Combining Declarative and Procedural Views in the Feature-Oriented Specification and Analysis of Product Families

Maurice H. ter Beek[1], Alberto Lluch Lafuente[2], and Marinella Petrocchi[3]

[1] ISTI–CNR, Pisa, Italy
maurice.terbeek@isti.cnr.it
[2] IMT Institute for Advanced Studies Lucca, Italy
alberto.lluch@imtlucca.it
[3] IIT–CNR, Pisa, Italy
marinella.petrocchi@iit.cnr.it

Abstract. We present the recently introduced feature-oriented language FLan as a proof of concept for specifying both declarative aspects of product families, namely constraints on their features, and procedural aspects, namely feature configuration and run-time behaviour. FLan is inspired by the concurrent constraint programming paradigm. A store of constraints allows one to specify in a declarative way all the constraints on features that are commonly used in software product line engineering, including the cross-tree constraints well known from feature models. A standard yet rich set of process-algebraic operators allows one to specify in a procedural way the configuration and behaviour of products. There is a close interaction between these two views: (i) the execution of a process is constrained by its store to forbid undesired configurations; (ii) a process can query a store to resolve design and behavioural choices; (iii) a process can update the store by adding new features.

1 Introduction

The last decades have witnessed a paradigm shift from mass production to mass customization to serve as many individual customer's needs as possible. Software Product Line Engineering (SPLE) has translated this into a software engineering approach aimed at developing, in a cost effective way, a variety of software-intensive products that share an overall reference model, i.e. that together form a product family. Usually, commonality and variability are defined in terms of features, and managing variability is about identifying variation points in a common family design to encode exactly those combinations of features that lead to valid products. The actual configuration of the products during application engineering then boils down to selecting desired options in the variability model.

Feature models are the most widely used variability model [15]. They provide a compact representation of all products of a family in terms of features, and additional constraints among them. Graphically, features are drawn as nodes of a tree, with the family as its root and relations between these features representing constraints. However, there may be thousands of features, requiring models with thousands of options, which easily leads to anomalies such as superfluous or—worse—contradictory variability information (e.g. so-called false optional features and dead features). There is a large body of literature on computer-aided analyses of feature models to extract valid products and to detect anomalies [4].

None of these analyses consider behavioural variability, though, in the sense that only the presence of the software implementing the features is considered—not their ordering in time. Indeed, research on applying formal methods in SPLE traditionally focusses on modelling and analysing structural rather than behavioural constraints in product families. However, many software-intensive systems are embedded, distributed and safety critical, making it important to be able to model and analyse also their behaviour, as a form of quality assurance.

Recent years have witnessed a growing interest in specifically considering also the behavioural variability of product families. This has resulted in, among others, extensions of Petri nets [13] and a variety of frameworks with an LTS-like semantics [8,11,10,12,7,1]. As a result, behavioural analysis techniques such as model checking have become available for the verification of (temporal) logic properties of product families. Specifying a product family directly in an operational model is often not feasible. Therefore it can be useful to resort to high-level formal languages with semantics over those operational models, as is common in the context of process algebra. Several extensions of CCS have been proposed to model product families [10,9], but none can combine behavioural constraints with all common structural constraints known from feature models [15].

We present here our current research on FLan: a feature-oriented language for specifying product families by taking structural *and* behavioural constraints into account [2]. It is inspired by concurrent constraint programming [14] and its adoption in process calculi [5]. A store of constraints allows one to specify all common structural constraints known from feature models in a declarative way, incl. cross-tree constraints. Moreover, a rich set of process-algebraic operators allows one to specify in a procedural way both the configuration and behaviour of products. These declarative and procedural views are closely related: (i) the execution of a process is constrained by its store, e.g. to avoid introducing inconsistencies; (ii) a process can query a store to resolve options regarding the design and behaviour; (iii) a process can update the store by adding new features. To this aim, the semantics of FLan unifies static and dynamic feature selection in an elegant fashion. Inspired by [9], we implemented FLan in the executable modelling language Maude (`http://maude.cs.uiuc.edu/`), whose rich toolkit allows the application of a variety of formal automated analysis techniques, from consistency checking to model checking, to product families specified in FLan.

