Using FMC for Family-Based Analysis of Software Product Lines

Maurice H. ter Beek
ISTI–CNR, Pisa, Italy
terbeek@isti.cnr.it

Alessandro Fantechi
DSI, Università di Firenze
and ISTI–CNR, Pisa, Italy
fantechi@dsi.unifi.it

Stefania Gnesi
ISTI–CNR, Pisa, Italy
gnesi@isti.cnr.it

Franco Mazzanti
ISTI–CNR, Pisa, Italy
mazzanti@isti.cnr.it

ABSTRACT

We show how the FMC model checker can successfully be used to model and analyze behavioural variability in Software Product Lines. FMC accepts parameterized specifications in a process-algebraic input language and allows the verification of properties of such models by means of efficient on-the-fly model checking. The properties can be expressed in a logic that allows to correlate the parameters of different actions within the same formula. We show how this feature can be used to tailor formulas to the verification of only a specific subset of products of a Software Product Line, thus allowing for scalable family-based analyses with FMC. We present a proof-of-concept that shows the application of FMC to an illustrative Featured Transition System from the literature.

CCS Concepts

•General and reference → Verification; •Theory of computation → Verification by model checking; *Modal and temporal logics;* Process calculi; Operational semantics; **•Software and its engineering → Model checking; Software product lines;** *Model-driven software engineering;*

Keywords

Variability, Features, Featured Transition Systems, Process algebra, Model transformation

1. INTRODUCTION

Featured Transition Systems (FTSs) were originally introduced in [25] for the concise description of the behaviour of software product lines (SPLs). Formally, an FTS is a doubly-Labelled Transition System (L^2TS) equipped with an additional feature diagram. Each state is labelled by an atomic proposition while each transition is labelled by an action and, using the improved definition from [24], an associated *feature expression* (a Boolean formula defined over the set of features) that needs to hold for this specific transition to be part of the executable product behaviour. Hence an FTS models a family of Labelled Transition Systems (LTSs), one per product, which can be obtained by projection: all transitions whose feature expression is not satisfied by the specific product's set of features are removed, as well as all states and transitions that because of this become unreachable. In this paper, we consider a subclass of 'action-based' FTSs obtained by ignoring state labels but considering only transition labels (i.e. actions and feature expressions).

An FTS is a popular model in SPLE that comes with dedicated model checkers. FTS model checkers like SNIP [23], now integrated and reengineered in the product line of model checkers ProVeLines [26], allow efficient *family-based* analysis capable of relating errors and undesired behaviour to the exact sets of products in which they occur. Such techniques verify properties directly over an entire SPL by using variability knowledge about valid feature configurations to deduce results for products, as opposed to enumerative *product-based* analysis in which properties are verified over individually generated products (or at most a subset) [43].

In this paper, we show how the model checker FMC [32] can be used successfully to model and analyze behavioural variability in SPLs. FMC (http://fmt.isti.cnr.it/fmc) is a modelling and verification framework for the definition, exploration, analysis and model checking of system designs modelled as parallel compositions of sequential terms. The process algebra used to model a system is rooted in CCS [39], CSP [41] and LOTOS [31], and inherits from these languages its concurrent and sequential constructs. FMC accepts parameterized specifications and it supports the verification of properties expressed in a logic that specifically allows to correlate the parameters of different actions within a formula. We will see how this feature can be used to tailor formulas to the verification of a specific subset of products of an SPL, thus allowing for family-based analyses of SPLs. The complexity of verifying a formula for a subfamily obviously does not depend on the size of the rest of the family. In combination with the on-the-fly model-checking algorithm of FMC, this means that more often than not only part of the complete state space needs to be inspected, which considerably improves the scalability of our approach.

SPLC 2015, July 20 - 24, 2015, Nashville, TN, USA

ISBN 978-1-4503-3613-0/15/07…$15.00

DOI: http://dx.doi.org/10.1145/2791060.2791118

While the behaviour of an SPL can of course be directly specified in FMC, in this paper we present a technique to automatically transform an FTS into a process-algebraic model in the specific format accepted by FMC, thus paving the way for a comparison of the modelling and analysis of SPL behaviour in two different model-checking frameworks.

The paper outline is as follows. In Sect. 2 we introduce FMC in the context of the KandISTI family of model checkers. Section 3 contains our running example taken from [24]. The first contribution of this paper, the process-algebraic interpretation of FTSs, is presented in Sect. 4 on the basis of our running example. In Sect. 5, we provide for the first time the full syntax of the KTL logic accepted by KandISTI. The model resulting from Sect. 4 is used in Sect. 6 to illustrate the main contribution of this paper, the feasibility of efficient family-based analyses with FMC. Section 7 discusses related work. We conclude with future work in Sect. 8.

Note that we assume some minimal familiarity with just the basic principles of process algebras, transition systems, model checking and logics like Hennessy–Milner logic, CTL, ACTL and the modal μ-calculus [3–5,20–22,28,29,36,38,39].

2. FMC: ON-THE-FLY MODEL CHECKER

FMC is a product of the KandISTI family of model checkers [13, 33] (http://fmt.isti.cnr.it/kandisti) we developed at ISTI–CNR over the past two decades, which includes also UMC [11, 18], CMC [18, 30] and VMC [17, 19]. Each of them allows the efficient verification, by means of explicit-state on-the-fly model checking, of functional properties expressed in the action-based and state-based branching-time temporal logic KTL (defined in Sec. 5) derived from the family of logics based on ACTL [28, 29], i.e. action-based CTL.

