Compositional Verification of Evolving Software Product Lines

Jean-Vivien Millo\textsuperscript{1} and S Ramesh\textsuperscript{2} and Shankara Narayanan Krishna\textsuperscript{3} and Ganesh Khandu Narwane\textsuperscript{4}

\textsuperscript{1} EPI AOSTE, INRIA Sophia-Antipolis, France
\textsuperscript{2} Global General Motors R&D, TCI Bangalore, India
\textsuperscript{3} Department of CSE, IIT Bombay, Mumbai, India
\textsuperscript{4} Homi Bhabha National Institute, Mumbai, India

Abstract. This paper presents a novel approach to the design verification of Software Product Lines (SPL). The proposed approach assumes that the requirements and designs are modeled as finite state machines with variability information. The variability information at the requirement and design levels are expressed differently and at different levels of abstraction. Also the proposed approach supports verification of SPL in which new features and variability may be added incrementally. Given the design and requirements of an SPL, the proposed design verification method ensures that every product at the design level behaviorally conforms to a product at the requirement level. The conformance procedure is compositional in the sense that the verification of an entire SPL consisting of multiple features is reduced to the verification of the individual features. The method has been implemented and demonstrated in a prototype tool SPLEnD (SPL Engine for Design Verification) on a couple of fairly large case studies.

1 Introduction

Large industrial software systems are often developed as Software Product Line (SPL) with a common core set of features which are developed once and reused across all the products. The products in an SPL differ on a small set of features which are specified using variation points. The focus of this paper is on modeling and analysis of SPLs which have drawn the attention of researchers recently [1-3].

Many approaches have been proposed to describe SPLs, the most prominent one being feature diagrams. All these proposals seem to assume a global view of SPL as they start with a complete list of features and the variation points using a single vocabulary. All the subsequent SPL assets, like requirement documents, design models, source codes, test cases, documentations, share the same definition and vocabulary [4]. The assumption of a single homogeneous and global view of variability description is inapplicable in many practical settings, where there is no top level complete description of features and variabilities. They often evolve during the long lifetime of an SPL as new features and variabilities
are added during the evolution. Further, SPL developers tend to use different representations and vocabulary of variability at different stages of development: at the requirement level, a more abstract and intuitive description of variation points are used, while at the design level, the efficiency of implementation of variation points is of primary concern. For example, consider the case of an automotive SPL, where one variation point is the region of sale (e.g., Asia Pacific, Europe, North America etc). At the requirement level, this variation point is expressed directly as an enumeration variable assuming one value for every region. Whereas, at the design level, the variation point is expressed using two or three boolean variables; by setting the values of the boolean variable appropriately, the behavior specific to a region is selected at the time of deployment.

We present a design verification approach that is more suited to the above kind of evolving SPLs in which different representation of variabilities would be used at the requirement and design level. One natural and unique problem that arises in this context is to relate formally the variation points expressed at different levels of abstractions. Another challenge is the analysis complexity: the number of products is exponential in the number of variation points and hence product centric analyses are not scalable. We propose a compositional approach in which every feature of the SPL is first analyzed independently; the per-feature analysis results are then combined to get the analysis result for the whole SPL.

For capturing variability in the behavior of an SPL, we have extended the standard finite state machine model, which we call Finite State Machines with Variability, in short, FSMv. The behavior and variability of a feature at the requirement and design level can be modeled using FSMv. We define a conformance relation between FSMvs to relate the requirement and design models. This relation is based upon the standard language containment of state machines.

One unique feature of FSMv is that it provides a compositional operator for composing the feature state machines to obtain a model for an SPL. This operator thus enables incremental addition of features and variabilities. The proposed verification approach exploits the compositional structure of the SPL models to contain the analysis complexity.