This paper is an extended abstract of [2]. It is organised as follows. §2 describes a running example of a family of coffee machines. In §3, we present the syntax and semantics of FLan and a specification of the example, while we refer to [2] for an illustration of its Maude-supported automated analyses. We report some concluding remarks and list promising future work in §4, while we refer to [2] for a detailed discussion of related work.

2 A Family of Coffee Machines

We use a popular running example in the style of [1,3,7,9,13]. It describes a (simplified) family of coffee machines in terms of the following requirements.

1. Initially, a coin must be inserted: either a euro, exclusively for European products, or a dollar, exclusively for Canadian products;
2. Upon the insertion of a coin, a choice for sugar must be offered, followed by a choice of beverages;
3. The choice of beverage (coffee, tea, cappuccino) varies, but every product must offer at least one beverage, tea may be offered only by European products, and all products that offer cappuccino must also offer coffee;
4. Optionally, a ringtone may be rung after the delivery of a beverage. However, a ringtone must be rung after serving a cappuccino;
5. After the beverage is taken, the machine returns idle.

These requirements define products by combining structural constraints defining valid feature configurations (e.g. *"every product must offer at least one beverage"*) with temporal constraints defining valid behaviour (e.g. *"a ringtone must be rung after serving a cappuccino"*). Behaviour is not captured at all in feature models.

3 FLan: Syntax and Semantics

The feature-oriented language FLan we propose here is loosely inspired by the CCS-like process algebra CL4SPL [9], but it differs in its treatment of the cross-tree constraints known from feature models and in the separation of declarative and procedural aspects inspired by the concurrent constraint programming paradigm [14] and its adoption in process calculi [5]. FLan's core notions are *features*, *constraints*, *processes* and *fragments*, which can be identified in the syntax of FLan in Fig. 1. More precisely, f and g range over features and syntactic categories S, P and F correspond to constraints, processes, and fragments.

A *feature* is a term describing specific elements or properties of a product. The universe of features is denoted by $\mathcal{F}$. The features of our running example are the coins accepted (i.e. *euro* and *dollar*), the products offered (i.e. *coffee*, *tea* and *cappuccino*) and additional elements such as *sugar* (the capability to regulate the delivery of sugar) and *ringtone* (the capability to emit a ringtone).

The declarative part of FLan is represented by a *store of constraints* which defines both constraints on features extracted from the product requirements and additional information (e.g. about the context wherein the product will operate). Two important notions of constraint stores are (i) the *consistency* of a store S, denoted by *consistent(S)*, which in our case amounts to logical satisfiability of all constraints forming S; and (ii) the *entailment* $S \vdash c$ of constraint c in store S, which in our case amounts to logical entailment. A constraint store is any term generated by S in the grammar of FLan. The most basic constraint stores are $\top$ (no constraint at all), $\bot$ (inconsistent) and

$$\begin{array}{rcl}
F & ::= & [S \parallel P] \\
S, T & ::= & K \mid f \triangleright g \mid f \otimes g \mid S\ T \mid \top \mid \bot \\
P, Q & ::= & \mathit{0} \mid X \mid A.P \mid P + Q \mid P;Q \mid P \,|\, Q \\
A & ::= & \mathsf{install}(f) \mid \mathsf{ask}(K) \mid a \\
K & ::= & p \mid \neg K \mid K \vee K
\end{array}$$

Fig. 1. The syntax of FLan, where $a \in \mathcal{A}$, $p \in \mathcal{P}$ and $f, g \in \mathcal{F}$

ordinary boolean propositions (generated by K). Constraints can be combined by juxtaposition.

We assume that all standard *structural constraints* known from feature models (optional, mandatory and alternative—or and xor—features) are expressed using boolean propositions (e.g. as explained in [15]). For this purpose, we assume that the universe $\mathcal{P}$ of propositions contains a Boolean predicate $has(\cdot) : \mathcal{F} \to \mathbb{B}$ that can be used to denote the presence of a feature in a product. Boolean propositions can also be used to represent additional information such as contextual facts. Examples from our running example are $in(Europe)$ and $in(Canada)$, respectively used to state the fact that the coffee machine being configured is meant to be used in Europe or in Canada. Boolean propositions can state relations between contextual information and features, like $in(Europe) \to has(euro)$ (i.e. a coffee machine for the European market needs a euro coin slot).