The model checkers of KandISTI share an optimized verification engine, as a result of which tens of millions of states can now be verified in a few minutes. The on-the-fly nature of KandISTI means that often not the complete state space needs to be generated and explored. This feature improves performance and it moreover allows to partially verify also finite fragments of infinite-state systems. Model checking the (A)CTL fragment of KTL has a complexity that is at most linear with respect to the size of the state space and linear with respect to the size of the formula. KandISTI moreover offers advanced explanation techniques, such as the step-by-step illustration of counterexamples, which is particularly useful when model checking branching-time formulas.

Note that while the model checkers of the KandISTI family all share a common verification engine, the input models of the members other than FMC are rather different, due to the varying fields of application for which they were developed: UMC accepts systems specified as sets of communicating UML-like state machines, CMC accepts systems specified in a calculus for the orchestration of web services and VMC, finally, accepts process-algebraic interpretations of Modal Transition Systems (MTSs), possibly enriched with variability constraints known from SPLE.

3. RUNNING EXAMPLE

We illustrate our contribution on a well-known example: the vending machine product line from [24]. Its valid products are modelled by the feature diagram in Fig. 1, which defines a set of 12 vending machines based on the primitive features `Soda`, `Tea`, `FreeDrinks` and `CancelPurchase`.

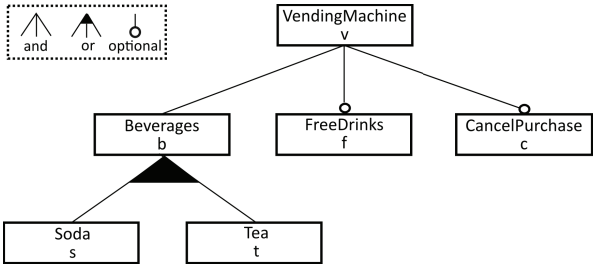

Figure 1: Feature diagram of vending machine SPL

The allowed product behavior is modelled by the L^2TS in Fig. 2, in which we have omitted the state labels since they are not considered in this paper. Ignoring its state labels, an FTS is an LTS with an associated feature diagram and a function that labels each transition with an action and an additional feature expression (i.e. a Boolean expression over the features, using the abbreviations introduced in Fig. 1). The transition $(1) \xrightarrow{\texttt{pay/v}\wedge\neg\texttt{f}} (2)$ in Fig. 2, e.g., means that payment is required/offered only in vending machines that lack the `FreeDrinks` feature.

4. FROM FTSs TO PROCESS ALGEBRA

Following [8–10, 14, 16], we model the configuration of a product separate from its actual behaviour. The breadth-first node traversal of the feature diagram of the FTS is directly translated into a process-algebraic interpretation of an LTS leading from an initial state to a final state (with no outgoing transitions). A (temporary) final state is valid if the selected features meet the constraints imposed by the feature model, otherwise it is a deadlock (sink) state.

For our example, we define a process `FMCModel` with (upto) four parameters holding abbreviations of the features included (so far). We ignore the root feature `v` and the compound feature `b`, since their only purpose is to group the (primitive) features `s` and `t`. The primitive features (next to `s` and `t` also `f` and `c`) are those that actually define user observable configuration parameters [6, 42]. The possible inclusion of an optional feature leads to a non-deterministic choice (`+`). Hence, initially, we can either include or not include `Soda`, resulting in either `ConfiguredSoda(1)` or `ConfiguredSoda(0)`, respectively, after which we need to decide upon the inclusion of `Tea`, `Free` and `Cancel`.

```
FMCModel =
   ConfiguredSoda(1) +
   ConfiguredSoda(0)

ConfigureSoda(s) =
   ConfiguredTea(s,1) +
   ConfiguredTea(s,0)

ConfiguredTea(s,t) =
   ConfigureFree(s,t,1) +
   ConfigureFree(s,t,0)

ConfiguredFree(s,t,f) =
   ConfiguredCancel(s,t,f,1) +
   ConfiguredCancel(s,t,f,0)

ConfiguredCancel(s,t,f,c) = Valid(s,t,f,c)
```

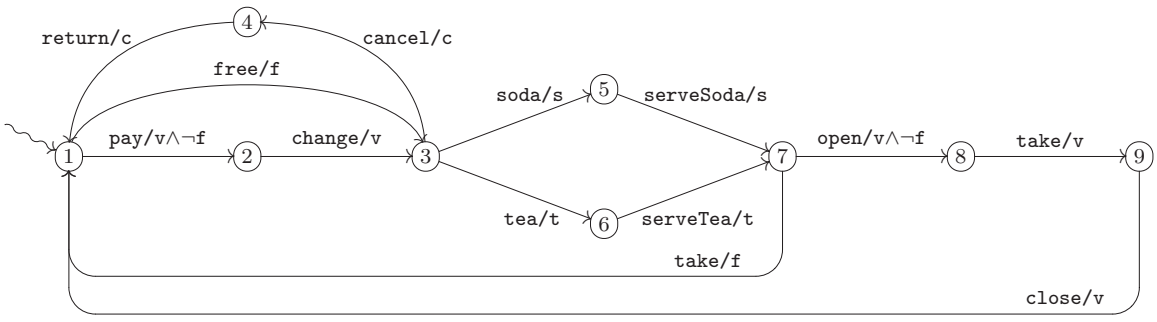

Figure 2: FTS of vending machine SPL

This procedure leads to non-deterministic choices for optional features, whereas mandatory features would result in determinism. Once all nodes in the feature diagram have been visited, a resulting feature set `Valid(s,t,f,c)` still may or may not satisfy the (cross-tree) constraints of the feature diagram. In our example, the only constraint is `Soda` or `Tea`, which is verified by means of an explicit pair of guards on these two features that require the corresponding parameter to be `0` if the feature is to be absent and `1` if it is to be present. If the constraint is satisfied, then action `checkOK(s,t,f,c)` passes the parameters to process `FTSModel(s,t,f,c)`, which models the behaviour of the configured products based on the transition system of the FTS.

```
Valid(s,t,f,c) =
   [s=0][t=1] checkOK(s,t,f,c).FTSModel(s,t,f,c) +
   [s=1][t=0] checkOK(s,t,f,c).FTSModel(s,t,f,c) +
   [s=1][t=1] checkOK(s,t,f,c).FTSModel(s,t,f,c)
```

