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

Maurice H. ter Beek
ISTI–CNR, Pisa, IT
LIACS, Leiden University, NL
maurice.terbeek@isti.cnr.it

Alberto Lluch Lafuente
IMT
Lucca, IT
alberto.lluch@imtlucca.it

Marinella Petrocchi
IIT–CNR
Pisa, IT
marinella.petrocchi@iit.cnr.it

ABSTRACT

We introduce the 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 design processes 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 common constraints on features, including cross-tree constraints as 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 both 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, for instance to add new features. An implementation in the Maude framework allows for a variety of formal automated analyses of product families specified in FLan, ranging from consistency checking to model checking.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Formal methods, Model checking, Validation*

General Terms

Design, Experimentation, Verification

Keywords

Product families, Variability, Process algebra, Concurrent constraint programming, Behavioural analyses, Maude

1. INTRODUCTION

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 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 variants of UML diagrams [31], extensions of Petri nets [25, 26] and a variety of frameworks with transition system semantics [12, 18, 15, 20, 10, 3]. 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 easily 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 [24] have been proposed to model product families [13, 15, 16, 21], but none of these can combine behavioural constraints with all common structural constraints known from feature models.

We introduce here the feature-oriented language FLan as a proof of concept for specifying product families by taking both structural and behavioural constraints into account. It is inspired by concurrent constraint programming [28] and its application in process algebra [8]. A store of constraints allows one to specify in a declarative way all common structural constraints known from feature models, including 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.

The 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, e.g. to add new features.

Inspired by [13], we implemented FLan in the executable modelling language Maude [11], whose rich toolkit enables the application of a variety of formal automated analysis techniques to product families specified in FLan, from consistency checking to model checking.

The paper is organised as follows. Section 2 describes a running example of a family of coffee machines. In Sect. 3, we present the syntax and semantics of FLan and a specification of the example. Section 4 illustrates the Maude-supported automated analyses of the example. We discuss related work in Sect. 5, report some concluding remarks in Sect. 6 and list promising future work in Sect. 7.[1]

[1]For the convenience of the reviewers the manuscript in [4] contains the complete Maude implementation.

2. A FAMILY OF COFFEE MACHINES

We use a popular running example in the style of [2, 3, 5, 6, 10, 13, 25, 26]. It describes a (simplified) family of coffee machines in terms of the following list of requirements:

1. Initially, a coin must be inserted: either a euro, exclusively for products for the European market, 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, i.e. action sequences (e.g. *"a ringtone must be rung after serving a cappuccino"*).

3. FLan: SYNTAX AND SEMANTICS

The feature-oriented language FLan we propose here is loosely inspired by the CCS-like process algebra CL4SPL presented in [13], but it strongly 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 [28] and its adoption in process calculi [8].

The core notions of FLan are *features*, *constraints*, *processes* and *fragments*, which can all be identified in the syntax of FLan given in Fig. 1. More precisely, features range over f and g and constraints, processes and fragments correspond to the syntactic categories S, P and F, respectively.

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

where $a \in \mathcal{A}$, $p \in \mathcal{P}$ and $f, g \in \mathcal{F}$

Figure 1: The syntax of FLan

Features. 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 quantity of sugar) and *ringtone* (the capability to emit a ringtone).

Constraints. 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 *consistent(S)* and which in our case amounts to logical satisfiability of all constraints forming S; and (ii) *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 ordinary boolean propositions (generated by K). Constraints can be combined by juxtaposition (its semantics amounts to logical conjunction).

We assume that the standard structural constraints on features (like options, obligations and alternatives) are expressed using boolean propositions (e.g. as explained in [29]). 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 (in grounded form) 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).

Cross-tree constraints, instead, are handled as first-class citizens. A constraint $f \rhd g$ 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 $f \rhd g$ 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). However, we prefer to keep here this first-class treatment in order to emphasize their use in the presentation of our work.

We also consider a class of *action constraints*, reminiscent of Featured Transitions Systems [10], 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 $do(a) \to p$. Such constraints act as a sort of guard to allow or forbid the execution of actions (as illustrated later on in the discussion of rule Act).

The constraint store S in Fig. 4 formalises part of the requirements specified in Sect. 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 exclusive features (formalised as $dollar \otimes euro$),

$$
\begin{array}{rcl@{\qquad}rcl}
P+Q & \equiv & Q+P & P+(Q+R) & \equiv & (P+Q)+R \\
P\,|\,Q & \equiv & Q\,|\,P & P\,|\,(Q\,|\,R) & \equiv & (P\,|\,Q)\,|\,R \\
P+0 & \equiv & P & P;(Q;R) & \equiv & (P;Q);R \\
0;P & \equiv & P & P;0 & \equiv & P \\
P\,|\,0 & \equiv & P & P & \equiv & P[^{Q}/_{X}] \text{ if } X \doteq Q
\end{array}
$$

Figure 2: Structural congruence in FLan

while from requirement 3 we understand that *cappuccino* requires *coffee* (formalised as *cappuccino* $\triangleright$ *coffee*).

Processes. The procedural part of FLan is represented by *processes*. A process can be one of the following:

- *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 $X \doteq P$. 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 or every process definition with an action);
- $A.P$, 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 as Q;
- $P;Q$, a process that must progress first as P and then as Q;
- $P\,|\,Q$, a process formed by the parallel composition of P and Q, which evolve independently.

It is worth remarking that we distinguish between ordinary actions (from a universe $\mathcal{A}$) and the special actions $\mathsf{install}(f)$ (used to denote the 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 will consider the following actions: *euro* and *dollar* (insertion of the respective coin); *sugar* (sugar selection); *coffee*, *tea*, and *cappuccino* (beverage selection); and *ringtone* (ringtone emission).

Fragments. Finally, a *fragment* F is a term $[S \parallel P]$, composed by a store of constraints S and a process P. Each of the components of a fragment may influence each other, along the lines of the concurrent constraint programming paradigm [28]: 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 reduction semantics) is formalised in terms of the state transition relation $\rightarrow \subseteq \mathbb{F}\times\mathbb{F}$ illustrated in Fig. 3, where $\mathbb{F}$ denotes the set of all terms generated by F in the grammar of Fig. 1. Technically, such reduction relation is defined in Structural Operational Semantics (SOS) style (i.e. by induction on the structure of the terms denoting a fragment) modulo a structural congruence relation $\equiv \subseteq \mathbb{F}\times\mathbb{F}$. As usual, the reduction relation implicitly defines an unlabelled transition system.