The *cross-tree constraints* known from feature models (*requires* and *excludes*) are instead handled as first-class citizens to emphasise the way we deal with them. A constraint $f \triangleright g$ expresses that feature f requires the presence of feature g while a constraint $f \otimes g$ expresses that features f and g mutually exclude each other's presence (i.e. they are incompatible). Of course, also these constraints can be encoded as boolean propositions. For instance, $f \otimes g$ and $f \triangleright g$ can equivalently be expressed as $has(f) \leftrightarrow \neg has(g)$ and $has(f) \to has(g)$, respectively. We use indeed such logical encoding to reduce consistency checking and entailment to logical satisfiability (and hence exploit Maude's SAT solver).

We also consider a class of *action constraints*, reminiscent of Featured Transitions Systems [7], where transitions are subject to the presence of features. For instance, in a coffee machine equipped with a slot for euro coins we will use *euro* for the action of inserting a euro coin and $do(euro)$ as a proposition stating the execution of that action. The relations between the action *euro* and the presence of the corresponding feature *euro* can be formalised as $do(euro) \to has(euro)$, i.e. the insertion of a euro coin requires the presence of an appropriate coin slot. In general, we assume that each action a may have a constraint $do(a) \to p$. These act as a sort of guard to allow or forbid the execution of actions (see later).

The constraint store S in Fig. 2 formalises part of the requirements specified in §2 for our running example. It contains both contextual information (e.g. $in(Europe)$) and action constraints (e.g. $do(euro) \to has(euro)$). For instance, from requirement 1 we extract that *euro* and *dollar* are mutually

$$
\begin{aligned}
F &\doteq [S \parallel D; R]\\
S &\doteq S_1\ S_2\\
S_1 &\doteq \mathit{has}(\mathit{euro}) \vee \mathit{has}(\mathit{dollar}) \quad \mathit{in}(\mathit{Europe}) \to \mathit{has}(\mathit{euro}) \quad \mathit{in}(\mathit{Canada}) \to \mathit{has}(\mathit{dollar})\\
&\quad\ \mathit{has}(\mathit{coffee}) \vee \mathit{has}(\mathit{cappuccino}) \vee \mathit{has}(\mathit{tea}) \quad \mathit{has}(\mathit{tea}) \to \mathit{in}(\mathit{Europe})\\
&\quad\ \mathit{dollar} \otimes \mathit{euro} \quad \mathit{cappuccino} \triangleright \mathit{coffee}\\
&\quad\ \mathit{do}(\mathit{euro}) \to \mathit{has}(\mathit{euro}) \quad \mathit{do}(\mathit{dollar}) \to \mathit{has}(\mathit{dollar})\\
&\quad\ \mathit{do}(\mathit{coffee}) \to \mathit{has}(\mathit{coffee}) \quad \mathit{do}(\mathit{cappuccino}) \to \mathit{has}(\mathit{cappuccino})\\
&\quad\ \mathit{do}(\mathit{tea}) \to \mathit{has}(\mathit{tea}) \quad \mathit{do}(\mathit{sugar}) \to \mathit{has}(\mathit{sugar}) \quad \mathit{do}(\mathit{ringtone}) \to \mathit{has}(\mathit{ringtone})\\
S_2 &\doteq \mathit{in}(\mathit{Europe})\\
D &\doteq \mathsf{install}(\mathit{euro}).0 \mid \mathsf{install}(\mathit{dollar}).0 \mid \mathsf{install}(\mathit{sugar}).0\\
&\quad\ \mid \mathsf{install}(\mathit{coffee}).0 \mid \mathsf{install}(\mathit{tea}).0 \mid \mathsf{install}(\mathit{cappuccino}).0\\
R &\doteq (\mathit{euro}.0 + \mathit{dollar}.0); (P_2 + P_3)\\
P_2 &\doteq \mathit{sugar}.P_3\\
P_3 &\doteq \mathit{coffee}.P_4 + \mathit{tea}.P_4 + \mathit{cappuccino}.P_5\\
P_4 &\doteq P_5 + R\\
P_5 &\doteq \mathsf{install}(\mathit{ringtone}).\mathit{ringtone}.R
\end{aligned}
$$

Fig. 2. A FLan specification of the coffee machine

exclusive features (formalised as *dollar* $\otimes$ *euro*), while from requirement 3 we understand that *cappuccino* requires *coffee* (formalised as *cappuccino* $\triangleright$ *coffee*).