![Fig. 1. The proposed verification framework.](image-url)
Figure 1 summarizes the proposed approach. It shows an SPL composed of features $f_1$ to $f_n$. Each feature has an FSMv model of its requirements (called FSMr) and an FSMv model derived from its design (called FSMd). The proposed analysis method checks whether the FSMd of every feature conforms to its FSMr (1$^{st}$ check). The output of this first step is a conformance relation between each pair of FSMr and FSMd. The obtained conformance relations are then used to check whether the actual behavior of the entire SPL conforms to the expected one (2$^{nd}$ check). We reduce this check to checking the satisfiability of a Quantified Boolean Formula (QBF). There is no need to build the entire behavioral model of the SPL in the second step.

We have built a prototype tool SPLEnD based upon this approach. This tool performs the first check using SPIN [6] while the the well-known QBF SAT solver CirQit [7] is used for the second step. We have experimented with the tool using modest industrial size examples with very encouraging results. An earlier version of this work (October 2012) can be found at [8].

1.1 Related works

FSMv and the proposed design verification approach were developed independently but has some apparent similarities with the FTS$^+$ model [2], which also extends finite state machines to include certain product variability information. However, there is a motivational difference between the two formalisms. The aim of FTS$^+$ is to model the entire SPL and hence there is a single global machine with a single global vocabulary for expressing variabilities; the variability information represents the presence/absence of features in the SPL. In contrast, our approach is based upon a different view of SPL: a feature with variability is an increment in functionality and an SPL is a collection of features. We use a single FSMv to model a feature and a whole SPL is modeled as a parallel composition of FSMv machines.

The difference in viewpoint has another consequence: FTS$^+$ models, since they model the entire SPL, tend to be large and hence has high analysis complexity. Efficient abstraction techniques are hence used for solving this problem [3]. Whereas, each FSMv models a fraction of functionality and hence can be analysed easily. Further, the entire SPL can be modeled as composition of FSMvs and can be efficiently analysed using composition techniques.

Many other behavioral models have also been proposed [9,10,11,12] which are usually coupled with a variability model such as OVM [5], the Czarnecki feature model [4], or VPM [13] to attain a fair level of variability expressibility. Unlike all these approaches, FTS$^+$ [2] and FSMv capture the variability in an explicit way which we find more intuitive.

The Variation Point Model (VPM) of Hassan Gomaa [13] distinguishes between variability at the requirement and design levels but no design verification approach has been presented. Kathrin Berg et al. [14] propose a model for variability handling throughout the life cycle of the SPL. Andreas Metzeger et al. [15] and M Riebisch et al. [16] provide a similar approach but they do not consider
the behavioral aspect. In the proposed approach, we extract the relation between requirement and design level variability from a behavioral analysis.

Kathi Fisler et al. \cite{fisler2008} have developed an analysis based on three-valued model checking of automata defined using step-wise refinement. Later on, Jing Liu et al. \cite{liu2014} have revisited Fisler’s approach to provide a much more efficient method. Recently, Maxime Cordy et al. have extended Fisler’s approach to LTL formula \cite{cordy2015}. Kim Lauenroth et al. \cite{lauenroth2016} as well as Andreas Classen et al. \cite{classen2017}, and Gruler et al. \cite{gruler2018} have developed model checking methods for SPL behavior. These methods are based on the verification of LTL/CTL/modal $\mu$ calculus formula.

All these verification methods assume a global view of variability and hence the representation of variability information is identical in both specification and the design. In contrast, in our work the specification and design involve variability information at different levels of abstraction and hence one needs mapping information between the two levels. Furthermore, our formalism allows incremental addition of functionality and variability and enables compositional verification.

### 2 Design Verification of a Single Feature

An SPL, in general consists of multiple features, each feature having different functionality and variability. A typical body control software of an automotive system is an SPL that has several features such as door lock, lighting, seat control etc. 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. In this section we will focus on modeling and relating the design of a single feature to its requirement.