Considering terms up to a structural congruence allows to identify different ways of denoting the same fragment. Here we consider the least congruence on fragments closed

$$
(\textsc{Inst})\ \frac{\mathit{consistent}(S\ \mathit{has}(f))}{[S \parallel \mathsf{install}(f).P] \longrightarrow [S\ \mathit{has}(f) \parallel P]}
$$

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

$$
(\textsc{Act})\ \frac{S = S'\ (\mathit{do}(a) \rightarrow K)\ \wedge\ S \vdash K}{[S \parallel a.P] \longrightarrow [S \parallel P]}
$$

$$
(\textsc{Or})\ \frac{[S \parallel P] \longrightarrow [S' \parallel P']}{[S \parallel P+Q] \longrightarrow [S' \parallel P']}
$$

$$
(\textsc{Seq})\ \frac{[S \parallel P] \longrightarrow [S' \parallel P']}{[S \parallel P;Q] \longrightarrow [S' \parallel P';Q]}
$$

$$
(\textsc{Par})\ \frac{[S \parallel P] \longrightarrow [S' \parallel P']}{[S \parallel P|Q] \longrightarrow [S' \parallel P'|Q]}
$$

Figure 3: Reduction semantics of FLan

with respect to the commutativity and associativity of non-deterministic and parallel composition of processes; the associativity of sequential composition of processes; the identity of non-deterministic choice, sequential and parallel composition of processes; and the expansion of recursive process definitions. This choice of axioms (some of which may seem unusual) is not accidental. Indeed, all can be naturally and efficiently treated by Maude so that 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 the Maude implementation of FLan; (3) the number of reduction rules is small and the semantics and its implementation are thus compact and easy to read.

As usual, reduction rules are expressed in terms of a set of (possibly empty) premises (above the line) and a conclusion (below the line).

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 [28] instantiated as $\mathsf{tell}(\mathit{has}(f))$.

Rule Act forbids inconsistencies with respect to action constraints. We remark that a typical case of action constraint is $\mathit{do}(a) \rightarrow \mathit{has}(f)$, i.e. action a is subject to the presence of feature f. However, this does not necessarily mean that feature f has been explicitly installed: its presence may be derived as a consequence of further constraints. This would be the case, for instance, of a store contaning the constraints $g \triangleright f$ and $\mathit{has}(g)$.

Rule Ask formalises the semantics of the usual $\mathsf{ask}(\cdot)$ operation as known from concurrent constraint programming [28]. 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 later on).

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

Figure 4: Initial specification of the coffee machine

Rules SEQ and PAR are standard. The former formalises the usual sequential composition, while the latter formalises an interleaving parallelism.

Example. Figure 4 shows an initial comprehensive specification of the coffee machine. The fragment F is composed by the store S and the concatenation of two processes, namely D, which specifies an initial design phase, and R, which specifies the run-time behaviour of the coffee machine.

The store S is made of two parts: some constraints derived from the requirements specification (S_1), and some contextual information and initial configurations (S_2).

The design 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 run-time operation of the coffee machine. Depending on the country it is meant for, the machine may either accept a euro or a dollar. After that, it may be subject to a sugar regulation (P_2) or not (P_3). The next step is the beverage selection and delivery, which may be followed by a ringtone (P_5) or not, after which it returns to its initial state.

$$
\begin{array}{rcl}
F' & \doteq & [S' \parallel D'; R'] \\
\\
S' & \doteq & S_1\ S_2' \\
S_2' & \doteq & \mathit{in}(\mathit{Europe}) \\
\\
D' & \doteq & \mathsf{install}(\mathit{euro}).0\ \mid \mathsf{install}(\mathit{dollar}).0 \\
 & & \ \mid \mathsf{install}(\mathit{sugar}).0 \mid \mathsf{install}(\mathit{coffee}).0 \mid \mathsf{install}(\mathit{tea}).0 \\
 & & \ \mid \mathsf{install}(\mathit{cappuccino}).0 \\
\\
R' & \doteq & (\mathit{euro}.0 + \mathit{dollar}.0); (P_2 + P_3)
\end{array}
$$

Figure 5: Final specification of the coffee machine

It is worth noting that D and R are not *pure* design 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 design 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.

In the next section, we will see that this specification has some flaws that can be spotted with our implementation in Maude. This will eventually lead to the corrected specification that follows from the modified parts depicted in Fig. 5.

4. MAUDE: AUTOMATED ANALYSES

In this section we describe some automated analysis activities supported by the implementation of our approach in Maude's formal environment.

We illustrate the use of some of the tools in what could be a typical specification and analysis life-cycle of a product family within our framework: (i) an initial constraint store (capturing the feature constraints described in the requirements) is specified and checked for consistency; (ii) a design process is specified and executed step-by-step; (iii) a consistency check is performed on all possible configurations allowed by the design process; and (iv) the product behaviour is specified and checked with respect to its requirements (that may include temporal requirements in addition to features constraints). We underline that this is only an example. The tools and techniques we illustrate can be combined and applied in many other ways.

Checking the consistency of the initial constraints. The consistency of a store is implemented by a function `consistent` that, given a constraint store, returns `true` if the store is consistent and `false` otherwise. This function can be used to check, e.g., the consistency of the initial store `S` presented in Fig. 4 as follows.

```
Maude> red in ANALYSIS-KRIPKE : inconsistency(S) .
...
result neConstraints: has(dollar) has(euro)
                      dollar * euro
```

The analysis spots the inconsistency of assuming the presence of two mutually excluding features (the euro and dollar coin slots) by reporting the subset of constraints formed by *has(dollar)*, *has(euro)* and *dollar* $\otimes$ *euro*.[2] There were,

[2] Maude uses * to denote $\otimes$.

of course others, like the presence of both $has(dollar)$ and $in(Europe)$ forbidden by $in(Europe) \to has(euro)$.

