation τ requires a transformation τ' if the model elements handled by τ are produced by τ'.

To introduce flexibility and reusability, the Gaspard2 environment has been re-engineered to rely on smaller transformations. Each transformation has a single intention such as memory management or scheduling and corresponds to 150 lines of code in average. Finally, 19 transformations including 4 *model to text* (M2T) transformations, and thus 15 *model to model* (M2M) transformations were defined. The number of chains that can be constructed from them is humongous. Let $T = \{\tau_1, \ldots, \tau_n\}$ a set of model to model transformations, and $M = \{\mu_1, \ldots, \mu_m\}$ a set of model to text transformations. We denote as $N_{T \cup M}$ the number of chains available in this context. The number of potential model to model chains is equal to the number of sequences without repetition that involve elements defined in T (denoted as $P(k, n)$). Secondly, There is $(m+1)$ potential targets for the previously defined sub-chain (as a transformation chain may not generate text). Finally, it is also possible to only generate text without involving other model transformation (thus, m chains).

$$N_{T \cup M} = m + (m+1)\sum_{k=1}^{n} P(k,n), \qquad P(k,n) = \frac{n!}{(n-k)!}$$

$N_{T \cup P}$ is hardly computable generically. Nevertheless, a sub-optimal approximation is to consider $N_{T \cup P}$ bigger than $(m+1)$ times the highest term of the sum $P(k, n)$ (*i.e.*, $P(n, n)$, that in our case is equals to $5 \times 15!$).

$$N_{T \cup P} \gg (m+1) \times P(n,n) = (m+1) \times n!,\ n = 15, m = 4, N_{T \cup P} \gg 6,5 \times 10^{12}$$

But only a few chains make sense! It becomes crucial to help the designer to built such chains. Thus, the definition of transformation libraries raises new issues such as *(i)* the representation of the transformations highlighting their purpose and the relationships between them; *(ii)* their appropriate selection according to the characteristics of the expected targeted system and *(iii)* their composition in a valid order.

Traditionally, transformations are represented in chains or with their metamodels. Such representations are not adapted to the description of transformation libraries. In preparation for chaining the transformations, it seems indispensable to specify their purpose (*i.e.*, what they handle), in addition to their associated metamodels. To generate systems with their own characteristics (*e.g.*, management of distributed versus shared memory, optimised vs simple scheduling), transformations have to be consequently selected. Thus the *end user* has to select the transformations not only according to the characteristics of the resulting system she would like, but also to the relationships between the transformations. Manually performed, this selection may be tedious and error prone. From the selected transformations, several chains can be built. Transformations cannot be chained arbitrarily, some constraints must be fulfilled [7, 11]. If it is often simple to identify the first transformation of the chain (depending on the input metamodels) and the last one (that is a model to text transformation, if code has to be generated), establishing a valid order between the other selected transformations may be difficult. Indeed, existing approaches check if the proposed order is valid, but do not automatically provide a valid one.

In order to support the *end user* in the design of transformation chains, the following challenges have to be addressed:

C_1 Propose to the *end user* a library in which each transformation can be easily identified according to the characteristics of the expected final system (Section 3.1).

C_2 Help the *end user* to select transformations while automatically taking into account the relationships between transformations (Section 3.2).

C_3 Automatically derive the transformation chain from the characteristics selected by the *end user* (Section 3.3).

3 Solution

To tackle the aforementioned challenges, we propose a feature-oriented approach and the associated too set to automatically generate accurate model transformation chains as depicted in Figure 1.

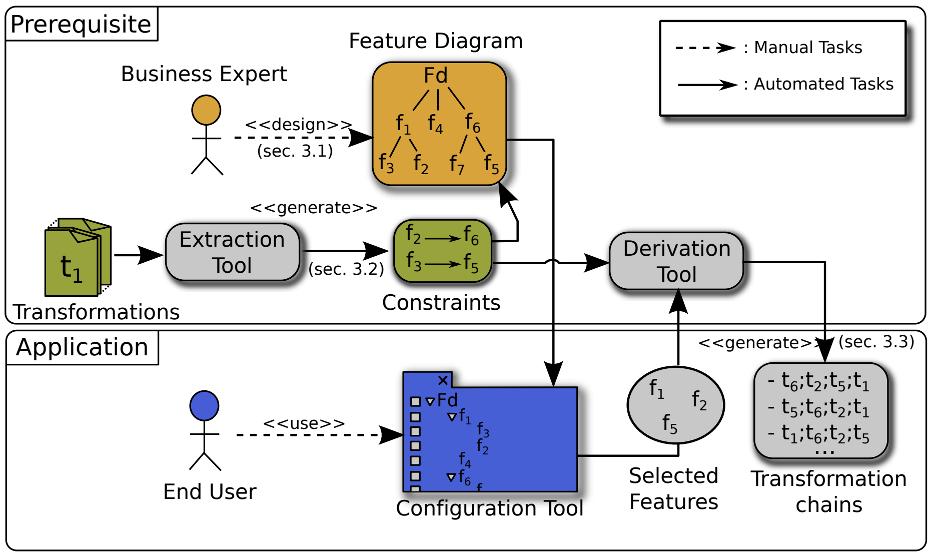


Fig. 1. Approach Process Overview

This approach relies on three pillars: *(i)* the classification of the available transformations as a *Feature Diagram* (FD) produced by the *business expert*, *(ii)* the reification of requirement relationships between transformation (directly generate from the *Transformations* set by the *Extraction Tool*) and *(iii)* the automated generation of transformation chains for a given product (using our *Derivation Tool*) from features selected by the *end user*.