The procedural part of FLan is represented by *processes* of the following type:

- *0*, the empty process that can do nothing;
- X, where X is a process identifier (we assume that there is a set of process definitions of the form $X \doteq P$ and we also assume that recursively defined processes are finitely branching, which can be ensured in standard ways, e.g. prefixing every occurrence of a process identifier X with an action or constraining process definitions to be of the form $X \doteq A.P$);
- $A.P$, a process willing to perform the action A and then to behave as P;
- $P + Q$, a process that can non-deterministically choose to behave as P or Q;
- $P; Q$, a process that must progress first as P and then as Q;
- $P \mid Q$, a process formed by the parallel composition of P and Q, which evolve independently.

Note that we distinguish between ordinary actions (from a universe $\mathcal{A}$) and the special actions $\mathsf{install}(f)$ (used to denote the dynamic installation of a feature f) and $\mathsf{ask}(K)$ (used to query the store). We will see that each action type is treated differently in rules of the operational semantics. In our example we consider the actions *euro*, *dollar* (insertion of the respective coin); *sugar* (sugar selection); *coffee*, *tea*, *cappuccino* (beverage selection); and *ringtone* (ringtone emission).

A *fragment* F is a term $[S \parallel P]$, formed by a store of constraints S and a process P, which may influence each other, as in concurrent constraint programming [14]: a process may update its store which, in turn, may condition the execution of process actions. The operational semantics of closed fragments (i.e. its

$$(\textsc{Inst})\ \frac{consistent(S\ has(f))}{[S \parallel \mathsf{install}(f).P] \longrightarrow [S\ has(f) \parallel P]} \qquad (\textsc{Or})\ \frac{[S \parallel P] \longrightarrow [S' \parallel P']}{[S \parallel P+Q] \longrightarrow [S' \parallel P']}$$

$$(\textsc{Ask})\ \frac{S \vdash K}{[S \parallel \mathsf{ask}(K).P] \longrightarrow [S \parallel P]} \qquad (\textsc{Seq})\ \frac{[S \parallel P] \longrightarrow [S' \parallel P']}{[S \parallel P;Q] \longrightarrow [S' \parallel P';Q]}$$

$$(\textsc{Act})\ \frac{S \vdash (do(a) \to K) \qquad S \vdash K}{[S \parallel a.P] \longrightarrow [S \parallel P]} \qquad (\textsc{Par})\ \frac{[S \parallel P] \longrightarrow [S' \parallel P']}{[S \parallel P|Q] \longrightarrow [S' \parallel P'|Q]}$$

Fig. 3. The reduction semantics of FLan

reduction semantics) is formalised by the transition relation $\to \subseteq \mathbb{F} \times \mathbb{F}$ of Fig. 3, with $\mathbb{F}$ denoting the set of all terms generated by F in the grammar of Fig. 1. Technically, such reduction relation is defined in SOS style modulo a structural congruence relation $\equiv \subseteq \mathbb{F} \times \mathbb{F}$, which allows to identify different ways to denote the same fragment. We consider the least congruence on fragments closed w.r.t. commutativity and associativity of non-deterministic and parallel composition of processes ($P+Q \equiv Q+P$, $P+(Q+R) \equiv (P+Q)+R$, $P \,|\, Q \equiv Q \,|\, P$ and $P \,|\, (Q \,|\, R) \equiv (P \,|\, Q) \,|\, R$); associativity of sequential composition of processes ($P;(Q;R) \equiv (P;Q);R$); identity of non-deterministic choice, sequential and parallel composition of processes ($P{+}0 \equiv P$, $0;P \equiv P$, $P;0 \equiv P$ and $P \,|\, 0 \equiv P$); and expansion of recursive process definitions ($P \equiv P[^Q/_X]$ if $X \doteq Q$). This choice is not accidental: all can be naturally and efficiently treated by Maude, so our semantics enjoys several nice properties: (1) it is (efficiently) executable; (2) each semantic rule of Fig. 3 corresponds to exactly one conditional rewrite rule in Maude's implementation of FLan (cf. [2]); (3) the number of reduction rules is small, so semantics and implementation are compact and easy to read.