#### 2.1 FSMv and language refinement

*Finite State Machines with Variability (FSMv)* is an extension of finite state machines, to represent all possible behaviours of a feature. Let $Var$ be a finite set of variables, each taking a value ranging over a finite set of values. Let $x \in Var$, and let $Dom(x)$ be the finite set of values that $x$ can take. The set of atomic formulae we consider are $x = a$, $x \neq a$, $x = y$, $x \neq y$ for $a \in Dom(x)$, and $x,y \in Var$. Let $A_{Var}$ denote the set of atomic formulae over $Var$. Let $\alpha$ represent a typical element of $A_{Var}$. Define

$$\Delta ::= \alpha \mid \neg \Delta \mid \Delta \land \Delta \mid \Delta \lor \Delta \mid \Delta \Rightarrow \Delta$$

to be the set of all well formed predicates over $Var$. 

Definition 1 (FSMv) An FSMv is a tuple $\mathcal{A} = \langle Q, q_0, \Sigma, \text{Var}, E, \rho \rangle$ where:

1. $Q$ is a finite set of states; $q_0$ is the initial state;
2. $\Sigma$ is a finite set of events;
3. Var is a finite set of variables;
4. $E \subseteq Q \times \Delta \times \Sigma \times Q$ gives the set of transitions. A transition $t = (s, g, a, s')$ represents a transition from state $s$ to state $s'$ on event $a$; the predicate $g$ is called a guard of the transition $t$; $g$ is consistent and defines the variability domain of the transition;
5. $\rho \in \Delta$ is a consistent predicate called the global predicate.

The variables in $\text{Var}$ determine the variability allowed in the feature with each possible valuation of the variables corresponding to a variant. The allowed values of the variables are constrained by the global predicate $\rho$. For example, if $\rho$ is $((x = 1) \lor (x = 2)) \land (x = y - 1)$, then the allowed variants are those for which the values for the pairs $(x, y)$ are $(1, 2), (2, 3)$. The predicate in a transition determines the variants to which the transition is applicable. While drawing a transition $t = (s, g, a, s')$, the edge connecting $s$ to $s'$ is decorated with $g : a$.

When $g$ is true, we simply write $a$ on the edge.

Definition 2 (Configuration) A configuration, denoted by $\pi$, is an assignment of values to the variables in $\text{Var}$. The set of all configurations is denoted by $\Pi_{\text{Var}}$, or $\Pi$, when $\text{Var}$ is clear from the context. Define $\Pi(\rho) = \{\pi | \pi \models \rho\}$ to be the set of all those configurations that satisfy $\rho$. The elements of $\Pi(\rho)$ are called valid configurations. Given a valid configuration $\pi$ and a transition $t = (s, g, a, s')$, we say that $t$ is enabled by $\pi$ if $\pi \models g$.

As a concrete example of an FSMv, consider the feature Door lock in automotive SPL which controls the locking of the doors when the vehicle starts. The expected behavior of this feature is modeled using the FSMv $\text{Req}_{dl}$ described pictorially in Figure 2. 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: $\text{DL}_{\text{Enable}}, \text{Transmission}_{dl}$ and $\text{DL}_{\text{User}_\text{Pref}}$. The top box denotes the values that these variables can assume, and the bottom box gives the global predicate ($\rho$) associated with the machine. $\rho$ ensures that in every valid configuration, the variable $\text{Transmission}_{dl}$ having the value $\text{Manual}$ implies that $\text{DL}_{\text{User}_\text{Pref}}$ takes the value $\text{Speed}$. This captures the fact that in manual transmission, there is no park position on the gearbox. To avoid clutter, we have replaced guards of the form $x = i$ with $i$ in the figure. The transition labeled with $\text{Disable} : *$ means that when $\text{DL}_{\text{Enable}}$ assumes the value $\text{Disable}$, it stalls on any event.

Requirement against Design 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 $\text{Manual} \implies \text{Speed}$ in the Door lock example.
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 *calibration parameters* ranging over a set of boolean values. Further, the constraint on the calibration parameters ($\rho$) takes the special form of the list of the possible configurations of the calibration parameters in order to easily configure the design.