We can fix this issue and produce a new initial store S' (cf. Fig. 5) in which the installation of euro and dollar is delegated to the new design process D', cf $\mathsf{install}(euro).0$ and $\mathsf{install}(dollar).0$. Indeed, the latter will not be executed, since it would make the store inconsistent.

We can verify the consistency of S' as follows:

```
Maude> red in ANALYSIS-KRIPKE : consistent(S') .
...
result Bool: true
```

The result confirms that the initial store `S'` is consistent.

Executing the design process. Starting from a consistent store, the user may want to specify and execute a design process that ends up with a maximally configured product. Consider for instance the initial store `S'` and the new design process `D'` presented in Fig. 4.

The Maude command `rew` can be used to execute the fragment `[ S' | D' ]` as follows.

```
rewrite in ANALYSIS-KRIPKE : ! [S' | D'] .
...
result KFragment: ! [has(dollar)  has(coffee)
has(tea) has(cappuccino) has(sugar) ... | 0]
```

The fragment runs until the underlying process becomes the empty process resulting in a product configured with several features (for reading purposes, the part of the store that has not changed is abbreviated with `...`). Clearly, such a configuration is ensured to be consistent since it was derived from a consistent store.

Checking the consistency of all configurations. Of course, even if we are ensure the semantics of FLan preserves consistency we can use the reachability command `reach` to check consistency of all reachable configurations as follows.

```
search in ANALYSIS-KRIPKE : ! [S' | D'] =>*
x:KFragment such that consistent(x:KFragment) ==
false = true .
...
No solution
```

The absence of solutions ensures that no inconsistent configuration is reachable.

Checking behavioural properties. After fixing the specification of the design we can analyse the run-time behaviour of the product. We can now check, for instance, that the run-time behaviour does not introduce inconsistencies. We can due it as before but we can also resort to the LTL model checker of Maude. The property we check is `[] isConsistent`, i.e. consistency is an invariant.

```
Maude> red in ANALYSIS-KRIPKE : modelCheck( ( ! [ S'
| D' ; R ] ) , [] isConsistent ) .
...
result Bool: true
```

The results confirms that consistency is preserved during the run-time operation of the coffee machine.

We may however notice that the conditional statement used to accept a dollar or a euro is actually redundant due to the introduced constraints. A possible, simpler run-time process is R' (cf. Fig. 5). It is very much like R, but the conditional statement has been replaced by 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 $do(dollar) \to has(dollar)$ in the store, which will forbid the use of the actions *euro* or *dollar* if the corresponding feature has not been installed. This time, contrary to what we did earlier for the initial store and design process, we are replacing procedural information by declarative information. The resulting process enjoys the property of preserving consistency, which can be checked as follows.

```
Maude> red in ANALYSIS-KRIPKE : modelCheck( ( ! [ S'
| D' ; R' ] ) , [] isConsistent ) .
...
result Bool: true
```

The results confirms that consistency is still preserved during the run-time operation of the coffee machine.

The LTL model checker can of course be used to check additional requirements. For instance, we can check the temporal requirement 4 of our case study (i.e. *"a ringtone must be rung after serving a cappuccino"*).

```
Maude> red in ANALYSIS-LTS : modelCheck( ( !
({do('machine)}[S' | D' ; R']) ) , [] ({cappuccino}
-> <> {ringtone}) ) .
...
result Bool: true
```

The results confirms that a ringtone eventually follows (the delivery of) a cappuccino.

The above example analyses illustrate how the implementation of FLan in Maude allows us to exploit Maude's rich analysis toolset. In this respect, it is worth noting that in the above analyses we have made use of only a limited number of Maude tools, namely its SAT solver, its reachability analyser and its LTL model checker. There are several other Maude tools whose use remains to be investigated.

5. RELATED WORK

There is an increasing body of research on how to successfully apply automated behavioural verification techniques, like model checking, in the particular context of (software) product families. The challenge, to the best of our knowledge first recognised in [22, 23], is to develop formal and modular modelling and verification approaches which specifically take cross-cutting feature constraints into account. In this section, we discuss a number of formal methods and analysis techniques that have been applied in SPLE.

There are two well-known lines of research on modelling product families in terms of extensions of LTSs, which both define family behaviour as actions (features) and use advanced model-checking techniques for the verification of behavioural properties. One makes use of extensions of Modal Transition Systems (MTSs) [12, 18, 20, 3], the other of Featured Transition Systems (FTSs) [10].

Modal Transition Systems. MTSs [19] were recognised as a suitable behavioural model for describing product families in [12]. A fixed-point algorithm, implemented in a tool, is defined to check whether an LTS conforms to an MTS with respect to several different branching relations. In the context of SPLE, it allows to check the conformance of the behaviour of a product against that of its product family.

VMC (`http://fmt.isti.cnr.it/vmc/`) [5, 6] is a tool for modelling and analysing behavioural variability in product families modelled as MTSs [3]. VMC thus accepts a product family specified as an MTS, possibly with additional variability constraints, after which it allows the user to interactively explore this MTS; efficiently model check properties (branching-time temporal logic formulae) over an MTS; visualise the (interactive) explanations of a verification result; automatically generate one, some, or all of the family's valid products (represented as LTSs); browse and explore these; efficiently model check whether or not products (one, some, or all) satisfy certain properties; and, finally, help the user to understand why a certain valid product does or does not satisfy specific verified properties, by allowing such a product to be inspected individually.

Featured Transition Systems. An FTS [10] is a doubly labelled transition system with an associated feature diagram. Its states are labelled with atomic propositions, while a specific distinction among its transitions is obtained by an edge-labelling defining which transitions refer to which features.

SNIP [9] is a model checker for product families modelled as FTSs specified in a language based on that of the SPIN model checker (`http://spinroot.com/`). Features are declared in the Text-based Variability Language (TVL) and are taken into account by the explicit-state model-checking algorithm of SPIN for verifying properties expressed in fLTL (feature LTL) interpreted over FTSs (e.g. to verify a property over only a subset of the set of all valid products). Exhaustive model-checking algorithms (which continue their search also after a violation was found) moreover allow to verify all products of a family at once and to output all of the products that violate a property. Unlike VMC, SNIP is a command-line tool without a GUI. SNIP, however, treats features as first-class citizens, with built-in support for feature diagrams, and it implements model-checking algorithms specifically tailored for product families.