We now define the product behaviour of our example product line in a process-algebraic setting, which can be seen as the natural encoding of the graph (FTS) of Fig. 2, with the process terms corresponding to the nodes of the graph and the guards verifying the respective feature expressions. In this paper, we will often simply speak of an FTS when we actually intend only the $L^{(2)}$TS, i.e. ignoring the feature diagram that is formally part of the definition of an FTS. Finally, we need to use `net SYS` to indicate the initial process of a process model for FMC.

```
FTSModel(s,t,f,c) = T1(s,t,f,c)

T1(s,t,f,c) =
   [f=0] pay.T2(s,t,f,c) +
   [f=1] free.T3(s,t,f,c)

T2(s,t,f,c) = change(s,t,f,c).T3(s,t,f,c)

T3(s,t,f,c) =
   [c=1] cancel.T4(s,t,f,c) +
   [s=1] soda.T5(s,t,f,c) +
   [t=1] tea.T6(s,t,f,c)

T4(s,t,f,c) = [c=1] return.T1(s,t,f,c)

T5(s,t,f,c) = [s=1] serveSoda.T7(s,t,f,c)

T6(s,t,f,c) = [t=1] serveTea.T7(s,t,f,c)

T7(s,t,f,c) =
   [f=1] take.T1(s,t,f,c) +
   [f=0] open.T8(s,t,f,c)

T8(s,t,f,c) = take.T9(s,t,f,c)

T9(s,t,f,c) = close.T1(s,t,f,c)

net SYS = FMCModel
```

Note that we simplified the feature expressions by ignoring the abstract feature `v`, which is (by definition) always present. Moreover, since the guards can only verify the presence or absence of a feature (through the Boolean value of the parameter), in general we first need to transform the feature expressions of an FTS into conjunctive normal form.

Taking the above FMCModel as input, FMC generates the L^2TS of all (valid) products of the vending machine SPL depicted in Fig. 3.

For this paper, the encoding of an FTS (and feature model) into FMC's input language, as described in this section, was performed manually. However, the procedure can easily be automated, such that SPL developers would not even have to look at or understand the encoding. In [7], an automatic technique is provided that transforms any action-based FTS (with an associated feature model) into an MTS (with additional sets of variability constraints) in the specific format accepted by the Variability Model Checker VMC [17, 19].

5. KTL: KandISTI TEMPORAL LOGIC

KTL is the common temporal logic shared by all tools of the KandISTI framework. It is the result of several years of evolution of the KandISTI tools [11, 17–19, 28, 30, 32].

Even though the various tools make use of different specification languages, the semantic model of their input specifications is uniformly seen as an L^2TS, which thus constitutes the abstract underlying model of the logic KTL.

The logic KTL includes the following rich set of features:

- Parametric state predicates (represented by the state labels of the L^2TS)
  e.g. `pred1(arg1, arg2)`, `pred2` and `pred3(*, arg3)`
- Special-purpose predefined state predicates (more on `PRINT` below, while `FINAL` is shorthand for a final state)
  e.g. `PRINT(message, arg1, arg2)` and `FINAL`
- Parametric action formulas (represented by Boolean expressions over the transitions labels of the L^2TS)
  e.g. (`act1(arg1, arg2)` or `act2`) and `not act3(arg3, *, *)`