The FD is designed once for all by the *business expert* as a prerequisite. It is nevertheless possible to modify it when new transformations and thus new features become available. The requirement relationships are expressed between the features and automatically computed from the transformation codes by the *Extraction Tool* we provide. The extracted relations enable to derive other features (and then the associated transformation) from the ones selected by the *end user* using a *Configuration Tool* (*e.g.*, FeatureIDE[2]). The requirement relationships are also used by our *Derivation Tool* to order the selected features in order to design valid chains.

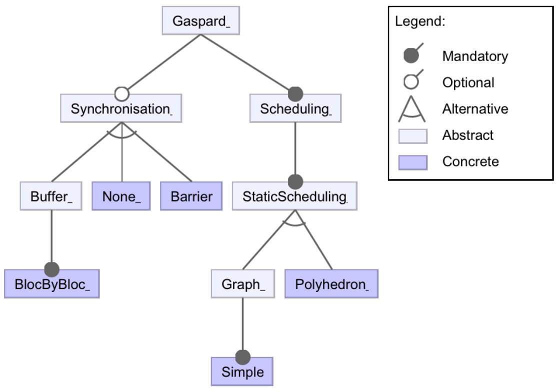
3.1 Structuring the Transformation Set as a Feature Diagram

As a transformation is used to support a given intention according to a business domain, a set of transformations implicitly model the variability of the different intentions associated to a domain. FD were defined to model such a variability, so its use is natural. We represent in Figure 2 an excerpt of the complete FD associated to Gaspard2. Using FD, features (represented as nodes) are classified among others according to constraints such as exclusiveness or optionality. Model transformations are bound to features as assets. A feature f holds a link to the actual model transformation to be used to implement the intention captured by f at run-time. Normally, each feature corresponds to a single transformation and vice versa. However, it occurs in practise that a single transformation may catch many intentions and thus corresponds to many features.

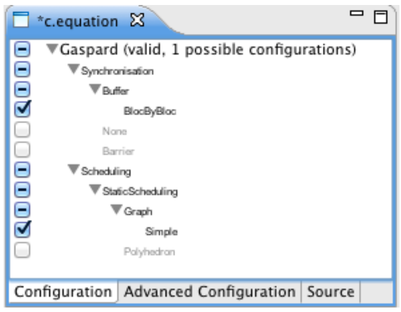
For example, in Figure 2(a), the FD models that a given product *must* contain a `Scheduling` feature, and *may* contain a `Synchronisation` feature. The features `Graph` and `Polyhedron` are exclusive, *i.e.*, the use of one in a given product implies that the other cannot be used in this particular product. We call a product a set of features that respects the constraints modelled in the FD. For example, Figures 2(b) and 2(c) represent two products among the eight valid *w.r.t.* the modelled FD. The first one (Figure 2(b)) considers a system synchronised using a `BlocByBloc` method, and scheduled with a simple `Graph`. The second product (Figure 2(c)) considers a system synchronised with a `Barrier` method, and scheduled with a `Polyhedron` approach. In our context, features reify model transformations: the actual implementation of the transformation is bound as an asset of the associated feature node. Thus, considering a given product, it is possible to automatically infer the set of transformations involved in the transformation chain that supports it.

Key Points. The role of the FD is to capture the business knowledge associated to a given set of transformations. It actually transforms a flat set of transformations into an organised family of products.This classification is done by the *business expert*, that is, someone who deeply knows the different transformations, their underlying intentions, as well as the artifact they are handling. The key idea here is that this work is done once by the *business expert*, and capitalised in the

[2] http://wwwiti.cs.uni-magdeburg.de/iti_db/research/featureide/

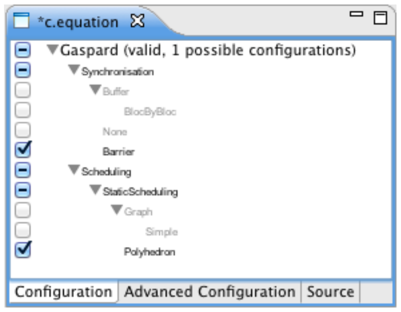


(a) Gaspard2 feature diagram (excerpt)



```
*c.equation
⊟ ▼Gaspard (valid, 1 possible configurations)
⊟     ▼Synchronisation
⊟         ▼Buffer
✓              BlocByBloc
☐           None
☐           Barrier
⊟     ▼Scheduling
⊟         ▼StaticScheduling
⊟             ▼Graph
✓                  Simple
☐              Polyhedron
Configuration | Advanced Configuration | Source
```