FSMv can capture both the design as well as the requirements of a feature. We distinguish the requirement and design models by denoting them FSMr and FSMd respectively. Figure 2 presents the FSMr, $Req_{dl}$, of the feature *Door lock*. The FSMd, $Des_{dl}$, of the feature *Door lock* is presented in Figure 3. The structure of $Des_{dl}$ is similar to $Req_{dl}$ except that the top elliptical shaped state in Figure 2 is split into two states (the top and the bottom elliptical shaped states) in Figure 3. The top 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. Two variables $Cp1$ and $Cp2$ encode the possible configurations in the FSMd. The box in Figure 3 depicts the set of possible values of these. $Cp1 = Auto$ corresponds to the configuration in which the transmission is *Auto* whereas $Cp1 = Moff$ corresponds to either the manual transmission or the case when $Cp1$ is disabled; similarly, $Cp2 = Speed$ means that the user preference is set on *Speed*, while $Cp2 = Poff$ means either *Park* or the case when $Cp2$ is disabled.
2.2 Variants of FSMv and Conformance

Having described the design and requirement behaviour of a feature \( f \) using FSMd and FSMr respectively, we now define the notions of variants and conformance. A variant of an FSMv corresponds to one of the several possible behaviours of the feature (at the design, requirement level respectively). Given a feature \( f \), and a (FSMd, FSMr) pair corresponding to \( f \), we say that the design of \( f \) conforms to the requirements of \( f \) provided every variant of the FSMd has a corresponding FSMr variant.

Definition 3 (Variant of an FSMv) Let \( A = \langle Q, q_0, \Sigma, \text{Var}, E, \rho \rangle \) be an FSMv and \( \pi \in \Pi(\rho) \) be a valid configuration of \( A \). A variant of \( A \) is an FSM obtained by retaining only transitions \( t = (s, g, a, s') \), and states \( s, s' \) such that \( g \models \pi \).

Once the relevant states and transitions are identified, we remove the guards \( g \) from all the transitions; \( \rho \) is also removed. The resultant FSM is denoted \( A \downarrow \pi \).

In the example of FSMr for the feature Door lock, the variant \( \text{Req}_{dl} \downarrow \langle \text{Enable}, \text{Auto}, \text{Park} \rangle \) does not contain the transitions with the event \( \text{Speed} > n \) and \( * \). We compare the FSMd and FSMr of a feature \( f \) using their variants. Given an FSMv \( A \), we associate with each configuration \( \pi \) of \( A \) the language of the FSM \( A \downarrow \pi \), denoted by \( L(A \downarrow \pi) \). We say that an FSMd \( A_d \) conforms to an FSMr \( A_r \) if and only if the behaviour of every variant of \( A_d \) is contained in the behaviour of some variant of \( A_r \).

Definition 4 (The conformance mapping \( \Phi \)) Let \( A_r \) and \( A_d \) be a pair of FSMr and FSMd respectively with global predicates \( \rho^d \) and \( \rho^r \). Let \( \Pi_d, \Pi_r \) be the set of all design, requirement configurations. Then \( A_d \) conforms to \( A_r \) denoted \( A_d \leq \Phi A_r \) if there exists a mapping \( \Phi : \Pi_d(\rho^d) \to 2^{\Pi_r(\rho^r)} \) such that \( \forall \pi_d \in \Pi_d(\rho^d), \exists \pi_r \in \Pi_r(\rho^r) \) satisfying \( L(A_d \downarrow \pi_d) \subseteq L(A_r \downarrow \pi_r) \). \( \Phi \) is called the conformance mapping.

In the feature Door lock, \( \Phi(\langle \text{Moff}, \text{Speed} \rangle) \) contains \( \langle \text{Enable}, \text{Manual}, \text{Speed} \rangle \) since \( L(\text{Des}_{dl} \downarrow \langle \text{Moff}, \text{Speed} \rangle) \subseteq L(\text{Req}_{dl} \downarrow \langle \text{Enable}, \text{Manual}, \text{Speed} \rangle) \).

