Relating Requirement and Design Variabilities

Jean-Vivien Millo
EPI AOSTE, INRIA Sophia-Antipolis
2004 routes des lucioles, BP93
06902, Sophia-Antipolis, France
Email: jean-vivien.millo@inria.fr

S. Ramesh
Global General Motors R&D
General Motors Technical Center India
ITPL, Whitefield main road, 560066, Bangalore, India
Email: ramesh.s@gm.com

***Abstract*—This paper presents a novel approach to relate the variabilities that exist at the requirement and design levels in a Software Product Line (SPL). This approach is based upon two key observations: (i) it is not only the requirements, but also the design contains variability information, (ii) The variability information at the requirement and design levels are expressed differently and at different levels of abstraction.**

In the context of an SPL composed of features including variability, the proposed method relates every feature configuration (evaluation of the configuration parameters) at the design level with a feature instantiation at the requirement level. The core step in the method is a conformance checking procedure that is based upon the well-known automata containment algorithm used in formal verification of finite state systems.

The method has been implemented on top of the well-known verification tool, SPIN, and then experimented on an industrial example with encouraging results.

I. Introduction

This paper focuses on relating the variability of features at the requirement and design level in a Software Product Lines (SPL). An SPL consists of multiple features that can be assemble together. A feature, as it is defined in this paper, is a piece of functionality including variability.

The SPL literature is full of models and formalisms for expressing the variability information, the prominent ones among them being the Czarnecki's feature model [1] and the Pohl's OVM [2]. The requirements of a feature describe the behavior of its different configuration. Requirement level promotes a high level of abstraction where expressibility, clarity, and ease of analysis are of importance. The following sentence is an example of a typical SPL requirement containing functionality and variability information: *"The Door Lock feature in an automobile SPL, when enabled, will lock the door when a shift out of park occurs for automatic transmission vehicles or when a certain calibratable speed (e.g. 5 or 8 mph) is reached for vehicles with manual transmission."*

The design of a feature realizes the functionality and variability specified in the requirements taking into account the implementation and platform constraints. Often in many application domains, e.g. automotive applications, like the requirements, the design still contains variability information. The variability is implemented as "jumper settings" or values in a ROM. At the deployment stage, the variability is resolved and a specific product is derived.

In order to relate the requirement and design of a feature, we define a conformance relation which ensures, for a given feature, that each configuration at the design level realizes a configuration at the requirement level. This relation captures the fact that the variability at the design level is the one implied at the requirement level. This relation makes use of the standard automata containment relation used in traditional model-checking approach to design verification. The goal is to check that every design configuration conforms to at least one requirement configuration.

Thus, it involves checking whether the *actual* behavior and variability, as realized at the design level, conforms to the *expected* behaviour and variability given in the requirements.

For capturing variability in the behavior of a feature, we have extended the well-known model of finite state machines, and proposed a model called *Finite State Machines with Variability*, in short, *FSMv*. FSMv is flexible enough to represent the behavior and variability information at both the requirement and design level. We define two sub-classes of FSMv, FSMr and FSMd, to represent the behavior and variability of a feature at the requirement and design level respectively.

The proposed approach still suffers from some limitations. One needs to trace a feature at the requirement level with its corresponding module(s) at the design level. Some initiatives such as [3] or [4] pave the way to the automation of this step. In addition, the conformance relation requires that the compared automata have comparable alphabet. One have to identify the correspondance between events at the requirement level with the signals at the design level. Lastly, the design may include implementation details that has to be abstracted out to perform the proposed verification approach.

The proposed approach has been implemented on top of SPIN [5], a well-known tool for traditional software design verification. We have conducted experiments on an industrial example with encouraging results.

In summary, the main contributions of this paper are i) a method to relate the variability from design to requirement based on a conformance relation between design and requirement models of SPL's features, ii) a scalable implementation of the conformation algorithm and iii) a prototype implementation of the tool.

II. Related works

1) SPL variability analysis between requirement and design: The Variation Point Model (VPM) of Hassan Gomaa

[6], [7] distinguishes between variability at the requirement and design levels but no design verification approach has been presented. Kathrin Berg *et al.*[8] propose a model for variability handling throughout the life cycle of the SPL. Andreas Metzger *et al.*[9] and M Riebisch *et al.*[10] provide a similar approach but they do not consider the behavioral aspect. In the proposed approach, we extract the relation between expected and actual variability from a behavioral analysis.

2) Model-Checking of SPL behavior: Kathi Fisler *et al.* [11], [12] have developed an analysis based on three-valued model checking of automata defined using step-wise refinement. Later on, Jing Liu *et al.* [13] have extended Fisler's approach to provide a much more efficient method. Kim Lauenroth *et al.* [14], [15] as well as Andreas Classen *et al.* [16], [17], [18], and Gruler *et al.* [19] have developed model checking methods for SPL behavior. These methods are based on the verification of LTL/CTL/modal μ calculus formula. In these verification methods, the requirement and design share the same variability and only the behavior is checked.