(b) $p = \{BlocByBloc, Simple, \dots\}$



```
*c.equation
⊟ ▼Gaspard (valid, 1 possible configurations)
⊟     ▼Synchronisation
☐         ▼Buffer
☐              BlocByBloc
☐           None
✓           Barrier
⊟     ▼Scheduling
⊟         ▼StaticScheduling
☐             ▼Graph
☐                  Simple
✓              Polyhedron
Configuration | Advanced Configuration | Source
```

(c) $p' = \{Barrier, Polyhedron, \dots\}$

Fig. 2. Gaspard2: Feature diagram and associated products (using FeatureIDE)

FD. Without the use of a FD to support such a classification, it would be up to the *end users* to guess how the different transformations cope with each others before assembling them.

3.2 Recovering Require Relationship from Transformations

On top of constraints expressing the mandatory/optional character of the features as well as the and/or relationships between them, "require" relationships can also be captured in FD. They enable to automatically deduced other features from the selected ones, independently of the tree structure of the FD. Require relationships can be determined manually by the *business expert*. However, when the number of features is huge, omission can happen leading to erroneous products determination. Therefore, we provide an automatic analysis of the transformations to recover the require relationships (bijection) between the features associated to them. A requirement between two features f and f' (denoted as a logical implication, *i.e.*, $f \Rightarrow f'$) means that the transformation bound to f requires the transformation bound to f'. The following question is raised: "*When does a require relationship between two transformations exists?*". In fact, it relies on the element type production and consumption. For two transformations τ and τ', if τ' consumes types created by τ, then it implies that a require relationship exists between τ and τ', denoted as $\tau' \rightarrow \tau$ (for τ' requires τ). For each transformation, it is thus mandatory to automatically determine the element types it produces and it consumes to provide an automatic require relationships determination.

This automatic analysis relies on the different actions performed on element types by a transformation. Four actions are classically performed by transformations: *reading*, *creating*, *deleting* and *modifying.* This analysis does not rely on transformation execution but on static code analysis. Thus an element of the input or the output metamodel of a transformation is considered read if the presence of one on its instance enables the application of a transformation rule. An element is considered created, if at least one of its instance can be created by the transformation, and so on. Thus, τ' requires τ if τ' reads some elements created by τ [6]. As we stated in the previous section, for a feature f from the FD, it exists at most one transformation τ. So, considering two features f, f' in the FD and two transformations τ, τ' mapped to f, respectively f', if $\tau \rightarrow \tau'$, it implies that $f \Rightarrow f'$.

Key Points. The proposed generation of the require relationships relies on a static analysis of the transformation codes. Once the FD designed and the constraints generated, the *end user* can use a *Configuration Tool* to select the features she wants for her transformation chain. The *Configuration Tool* is parametrised by the feature diagram and the generated constraints. Thus, by taking into account the generated constraints, during the feature selection, the *Configuration Tool* can either invalidate features or add required features according to the ones already selected by the *end user.* The automatic characteristic of the generation enables a certain evolutivity of the FD.

3.3 Generating Transformation Chains

Based on the two previous parts of the contribution, it is now possible to *(i)* consider a set of model transformations as a product family and *(ii)* automatically infer the requirement relationships that exist inside the product family. These two contributions act at the level of the FD. According to the global process, the selected features are then passed to a *Derivation Tool*, which uses the generated constraints to propose transformation chains from the selected features.

We consider now a given product $p = \{f_1, \ldots, f_n\}$, *i.e.*, a subset of features that satisfies the constraints modelled in the FD. As stated in Section 3.1, model transformations are bound to features. It is then possible to obtain the set of model transformations associated to p (denoted as T_p) by mapping each feature to its associated transformation: $T_p = \{\tau_1, \ldots, \tau_m\}$. As some features are only used to structure the FD and are not related to any concrete transformation, it should be noted that the cardinality of T_p may be lesser than the cardinality of p. But this set of transformations is not sufficient to properly derive a concrete transformation chain from a given product. The requirement constraints identified in Section 3.2 must be taken into account. Considering two features f and f', if the requirement $f \Rightarrow f'$ exists, then the transformation τ' mapped to f' must be executed before the transformation τ mapped to f. As a consequence, the analysis of the set of requirement constraints leads to the identification of sequences of model transformations. Two situations can be encountered. If the requirement constraints implement a total order on the set of transformations, only one sequence will be identified, *i.e.*, the proper transformation chain to be executed to support the intentions captured by this product. But if the requirement constraints implement a partial order, only *partial sequences* can be identified automatically. But as there is no requirement between these different sub-sequences, their order is not important. Consequently several valid chains are generated. This "subchains approach" is useful to support the *business expert* while assessing the FM consistency. It also helps the non-expert *end user* to construct a valid chain: any of the chains built upon these sub-chains, will by essence respect the dependencies captured by the FM.