In this paper, we proposed to specify product families in a high-level formal process-algebraic language, FLan, which has transition systems as semantic domain. While, in principle, product family behaviour could be directly specified using transition systems from a practical point of view it is more convenient to resort to some more intuitive linguistic formalism. In fact, when used as a specification formalism, transition systems are too low level and, above all, suffer from the lack of compositionality—in the sense that they offer no means for constructing the transition system of a (sub)family in terms of that of its components. On the contrary, the process-algebraic linguistic terms offered by FLan are more intuitive and concise notations. Using them, product families can be built in a compositional way.

Like the approach based on FTSs, we thus use a high-level language for modelling, treating features as first-class citizens, and a transition system semantics for analysis. While we currently use Maude for the automated verification of behavioural properties of product families specified in FLan, in the future we hope to make their semantic models (LTSs, basically) amenable to model checking with VMC. FLan is loosely inspired by the CCS-like process algebra CL4SPL presented in [13]. Unlike FLan, however, CL4SPL has no language constructs for the cross-tree constraints known from feature models nor a store of constraints to separate the declarative aspects of a product family from its procedural aspects.

Feature-aware verification. Tool suite SPLverifier [1] uses standard off-the-shelf model-checking techniques to verify the absence of feature interactions by means of an approach called feature-aware verification. To this aim, the AutoFeature automata language for specifying features in separate and composable units was developed, while a variant of abstract syntax trees, called Feature Structure Trees (FSTs), forms the basis for encoding the variability. SPLverifier offers two methods: a brute-force one generates and verifies all valid products, while an alternative one avoids the generation of all individual products as it verifies all possible feature combinations on a single product that is purpose-built to contain all the family's features. Like SNIP and FLan, features are central to SPLverifier, but only the (renowned) problem of detecting feature interactions is addressed. Unlike VMC, SNIP and FLan, behavioural variability is not considered.

Process-algebraic approaches. A process-algebraic theory for the modelling and analysis of product families was developed also in [15, 16, 21]. PL-CCS extends CCS by a variant operator that allows to model alternative behaviour in the form of alternative processes, with the meaning that only one of the alternative processes will exist at run-time. PL-CCS has an SOS semantics defined over multi-valued MTSs. To reason on the behaviour of product families specified in PL-CCS, a multi-valued version of the modal μ-calculus is defined, i.e. the interpretation of a logic formula over a product family no longer yields true or false, but rather a set of configurations characterising exactly those products of the family which satisfy the behavioural property under verification. Unlike FLan, PL-CCS however does not cater for the cross-tree constraints known from feature models. Also, the analysis is limited to verification by model checking which is moreover not implemented.

Petri net-based approaches. The same idea underlying FTSs, namely to explicitly label the transitions of an LTS with the set of features (i.e. products) for which the transition is available, was also applied to Petri nets in [25, 26], resulting in feature (Petri) nets. Larger feature nets can be constructed from smaller ones to model the addition of new features to a product family, while correctness criteria can ensure that the resulting composition preserves the original behaviour. An extension can capture the dynamic reconfiguration of products by associating to each transition of a feature net also an update expression that describes how the feature selection evolves after firing (executing) the transition. The resulting feature reconfiguration model may remain disconnected from the ordinary behavioural model, thus offering orthogonality but at the same time allowing the reconfiguration to depend upon the underlying behaviour and vice versa. This has some similarities with the combination of declarative and procedural views that is at the heart of FLan. Efficient formal analysis and verification techniques from Petri nets of course become available to feature nets, but their application in the specific context of product families has not yet been studied.

In [30], FTSs are translated into so-called adaptable featured Petri nets, after which projection and reachability techniques from Petri nets become available for product derivation and liveness analysis.

Other approaches. In [17], FTSs (including their associated feature diagrams) are translated into Maude specifications by graph transformation. Starting from a set of requirements, this means that first a feature diagram needs to be extracted (to model the variability) and only then the desired run-time behaviour can be specified (as an FTS). FLan, on the contrary, allows to combine the specification of design and run-time processes directly from a given set of requirements, which may be very convenient, for instance to specify the behaviour of partially configured or run-time configurable products. Another difference is that the semantic foundation of our approach is based on techniques from concurrent constraint programming and process algebras rather than graph transformation.

In [14], a feature-oriented approach to modelling product families in Event-B by means of a chain of refinements is explored by applying existing Event-B (de)composition techniques to two case studies, using a prototypical feature composition tool. Behavioural variability is not considered, but it would be interesting to explore the feasibility of using this Feature Event-B as a high-level specification language on top of one of the aforementioned semantic models.

6. CONCLUDING REMARKS

We have introduced the feature-oriented language FLan as a proof of concept for specifying and analysing both declarative and procedural aspects of product families.

We do not envisage FLan to become *the* feature-oriented language, but we advocate that some of its features are very convenient and may be adopted by existing languages.

First, we think that the concurrent constraint programming paradigm 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 where features may be added while the system operates. Furthermore, the run-time specification can be discharged from design decisions such as feature constraints thus resulting in light-weight, understandable specifications.

Second, the implementation of FLan in Maude allows one to exploit the rich analysis toolset of this framework. In this paper, we have essentially restricted ourselves to its SAT solver, its reachability analyser and its LTL model checker. However, there are other Maude tools whose use may be worth investigating. The statistical model checker PVeSta, for instance, could be used for evaluating the performance of product families in variants of FLan with stochastic and quantitative aspects.

7. FUTURE WORK

We envisage several potentially interesting extensions of FLan. For one, we can adopt further primitives and mechanisms from the concurrent constraint programming tradition. The concurrent constraint π-calculus [8], for instance, 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). Such features have been shown successful for the specification of service level agreements and negotiation processes [7]. This may thus turn out to be useful when product families are to be designed by cooperating partners and are hence subject to negotiation mechanisms.

Another promising line of research is to provide an FTS and an MTS semantics of FLan so that (i) FLan becomes a high-level language for those semantic models and (ii) we can exploit the specialised analysis tools developed for them.