2.3 Checking the conformance

Let \( f \) be a feature with FSMr \( \text{Req}_f \) and FSMd \( \text{Des}_f \). Then the conformance checking problem is to compute a mapping \( \Phi \) such that \( \text{Des}_f \leq \Phi \text{Req}_f \).

The conformance mapping is computed by comparing every projection of \( \text{Des}_f \) with every projection of \( \text{Req}_f \). Algorithm 1, given below, presents a possible implementation using the standard automata containment algorithm[22], as implemented in the SPIN model checker[6]. To use SPIN, one should describe the system along with the checked property in the Promela language[6]. Out of this description, SPIN generates the pan.c file which is the verifier for the system. After compilation, the pan(.exe) executable performs the verification.
Algorithm 1 starts by generating a Promela file containing the definition of (i) the environment, (ii) Des$_f$, (iii) Req$_f$, (iv) the initialization sequence and (v) 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 random couple of configurations. Then the environment, followed by Des$_f$ and Req$_f$ are run atomically. The never claim assertion is: \(\text{never}((\neg \text{error}(\text{Des}_f) \land \text{error}(\text{Req}_f)))\), where \(\text{error}(X)\) means that \(X\) is in error state. The never claim is violated when the design is not in the error state but the requirement process is in the error state. This corresponds to a design configuration \(\pi_d\) such that \(\text{Des}_f \downarrow \pi_d\) handles an event, while \(\text{Req}_f \downarrow \pi_r\) does not, for all possible requirement configurations \(\pi_r\).

Algorithm 1 runs the full verification algorithm of SPIN for every pair \((\pi_d, \pi_r)\) of design and requirement configurations. SPIN (i.e. \texttt{pan.exe}) returns the list of pairs for which the conformance condition is violated. Every other pair is added to the conformance mapping \(\Phi\). Lemma 5 proves the correctness of Algorithm 1.

Algorithm 1 implements the conformance checking using SPIN.

\begin{algorithm}
\textbf{Input}: Des$_f$, Req$_f$.
\textbf{Output}: The mapping \(\Phi\) 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.
2. Launch the full verification algorithm of SPIN.
3. Build the mapping \(\Phi\) from the output of SPIN.
4. Conclude whether the design conforms to the requirement
   \begin{verbatim}
   if \(\forall \pi_d \in H(\pi_d), \Phi(\pi_d) \neq \emptyset\) then
       return true along with (\(\Phi\))
   else
       return false along with \((\pi_d)\) (where \(\pi_d\) has no correspondence through \(\Phi\))
   end if
   \end{verbatim}
\end{algorithm}

\textbf{Lemma 5} Given FSM$_d$ Des$_f$ and FSM$_r$ Req$_f$ for a feature \(f\), let \((\pi_d, \pi_r)\) be a pair of design and requirement configurations. Then, \(L(\text{Des}_f \downarrow \pi_d) \nsubseteq L(\text{Req}_f \downarrow \pi_r)\) if and only if \(\neg \text{error}(\text{Des}_f) \land \text{error}(\text{Req}_f)\).

\textit{Proof.} Assume \(L(\text{Des}_f \downarrow \pi_d) \nsubseteq L(\text{Req}_f \downarrow \pi_r)\). Then there exists a word \(w \in L(\text{Des}_f \downarrow \pi_d)\) which is prefixed by \(u.e\), with \(u\) a finite prefix of a word in \(L(\text{Req}_f \downarrow \pi_r)\), and \(e\) an event such that \(u.e\) is not a prefix of any word in \(L(\text{Req}_f \downarrow \pi_r)\). In such a situation, \(\text{Des}_f\) does not go to the error state but \(\text{Req}_f\) does.