Key Points. A concrete chain of model transformations is automatically derived, through the FD, from the transformation set selected by the *end user*. First, the knowledge of a *business expert* is captured in the FD, and then an automatic static analysis is used to properly extract technical constraints from the implementation of the transformations. Finally, it is possible to automatically derive the chain, through the systematic exploration of the identified constraints. As a consequence,the generation of the concrete model transformation chain is automated, and the *end user* does not require any knowledge of model transformation from a technical point of view.

4 Validation: The Gaspard2 Case Study

Gaspard2 is a co-design environment dedicated to high performance embedded systems based on massively regular parallelism. From high level specifications, it automatically generates code for high performance computing, hardware-software co-simulation, functional verification or hardware synthesis using model transformations. Such generations are complex and require intermediary steps, *e.g.*, the explicit mapping of application tasks onto processing units, the mapping of the data onto memories or the scheduling of the tasks. Each transformation has a specific intention and deals with few concepts. Nineteen transformations have been implemented for now but the framework may support even more of them in the upcoming months. It is difficult for a non expert user to easily understand the purpose of each transformation, to select the ones useful to reach the desired platform and finally to order them in order to compose a chain.

In this paper, we used the Familiar tool suite [1] to manipulate feature diagrams. This tool allows us to model FD, and is well integrated in the Eclipse platform. Thus, standard *Configuration Tools* (*e.g.*, FeatureIDE) can be used to allow the *end user* to configure products. But it should be noted that the approach is not bound to this tool nor to this case study from a theoretical point of view, as described in the previous section.

4.1 Step #1: Capturing Business Expert Knowledge in a FD

Embedded systems designers usually do not master model transformation paradigm and underlying technologies. It is then essential to support them while designing the transformation chains used to generate code from high level specifications. The design of these chains consists in the selection of relevant transformations available in a library and in the computation of a valid order. Selecting a transformation requires to easily distinguish one transformation from another and to quickly identify its intention. In order to help the embedded systems designers, we have classified the available transformations based on embedded characteristics using feature model. It is up to the *business expert* to find the most appropriate classification method to be used to support the *end user*. During the implementation of the case study, we applied an incremental definition of the FD. We produced 12 successive versions of the FD. Excepting from one deep refactoring to better handle business requirement constraint, the implementation of the FD by the *business expert* was straightforward.

Most transformations of Gaspard2 have a unique intention representing a specific characteristics of the produced systems such as memory management. The transformations and their associated intentions are listed in Table 1. The Gaspard2 transformation library counts 19 intentions through 15 M2M and 4 M2T transformations. For example, the *scheduling* transformation has the following intention: it manages a simple scheduling of application tasks on computing units. As a consequence of the non mandatory bijection between features and transformations, the *barrier* and the *openMP* features are implemented by a single transformation. This many-to-one (surjection) relationship corresponds to a lack of modularisation of the transformation.

From these intentions, the *business expert* builds the feature diagram by associating a feature to each intention. Moreover, some features are added in the hierarchy in order to specify the relationship AND/OR/XOR between features. Indeed, as stated in Section 3.1, each feature represents at most one transformation. The resulting FD, depicted in Figure 3, gathers, in an non exhaustive way, some characteristics that an embedded system produced by Gaspard2 may possess. For example, the `OpenCL` and `OpenMP` features, introduce a scientific computation intention. However, only one of these two features can be selected. Indeed, the target language is either OpenCL, or OpenMP. In the FD, this choice is designed by the introduction of an intermediary abstract node `ScientificComputation` and an alternative between the two features.

The associated tooling provided by the Familiar platform can be used to query the model, as shown in Figure 4. This FD models up to 200 different available configurations (obtained by the Familiar `counting` algorithm). The `configs` command computes all the available products, returning the set of valid products defined by this FD.