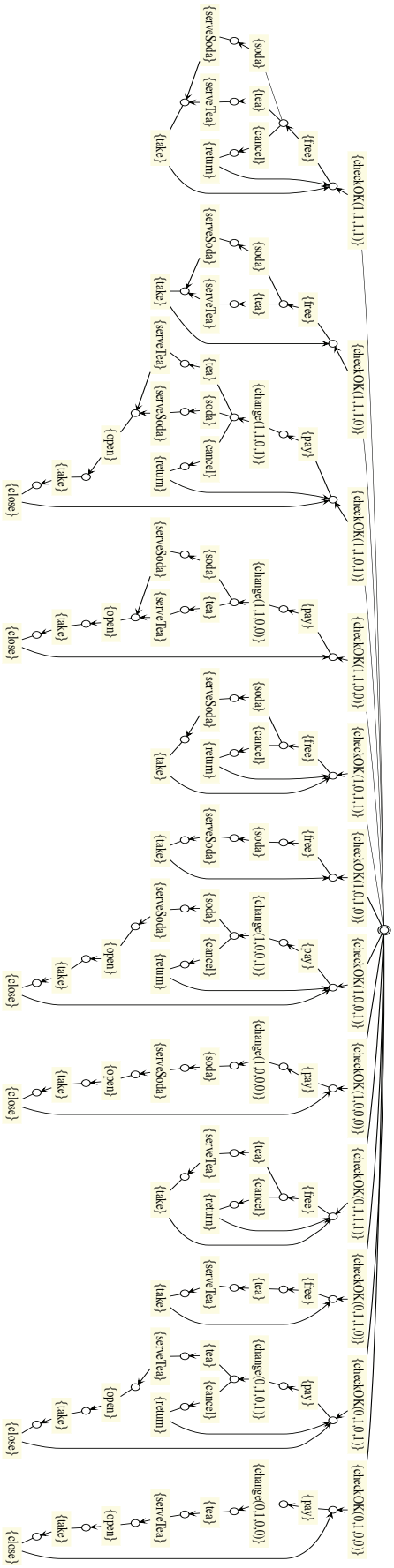

Figure 3: L^2TS of all products of the vending machine SPL as generated by FMC

- Classical Box, Diamond and Fixed-point operators (i.e. the full modal μ-calculus)
  e.g. $\texttt{max}\, Y\texttt{: max}\, Z\texttt{: (}\, (\, \langle \texttt{act2(arg1)} \rangle\, Y\,)\ \texttt{or}\ \langle \texttt{act2} \rangle\, Z\, \texttt{)}$
- Classical high-level CTL operators (e.g. Next, Always, Until, Globally, Eventually, Weak Until)
  e.g. $EX\,\texttt{pred1}$, $A\,[\texttt{pred1(arg1)}\,U\,\texttt{pred2}]$, $AG\,EF\,\texttt{pred1}$ and $E\,[\texttt{pred1(arg1)}\,W\,\texttt{pred2}]$
- High-level ACTL-like operators (i.e. the action-based variants of the above CTL operators)
  e.g. $EX\,\{\texttt{act1}\}\,true$, $E\,[\texttt{pred1(arg1)}\,\{\texttt{act1}\}\,W\,\texttt{pred2}]$, $AG\,EF\,\{\texttt{act1}\}\,\texttt{pred1}$, $A\,[\texttt{pred1(arg1)}\,\{\texttt{act1}\}\,U\,\texttt{pred2}]$ and $A\,[\texttt{pred1(arg1)}\,\{\texttt{act1}\}\,U\,\{\texttt{act2}\}\,\texttt{pred2}]$
- Parametric formulas that express data correlations between actions and subformulas
  e.g. $[\,\texttt{act1}(\$1, \$2)\,]\ AF\ \{\texttt{act2}(\%1, \%2)\}\ true$ and $EF\,\{\$1\}\ EF\,\{\%1\}\ true$
- Deontic variants of some of the above operators (which allow to reason on MTSs, distinguishing mandatory from optional transitions)
  e.g. $\langle \texttt{act1} \rangle \#\, true$ and $EF\#\,\{\texttt{act}\}\,\texttt{pred1}$

Clearly not all KandISTI tools are able to fully exploit all KTL features (cf. Sect. 4). For instance, VMC and FMC specifications do not support state labelling (and therefore neither state predicates), whereas FMC, CMC and UMC specifications do not support variability-related aspects (e.g. the deontic variants).

The syntax of KTL is given by the next three definitions (*var* abbreviates variable, more comment below).

Definition 1. *State predicates are built as follows over a set $\mathcal{L}_s$ of state labels.*

$$\begin{aligned} P &::= \ell(e, \ldots) \\ e &::= \mathit{literal} \mid * \mid \%\mathit{var} \end{aligned}$$

Definition 2. *Action formulas are built as follows over a set $\mathcal{L}_t$ of transition labels.*

$$\begin{aligned} \psi &::= \mathit{true} \mid \mathit{false} \mid \ell(e, \ldots) \mid \$\mathit{var} \mid \neg\,\psi \mid \psi \wedge \psi \mid \psi \vee \psi \\ e &::= \mathit{literal} \mid * \mid \$\mathit{var} \mid \%\mathit{var} \end{aligned}$$

Here $var denotes a free variable, while %var denotes a bound variable which must appear in a subformula of a formula that has introduced the corresponding free variable. Inside an action formula, $var may only occur inside certain contexts (Box, Diamond, Next, Eventually, and on the right side of Until operators) and not inside Boolean expressions.

Definition 3. *The full syntax of KTL is as follows:*

$$\begin{aligned} \phi ::=\ & \mathit{true} \mid \mathit{false} \mid P \mid \neg\,\phi \mid \phi \wedge \phi' \mid \phi \vee \phi' \mid \phi \rightarrow \phi' \mid \\ & [\psi]\,\phi \mid \langle \psi \rangle\,\phi \mid \mu\, Y.\phi(Y) \mid \nu\, Y.\phi(Y) \mid \\ & E\,\pi \mid A\,\pi \\ \pi ::=\ & X\,\phi \mid [\phi\ U\ \phi'] \mid [\phi\ W\ \phi'] \mid F\,\phi \mid G\,\phi \mid \\ & X\,\{\psi\}\ \phi \mid [\phi\ \{\psi\}\,U\,\{\psi'\}\ \phi'] \mid [\phi\ \{\psi\}\,W\,\{\psi'\}\ \phi'] \mid \\ & [\phi\ \{\psi\}\,U\ \phi'] \mid [\phi\ \{\psi\}\,W\ \phi'] \mid F\,\{\psi\}\ \phi \mid \\ & [\psi]^{\Box}\,\phi \mid \langle \psi \rangle^{\Box}\,\phi \mid F\,\{\psi\}^{\Box}\ \phi \end{aligned}$$

where Y is a propositional variable and $\phi(Y)$ is syntactically monotone in Y.

In KandISTI, $\neg$, $\vee$, $\wedge$, $\rightarrow$, $^{\square}$, μ and ν must be written as `not`, `or`, `and`, `implies`, `#`, `min` and `max`, respectively.

It is outside the scope of this paper to describe all details of the semantics of the full KTL logic, which has already been presented (though incrementally) in [11, 17, 30]. However, we now recall in some detail a few aspects which are less known but which are used in this paper.

The first aspect is related to the presence of free variables inside a formula which are dynamically bound to some value during the evaluation of that formula. Consider the formula:

$$[\texttt{act1}(\$1)]\ (EF\ \langle \texttt{act2}(\%1)\rangle\ \mathit{true}) \tag{1}$$

The meaning of this formula is as follows: if from the current (initial) state s_1 there exists an outgoing transition labelled with `act1(arg)` (where `arg` is some literal), then the target state of this transition must satisfy the subformula $EF\ \langle \texttt{act2}(\texttt{arg})\rangle\ \mathit{true}$.