3) Behavior model for SPL: Ulrik Nyman [20], [21] and later, Jean-Baptiste Raclet et al. [22] have defined the Modal I/O Automata (MIOA) to express in one automaton all the possible behaviors of a feature. Similarly, Alessandro Fantechi and Stefania Gnesi [23] have defined the GEMTS model which expresses the variability better than MIOA using the *at least*, *at most* operators but GEMTS is not compositional. Patrizia Asirelli *et al.* [24], [25] proposes LTS based on deontic logic. Kathrin Scheidemann *et al.* [26], [19] present a behavioral model (PL-CCS) inspired from CCS and introducing a variation operation. These models are usually coupled with a variability model such as OVM [2], the Czarnecki feature model [1], or VPM [6], [7] to attain a fair level of variability expressibility. Unlike all these approaches, FSMv captures the variability in an explicit way which we find more intuitive. Also the FSMv models can be automatically derived from design models like UML models.

FSMv is similar to FTS [16] and FTS$^+$ [17]. However, FTS$^+$ is applicable to model the behavior of the entire SPL while FSMv focuses on single feature. In addition, FTS$^+$ and its associated methodology is more appropriate to check for properties focusing on the reachable states of the system while FSMv focuses on simulation relation between requirement and design and thus infinite sequence of events.

4) Automatic analysis of feature diagram: In the literature, variability is usually given by a feature diagram [1] and translated into a propositional formula. Even though feature diagrams are not considered in the proposed approach, existing works about automatic reasoning on feature diagram [27], [28], [29], [30], [31] have been source of inspiration for the present article. All these works are mostly based on [32]. [33] gives an exhaustive survey of the works conducted in this area.

III. RELATING VARIABILITIES

In general, an SPL consists of a collection of features, each feature having different functionality as well as variability. For example, consider the body control function product line used in automotive applications. A typical body control function has several features, like, door lock, lighting and seat control function. Each of these features has a distinct function and variability. For example, the locking behaviour of a door lock function has a variation point called *transmission type*. If the transmission type is manual then the door is locked after the speed of the vehicle exceeds a certain threshold value; for automatic transmission, the door is locked when the gear position is shifted out of park.

As mentioned in the introduction, we make use of a novel formal model called FSMv for precisely capturing the behavior of a feature.

A. FSMv and language refinement

FSMv is an extension of the classical FSM model. The distinguishing aspect of FSMv is its ability to specify variability. The variability information is captured using a finite set of variables associated with an FSMv.

Let Var be a finite set of variables each taking a value ranging over a finite set of values. Let $Pred_{Var}$ denote the set of all predicates over Var using propositional logic; we assume a set of atomic propositions involving the variables and the standard relational operators like $=, >, <, \in$ etc.

Definition 1 (FSMv): An FSMv is a tuple $\langle Q, q_0, \Sigma, \delta, Var, \rho, \Delta \rangle$ where:

- Q is a finite set of states. q_0 is the initial state.
- Σ is a set of events.
- $\delta = Q \times \Sigma \times Q$ is a set of transitions. For $t = s_1 \times e \times s_2 \in \delta$, we define $\bullet t = s_1$, $t\bullet = s_2$, and $\dot{t} = e$.
- Var is a finite set of (variant) variables.
- $\rho \in Pred_{Var}$ is a consistent predicate called the *global predicate*.
- Δ is a mapping from every transition to a consistent predicate on the variables of Var. Δ: $\delta \rightarrow Pred_{Var}$ defines the variability domain of a transition.

Note that there is the global predicate (ρ) for the whole FSMv and local transition predicates (Δ) each for a transition. A simple example of a global predicate is: $\neg(var_1 = a) \wedge (var_2 = var_3)$. As this example shows, the global predicate restricts the possible values that the variables can take. A predicate is consistent if there exists at least a valuation of the variables that satisfies the predicate. Each valuation of the variables satisfying the global predicate gives rise to a product variant whose behavior is described by the FSMv with only and all those transitions whose variability domains (Δ) hold for the given valuation. Thus using the variables one can represent the behavior of all possible configurations of a feature in a single FSMv description.

Definition 2 (Configuration): A configuration, denoted by π, is a mapping that associates each variable in Var with an appropriate value.

The set of all configurations is denoted by Π_{Var}. Let $\pi \in \Pi_{Var}$ be a configuration. $\pi(var)$ is the value associated to var in π.

We define $\Pi_{Var}(\rho)$ as the set of all the configurations of Var that satisfy the predicate ρ. $\Pi_{Var}(\rho) = \{\pi | \pi \models \rho\}$ where "$\models$" denotes the standard Boolean satisfiability relation.

We define ρ_π as the predicate which is satisfied only by π. If $\forall i \in [1, |Var|]$, $\pi(var_i) = a_i$, then $\rho_\pi : \bigwedge_{i=1}^{|Var|}(var_i = a_i)$.

Given a configuration π and a transition t, we say that t is allowed in π if $\pi \models \Delta(t)$.

An FSMv is said to be nondeterministic, if there exists a configuration, an event, and a state in the state machine from which two distinct transitions triggered by the same event are allowed in the same configuration.