Conversely, if \(L(\text{Des}_f \downarrow \pi_d) \subseteq L(\text{Req}_f \downarrow \pi_r)\), then whenever \(\text{Des}_f\) is not in an error state, \(\text{Req}_f\) will also not be in an error state. \(\Box\)
3 Design Verification of SPL

In the previous section, we looked at individual features in an SPL and provided a method for comparing the design and requirements of a feature, both containing variabilities. In this section, we extend this method to verifying a whole SPL design against its requirements. An SPL is essentially a composition of multiple features satisfying certain constraints. We define a parallel composition operator over FSMv to model an SPL. The features in an SPL can interact and we follow one of the standard methods of allowing the composed FSMv models to share some common events, which correspond to two-party handshake communication events. A distinguishing aspect of the proposed parallel operator is that it takes into account the constraints across the composed machines. The constraints could be of various types, e.g. dependency and exclusion relations, and are modeled as predicates over variables of the composed features.

Definition 6 (Parallel composition of FSMv)

Let \( \mathcal{A}_x = (Q_x, \delta_x, \Sigma_x, Var_x, E_x, \rho_x) \), \( x \in \{1, 2\} \) be two FSMv’s with \( Var_1 \cap \) \( Var_2 = \emptyset \). Let \( H = \Sigma_1 \cap \Sigma_2 \) be the set of handshaking events. Let \( \rho_{12} \) be a predicate over \( Var_1 \cup \) \( Var_2 \), such that \( \rho_{12} \wedge \rho_1 \wedge \rho_2 \) is consistent. \( \rho_{12} \) is the composition predicate capturing the possible constraints between the variabilities of the two composed features. Let \( \rho = \rho_{12} \wedge \rho_1 \wedge \rho_2 \).

The parallel composition of \( \mathcal{A}_1 \) and \( \mathcal{A}_2 \) denoted by \( \mathcal{A} = \mathcal{A}_1 \parallel \mathcal{A}_2 \) is a tuple \( \langle Q_1 \times Q_2, (q_0^1, q_0^2), \Sigma_1 \cup \Sigma_2, Var_1 \cup \) \( Var_2, E, \rho \rangle \) with transitions defined as follows: Consider a state \((s_1, s_2) \in Q_1 \times Q_2\), and transitions \((s_1, g_1, a_1, s'_1, s'_2) \in E_1 \) and \((s_2, g_2, a_2, s'_2) \in E_2\).

1. If \( a_1 = a_2 = a \in H \), define \( ((s_1, s_2), g_1 \wedge g_2, a, (s'_1, s'_2)) \in E \), provided \( g_1 \wedge g_2 \) is consistent and \( g_1 \wedge g_2 \models \rho \).
2. If \( a_1 \in \Sigma_1 \setminus H \), define \( ((s_1, s_2), g_1, a_1, (s'_1, s'_2)) \in E, \) \( g_1 \models \rho \).
3. If \( a_2 \in \Sigma_2 \setminus H \), define \( ((s_1, s_2), g_2, a_2, (s'_1, s'_2)) \in E, \) \( g_2 \models \rho \).

For illustration, consider the feature Door unlock which automates the unlocking of the doors in a vehicle. Figure 4-a gives the FSMr of the feature extracted from the requirements. From the initial state, the feature becomes active when the event Lock happens. As soon as either the key is removed from ignition or the gear is shifted to park position, the doors get unlocked and the feature Door unlock becomes inactive. Figure 4-b presents the FSMd of the feature Door unlock. It is quite similar to the requirement except that the active state is split in two: the feature reacts to the ignition Off event in one state, and to the Shift Into Park event in another state.

Let us consider the composition of the two FSMrs of the features Door lock and Door unlock. The handshake events between the two features are Lock, Unlock, Door unlock, and Door unlock. In the composition, we introduce the following composition predicate: \( (DU\_Enable = Enable \Rightarrow DL\_Enable = Enable) \wedge Transmission_{dl} = Transmission_{du} \), which brings out the natural constraints that Door lock feature is enabled if and only if Door unlock is also enabled and the transmission status has to be the same.
The valid configurations after composition are restricted by the composition predicate. We provide a few definitions to define composite valid configurations.