Note that the literal `arg` (possibly more than one) is taken from the labels of the outgoing transitions from s_1 (which match the label template `act1`($1)) and used to instantiate the parametric subformula $EF\ \langle \texttt{act2}(\%1)\rangle\ \mathit{true}$ into the non-parametric subformula $EF\ \langle \texttt{act2}(\texttt{arg})\rangle\ \mathit{true}$. To make this more clear, consider the L^{2}TS in Fig. 4.

In this case, the evaluation of Formula 1 in s_1 thus becomes equivalent to the evaluation of the following formula:

$$(\ [\texttt{act1}(\texttt{111})]\ (EF\ \langle \texttt{act2}(\texttt{111})\rangle\ \mathit{true})\ ) \wedge$$
$$(\ [\texttt{act1}(\texttt{222})]\ (EF\ \langle \texttt{act2}(\texttt{222})\rangle\ \mathit{true})\ )$$

For all labels matching the label template a variable binding is generated and used to instantiate the rest of the formula.

It is interesting to recall the origin of this KTL feature, initially introduced in CMC to support the verification of service-oriented systems. In that case, there was the need to be able to verify that in a system any action of the type `request(session_id, operation)` is always followed by an action of the type `response(session_id, result)`, where `session_id` is a dynamically generated value.

Using the above parameterization mechanism, we are able to express such a property by a formula of the following form:

$$AG\,[\texttt{request}(\$s, *)]\ AF\,\{\texttt{response}(\%s, *)\}\ \mathit{true}$$

Note how $*$ can be used as a placeholder for identifying label parameters not used in the variable bindings (this is commonly referred to as a *don't care* symbol).

The second aspect we want to describe in detail is related to the special-purpose predefined state predicate of the form `PRINT(arg1, arg2, ...)`, in which `arg`'s can be identifiers, literals or bound variables (i.e. $\%var$). The evaluation of such a `PRINT` predicate on a state s simply returns the value *false*. However, its evaluation in state s also has the side-effect of printing the following text:

`MESSAGE from state` s`: PRINT(arg1, arg2, ...)`

(in which any bound variable in the list of `arg`'s is obviously replaced by its currently instantiated value in s).

In case of the L^{2}TS depicted in Fig. 4, the evaluation of the formula $\langle \texttt{act1}(\$1)\rangle$ `PRINT`$(\%1)$ in state s_1 generates the following output:

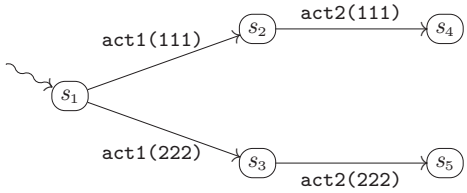

Figure 4: Dynamic variable binding in L^{2}TS

```
MESSAGE from state s2: PRINT(111)
MESSAGE from state s3: PRINT(222)
The formula:
⟨act1($1)⟩ PRINT(%1)
is FALSE
```

Note that in the end the formula evaluates to *false* because each time a transition satisfying `act1`($1) is found a `PRINT` predicate is evaluated, which returns *false*. So the final result (*false*) is actually irrelevant, while the interesting part is the preceding list of messages generated as side-effect.

Whenever a KTL formula contains a `PRINT` predicate, all its Boolean conjunction and disjunction operators are evaluated in a lazy, left-to-right order. This means that the evaluation of the formula *false* $\wedge$ `PRINT(OK)` only returns *false*, without printing anything, whereas an evaluation of the formula *true* $\wedge$ `PRINT(OK)` has the effect of still returning *false*, but it subsequently prints the message `PRINT(OK)` from the state in which it is evaluated. It is interesting to see that evaluation of the formula $EF\,\phi$ $\wedge$ `PRINT(OK)` thus has the effect of always returning a final value *false* and afterwards printing the message `PRINT(OK)` from all the states in which the formula ϕ actually holds.

These two aspects will be applied in the next section for the family-based analysis of our FMC model of Sect. 4.

6. MODEL CHECKING FTSs WITH FMC

In this section, we show how FMC can be used to perform family-based analyses of SPL behaviour and we illustrate its usage on our running example.

As usual, a product is identified by the set of features that it contains, i.e. in our example $(1, 0, 1, 0)$ refers to the product that offers free soda (cf. Fig. 3).

We now provide three patterns of KTL formula that are useful for performing family-based analyses with FMC, after which we show an example of their usage by applying them to the running example. Note how we use the predefined state predicate `PRINT` to output the products that do or do not satisfy a certain property.

First, a KTL formula of the format

$$\langle \texttt{checkOK}(\$1, \ldots, \$n)\rangle\ (\phi\ \texttt{and PRINT}(\texttt{OK}, \%1, \ldots, \%n))$$

prints all products for which ϕ holds.

Second, a KTL formula of the format

$$\langle \texttt{checkOK}(\$1, \ldots, \$n)\rangle\ (\texttt{not}\ \phi\ \texttt{and PRINT}(\texttt{KO}, \%1, \ldots, \%n))$$

prints all products for which ϕ does not hold.

Third, a KTL formula of the format

$$\langle \texttt{checkOK}(\$1,\ldots,\$n)\rangle\ ((\phi\ \texttt{and PRINT(OK},\%1,\ldots,\%n))\ \texttt{or}\ (\texttt{not}\ \phi\ \texttt{and PRINT(KO},\%1,\ldots,\%n)))$$

prints for all products whether or not they satisfy ϕ. Note that it moreover does so in a single computation.