For a transition t, if $\Delta(t)$ is inconsistent with ρ, then t is not allowed in any valid configuration. Likewise, a transition can be allowed in multiple valid configurations.

As a concrete example of an FSMv, consider the feature *Door lock* in automotive software which controls the locking of the doors when the vehicle starts. This is one of the features in the Entry Control Product Line to be discussed later as a case study.

The expected behavior of this feature is modeled using the FSMv Req_{dl} described pictorially in Figure 1. In the initial state, this feature becomes active when all the doors are closed. The doors are locked when either the speed of the vehicle exceeds a predefined value or the gear is shifted out of park. An unlock event reactivates the feature.

There are four configurations for this feature all of which are described using the three variables: DL_Enable, Transmission and DL_User_Pref; see the box at the top left in the figure for the possible values of these variables. The bottom box gives the global predicate associated with the machine. It ensures that in every valid configuration, the variable Transmission has the value Manual implies that DL_User_Pref takes the value Speed. This captures the fact that in manual transmission, the door is closed when speed reaches a threshold value. Note that to avoid clutter, the transition predicates use a short hand notation: for the edge labeled *Park:ShiftOutOfPark*, the predicate is DL_User_Pref = Park, i.e., $\Delta(t) = (DL\_User\_Pref = Park)$ and $\overset{\bullet}{t} = ShiftOutOfPark$. The transition labeled with *Disable:** means that when DL_Enable = Disable, it stalls on any event.

B. The SPL requirement and design models

In the requirement of a product line, the variability is usually discussed in terms of variation points, which are at a high level of abstraction and focused on clarity and expressibility. The restriction of the possible configurations is expressed as general constraints on these variation points, e.g., the global predicate $Manual \implies Speed$ in the *Door lock* example, without concerning about how they are implemented. In contrast, in a design, the variability description is constrained by efficiency, implementability, ease of reconfiguration and deployment considerations. For instance, in the automotive applications, one often finds that the variation points are expressed using a set of variables, called *calibration parameters*. The calibration parameters range over a set of simple values and each setting of the parameters corresponds to a reconfiguration to a specific product.

In order to capture the difference between the requirements and design of an SPL, we define a subclass of FSMv, called FSMd.

Definition 3 (FSMd): An FSMd (d stands for design) is an FSMv where the global predicate ρ takes the simple form $\rho = \bigvee_{i=1}^{m} \rho_{\pi_i}$ where each ρ_{π_i} is a conjunction of equality constraints of the form $(v = V)$, where v is a variable and V is a specific value that can be assigned to the variable. Further each transition predicate is also conjunction of such equality constraints.

ρ has the same expresivness in FSMd than in a general FSMv but we want to capture the fact that, at the design level, the possible configurations are given as an exhaustive list of tuples of values but not as general constraint.

Let us illustrate FSMd using the *Door lock* example. Des_{dl} Figure 2 describes the behaviour of a design of *Door lock*. This is similar to Figure 1 except that the active state (top elliptical shaped state in Figure 2) is split into two states (the top and the bottom elliptical shaped states in Figure 2. The top active state is for auto-transmission whereas the bottom one is for manual transmission as can be seen from the configuration label of the two transitions going from the initial state. The major difference to be noted in these two machines is the variability representation. Two parameters ($Cp1$ and $Cp2$) encode the possible configurations in the FSMd. $Cp1 = Auto$ corresponds to the configuration in which the transmission is Auto whereas $Cp1 = Moff$ corresponds to either the manual transmission or that the feature is disabled according to the value of $Cp2$. Similarly, $Cp2 = Speed$ means that the user preference is set on $Speed$ but $Cp2 = Poff$ means either Park or that the feature is disabled.

In our discussions, we often distinguish an FSMd by using the tuple $\langle Q_d, q_0^d, \Sigma, \delta_d, Var_d, \rho_d, \Delta_d \rangle$ using the subscript d at appropriate places. Further we use the notation FSMr to denote the configuration state machines whose global predicates are not of the special form assumed in FSMd. Any such machine is called a requirement state machine and a typical requirement state machine, is denoted by $\langle Q_r, q_0^r, \Sigma, \delta_r, Var_r, \rho_r, \Delta_r \rangle$.

C. Variant and language

Given the behavior of a feature with multiple variants, we can obtain the behavior of a single variant or instance by *configuring* it. A variant is a standard FSM corresponding to a given configuration. In the sequel, we consider only FSMv*s* with deterministic variants.

Definition 4 (Variant of an FSMv): Let $M = \langle Q, q_0, \Sigma, \delta, Var, \rho, \Delta \rangle$ be an FSMv and $\pi \in \Pi_{Var}(\rho)$ be a configuration of M. The variant of M is a standard FSM obtained from M by retaining only those transitions (and the respective source and target states) that are allowed in π. The transition predicates and global predicates are also removed from the tuple as they do not play any role in the resultant state machine. We denote it $M \downarrow \pi$.