8. ACKNOWLEDGEMENTS

Maurice ter Beek conducted his work while on sabbatical leave at Leiden University. He gratefully acknowledges the hospitality and support during his stay in Leiden.

Maurice ter Beek and Alberto Lluch Lafuente were supported by the Italian Ministry of Instruction, University and Research project CINA (PRIN 2010LHT4KM) and by the European project QUANTICOL (FP7 600708).

Alberto Lluch Lafuente is also supported by the European project ASCENS (FP7 257414).

Marinella Petrocchi acknowledges the European projects NESSoS (FP7 256980) and Aniketos (FP7 257930).

9. REFERENCES

[1] S. Apel, H. Speidel, P. Wendler, A. von Rhein, and D. Beyer. Detection of feature interactions using feature-aware verification. In *ASE'11*. IEEE, 2011, 372–375.

[2] P. Asirelli, M. H. ter Beek, A. Fantechi, and S. Gnesi. A Logical Framework to Deal with Variability. In *IFM'10*, *LNCS* 6396. Springer, 2010, 43–58.

[3] P. Asirelli, M. H. ter Beek, A. Fantechi, and S. Gnesi. Formal Description of Variability in Product Families. In *SPLC'11*. IEEE, 2011, 130–139.

[4] M. H. ter Beek, A. Lluch Lafuente, M. Petrocchi. Combining Declarative and Procedural Views in the Specification and Analysis of Product Families Submitted to *FMSPLE'13*. Draft available at `http://eprints.imtlucca.it/1589/`

[5] M. H. ter Beek, S. Gnesi, and F. Mazzanti. Demonstration of a model checker for the analysis of product variability. In *SPLC'12*. ACM, 2012, 242–245.

[6] M. H. ter Beek, F. Mazzanti, and A. Sulova. VMC: A Tool for Product Variability Analysis. In *FM'12*, *LNCS* 7436. Springer, 2012, 450–454.

[7] M. G. Buscemi and U. Montanari. QoS negotiation in service composition. *Journal of Logic and Algebraic Programming* 80, 1 (2011), 13–24.

[8] M. G. Buscemi and U. Montanari. CC-Pi: A Constraint-Based Language for Specifying Service Level Agreements. In *ESOP'07*, *LNCS* 4421. Springer, 2007, 18–32.

[9] A. Classen, M. Cordy, P. Heymans, A. Legay, and P.-Y. Schobbens. Model Checking Software Product Lines with SNIP. *Software Tools for Technology Transfer* 14, 5 (2012), 589–612.

[10] A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, and J.-F. Raskin. Model Checking Lots of Systems. In *ICSE'10*. ACM, 2010, 335–344.

[11] M. Clavel *et al.*, Eds. *All About Maude*, *LNCS* 4350. Springer, 2007.

[12] D. Fischbein, S. Uchitel, and V. A. Braberman. A Foundation for Behavioural Conformance in Software Product Line Architectures. In *ROSATEA'06*. ACM, 2006, 39–48.

[13] S. Gnesi and M. Petrocchi. Towards an Executable Algebra for Product Lines. In *FMSPLE'12*. ACM, 2012, 66–73.

[14] A. Gondal, M. Poppleton, and M. Butler. Composing Event-B Specifications - Case-Study Experience. In *SC'11*, *LNCS* 6708. Springer, 2011, 100–115.

[15] A. Gruler, M. Leucker, and K. D. Scheidemann. Modeling and Model Checking Software Product Lines. In *FMOODS'08*, *LNCS* 5051. Springer, 2008, 113–131.

[16] A. Gruler, M. Leucker, and K. D. Scheidemann. Calculating and Modeling Common Parts of Software Product Lines. In *SPLC'08*. ACM, 2008, 203–212.

[17] K. Khalfaoui, A. Chaoui, C. Foudil, and E. Kerkouche. Formal Specification of Software Product Lines: A Graph Transformation Based Approach. *Journal of Software* 7, 11 (2012), 2518–2532.

[18] K. G. Larsen, U. Nyman, and A. Wąsowski. Modal I/O Automata for Interface and Product Line Theories. In *ESOP'07*, *LNCS* 4421. Springer, 2007, 64–79.

[19] K. G. Larsen and B. Thomsen. A modal process logic. In *LICS'88*. IEEE, 1988, 203–210.

[20] K. Lauenroth, K. Pohl, and S. Töhning. Model Checking of Domain Artifacts in Product Line Engineering. In *ASE'09*. IEEE, 2009, 269–280.

[21] M. Leucker and D. Thoma. A Formal Approach to Software Product Families. In *ISoLA'12*, *LNCS* 7609. Springer, 2012, 131–145.

[22] H. C. Li, S. Krishnamurthi, and K. Fisler. Verifying Cross-Cutting Features as Open Systems. In *FSE'02*. ACM, 2002, 89–98.

[23] H. C. Li, S. Krishnamurthi, and K. Fisler. Modular Verification of Open Features Using Three-Valued Model Checking. *Automated Software Engineering* 12 (2005), 349–382.

[24] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[25] R. Muschevici, D. Clarke, and J. Proença. Feature Petri Nets. In *FMSPLE'10*. University of Lancaster, 2010.

[26] R. Muschevici, J. Proença, and D. Clarke. Modular Modelling of Software Product Lines with Feature Nets In *SEFM'11*, *LNCS* 7041. Springer, 2011, 318–333.

[27] G. D. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming* 60–61 (2004), 17–139.

[28] V. A. Saraswat and M. C. Rinard. Concurrent Constraint Programming. In *POPL'90*. ACM, 1990, 232–245.

[29] P.-Y. Schobbens, P. Heymans, and J.-C. Trigaux. Feature Diagrams: A Survey and a Formal Semantics. In *RE'06*. IEEE, 2006, 136–145.

[30] H. Zhang, H. Zou, F. Yang, and R. Lin. Modeling and Analysis of Behavioral Variability in Product Lines. *Journal of Information & Computational Science* 9, 12 (2012), 3589–3600.

[31] T. Ziadi and J. M. Jézéquel. Software Product Line Engineering with the UML: Deriving Products. In *SPLC'06*. Springer, 2006, 557–588.