If, instead, we are not interested in pinpointing the precise set of products for which a property holds, but simply want to know whether a property holds for all valid products, then we can verify a formula of the format

$$[\texttt{checkOK}]\ \phi$$

Note that this is logically equivalent to verifying the formula $[\texttt{checkOK}(*,*,*,*)]\ \phi$. This specific format paves the way to verification of properties for only a subset of valid products. Suppose that we want to verify ϕ for all products that serve tea but not soda and which do not allow to cancel a purchase, while we don't care about whether or not a beverage needs to be paid for. We can verify this with the following formula:

$$[\texttt{checkOK}(0,1,*,0)]\ \phi$$

The validity of a formula for a subset of valid products (i.e. a subfamily) obviously does not depend on the rest of the family and, hence, neither does the complexity of verifying the formula. In combination with the on-the-fly model-checking algorithm of FMC, this means that more often than not only part of the complete state space needs to be inspected, which considerably improves the scalability of our approach.

We now show how to verify the following example property from [24]: *"After selecting a beverage, the machine will always open the beverage compartment to allow the customer to collect his purchase."*

Analogous to [24], we first verify it for all valid products:

$$[\texttt{checkOK}]\ AG\ [\texttt{soda or tea}]\ AF\ \{\texttt{open}\}\ \texttt{true}$$

Not surprisingly, FMC concludes that this formula is false. Actually, if we request FMC to explain the result (which we recall is a very useful feature of KandISTI), it comes up with a counterexample, similar to the one presented in [24].

In brief, FMC states that the execution sequence

```
C1   ->  C2     {checkOK(1,1,1,1)}
C2   ->  C14    {free}
C14  ->  C16    {soda}
```

leads to a state (C16) in which the formula $AF\ \{\texttt{open}\}\ \texttt{true}$ is not satisfied. Literally, the reason provided is "there exists at least one full path from C16 in which all transitions have the label which does NOT satisfy the action open. For example:

```
C16  ->  C18    {serveSoda}
C18  ->  C2     {take}
C2   ->  C14    {free}
C14  ->  C15    {cancel}
C15  ->  C2     {return}    (C2 closes a loop)
```

is one of the above mentioned failing paths".

As in [24], we thus need to restrict verification of the aforementioned property to products without the FreeDrinks feature if it were to hold. This can be formalized in KTL as:

$$[\texttt{checkOK}(*,*,0,*)]\ AG\ [\texttt{soda or tea}]\ AF\ \{\texttt{open}\}\ \texttt{true}$$

FMC reports that this formula is indeed true.

If we are interested in knowing precisely for which of the products of the SPL the property holds and/or for which it does not hold, then we can use one of the above patterns of KTL formula. For example, upon verifying the KTL formula

$$\langle \texttt{checkOK}(\$1,\ldots,\$4)\rangle\ ((\phi\ \texttt{and PRINT(OK},\%1,\ldots,\%4))\ \texttt{or}\ (\texttt{not}\ \phi\ \texttt{and PRINT(KO},\%1,\ldots,\%4)))$$

for $\phi \equiv AG\ [\texttt{soda or tea}]\ AF\ \{\texttt{open}\}\ \texttt{true}$, FMC reports:

```
MESSAGE from state C2: PRINT(KO,1,1,1,1)
MESSAGE from state C3: PRINT(KO,1,1,1,0)
MESSAGE from state C4: PRINT(OK,1,1,0,1)
MESSAGE from state C5: PRINT(OK,1,1,0,0)
MESSAGE from state C6: PRINT(KO,1,0,1,1)
MESSAGE from state C7: PRINT(KO,1,0,1,0)
MESSAGE from state C8: PRINT(OK,1,0,0,1)
MESSAGE from state C9: PRINT(OK,1,0,0,0)
MESSAGE from state C10: PRINT(KO,0,1,1,1)
MESSAGE from state C11: PRINT(KO,0,1,1,0)
MESSAGE from state C12: PRINT(OK,0,1,0,1)
MESSAGE from state C13: PRINT(OK,0,1,0,0)
The formula:
```

$$\langle \texttt{checkOK}(\$1,\ldots,\$4)\rangle\ ((\phi\ \cdots\ \texttt{PRINT(KO},\%1,\ldots,\%4)))$$

```
is FALSE
```

Hence this property actually only holds for products without the FreeDrinks feature (as was known from [24]).

We conclude with three more complex properties not taken from [24].

The first property is as follows: *For all beverage vending machines, it is always true that, if the customer pays or skips payment, and he does not cancel his purchase, then he will surely take the purchased beverage without being asked again to pay or skip payment.* In KTL, this can be formalized as:

$$[\texttt{checkOK}]\ AG\,[\texttt{pay or free}]$$
$$A\,[\mathit{true}\ \{(\texttt{not pay})\ \texttt{and}\ (\texttt{not free})\}\ U\ \{\texttt{take or cancel}\}\ \mathit{true}]$$

As desired, FMC reports that this is a property that holds.

The second property is as follows: *For all beverage vending machines, the machine will always eventually become idle again.* This property, which verifies that any execution sequence of a product of our FMC model will eventually lead back to the product's initial state, is formalized in KTL as:

$$[\texttt{checkOK}]\ AG\ AF\ \langle \texttt{pay or free}\rangle\ \mathit{true}$$

FMC reports that also this property holds, thus confirming the absence of deadlocks in our FMC model.

The third property, finally, is as follows: *For all beverage vending machines, the machine will deliver an infinite number of beverages if at each cycle all internal loops are executed just a finite number of times.* If we measure a cycle as the execution sequence between a customer taking two different beverages, then this property can be formalized as:

$$[\texttt{checkOK}]\ AG\ EF\ \{\texttt{take}\}\ \mathit{true}$$

Note that the only possible internal loops are those sequences that involve a purchase being cancelled. This formula thus concerns the evaluation of a property that holds only for 'fair' executions.