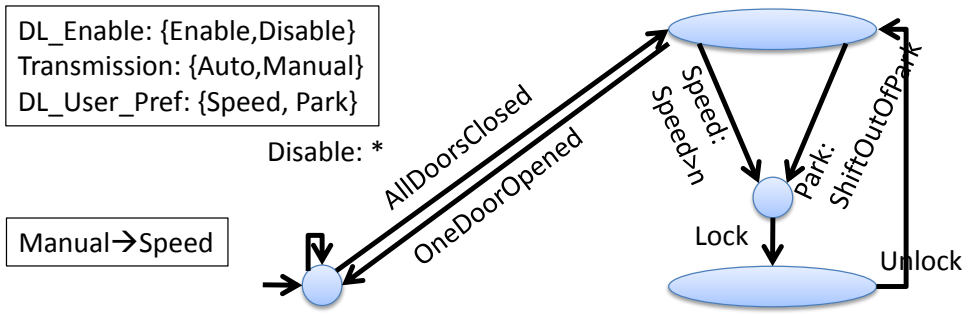

Fig. 1. The FSMv of the feature *Door lock*.

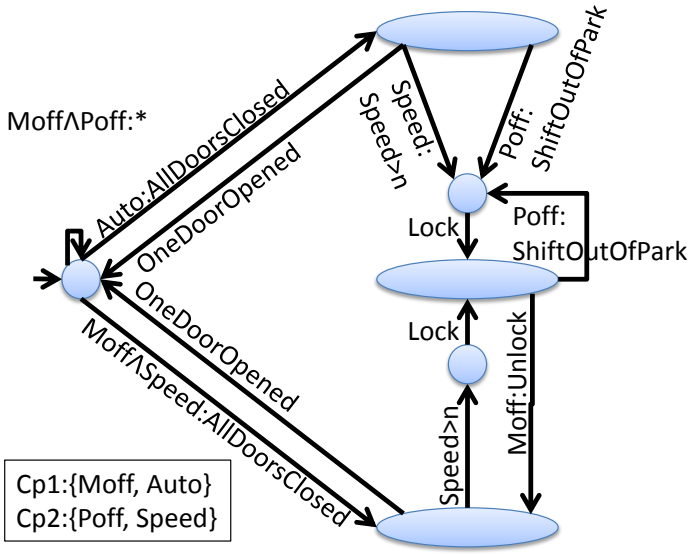

Fig. 2. Des_{dl}: the FSMd abstracted from the design of the feature *Door lock*.

In the example of the feature *Door lock*, the variant $Req_{dl} \downarrow \langle Enable, Auto, Park\rangle$ does not contain the transitions with the event $Speed > n$ and $*$. The variant $Req_{dl} \downarrow \langle false, X, X\rangle$ contains only the initial state with the self-loop labeled (*Disable:**).

The configuration operation forms the basis for defining the semantics of FSMv and comparing FSMd and FSMr. Since a variant of an FSMv is a standard FSM, we can give a language-theoretic semantics to FSMv. Given an FSMv machine M, the semantics of M is a mapping that associates with each configuration, π, of M, the language of the FSM $M \downarrow \pi$, denoted by $L(M \downarrow \pi)$.

D. Language refinement

An FSMd conforms to an FSMr if and only if there exists a mapping Φ relating every configuration of the FSMd with some configuration of the FSMr such that the language of the variant of the FSMd on a given configuration is a subset of the language of the variant of the FSMr on the corresponding configuration. Φ is a total mapping from the configuration of an FSMd to the configuration of its corresponding FSMr. Φ embodies the conformance relation.

Definition 5 (The mapping Φ): Let Req and Des be a pair of FSMr and FSMd respectively. $\Phi : \Pi_{Var_d}(\rho_d) \rightarrow 2^{\Pi_{Var_r}(\rho_r)}$

Definition 6 (Language refinement): Let Des and Req be a pair of FSMd and FSMr respectively.

Des conforms to Req denoted $Des \leq_\Phi Req$ if $\exists\Phi$ such that for all the configurations π_d of Des, there exists π_r in $\Phi(\pi_d)$ and $L(Des \downarrow \pi_d) \subseteq L(Req \downarrow \pi_r)$[1].

In the feature *Door lock*, $\Phi(\langle Moff,\ Speed\rangle) = \{\langle Enable, Manual, Speed\rangle\}$ since $L(Des_{dl} \downarrow \langle Moff, Speed\rangle) \subseteq L(Req_{dl} \downarrow \langle Enable, Manual, Speed\rangle)$.

IV. The Conformance Algorithm

Let f be a feature. Suppose that the FSMr Req_f represents the expected behavior of f and the FSMd Des_f the actual behavior. Then the conformance checking problem is to compute a mapping Φ such that

$$Des_f \overset{?}{\leq_\Phi} Req_f$$

We say that Φ is the conformance relation between Des_f and Req_f.

The conformance relation is computed by comparing every projection of Des_f with every projections of Req_f. Algorithm 1, given below, presents a possible implementation using the standard automata containment algorithm[34], as implemented in the SPIN model checker [5]. To use SPIN, one should describe the system along with the checked property in the Promela language [5]. Out of this description, SPIN generates the *pan.c* file which is the verifier for your system. After compilation, the *pan(.exe)* executable performs the verification.