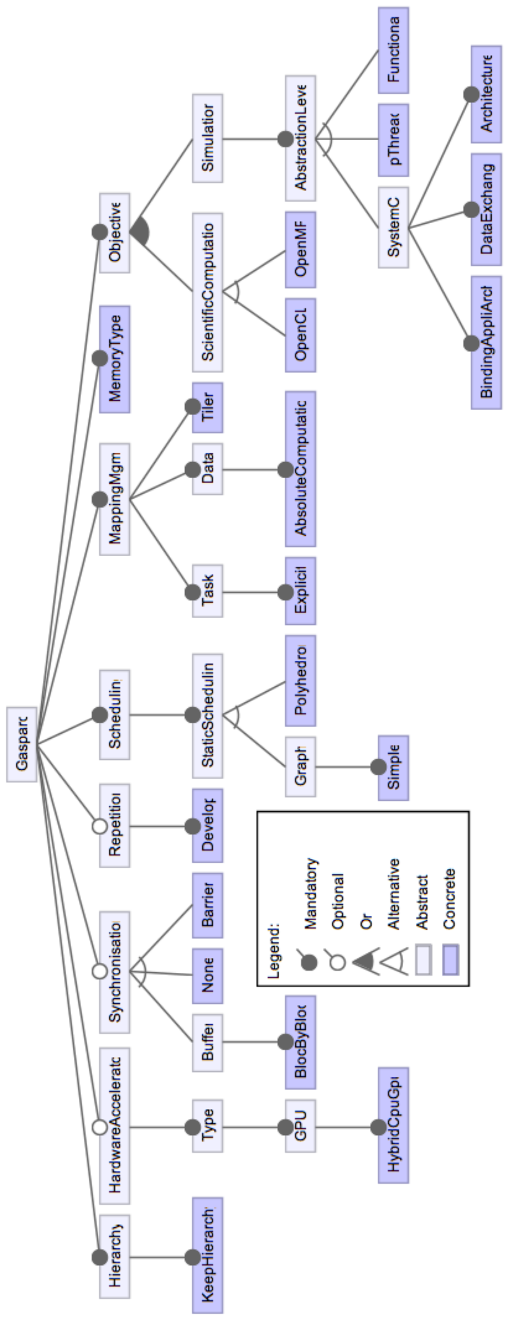
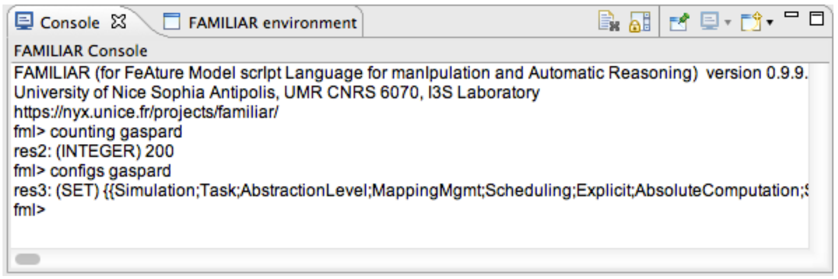


Fig. 3. Feature model associated to Gaspard2.

Transformation	**Intention**
tiler2task	- Keep repetitions `hierarchy`
gpuApi	- Manage `hybrid GPU-CPU` computing
pThread	- Manage `buffered synchronisation by bloc`
sequentialC	- Generate `sequential` C code
barrier	- Manage `barrier synchronisation` for `OpenMP`
shape2loop	- `Develop repetitions` in the generated systems
scheduling	- Manage `simple` scheduling
poly_loop	- Manage `polyhedron` optimised `scheduling`
explicitAllocation	- `Explicitly` place tasks on processors
memorymapping	- Manage `absolute` memory `addresses`
tilerMapping	- Manage `tiler` (*i.e.* task distributing data) mapping on computing unit
shared	- Manage the shared `memory type`
openCL	- Generate `OpenCL` code for `scientific computation` purposes
openMP	- Generate `OpenMP` code for `scientific computation` purposes
systemcPA	- `Bind` SystemC `architecture` with SystemC `application`
systemcBind	- Manage SystemC `data exchanges`
systemcStruct	- Manage SystemC `architecture`
pthreadGen	- Generate `pthread` code for simulation purposes
functional	- Introduce `functional` abstraction

Table 1. Gaspard2 transformation set



```
FAMILIAR (for FeAture Model scrIpt Language for manIpulation and Automatic Reasoning)  version 0.9.9.
University of Nice Sophia Antipolis, UMR CNRS 6070, I3S Laboratory
https://nyx.unice.fr/projects/familiar/
fml> counting gaspard
res2: (INTEGER) 200
fml> configs gaspard
res3: (SET) {{Simulation;Task;AbstractionLevel;MappingMgmt;Scheduling;Explicit;AbsoluteComputation;
fml>
```

Fig. 4. Using the Familiar shell to interact with the FD.