The reader is warmly invited to experiment with FMC: the running example is available among the examples offered upon clicking 'Model Definition' in FMC's menu panel.

7. RELATED WORK

Our approach to model checking SPLs falls in the category of using existing model checkers, that have been optimized for single system engineering, for SPLE, based on encoding the product variability directly into the specification model to be verified [1, 2, 8–10, 12, 14–17, 19, 34, 37, 40]. Consequently, these model checkers stop once a violating product is found, without outputting the set of products that do satisfy the property. This drawback is overcome in the dedicated SPL model checker SNIP [23], which has recently been reengineered and the resulting tool suite ProVeLines [26] now supports discrete as well as real-time models, various types of computations and advanced feature notions. In this paper, we have shown how to overcome this drawback with FMC by exploiting the advanced correlation features of KTL, which allow to print for all products whether or not they satisfy a formula as the result of a single computation.

8. CONCLUSIONS AND FUTURE WORK

We have shown how the FMC model checker (http://fmt.isti.cnr.it/fmc) and its KTL logic can be quite easily used for the modelling and (family-based) analysis of behavioural variability in SPLs. We have also shown how to automatically obtain a process-algebraic model from an FTS that can directly serve as input for FMC. In [7], we undertook a similar approach by providing an automatic technique to transform FTSs into MTSs with additional sets of variability constraints in the specific format accepted by VMC. This technique was illustrated on the same example.

In FMC, parametric formulas, lazy evaluation and special-purpose `PRINT` predicates allow to identify all the (reachable) states in which a certain formula is satisfied, paving the way for an automatic analysis for deciding which specific products of an SPL do or do not satisfy a given property.

The on-the-fly evaluation of the KTL logic also allows efficient verifications of properties over subfamilies of an SPL (i.e. over a subset of products characterized by a subset of features), in which case the state space of the rest of the family obviously need not be generated. In combination with on-the-fly model checking, this considerably improves the scalability of our approach.

The adopted approach is illustrated for a classical, though quite simple example SPL. It is left as future work to extend the analysis approach to models of larger size as well to pursue a quantitative/qualitative comparison with other tools that have been used for the (family-based) behavioural variability analysis of SPLs, such as the Variability Model Checker VMC [17, 19], the dedicated SPL model checker SNIP/ProVeLines [23, 26] and the industrial-strength modelling and analysis toolset mCRL2 [27, 35].

9. ACKNOWLEDGEMENTS

Research supported by EU project QUANTICOL (600708) and Italian MIUR project CINA (PRIN 2010LHT4KM). We thank the anonymous reviewers for their useful comments.

10. REFERENCES

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

[2] S. Apel, A. von Rhein, P. Wendler, A. Größlinger, and D. Beyer. Strategies for Product-Line Verification: Case Studies and Experiments. In *ICSE*, pages 482–491. IEEE, 2013.

[3] J. C. M. Baeten, T. Basten, and M. A. Reniers. *Process Algebra: Equational Theories of Communicating Processes*, volume 50 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2010.

[4] J. C. M. Baeten and W. P. Weijland. *Process Algebra*, volume 18 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1990.

[5] C. Baier and J. Katoen. *Principles of Model Checking*. The MIT Press, 2008.

[6] D. S. Batory. Feature Models, Grammars, and Propositional Formulas. In J. H. Obbink and K. Pohl, editors, *SPLC*, volume 3714 of *LNCS*, pages 7–20. Springer, 2005.

[7] M. H. ter Beek, F. Damiani, S. Gnesi, F. Mazzanti, and L. Paolini. From Featured Transition Systems to Modal Transition Systems with Variability Constraints. In R. Calinescu and B. Rumpe, editors, *SEFM*, LNCS. Springer, 2015.

[8] M. H. ter Beek and E. P. de Vink. Software Product Line Analysis with mCRL2. In A. Legay and E. de Vink, editors, *SPLC workshop SPLat*, volume 2 of *SPLC*, pages 78–85. ACM, 2014.

[9] M. H. ter Beek and E. P. de Vink. Towards Modular Verification of Software Product Lines with mCRL2. In T. Margaria and B. Steffen, editors, *ISoLA*, volume 8802 of *LNCS*, pages 368–385. Springer, 2014.

[10] M. H. ter Beek and E. P. de Vink. Using mCRL2 for the Analysis of Software Product Lines. In S. Gnesi and N. Plat, editors, *ICSE workshop FormaliSE*, pages 31–37. IEEE, 2014.

[11] M. H. ter Beek, A. Fantechi, S. Gnesi, and F. Mazzanti. A state/event-based model-checking approach for the analysis of abstract system properties. *Science of Computer Programming*, 76(2):119–135, 2011.

[12] M. H. ter Beek, A. Fantechi, S. Gnesi, and F. Mazzanti. Modelling and Analysing the Variability in Product Families: Model Checking of Modal Transition Systems, 2015.

[13] M. H. ter Beek, S. Gnesi, and F. Mazzanti. From EU Projects to a Family of Model Checkers: From Kandinsky to KandISTI. In R. De Nicola and R. Hennicker, editors, *Software, Services and Systems*, volume 8950 of *LNCS*, pages 312–328. Springer, 2015.

[14] M. H. ter Beek, A. Legay, A. Lluch Lafuente, and A. Vandin. Quantitative Analysis of Probabilistic Models of Software Product Lines with Statistical Model Checking. In J. M. Atlee and S. Gnesi, editors, *SPLC workshop FMSPLE*, volume 182 of *EPTCS*, pages 56–70, 2015.