APPENDIX

Implementation

This section documents the complete specification of our implementation of FLAN in Maude.

```
load model-checker.maude .

fmod FLAN-FEATURES is

  inc QID .
  including SATISFACTION .


  sort Feature .
  subsort Qid < Feature . --- We use quoted
       identifiers as features

  --- Actions
  sorts Action InstallAction OtherAction .
  subsort InstallAction OtherAction < Action .
  subsort Feature < OtherAction . --- Let us use
       features as actions

  op install : Feature -> InstallAction .
  op ask : Prop -> InstallAction .

  op do : Action -> Prop .    --- Predicate stating
       that fragment does an action
  op has : Feature -> Prop .  --- Predicate stating
       that fragment has the feature

endfm


fmod FLAN-CONSTRAINTS is

  pr FLAN-FEATURES .
  pr SAT-SOLVER .

  sorts Constraint Constraints neConstraints .
  subsort Constraint < neConstraints < Constraints .
  subsort Formula Prop < Constraint .

  --- Constraints over features
  op _ * _ : Feature Feature -> Prop [ctor comm] .
  op _ |> _ : Feature Feature -> Prop [ctor].

  --- Constraints set operators
  op _ _ : neConstraints Constraints  ->
       neConstraints [assoc comm] .
  op _ _ : Constraints neConstraints  ->
       neConstraints [assoc comm] .
  op _ _ : Constraints Constraints  -> Constraints [
       assoc comm] .

  vars Cons Cons' : Constraints .
  vars neCons : neConstraints .
  vars c1 c2  : Constraint .
  vars f g : Feature .
  vars formula1 : Formula .
  vars oa1 : OtherAction .

  --- Id of set union
  eq neCons True = neCons .

  --- Idempotency of set union
  eq neCons neCons = neCons .

  --- Some basic entailment reductions
  --- Since entailment is expensive, we declare the
       operator as "memo" to memoize results.
  op _ |= _ : Constraints Constraint -> Bool [memo] .

  --- Some trivial cases
  eq (neCons c1 |= c1) = true .
  eq (c1 |= c1) = true .
  eq (neCons (~ c1) |= c1) = false .
  eq ( (~ c1) |= c1) = false .
  --- Entailment via SAT
  ceq (Cons |= c1 ) = true
   if (satSolve(store2sat(Cons) /\ ~ c1) == false) .
  --- Default case
  eq (Cons |= c1) = false [owise] .

  --- A procedure for checking some local
      inconsistencies
  op check : Constraints Constraint -> Bool .
  eq check(True,c1) = true .
  --- Checking of for installation of new features (
      wrt. to "exclude" constraints only)
  eq check((f * g) has(f) Cons, has(f)) = false .
  eq check((f * g) has(f)     , has(f)) = false .
  eq check(Cons, has(f)) = true [owise] .
  --- Checking all other actions (wrt. to explicit
      constraints of the form oa1 -> ...)
  ceq check(Cons ((~ do(oa1)) \/ formula1), do(oa1))
      = false
   if (Cons |= formula1) =/= true .
  eq check(      ((~ do(oa1)) \/ formula1), do(oa1))
      = false .
  eq check(Cons, do(oa1)) = true [owise] .

  --- A procedure to check consistency with respect
      to feature constraints
  --- Other boolean inconsistencies are ignored (
      assumed to not exist)
  op consistent : Constraints -> Bool .
  eq consistent(Cons) = (inconsistency(Cons) == True)
        .

  --- A simple procedure to check and find cross-tree
       feature inconsistencies
  --- Other boolean inconsistencies are ignored (
      assumed to not exist)
  op inconsistency : Constraints -> Constraints .
  eq inconsistency(True) = True .
  eq inconsistency((f * g) has(f) has(g) Cons) = ((f
      * g) has(f) has(g)) .
  eq inconsistency((f * g) has(f) has(g)     ) = ((f
      * g) has(f) has(g)) .
  ceq inconsistency((f |> g) has(f) Cons) = ((f |> g)
       has(f))
   if Cons has(g) =/= Cons .
  eq inconsistency((f |> g) has(f)     ) = ((f |> g)
      has(f)) .
  --- Simplifications
  eq inconsistency((f |> g) has(g) Cons) =
      inconsistency(has(g) Cons) .
  eq inconsistency((f |> g) has(g)     ) =
      inconsistency(has(g)) .
  ceq inconsistency((f |> g) Cons) = inconsistency(
      Cons)
   if Cons has(f) =/= Cons .
  --- Default case true (all inconsistencies captured
       above)
  eq inconsistency(Cons) = True [owise] .

  --- A full consistency procedure (not only feature
      inconsistencies)
  op fully-consistent : Constraints -> Bool .
  --- We reduce the problem to SAT and use Maude's
      SAT solver
  eq fully-consistent(Cons) = ( satSolve(store2sat(
      Cons)) =/= false) .

  --- This functions essentially replaces constraint
      union with boolean conjunction
  op store2sat : Constraints -> Formula .
  eq store2sat(True) = True .
  eq store2sat(False) = False .
  eq store2sat( f * g ) = has (f) <-> ~ has(g) .
  eq store2sat( f |> g ) = has (f) -> has(g) .
  eq store2sat(c1) = c1 .
  eq store2sat(c1 neCons) = c1 /\ store2sat(neCons) .

endfm


fmod FLAN-SYNTAX is

  inc FLAN-CONSTRAINTS .
```

```
  --- Fragments
  sort Fragment . --- Syntactic category FT in Fig. 1

  op [ _ | _ ]: Constraints Process -> Fragment [ctor
      frozen] .

  --- Processes
  sort Process .
  sort ProcessId . --- Vocabulary A in Fig. 1
  subsorts Qid < ProcessId . --- Fragment ids are
      quoted identifiers.