**Definition 7 (Composing Configurations)** Let $A_1 = (Q_1, q_{01}, \Sigma_1, Var_1, E_1, \rho_1)$ be two FSMv’s, and let $A = A_1 \parallel A_2$ be as given by definition 6. Let $\rho = \rho_{12} \land \rho_1 \land \rho_2$ be the global predicate of $A$. Consider two valid configurations $\pi_1 \in \Pi(\rho_1)$ and $\pi_2 \in \Pi(\rho_2)$ of $A_1$ and $A_2$. The composition of $\pi_1$, $\pi_2$, denoted $\pi_{12}$ is a configuration over $Var_1 \cup Var_2$ such that $\pi_{12}$ agrees with $\pi_1$ over $Var_1$, and agrees with $\pi_2$ over $Var_2$, and $\pi_{12} \models \rho$. $\pi_{12}$ is a valid configuration of $A$ and we denote it by $\pi_{12} = \pi_1 + \pi_2$.

**Lemma 8** Let $A_1$ and $A_2$ be two FSMv’s. For each valid configuration $\pi$ of $A_1 \parallel A_2$, there are valid configurations $\pi_1$ of $A_1$ and $\pi_2$ of $A_2$ such that $\pi = \pi_1 + \pi_2$.

**Proof.** Let $\pi \in \Pi(\rho)$ with $\rho = \rho_{12} \land \rho_1 \land \rho_2$ be a valid configuration of $A_1 \parallel A_2$. $\rho_1$ and $\rho_2$ are the global predicates of $A_1$, $A_2$ respectively, and $\rho_{12}$ is the composition predicate of $A_1$, $A_2$. By definition of valid configuration, $\pi \models \rho$; hence $\pi \models \rho_1$ and $\pi \models \rho_2$. Since $\pi$ is a configuration over $Var_1 \cup Var_2$, let us consider the restriction of $\pi$ on $Var_1$, call the resulting configuration $\pi_1$. Then $\pi_1 \models \rho_1$. Similarly, call the restriction of $\pi$ on $Var_2$ as $\pi_2$. Then $\pi_2 \models \rho_2$. Then, $\pi_1, \pi_2$ are respectively valid configurations of $A_1$ and $A_2$. Hence, by definition 7 we obtain $\pi = \pi_1 + \pi_2$. □

In the example of feature Door Lock, the configuration $\langle Enable, Auto, Speed \rangle$ from $Req_{dl}$ can be composed with $\langle Enable, Auto, Key \rangle$ from $Req_{du}$ because the transmission is Auto in both (which is specified in the composition predicate). $\langle Enable, Auto, Speed, Enable, Auto, Key \rangle$ is a configuration of the parallel composition of $Req_{dl}$ with $Req_{du}$.

The parallel composition of FSMv’s is such that each variant of the composition of two FSMv’s is equal to the composition of variants of the individual FSMv’s.
Lemma 9 (Variants of a composed FSMv) Let $A_1$ and $A_2$ be two FSMv machines. Let $\pi$ be a valid configuration of $A_1 \parallel A_2$. Then $L([A_1 \parallel A_2] \downarrow \pi) = L(A_1 \downarrow \pi) \parallel L(A_2 \downarrow \pi)$.\footnote{The right hand side $\parallel$ refers to the standard communicating finite state machine composition.}

Proof. We review some preliminary definitions before the proof. In the following, the operation $\parallel$ stands for (i) shuffle of words, (ii) shuffle of languages, (iii) parallel composition of FSMs, and (iv) parallel composition of FSMv. The context is clear in each case; hence there is no confusion.