A. Promela Model of Des_f and Req_f

The model consists of three automata implemented as Promela processes: one process represents the environment, another represents Des_f and a third represents Req_f. In the environment process, at every step, one random event can be broadcast to Des_f and Req_f using synchronous channels. In the processes corresponding to Des_f and Req_f, the calibration parameters (Var_d) and variation points (Var_r) are represented

[1] $\subseteq$ expresses the refinement.

as read only variables. The states are encoded using a variable which initially holds the initial state. We assume an error state in both the models.

The main structure of either of these processes is a while loop with a switch statement inside. Each case represents a state. For each state, its output transitions are listed and can be triggered if the corresponding event is present and if the variability domain of the transition is respected by the current configuration. An additional transition goes to the error state when none of the above transitions can be taken. The error state has only one outgoing transition self-looping on any event.

Appendix A gives details about the generated Promela file.

B. The conformance checking

Algorithm 1 starts by generating a Promela file containing the three processes defined above: the environment, Des_f, and Req_f plus the initialization sequence and a never claim which holds for the language containment condition. During the initialization, the configuration of Des_f and Req_f are initialized with a given couple of configurations. Then the environment followed by Des_f and Req_f are run atomically. The never claim assertion is the following $never((\neg error(Des_f) \wedge error(Req_f))$
(where $error(X)$ means that X is in error state). The never claim is violated when the design is not in the error state while the requirement process is in. It means that, at some point, the design handles an event but the requirement does not. So, the language conformance condition is violated for the current couple of configurations.

Algorithm 1 runs the full verification algorithm of SPIN for every couple of configurations. SPIN(i.e. *pan(.exe)*) returns either one error, corresponding to a pair for which the conformance condition is violated, or no error when the conformance condition holds and thus this couple can be added to the conformance mapping Φ. One should note that the analysis can be fully parallelized. Since the analysis is feature-wise, the considered variability remains limited and thus the complexity though still exponential is scalable.

C. Correctness of Algorithm 1

The following lemma proves the correctness result.

Lemma 1: $\forall(\pi_d, \pi_r)$, $L(Des_f \downarrow \pi_d) \not\subseteq L(Req_f \downarrow \pi_r)$ if and only if $\neg error(Des_f) \wedge error(Req_f)$.

Proof:

($\Rightarrow$)
There exists a word in $L(Des_f \downarrow \pi_d)$ which is prefixed by $u.e$ with u a finite prefix of a word in $L(Req_f \downarrow \pi_r)$, and e an event such that $u.e$ is not a prefix of any word in $L(Req_f \downarrow \pi_r)$. In such a situation, Des_f does not go to the error state but Req_f does.

($\Leftarrow$)
If $L(Des_f \downarrow \pi_d) \subseteq L(Req_f \downarrow \pi_r)$ then whenever Req_f enters an error state then Des_f enters an error state. ■

Algorithm 1 implements the conformance checking using SPIN.

Input : Des_f, Req_f.
Output : The mapping Φ when $Des_f \leq_\Phi Req_f$
1. Generate a Promela file which contains Req_f, Des_f, the environment, the never claim and, the initialization sequence as described above.
for all $\pi_d \in \Pi_{Var_d}(\rho_d)$ **do**
  for all $\pi_r \in \Pi_{Var_r}(\rho_r)$ **do**
    2. Launch the full verification algorithm of spin for the configuration (π_d, π_r)
    3. Build the mapping Φ from the output of spin.
    if (π_d, π_r) does not return any error **then**
      $\Phi(\pi_d) = \Phi(\pi_d) \cup \{\pi_r\}$
    end if
  end for
end for
4. Conclude whether the design conforms to the requirement
if $\forall \pi_d \in \Pi_{Var_d}(\rho_d)$, $\Phi(\pi_d) \neq \emptyset$ **then**
  return $true$ along with (Φ)
else
  return $false$ along with (π_d) {where π_d has no correspondence through Φ}
end if

V. Experimental results

The Entry Control Product Line comprises all the features involved in the management of the locks in a car. In this study, we focus on the following features:

- *Power lock*: this is the basic locking functionality which manages the locking/unlocking according to key button press and courtesy switch press.
- *Last Door Closed Lock*: delays the locking of the doors until all the doors are closed. It is applicable when the lock command appends while a door is open.
- *Door lock*: automates the locking of doors when the vehicle starts.
- *Door unlock*: automates the unlocking of door(s) when the vehicle stops.
- *Anti-lockout*: is intended to prevent the inadvertent lockout situations: the driver is out of the car with the key inside and all the doors locked.
- *Post crash unlock*: unlocks all the doors in a post crash situation.
- *Theft security lock*: secures the car with a second lock.

Each feature is represented as a pair of state machines containing 3 to 8 states.