Rules Inst and Act are very similar, both allowing a process to execute an action if certain constraints are satisfied. In particular, rule Inst forbids inconsistencies due to the introduction of new features. Note that rule Inst can be seen as a particular instance of the rule for the tell operation of concurrent constraint programming [14] instantiated as $\mathsf{tell}(has(f))$. Rule Act forbids inconsistencies with respect to action constraints. A typical case of action constraint is $do(a) \to has(f)$, i.e. action a is subject to the presence of feature f. Rule Ask formalises the semantics of the usual $\mathsf{ask}(\cdot)$ operation as known from concurrent constraint programming [14]. It allows to block a process until a proposition can be derived from the store. Rule Or is quite straightforward. It allows the process to evolve as any of the branches. It is worth remarking that non-determinism can be solved at the procedural level (by relying on $\mathsf{ask}(\cdot)$ actions) or at the declarative level (by using a non-deterministic choice that may be solved by the constraint store), thus providing a lot of flexibility to fragment designers (as illustrated in [2]). Rules Seq, formalising the usual sequential composition, and Par, formalising an interleaving parallel composition, are standard.

Note the different ways in which feature f can be selected in configurations. First, this can be done in an *explicit* and *declarative* way by including the proposition $has(f)$ in the initial store. This would be the case of features that the system designer is sure to be mandatory for all the family's products. Second, the presence of feature f can be obtained in an *implicit* and *declarative* way, meaning that f may be derived as a consequence of further constraints. This would be the case of features that apparently seem not to be mandatory to the system designer, but that are indeed enforced by the constraints (e.g. in a store containing the constraints $g \rhd f$ and $has(g)$ the presence of f can be inferred). Third, feature f can be dynamically installed in a *procedural* way during process execution. This is a key aspect of our approach as it enables the designer to delay feature configuration decisions and to specify them procedurally. FLan's concurrent constraint approach allows to combine these three declarative and procedural forms of feature configuration in an elegant and consistent way.

Figure 2 shows a specification of the coffee machine. Fragment F is formed by store S and the sequential composition of processes D, specifying an initial design phase, and R, specifying the run-time behaviour of the coffee machine. The store S is made of two parts: constraints derived from the requirements specification (S_1) plus some contextual information and initial configurations (S_2). Note that the action constraints are quite simple (all are of the form $do(f) \to has(g)$) but recall that they could be more sophisticated if needed. For instance, one could specify the constraint on action *cappuccino* as $do(cappuccino) \to has(cappuccino) \wedge has(ringtone)$ thus requiring not only the presence of the corresponding feature but also that of the *ringtone* feature.

The configuration process D is quite simple. It is just formed by the parallel composition of the installation of some of the features that the coffee machine may exhibit. This specifies a sort of race between features and may be thought of as independent designers competing to install the features they are responsible for. The semantics of FLan ensures that all executions will end up with a consistent configuration if the process begins with a consistent store. For instance, the semantics will forbid the installation of mutually exclusive features. Process R describes the coffee machine's run-time operation. Depending on the country it is meant for, the machine may either accept a euro or a dollar. This is implemented as a non-deterministic choice that will be consistently solved at run-time due to the presence of the action constraints $do(euro) \to has(euro)$ and $do(dollar) \to has(dollar)$, which will forbid the use of actions *euro* or *dollar* if the corresponding feature has not been installed. After that, it may (P_2) or it may not (P_3) deliver sugar. The next step is beverage selection and delivery, which may be followed by a ringtone (P_5) or not, after which it returns idle. Note that D and R are not *pure* configuration and run-time processes. Indeed, feature *ringtone* is not installed by D but by R, i.e. the feature *ringtone* is installed dynamically and it can be thought of as, e.g., a software module. This is an interesting example of a partial configuration process, where some non-mandatory features are not installed and products are only partially configured, and a run-time configurable process that installs features when needed.