4.2 Step #2: Extracting Constraints from the Implementation

This feature model enables the classification and the distinction of the transformations the one from the others. However, in this primary form, it does not gather enough information to build the chains: some others may be required and the selection of one transformation may require the selection of others. Such dependencies between transformations have to be captured and the feature model tools enable to take them into account for the product configuration. Thanks to the *Extraction Tool*, the implementation of the available transformations is automatically analysed. The result of this analysis is a set of "require" constraints between the features modelled in the FD. We represent in Listing 1.1 the set of constraints obtained after the execution of the tool. These constraints are generated using the syntax of the Familiar tool, and thus can be automatically integrated in the FD. Contrarily to the initial FD that captures the knowledge of the *business expert*, these relations reify the implementation constraints that exist between the transformations, from a technical point of view. It then ensures that the products configured *w.r.t.* this FD will be valid at both level: *(i)* business domain and *(ii)* technical implementation.

```
1   AbsoluteComputation -> Develop
2   AbsoluteComputation -> KeepHierarchy
3   AbsoluteComputation -> Polyhedron
4   BindingAppliArchi ->
        AbsoluteComputation
5   BindingAppliArchi -> Architecture
6   BindingAppliArchi -> BlocByBloc
7   BindingAppliArchi -> MemoryType
8   BlocByBloc -> AbsoluteComputation
9   BlocByBloc -> Graph
10  BlocByBloc -> KeepHierarchy
11  BlocByBloc -> MemoryType
12  DataExchange -> Architecture
13  DataExchange -> KeepHierarchy
14  Functional -> Graph
15  Graph -> KeepHierarchy
16  Hybrid -> AbsoluteComputation
17  Hybrid -> Graph
18  Hybrid -> KeepHierarchy
19  Hybrid -> MemoryType
20  MemoryType -> KeepHierarchy
21  Simple -> Graph
22  Tiler -> Graph
```

Listing 1.1. Set of requirement constraints.

Considering this set of constraints, the *Configuration Tool* now proposes 37 available products to the *end user* (from 200 at the beginning). This highlights the fact that working with the implementation of the transformation is critical. The technical implementation of the transformations dramatically reduces the initial variability of the domain as it was designed by the *business expert*.

Taking into account the “real” features implementations in the FD (*i.e*, the transformations code in our context) through this set of automatically computed constraints also leads to interesting situations that help the *business expert*. We consider here the feature `Repetition`, defined as optional by the *business expert* (see Figure 3). The generated set of constraints identifies a requirement between the feature `AbsoluteComputation` and the feature `Develop` (line 1 in Listing 1.1). However, `AbsoluteComputation` is mandatory, and selecting `Develop` implies to select `Repetition`. Thus, the `Repetition` feature is automatically identified by the tool suite as a *false optional* feature, that is, a feature modelled as optional but enforced as mandatory by a requirement constraints. In this case, it helped the *business expert* to identify a missing artifact in the FD: it should also contain an alternative implementation for `Repetition` instead of only defining the `Develop` approach.

4.3 Step #3: Deriving Transformation Chains

Based on the FD enhanced with the implementation constraints, we can now ensure that the products configured by the *end user* through the configuration tool are valid. The final step is to use a derivation tool that properly builds the transformation chains associated to a given product. We consider here one of the 37 products available according to this FD, denoted as p corresponding for example to the set of the features selected by the *end user*. The first step is to translate p into T_p, that is, the set of transformations involved by this product. It should be noted that $|p| > |T_p|$, as several features are only used to structure the FD and consequently are not bound to concrete transformations. For example, for the following product, corresponds the associated T_p:

$$\begin{aligned} p = \{&Gaspard, MemoryType, Polyhedron, Data, Barrier, MappingMgmt,\\ &KeepHierarchy, Hierarchy, Tiler, Develop, StaticScheduling,\\ &AbsoluteComputation, Task, Explicit, ScientificComputation,\\ &Scheduling, Objective, Repetition, Synchronisation, OpenMP\}\\ T_p = \{&explicitAllocation, memMapping, openMP, poly\_loop,\\ &shape2loop, tilerMapping, tiler2task\} \end{aligned}$$

The second step is to map the constraints between features as a partial order among the transformations. The requirements involved in p are the following:

$$\begin{aligned} &\textbf{Feature Requirement} \leadsto \textbf{Transformation Ordering}\\ AbsoluteComputation \rightarrow Develop &\leadsto memMapping \rightarrow shape2loop\\ AbsoluteComputation \rightarrow Polyhedron &\leadsto memMapping \rightarrow poly\_loop\\ MemoryType \rightarrow KeepHierarchy &\leadsto memMapping \rightarrow tiler2task \end{aligned}$$

Based on this partial order, it is possible to compute[3] the following sets of "independent" sub-chains involved in this product, as a chain template, that is, a partition of the transformation set taking into account the partial order:

$$tpl_p = [\ [openMP], [explicitAllocation], [tilerMapping] \tag{1}$$
$$[memMapping, [shape2loop, poly\_loop, tiler2task]\ ]\ ] \tag{2}$$

Among the computed sub-chains, the *openMP* transformation is a "model to text" transformation and will always be the last one executed in the chain. In line 2, the partial order indicates that the *memMapping* transformation must be preceded by the 3 transformations listed, without specifying any order between them. Thus, there is up to 6 ways to combine these transformations according to this constraint. As the *explicitAllocation* and *tilerMapping* transformations can be executed independently of these sub-chains, they can be executed before, after or inside the previously described sub-chains. As a consequence, up to 180 chains can be obtained from this product. Following the sub-chains computed by our derivation tool, a valid transformation chain could be:

$$\begin{aligned} &explicitAllocatlion \rightarrow tiler2task \rightarrow tilerMapping \rightarrow \cdots\\ &\qquad\qquad \cdots poly\_loop \rightarrow shape2loop \rightarrow memMapping \rightarrow openMP \end{aligned}$$

Without any lead, the *end user* has only one constraint: the model to text transformation must be the last of the chain. From the product p and its associated set of transformations T_p, it means that the *end user* has the choice to

[3] We used a set of logical predicates implemented using he Prolog language to implement the *Derivation Tool*.

organise 6 transformations. Thus, she has $P(6, 6) = 720$ choices to organise the model to model transformations. Among the 720 chaining possibilities, many are not valid because the require relationships are not considered. So, without any indication, the *end user* has to choose from 720, potentially non valid, chains, whereas with our methodology, the choice is reduced to 180 valid chains only.