  subsort ProcessId < Process . --- Process Ids are
      processes

  --- Some structural axioms (idempotency and
      identity are handled with equations)
  vars P Q : Process .
  vars K : Prop .

  op 0 : -> Process [ctor] .
  op _ . _ : Action Process -> Process [ctor frozen
      prec 10 gather (e E) ] .
  op _ + _ : Process Process -> Process [assoc comm
      frozen prec 20 gather (E e) ] .
  op _ ; _ : Process Process -> Process [assoc frozen
      prec 20 gather (E e) ] .
  op _ | _ : Process Process -> Process [assoc comm
      frozen prec 20 gather (E e) ] .

  --- Derived operators
  op if _ then _ else _ fi : Formula Process Process
      -> Process [ctor frozen prec 15 gather (E E E)]
      .
  eq if K then P else Q fi = (ask(K) . P) + (ask(~ K)
      . Q) .

  eq P + 0 = P .
  eq P + P = P .
  eq P ; 0 = P .
  eq 0 ; P = P .
  eq P | 0 = P .

endfm


fmod FLAN-RECURSION is

  inc FLAN-SYNTAX .

  sorts ProcessDefinitions .

  vars PId1 PId2 : ProcessId .
  var P : Process .
  vars PD1 PD2 : ProcessDefinitions .

  --- We assume a global set of process definitions
  --- For the sake of simplicity
  --- "specification" is to be defined for each
      example
  op specification : -> ProcessDefinitions .

  op _=def_ : ProcessId Process -> ProcessDefinitions
      [ctor frozen prec 40] .

  op noProcessDefinition : -> ProcessDefinitions .
  op _ _ :  ProcessDefinitions ProcessDefinitions ->
      ProcessDefinitions [assoc comm id:
      noProcessDefinition prec 42] .

  --- Function to determine whether a process id is
      defined
  op _ definedIn _ : ProcessId ProcessDefinitions ->
      Bool .

  eq PId1 definedIn noProcessDefinition = false .
  eq PId1 definedIn ( (PId1 =def P) PD1) = true .
  eq PId1 definedIn PD1 = false [owise] .

  op def : ProcessId ProcessDefinitions -> [Process]
      .

  eq def(PId1, (PId1 =def P)) = P .
  ceq def(PId1, (PId1 =def P) PD2) = P
    if PD2 =/= noProcessDefinition .

endfm


---  transitions
mod FLAN-SEMANTICS is

  pr FLAN-SYNTAX .
  pr FLAN-RECURSION .

  --- The implementation of the SOS semantics follows
  --- the Verdejo&Oliet approach

  --- Labelled fragments are used to encode labelled
      transitions
  sort LabelledFragment .
  subsort Fragment < LabelledFragment .

  --- Labelling operator
  sort Label .
  op {_} _ : Label LabelledFragment ->
      LabelledFragment [ctor frozen] .

  --- Label constructors
  subsort Action < Label . --- Just use actions as
      labels

  vars f : Feature .
  vars act1 : Feature .
  vars a b c : Label .
  vars P P' Q Q' : Process .
  vars Cons Cons' Cons'' : Constraints .
  vars LabF LabF' : LabelledFragment .
  vars F F' : Fragment .
  vars K : Prop .
  vars PId1 PId2 : ProcessId .
  vars oa1 : OtherAction .

  --- Rule Install in Fig. 2
  crl [Install] : [ Cons | install(f) . P ] => {
      install(f)} [ Cons' | P ]
   if consistent(Cons has(f)) /\ Cons' := Cons has(f)
      .

  --- Rule Act in Fig. 2
  crl [Ask] : [ Cons | ask(K) . P ] => {'ask} [ Cons
      | P ]
   if Cons |= K .

  --- Rule Act in Fig. 2
  crl [Act] : [ Cons | oa1 . P ] => {oa1} [ Cons | P
      ]
   if check(Cons,do(oa1)) .

  --- Rule Or in Fig. 2
  crl [Or] : [ Cons | (P + Q) ] => {a} [ Cons' | P' ]
   if [ Cons | P ] => {a} [Cons' | P'] .

  --- Rule Seq in Fig. 2
  crl [Seq] : [ Cons | (P ; Q) ] => {a} [ Cons' | (P'
      ; Q) ]
   if [ Cons | P ] => {a} [Cons' | P'] .

  --- Rule Par in Fig. 2
  crl [Par] : [ Cons | (P | Q) ] => {a} [ Cons' | (P'
      | Q) ]
   if [ Cons | P ] => {a} [Cons' | P'] .

  --- This is only a test.
---   crl [ParC] : [ Cons | (P | Q) ] => {install(f)}
     [ Cons' Cons'' | (P' | Q') ]
---    if [ Cons | P ] => {install(f)} [Cons' | P'] /\
      [ Cons | Q ] => {install(g)} [Cons'' | Q']


  --- Auxiliary rules to expand definitions when
      needed
  crl [def] : [Cons | PId1 ] => {a} [Cons' | P]
   if (PId1 definedIn specification)
   /\ [Cons | def(PId1,specification) ] => {a} [Cons'
      | P ] .
```

```
  --- A function to check the consistency of a
       Fragment
  op consistent : LabelledFragment -> Bool .
  eq consistent({a} LabF) = consistent(LabF) .
  eq consistent([Cons | P]) = consistent(Cons) .

endm

mod FLAN-TRACES is

  vars a b c : Label .
  vars P P' Q Q' : Process .
  vars Cons Cons' : Constraints .
  vars LabF LabF' : LabelledFragment .
  vars F F' : Fragment .

  pr FLAN-SEMANTICS .

  sort TracedFragment .
  subsort LabelledFragment < TracedFragment .
  op !_ : TracedFragment -> TracedFragment [frozen] .

  crl [refl] : ! F => ! ({a} F')
   if F => {a} F' .

  crl [tran] : ! ({a} LabF) => ! ({b} ({a} LabF') )
   if ! LabF => ! ({b} LabF') /\ LabF =/= {b} LabF' .

   --- A function to check the consistency of a
       Fragment
  op consistent : TracedFragment -> Bool .
  eq consistent(! LabF) = consistent(LabF) .

endm


mod FLAN-KRIPKE is

  pr FLAN-SEMANTICS .

  sort KFragment .

  op !_ : Fragment -> KFragment [frozen] .

  vars a b c : Label .
  vars F F' : Fragment .

  crl ! F => ! F'
   if F => {a}F' .

   --- A function to check the consistency of a