A. Global analysis results

We have built an experimental tool called *FSMv-Verifier* to implement and validate the proposed method. We ran Algorithm 1 on the features *Power lock*, *Last Door Closed Lock*, *Post crash unlock*, *Door lock*, *Door unlock*, *Anti-lockout* and *Theft security lock* from the ECPL. A bug was found in the

Feature	*PL & LDCL*	*PCU*	*DL*	*DU*	*AL*	*TSL*
Exec. time	2546 ms	1267 ms	1468 ms	1453 ms	1177 ms	1980 ms
pan.exe	1730 ms	86 ms	377 ms	385 ms	28 ms	1130 ms

TABLE I
EXECUTION TIME OF FSMV-VERIFIER ON ALGORITHM 1

Door lock feature (see below). Figure I presents the execution time of Algorithm 1 for each feature including the calls to *SPIN*, *GCC*, *PAN* and the cleaning of temporary files. The execution time of the analysis itself (execution of *pan.exe*) is given in the second line.

B. Detailed analysis of Door lock

We ran Algorithm 1 on the feature *Door lock* and a bug was found. In Des_{dl}, the transition from middle oval state to the round state labeled with ***Poff:ShiftOutOfPark*** is incorrect. We have $\Phi(\langle Auto, Poff\rangle) \rightarrow \emptyset$. If we remove the concerned transition, and run again Algorithm 1, we get the following conformance relation:

- $\Phi(\langle Moff, Poff\rangle) \rightarrow \{\langle Disable, X, X\rangle\}$
- $\Phi(\langle Moff, Speed\rangle) \rightarrow \{\langle Enable, Manual, Speed\rangle, \langle Enable, Auto, Speed\rangle\}$
- $\Phi(\langle Auto, Poff\rangle) \rightarrow \langle Enable, Auto, Park\rangle$
- $\Phi(\langle Auto, Speed\rangle) \rightarrow \{\langle Enable, Auto, Speed\rangle, \langle Enable, Manual, Speed\rangle\}$

One can see that a design configuration may be associated to more than one requirement configuration.

VI. CONCLUDING REMARKS

The paper argued that in an SPL, both the design and requirements may contain variability information and that there is a need for relating these. The novel aspect of the proposed work is to relate the variability from the design level to the requirement level.

A novel model called FSMv is proposed for modeling both requirements and design with variabilities and a new conformance relation over FSMv models are also defined. Algorithm 1 for checking the conformance between the design and requirement models of a feature has been proposed. A prototype tool has been implemented and was used for verifying an industrial scale product line.

The proposed approach assumes that the requirements and designs are expressed as finite state machines and hence can be applied only on finite state systems. For general designs involving infinite states, one needs to build finite state abstractions and apply the proposed approach. In fact, for the case study, we indeed developed finite state abstractions of the entry control product line's features and applied the tool. We are also extending this work to general infinite state systems.

In this paper, the expected behavior is an early representation of the feature but it could also represent a safety property of the feature. For this, FSMv can be extended with an accepting condition such as that used in Büchi automata. This is an interesting future work direction. The conformance relation used in this paper, is based on the language refinement. Other relations like bi-simulation, failure traces could also be explored.

In many SPL, the configuration parameters are shared from a feature to another. Moreover, the features follow some composition constraints in order to *weave* a product. These constraints are usually given by a feature diagram. Still, these constraints exist differently at the requirement and design level. One could extend the proposed approach and check that every possible product at the design level conforms to a product at the requirement level. To do so, one can consider all the possible products (feature combination) at the design level that respect the composition constraint. When considering the ϕ mapping, one can relate every design product to one (or many) requirement product(s). Finally, one can check that the corresponding requirement product respects their composition constraint. If not, the design allows features composition that are not specified in the requirement.

REFERENCES

[1] K. Czarnecki and U. Eisenecker, *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley Professional, June 2000.

[2] A. Metzger and K. Pohl, "Variability management in software product line engineering," in *ICSE COMPANION '07: Companion to the proceedings of the 29th International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 186–187.

[3] K. Czarnecki, "Mapping features to models: A template approach based on superimposed variants," in *GPCE 2005 - Generative Programming and Component Enginering. 4th International Conference*. Springer, 2005, pp. 422–437.

[4] BigLeverSoftware, "The software product line lifecycle framework," 2008.

[5] G. J. Holzmann, *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, September 2003. [Online]. Available: http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0321228626

[6] H. Gomaa and D. L. Webber, "Modeling adaptive and evolvable software product lines using the variation point model," in *HICSS '04: Proceedings of the Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS'04) - Track 9*. Washington, DC, USA: IEEE Computer Society, 2004, p. 90268.3.

[7] H. Gomaa and E. Olimpiew, "Managing variability in reusable requirement models for software product lines," in *High Confidence Software Reuse in Large Systems*, ser. Lecture Notes in Computer Science, H. Mei, Ed. Springer Berlin / Heidelberg, 2008, vol. 5030, pp. 182–185. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-68073-4_17

[8] K. Berg, J. Bishop, and D. Muthig, "Tracing software product line variability: from problem to solution space," in *SAICSIT '05: Proceedings of the 2005 annual research conference of the South African institute of computer scientists and information technologists on IT research in developing countries*. , Republic of South Africa: South African Institute for Computer Scientists and Information Technologists, 2005, pp. 182–191.