4 Conclusion

The concurrent constraint programming paradigm adopted in FLan provides a flexible mechanism for separating and (when necessary) combining declarative and procedural aspects. For instance, design decisions can be delayed until run-time, which is very convenient for software product families that allow features to be added while the system operates. Also, the run-time specification can be relieved from design decisions like feature constraints, thus resulting in light-weight, understandable specifications. Moreover, FLan's implementation in Maude allows to exploit this framework's rich analysis toolset (as done in [2]).

We envisage several potentially interesting extensions of FLan. First, we can adopt further primitives and mechanisms from the concurrent constraint programming tradition. The concurrent constraint π-calculus [5], e.g., provides synchronisation mechanisms typical of mobile calculi (i.e. name passing), a check operation to prevent inconsistencies, a retract operation to remove (syntactically present) constraints from the store and a general framework for *soft* constraints (i.e. not only boolean). Second, we may equip FLan with a semantics over known suitable models for product families, like Featured Transition Systems [7] and Modal Transition Systems [8,11,12,1], so that FLan becomes a high-level language for them and we can exploit their specialised analysis tools (e.g. [6,3]).

References

1. P. Asirelli, M.H. ter Beek, A. Fantechi, and S. Gnesi. Formal description of variability in product families. *SPLC'11*. IEEE, 2011, 130–139.
2. M.H. ter Beek, A. Lluch Lafuente, and M. Petrocchi. Combining declarative and procedural views in the specification and analysis of product families. *FMSPLE'13*. ACM, 2013.
3. M.H. ter Beek, F. Mazzanti, and A. Sulova. VMC: A tool for product variability analysis. *FM'12*, *LNCS* 7436. Springer, 2012, 450–454.
4. D. Benavides, S. Segura, and A. Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. *Inf. Syst.* 35, 6 (2010), 615–636.
5. M.G. Buscemi and U. Montanari. CC-Pi: A constraint-based language for specifying service level agreements. *ESOP'07*, *LNCS* 4421. Springer, 2007, 18–32.
6. A. Classen, M. Cordy, P. Heymans, A. Legay, and P.-Y. Schobbens. Model checking software product lines with SNIP. *STTT* 14, 5 (2012), 589–612.
7. A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, and J.-F. Raskin. Model checking lots of systems. *ICSE'10*. ACM, 2010, 335–344.
8. D. Fischbein, S. Uchitel, and V.A. Braberman. A foundation for behavioural conformance in software product line architectures. *ROSATEA'06*. ACM, 2006, 39–48.
9. S. Gnesi and M. Petrocchi. Towards an executable algebra for product lines. *FM-SPLE'12*. ACM, 2012, 66–73.
10. A. Gruler, M. Leucker, and K.D. Scheidemann. Modeling and model checking software product lines. *FMOODS'08*, *LNCS* 5051. Springer, 2008, 113–131.
11. K.G. Larsen, U. Nyman, and A. Wąsowski. Modal I/O automata for interface and product line theories. *ESOP'07*, *LNCS* 4421. Springer, 2007, 64–79.
12. K. Lauenroth, K. Pohl, and S. Töhning. Model checking of domain artifacts in product line engineering. *ASE'09*. IEEE, 2009, 269–280.
13. R. Muschevici, J. Proença, and D. Clarke. Modular modelling of software product lines with feature nets *SEFM'11*, *LNCS* 7041. Springer, 2011, 318–333.
14. V.A. Saraswat and M.C. Rinard. Concurrent constraint programming. *POPL'90*. ACM, 1990, 232–245.

15. P.-Y. Schobbens, P. Heymans, and J.-C. Trigaux. Feature diagrams: A survey and a formal semantics. *RE'06*. IEEE, 2006, 136–145.