[15] M. H. ter Beek, A. Legay, A. Lluch Lafuente, and A. Vandin. Statistical Analysis of Probabilistic Models of Software Product Lines with Quantitative Constraints. In *SPLC*. ACM, 2015.

[16] M. H. ter Beek, A. Lluch Lafuente, and M. Petrocchi. Combining Declarative and Procedural Views in the Specification and Analysis of Product Families. In

D. Clarke, editor, *SPLC workshop FMSPLE*, volume 2 of *SPLC*, pages 10–17. ACM, 2013.

[17] M. H. ter Beek and F. Mazzanti. VMC: Recent Advances and Challenges Ahead. In A. Legay and E. de Vink, editors, *SPLC workshop SPLat*, volume 2 of *SPLC*, pages 70–77. ACM, 2014.

[18] M. H. ter Beek, F. Mazzanti, and S. Gnesi. CMC–UMC: a framework for the verification of abstract service-oriented properties. In C. Guidi, I. Lanese, and M. Mazzara, editors, *SAC track SOAP*, pages 2111–2117. ACM, 2009.

[19] M. H. ter Beek, F. Mazzanti, and A. Sulova. VMC: A Tool for Product Variability Analysis. In D. Giannakopoulou and D. Méry, editors, *FM*, volume 7436 of *LNCS*, pages 450–454. Springer, 2012.

[20] J. C. Bradfield and C. Stirling. Modal μ-calculi. In P. Blackburn, J. F. A. K. van Benthem, and F. Wolter, editors, *Handbook of Modal Logic*, volume 3 of *Studies in Logic and Practical Reasoning*, pages 721–756. Elsevier, 2007.

[21] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[22] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 1999.

[23] A. Classen, M. Cordy, P. Heymans, A. Legay, and P. Schobbens. Model checking software product lines with SNIP. *International Journal on Software Tools for Technology Transfer*, 14(5):589–612, 2012.

[24] A. Classen, M. Cordy, P. Schobbens, P. Heymans, A. Legay, and J. Raskin. Featured Transition Systems: Foundations for Verifying Variability-Intensive Systems and Their Application to LTL Model Checking. *IEEE Transactions on Software Engineering*, 39(8):1069–1089, 2013.

[25] A. Classen, P. Heymans, P. Schobbens, A. Legay, and J. Raskin. Model Checking Lots of Systems: Efficient Verification of Temporal Properties in Software Product Lines. In *ICSE*, pages 335–344. ACM, 2010.

[26] M. Cordy, A. Classen, P. Heymans, P. Schobbens, and A. Legay. ProVeLines: a product line of verifiers for software product lines. In *SPLC*, volume 2, pages 141–146. ACM, 2013.

[27] S. Cranen, J. F. Groote, J. J. A. Keiren, F. P. M. Stappers, E. P. de Vink, W. Wesselink, and T. A. C. Willemse. An Overview of the mCRL2 Toolset and Its Recent Advances. In N. Piterman and S. A. Smolka, editors, *TACAS*, volume 7795 of *LNCS*, pages 199–213. Springer, 2013.

[28] R. De Nicola, A. Fantechi, S. Gnesi, and G. Ristori. An Action Based Framework for Verifying Logical and Behavioural Properties of Concurrent Systems. In K. G. Larsen and A. Skou, editors, *CAV*, volume 575 of *LNCS*, pages 37–47. Springer, 1991.

[29] R. De Nicola and F. W. Vaandrager. Action versus State based Logics for Transition Systems. In I. Guessarian, editor, *Semantics of Systems of Concurrent Processes*, volume 469 of *LNCS*, pages 407–419. Springer, 1990.

[30] A. Fantechi, S. Gnesi, A. Lapadula, F. Mazzanti, R. Pugliese, and F. Tiezzi. A logical verification methodology for service-oriented computing. *ACM Transactions on Software Engineering and Methodology*, 21(3):16:1–16:46, 2012.

[31] C. Fidge. A Comparative Introduction to CSP, CCS and LOTOS. Technical Report 93-24, Software Verification Research Centre, University of Queensland, January 1994.

[32] S. Gnesi and F. Mazzanti. On the Fly Verification of Networks of Automata. In H. R. Arabnia, editor, *PDPTA*, pages 1040–1046. CSREA Press, 1999.

[33] S. Gnesi and F. Mazzanti. An Abstract, on the Fly Framework for the Verification of Service-Oriented Systems. In M. Wirsing and M. M. Hölzl, editors, *Rigorous Software Engineering for Service-Oriented Systems: Results of the SENSORIA Project on Software Engineering for Service-Oriented Computing*, volume 6582 of *LNCS*, pages 390–407. Springer, 2011.

[34] S. Gnesi and M. Petrocchi. Towards an executable algebra for product lines. In *SPLC workshop FMSPLE*, pages 66–73. ACM, 2012.

[35] J. F. Groote and M. R. Mousavi. *Modeling and Analysis of Communicating Systems*. The MIT Press, 2014.

[36] D. Kozen. Results on the Propositional mu-Calculus. *Theoretical Computer Science*, 27:333–354, 1983.

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

[38] R. Meolic, T. Kapus, and Z. Brezocnik. ACTLW: An action-based computation tree logic with unless operator. *Information Sciences*, 178(6):1542–1557, 2008.

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

[40] H. Post and C. Sinz. Configuration Lifting: Verification meets Software Configuration. In *ASE*, pages 347–350. IEEE, 2008.

[41] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1997.

[42] P. Schobbens, P. Heymans, and J. Trigaux. Feature Diagrams: A Survey and a Formal Semantics. In *RE*, pages 136–145. IEEE, 2006.

[43] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Computing Surveys*, 47(1):6:1–6:45, 2014.