       Fragment
  op consistent : KFragment -> Bool .
  eq consistent(! F) = consistent(F) .

endm

mod FLAN-LTS is

  pr FLAN-SEMANTICS .

  sort KFragment .

  op !_ : LabelledFragment -> KFragment [frozen] .

  vars a b c : Label .
  vars F F' : Fragment .

  crl ! ({a} F) => ! ({b}F')
   if F => {b}F' .

   --- A function to check the consistency of a
       Fragment
  op consistent : KFragment -> Bool .
  eq consistent(! ({a}F)) = consistent(F) .

endm

mod FLAN-KRIPKE-PREDS is

    protecting FLAN-KRIPKE .
    including SATISFACTION .
    subsort KFragment < State .

    op isConsistent : -> Prop .

    vars F : Fragment .

    eq ! F |= isConsistent = consistent(! F) .

endm


mod FLAN-LTS-PREDS is

    protecting FLAN-LTS .
    including SATISFACTION .
    subsort KFragment < State .

    op isConsistent : -> Prop .

    vars a b c : Label .
    vars F : Fragment .

    eq (! ({a} F) |= isConsistent) = consistent(! ({a
        } F)) .

    op { _ } : Label -> Prop [ctor] .
    eq ! ({a}F) |= {a} = true .
    eq ! ({a}F) |= {b} = false [owise] .

endm


mod FLAN-LTS-CHECK is

    protecting FLAN-LTS-PREDS .
    including MODEL-CHECKER .
    including LTL-SIMPLIFIER .

endm

mod FLAN-KRIPKE-CHECK is

    protecting FLAN-KRIPKE-PREDS .
    including MODEL-CHECKER .
    including LTL-SIMPLIFIER .

endm


mod FLAN-COFFE-MACHINE is

  pr FLAN-SEMANTICS .

  sort Region .
  ops Europe Canada : -> Region .

  sort Currency .
  ops dollar euro : -> Currency .

  sort Product .
  ops coffee tea cappuccino : -> Product .

  subsort Product Currency < Feature .

  --- Other features
  ops machine ringtone sugar : -> Feature .

  op in : Region -> Prop .

  --- Example from the paper, section 5
  ops F F' FD FR : -> Fragment .
  eq F   = [ S | D  ; R ] .
  eq F'  = [ S' | D' ; R' ] .
  eq FD  = [ S | D ] .

  ops S S' S1 S2 S2' : -> Constraints .
  eq S = S1 S2 .
  eq S' = S1 S2' .
  eq S1 =
--- either a euro, or a dollar
         ( dollar * euro )
--- at least one of euro or dollar
         ( has(euro) \/ has(dollar) )
```

```
--- euro, exclusively for products for the European
    market
          ( in(Europe) -> has(euro) )
--- dollar, exclusively for Canadian products
          ( in(Canada) -> has(dollar) )
--- every product must offer at least one beverage
          ( has(coffee) \/ has(cappuccino) \/ has(tea
              ) )
--- tea may be offered only by European product
          ( has(tea) -> in(Europe) )
--- all products that offer cappuccino must also
    offer coffee
          ( cappuccino |> coffee )
--- standard do(feature) -> has(feature)
          ( do(euro)       -> has(euro) )
          ( do(dollar)     -> has(dollar) )
          ( do(sugar)      -> has(sugar) )
          ( do(coffee)     -> has(coffee) )
          ( do(cappuccino) -> has(cappuccino) )
          ( do(tea)        -> has(tea) )
          ( do(ringtone)   -> has(ringtone) ) .

--- some contextual information
  eq S2 =
          ( in(Europe) ) ( ~ in(Canada) )
          ( has(euro) )
          ( has(dollar) ) .
  eq S2' =
          ( in(Europe) ) ( ~ in(Canada) ) .

  ops D D' R R' P1 P2 P3 P4 P5 : -> Process .

  eq D =  install(sugar) . 0 |
          install(coffee) . 0 |
          install(tea) . 0 |
          install(cappuccino) . 0 |
          install(euro) . 0 |
          install(dollar) . 0 .

  eq D' =  install(sugar) . 0 |
           install(coffee) . 0 |
           install(tea) . 0 |
           install(cappuccino) . 0 |
           ( (ask(in(Europe)) . install(euro) . 0) +
                (ask(in(Canada)) . install(dollar) .
                0) ) .

  eq R  =  ( (ask(in(Europe)) . euro . 0) + (ask(in(
      Canada)) . dollar . 0) ) ; 'P2 .

  eq R' = (euro . 'P2) + (dollar . 'P2) .
  eq P2 = sugar . 'P3 .
  eq P3 = (coffee . 'P4) + (tea . 'P4) + (cappuccino
      . 'P5) .
  eq P4 = 'P5 + 'R .
  eq P5 = install(ringtone) . ringtone . 'R .

  eq specification = ( 'D  =def D  )
                     ( 'D' =def D' )
                             ( 'R  =def R  )
                             ( 'R' =def R' )
                     ( 'P1 =def P1 )
                     ( 'P2 =def P2 )
                     ( 'P3 =def P3 )
                     ( 'P4 =def P4 )
                     ( 'P5 =def P5 ) .

endm


mod ANALYSIS-KRIPKE is

  pr FLAN-COFFE-MACHINE .

  pr FLAN-KRIPKE-CHECK .

endm

mod ANALYSIS-LTS is

  pr FLAN-COFFE-MACHINE .

  pr FLAN-LTS-CHECK .

endm

--- Commands exemplified in the paper

--- red in ANALYSIS-KRIPKE : consistent(S) .
--- red in ANALYSIS-KRIPKE : inconsistencies(S) .
--- red in ANALYSIS-KRIPKE : consistent(S') .
--- rew in ANALYSIS-KRIPKE : ! [ S' | D' ] .
--- search [1] in ANALYSIS-KRIPKE :  ! [ S' | D' ]
    =>* x:KFragment such that consistent(x:KFragment)
     == false .
--- red in ANALYSIS-KRIPKE : modelCheck( ( ! [ S' | D
    ' ; R' ] ) , ([] isConsistent) ) .
--- red in ANALYSIS-KRIPKE : modelCheck( ( ! [ S' | D
    ' ; R' ] ) , ([] isConsistent) ) .
--- red in ANALYSIS-LTS : modelCheck( ( ! ({'machine
    }[S' | D' ; R']) ) , [] ({cappuccino} -> <> {
    ringtone}) ) .
```