[9] A. Metzger, K. Pohl, P. Heymans, P.-Y. Schobbens, and G. Saval, "Disambiguating the documentation of variability in software product lines: A separation of concerns, formalization and automated analysis," in *Requirements Engineering Conference, 2007. RE '07. 15th IEEE International*, 2007, pp. 243–253. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4384187

[10] M. Riebisch and R. Brcina, "Optimizing design for variability using traceability links," in *ECBS '08: Proceedings of the 15th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 235–244.
[11] H. C. Li, S. Krishnamurthi, and K. Fisler, "Modular verification of open features using three-valued model checking," *Autom. Softw. Eng.*, vol. 12, no. 3, pp. 349–382, 2005.
[12] S. Krishnamurthi and K. Fisler, "Foundations of incremental aspect model-checking," *ACM Trans. Softw. Eng. Methodol.*, vol. 16, no. 2, p. 39, 2007.
[13] J. Liu, S. Basu, and R. Lutz, "Compositional model checking of software product lines using variation point obligations," *Automated Software Engineering*, vol. 18, pp. 39–76, 2011, 10.1007/s10515-010-0075-7. [Online]. Available: http://dx.doi.org/10.1007/s10515-010-0075-7
[14] K. Lauenroth, K. Pohl, and S. Toehning, "Model checking of domain artifacts in product line engineering," in *ASE '09: Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 269–280.
[15] K. Lauenroth, A. Metzger, and K. Pohl, "Quality assurance in the presence of variability," SSE, Institut fur Informatik und Wirtschaftsinformatik, univertitat Duisburg Essen, Tech. Rep., 2011.
[16] A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, and J.-F. Raskin, "Model checking lots of systems: efficient verification of temporal properties in software product lines," in *ICSE '10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*. New York, NY, USA: ACM, 2010, pp. 335–344.
[17] A. Classen, P. Heymans, P.-Y. Schobbens, and A. Legay, "Symbolic model checking of software product lines," in *33rd International Conference on Software Engineering, ICSE 2011, May 21-28, 2011, Waikiki, Honolulu, Hawaii, Proceedings*. ACM, 2011, pp. 321–330, acceptance rate: 14http://2011.icse-conferences.org/
[18] M. Cordy, A. Classen, G. Perrouin, P. Heymans, P.-Y. Schobbens, and A. Legay, "Simulation relation for software product lines: Foundations for scalable model checking (to appear)," in *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland, Proceedings*, June 2012, acceptance rate: 21[Online]. Available: http://www.ifi.uzh.ch/icse2012/
[19] A. Gruler, M. Leucker, and K. D. Scheidemann, "Modeling and model checking software product lines," in *FMOODS*, ser. Lecture Notes in Computer Science, G. Barthe and F. S. de Boer, Eds., vol. 5051. Springer, 2008, pp. 113–131.
[20] K. G. Larsen, U. Nyman, and A. Wasowski, "Modal i/o automata for interface and product line theories," in *ESOP*, ser. Lecture Notes in Computer Science, R. D. Nicola, Ed., vol. 4421. Springer, 2007, pp. 64–79.
[21] U. Nyman, "Modal transition systems as the basis for interface theories and product lines," Ph.D. dissertation, PhD thesis, Department of computer science, Aalborg University, Denmark, 2008.
[22] J.-B. Raclet, B. Caillaud, E. Badouel, A. Legay, A. Benveniste, and R. Passerone, "Modal interfaces: Unifying interface automata and modal specifications," in *EMSOFT*, C. M. Kirsch and R. Wilhelm, Eds. ACM, 2009.
[23] A. Fantechi and S. Gnesi, "Formal modeling for product families engineering," in *SPLC*, SPLC'08, Ed. IEEE Computer Society, 2008, pp. 193–202.
[24] P. Asirelli, M. H. ter Beek, S. Gnesi, and A. Fantechi, "Deontic logics for modeling behavioural variability," in *VaMoS*, 2009, pp. 71–76.
[25] P. Asirelli, M. H. ter Beek, A. Fantechi, and S. Gnesi, "A logical framework to deal with variability." in *IFM'10*, 2010, pp. 43–58.
[26] A. Gruler, M. Leucker, and K. D. Scheidemann, "Calculating and modeling common parts of software product lines," in *SPLC*, SPLC'08, Ed. IEEE Computer Society, 2008, pp. 203–212.
[27] D. Benavides, P. Trinidad, and A. Ruiz-Corts, "Using constraint programming to reason on feature models," in *in the seventeenth international conference on software engineering and knowledge engineering*, 2005.
[28] J.-V. Millo, S. Mohalik, and S. Ramesh, "Integrated analysis of software product lines: A constraint based framework for consistency, liveness, and commonness checking," in *ISEC: India Software Enginnering Conference*, 2011.
[29] T. Thum, D. Batory, and C. Kastner, "Reasoning about edits to feature models," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 254–264. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2009.5070526
[30] T. Than Tun, Q. Boucher, A. Classen, A. Hubaux, and P. Heymans, "Relating requirements and feature configurations: a systematic approach," in *Proceedings of the 13th International Software Product Line Conference*, ser. SPLC '09. Pittsburgh, PA, USA: Carnegie Mellon University, 2009, pp. 201–210. [Online]. Available: http://dl.acm.org/citation.cfm?id=1753235.1753263
[31] R. E. Lopez-Herrejon and A. Egyed, "Detecting inconsistencies in multi-view models with variability," in *ECMFA*, ser. Lecture Notes in Computer Science, T. Kühne, B. Selic, M.-P. Gervais, and F. Terrier, Eds., vol. 6138. Springer, 2010, pp. 217–232.
[32] D. S. Batory, "Feature models, grammars, and propositional formulas," in *SPLC*, ser. Lecture Notes in Computer Science, J. H. Obbink and K. Pohl, Eds., vol. 3714. Springer, 2005, pp. 7–20.
[33] D. Benavides, S. Segura, and A. Ruiz-Corts, "Automated analysis of feature models 20 years later: a literature review," *Information Systems*, vol. 35, no. 6, pp. –, 2010. [Online]. Available: http://dx.doi.org/10.1016/j.is.2010.01.001
[34] M. Y. Vardi and P. Wolper, "An automata-theoretic approach to automatic program verification," in *Proceedings of the First Annual IEEE Symposium on Logic in Computer Science*, Cambridge, Ed., June 1986, pp. 322–331.
[35] SPLC'08, Ed., *Software Product Lines, 12th International Conference, SPLC 2008, Limerick, Ireland, September 8-12, 2008, Proceedings*. IEEE Computer Society, 2008.