To sum up, our methodology and the associated tool have allowed the *end user* (without any knowledge about transformations) to easily build chains. She has selected transformations based on embedded system features *i.e.* using terms she is familiar with. Finally, she has to choose among 180 valid chains whereas initially she was confronted to a huge number of possible chains that she has to build by scrutinizing the transformation code.

5 State of the Art

In order to enhance the reusability of transformations, several authors promote the decomposition of transformations into smaller ones. However transformations have then to be chained. Vanhooff *et al.* proposed an approach based on the explicit and manual identification of the required and provided concepts by the chain developer for example using a profile in order to later build the chain [21]. Our approach relies on the feature model to compose the transformations.

Several approaches have been proposed to build chains. Transformations are considered as functions to compose if their domains are compliant [13] or UML activity that can be chained using different operators: composition, conditional composition, parallel composition and loop [16]. However, in both cases, the transformation chain has to be manually specify by the designer, without any specific help. In the latter case, they are executed using the provided model transformation orchestration tool. Our approach could be used upstream to identify the useful transformations and to compose them.

Transformation chaining relies on constraints that can be automatically identified, *e.g.* using the distinction between concepts copied and those mutated [4]. This approach only deals with endogenous transformations (even if a possible extension to heterogeneous transformations is suggested). With the "require" constraints, we have extended this approach to heterogenous transformations.

Several approaches propose to deal with the complexity of large systems with a feature-based approach. For example, feature models were accurately used to model the intrinsic variability of the Linux Kernel [10], and support end-user during the kernel configuration task. The approach proposed in this paper follow the same idea, that is, the use of feature modelling to leverage a highly variable systems into an entity configurable by the end-user.

Being able to extract the features from the implementation is a challenge [2]. The most difficult part is the extraction of the feature hierarchy from the "flattened" implementation [19]. Inferring such a hierarchy relies on domain heuristics that rank the possible hierarchy, and the final assessment of these ranks by a domain expert. In this paper, we do not consider the automatic extraction of the features from the transformation set, and only rely on the business expert to properly model the feature diagram. Being able to support the business expert during this task is an interesting perspective of this work.

Feature models are also used to support the reverse engineering of large scale systems [1]. For example, the FraSCAti platform (an open source implementation of the SCA standard) was accurately reverse-engineered to support its assessment. Based on a dedicated tool that extracts the architecture from the implementation, the authors confront the automatically extracted feature model with the one defined by the *business expert*. This approach complements ours, as we also rely on a tool that automatically infers feature information from the actual implementation of the system (in our case requirements between features). But instead of assessing the model defined by the business expert, we focused on its enrichment, by merging the set of automatically identified information in this feature model. We were able to identify several situations where the actual system was not "as variable" as the business expert thought.

6 Conclusions & Perspectives

In order to be reusable and maintainable, model transformations are written according to a single intention and complex transformations are built as the chaining of smallest ones. In this paper, we proposed an approach based on FD to support the design of model transformation chains. Based on a classification of the transformations made by a *business expert*, This approach allows an *end user* to build such chains, without any prior knowledge of model transformation technologies. The implementation of the transformations is also automatically taken into account to ensure that the built chains are valid from a run-time point of view. From an implementation point of view, the approach is independent of any tools and can be easily coupled to existing approaches (*e.g.*, FeatureIDE, Familiar). The approach was validated on the Gaspard2 case study, and we are currently pursuing another validation study in the domain of website engineering.

The resulting chains are valid according to a type based approach [7]. However, two transformations that can be chained into both orders from a syntactic perspective are not obviously commutable from a business point of view: the execution of the two successive transformations on whatever models may not always lead to the same result. A perspective of this work is to enhance the expressiveness of the requirement detection mechanisms to address this issue. Another perspective concerns the FD refinement. Indeed, the FD being manually designed by the *business expert*, some constraints between features may have been omitted. The automatic requirement relationships extraction could be a first help to highlight a badly / incompletely designed FD. To help the *business expert* in the definition or the refinement of the FD, we plan to automatically extract features from the documentation written by the transformation developers.

References

1. Mathieu Acher, Anthony Cleve, Philippe Collet, Philippe Merle, Laurence Duchien, and Philippe Lahire. Reverse engineering architectural feature models. In Ivica Crnkovic, Volker Gruhn, and Matthias Book, editors, *ECSA*, volume 6903 of *Lecture Notes in Computer Science*, pages 220–235. Springer, 2011.
2. Mathieu Acher, Anthony Cleve, Gilles Perrouin, Patrick Heymans, Charles Vanbeneden, Philippe Collet, and Philippe Lahire. On extracting feature models from product descriptions. In Ulrich W. Eisenecker, Sven Apel, and Stefania Gnesi, editors, *VaMoS*, pages 45–54. ACM, 2012.
3. Jean Bézivin and Olivier Gerbé. Towards a Precise Definition of the OMG/MDA Framework. In *16th IEEE International Conference on Automated Software Engineering (ASE 2001), San Diego, CA, USA*, pages 273–280, 2001.
4. Raphael Chenouard and Frédéric Jouault. Automatically Discovering Hidden Transformation Chaining Constraints. In Andy Schurr and Bran Selic, editors, *Model Driven Engineering Languages and Systems*, volume 5795 of *Lecture Notes in Computer Science*, pages 92–106. Springer Berlin / Heidelberg, 2009.
5. Jim Cordy. Eating our own Dog Food: DSLs for Generative and Transformational Engineering. In *GPCE*, 2009.
6. A. Etien, V. Aranega, X. Blanc, and R. Paige. Chaining Model Transformations. In *Submitted to ECMFA conference*, 2012.
7. A. Etien, A. Muller, T. Legrand, and X. Blanc. Combining Independent Model Transformations. In *Proceedings of the ACM SAC, Software Engineering Track*, pages pp. 2239–2345, 2010.
8. A. Gamatié, S. Le Beux, É. Piel, R. Ben Atitallah, A. Etien, P. Marquet, and J.-L. Dekeyser. A Model Driven Design Framework for Massively Parallel Embedded Systems. *ACM Transactions on Embedded Computing Systems*, 10(4), 2011.
9. Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-Oriented Domain Analysis (FODA) - Feasibility Study. Technical report, The Software Engineering Institute, 1990.
10. Rafael Lotufo, Steven She, Thorsten Berger, Krzysztof Czarnecki, and Andrzej Wasowski. Evolution of the linux kernel variability model. In Jan Bosch and Jaejoon Lee, editors, *SPLC*, volume 6287 of *Lecture Notes in Computer Science*, pages 136–150. Springer, 2010.
11. T. Mens, G. Taentzer, and O. Runge. Detecting Structural Refactoring Conflicts Using Critical Pair Analysis. *Electronic Notes in Theoretical Computer Science*, 127(3):113–128, April 2005.
12. J. Miller and J. Mukerji. MDA Guide Version 1.0.1. Technical report, Object Management Group (OMG), 2003.
13. Jon Oldevik. Transformation Composition Modelling Framework. In *Proceedings of the Distributed Applications and Interoperable Systems Conference*, volume 3543 of *Lecture Notes in Computer Science*, pages 108–114. Springer, 2005.
14. Harold Ossher, William Harrison, and Peri Tarr. Software engineering tools and environments: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 261–277, New York, NY, USA, 2000. ACM.
15. Jens Pilgrim, Bert Vanhooff, Immo Schulz-Gerlach, and Yolande Berbers. Constructing and Visualizing Transformation Chains. In *Proceedings of the European conference on Model Driven Architecture*, pages 17–32, 2008.
16. José E. Rivera, Daniel Ruiz-Gonzalez, Fernando Lopez-Romero, José Bautista, and Antonio Vallecillo. Orchestrating ATL Model Transformations. In *Proc. of MtATL 2009*, pages 34–46, Nantes, France, July 2009.

17. Douglas C. Schmidt. Guest Editor's Introduction: Model-Driven Engineering. *IEEE Computer*, 39(2):25–31, 2006.
18. Bran Selic. The Pragmatics of Model-Driven Development. *IEEE Software*, 20(5):19–25, 2003.
19. Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wasowski, and Krzysztof Czarnecki. Reverse engineering feature models. In Richard N. Taylor, Harald Gall, and Nenad Medvidovic, editors, *ICSE*, pages 461–470. ACM, 2011.
20. Bert Vanhooff, Dhouha Ayed, and Yolande Berbers. A Framework for Transformation Chain Development Processes. In *Proceedings of the ECMDA Composition of Model Transformations Workshop*, pages pp. 3–8, 2006.
21. Bert Vanhooff and Yolande Berbers. Breaking Up the Transformation Chain. In *Proceedings of the Best Practices for Model-Driven Software Development at OOPSLA 2005*, San Diego, California, USA, 2005.
22. Dennis Wagelaar, Ragnhild Van Der Straeten, and Dirk Deridder. Module Superimposition: a Composition Technique for Rule-based Model Transformation Languages. *Software and Systems Modeling*, 2009. Online First.