APPENDIX

The following presents the structure of the ProMela file called "fsmvInputForSpin" generated from the Door lock example presented in the paper. Algorithm 1 runs the three following command in sequence:

1) spin -a ./fsmvInputForSpin
2) gcc pan.c -DPRINTF -o pan
3) pan -X -m10000000 -w25 -c10000 -a -n

Thanks to the "-DPRINTF" and the "-c" options, the output of pan contains a line of comment with the values of the calibration parameters and variation points every times the never claim is violated, such as "vp_v_DL_ENA: 0, vp_v_transmission: 0, vp_v_configuration: 0,vp_cp1: 1, vp_cp2: 0" (the explicit values associated to the integer values are recovered from the #define statement at the begin of the "fsmvInputForSpin" file). A line of comment is generated for every couple of configuration which does not match.

```
/*The calibration parameter's values*/
#define  d_cp1_Moff 0
#define  d_cp1_Auto 1
//same for cp2 values

/*The variation point's values*/
#define  r_v_DL_ENA_Enable 0
#define  r_v_DL_ENA_Disable 1
//same for the other VP values

/*The Events*/
#define  evt_e 0
#define  evt_AllDoorsClosed 1
//...
#define  evt_Unlock 7

/*The states of the design model*/
```

```
#define  des_ADL_Inactive 0
//...
#define  des_ADL_Done 5
#define des_error 6

/*The states of the requirement model*/
#define  req_ADL_Inactive 0
//...
#define req_error 4

/*these channels are used to forward
the event in the design and requirement
from the environment*/
chan evts_req= [0] of {byte};
chan evts_des= [0] of {byte};

/*State variable*/
byte req_state;
byte des_state;

/*Initialization  variables*/
byte vp_v_DL_ENA;
byte vp_v_transmission;
byte vp_v_configuration;
byte vp_cp1;
byte vp_cp2;

/*This flag is used to ensure that the
never claim is checked only after that
the environment, the design, and the
requirement have run but not in between*/
byte flag;

proctype environmentModel(){
do
::flag==0 -> flag=1;
atomic{if
::(1)-> evts_des! evt_e;
        evts_req! evt_e;
//Same for every other events
fi;}
od;
 };

proctype requirementModel() {
mtype currentEvent;
req_state=req_ADL_Inactive;
do
::flag==2-> evts_req?currentEvent;
if
//A case for every valid state
//As described in Section IV-A
::else -> req_state = req_error;
fi;    flag=0;
od;
};

proctype designModel() {
mtype currentEvent;
des_state=des_ADL_Inactive;
do
::flag==1-> evts_des?currentEvent;
if
//A case for every valid state
//As described in Section IV-A
::else -> des_state = des_error;
fi;    flag=2;
od;
};

/*never claim definition*/
never {
TO_init:
if
  ::(flag==0
    && req_state==req_error
    && des_state!=des_error)
    ->//print the configuration

    goto accept_S9
  ::(1) -> goto TO_init
fi;

accept_S9:
if
  ::(1) -> goto TO_init
fi;
}

/*The initialization sequence*/
init{
flag=0;
atomic{ if
//All the possible configurations
//of the requirement
//are listed and selected randomly
fi;}

atomic{
if
//All the possible configurations
//of the design
//are listed and selected randomly
  fi;}

atomic
{
  run environmentModel();
  run requirementModel();
  run designModel();
 }
}
```