Definition 10 Let $\Sigma_1, \ldots, \Sigma_n$ be $n$ finite sets of symbols. Let $\Sigma$ be a finite set. Given a word $w \in \Sigma^*$, we denote by $w \downarrow \Sigma_i$, the unique subword of $w$ over $\Sigma_i$. For example, if $\Sigma_1 = \{a, b, c\}$, $\Sigma_2 = \{a, e, f\}$, and if we consider $w = aeefdefr \in \{a, d, e, f, r\}^*$, then $w \downarrow \Sigma_1 = ace$ and $w \downarrow \Sigma_2 = aeef$.\footnote{The right hand side $\parallel$ refers to the standard communicating finite state machine composition.}

Definition 11 (Asynchronous Shuffle) Let $\Sigma_1, \ldots, \Sigma_n$ be $n$ finite sets. Let $\Sigma = \bigcup_{i=1}^n \Sigma_i$. Consider $n$ words $u_1, u_2, \ldots, u_n, u_i \in \Sigma_i^*$. The asynchronous shuffle of $u_1, \ldots, u_n$ denoted $u_1 \parallel \cdots \parallel u_n$ is defined as $\{ w \mid w \downarrow \Sigma_i = u_i \}$.

As an example, consider $\Sigma_1 = \{a, b, c, f\}$, $\Sigma_2 = \{a, d, e, f\}$, $\Sigma_3 = \{c, d, f\}$, and the words $u_1 = abcf$, $u_2 = adfe$, $u_3 = def$. Then the word $w = abdefe$ is in $u_1 \parallel u_2 \parallel u_3$ since, $w \downarrow \Sigma_i = u_i$ for $i = 1, 2, 3$. Similarly, the word $w' = abcfde$ is also in $u_1 \parallel u_2 \parallel u_3$. However, the word $w'' = aebcfde$ is not in $u_1 \parallel u_2 \parallel u_3$, since $w'' \downarrow \Sigma_2 = aec$, not $u_2$.

The definition of shuffle can be extended from words to languages. We use the same notation $\parallel$ for the shuffle of sets, as well as for the shuffle of words.

The asynchronous shuffle of two languages $L_1, L_2$ is defined as $L_1 \parallel L_2 = \{w_1 \parallel w_2 \mid w_1 \in L_1, w_2 \in L_2\}$. For example, if $L_1 = \{abcf, abbf\}$ is a language over $\Sigma_1 = \{a, b, c, f\}$ and $L_2 = \{adfe\}$ is a language over $\{a, d, e, f\}$, then $L_1 \parallel L_2 = \{abcf \parallel adfe, abbf \parallel adfe\} = \{abcdefe, abcfde, abbcfe, abdefe, abdbfe, abffbe\}$.

Definition 12 Let $M_i = (Q_i, q_i, \Sigma_i, \delta_i)$ and $M_j = (Q_j, q_j, \Sigma_j, \delta_j)$ be complete FSMs. The asynchronous product of $M_i, M_j$ is defined as the FSM $M_i \parallel M_j = (Q_i \times Q_j, (q_i, q_j), \Sigma_i \cup \Sigma_j, \delta)$ where

1. $\delta((q, q'), a) = (\delta_i(q, a), \delta_j(q', a))$, $a \in \Sigma_i \cap \Sigma_j$,
2. $\delta((q, q'), a) = (\delta_i(q, a), q')$, $a \in \Sigma_i, a \notin \Sigma_j$,
3. $\delta((q, q'), a) = (q, \delta_j(q', a))$, $a \in \Sigma_j, a \notin \Sigma_i$.

On the common events, both FSMs move in parallel; otherwise, they move independent of each other.

It is known that $L(M_i \parallel M_j) = L(M_i) \parallel L(M_j)$. Now we start the proof of Lemma 9.

Consider a valid configuration $\pi$ of $A_1 \parallel A_2$. As seen in Lemma 8, we can find valid configurations $\pi_1$ of $A_1$ and $\pi_2$ of $A_2$ such that $\pi = \pi_1 + \pi_2$. The initial state of $A_1 \parallel A_2$ is $(q_0^1, q_0^2)$, where $q_0^1$ is the initial state of $A_1$ and $q_0^2$ is the initial...
Hence, \( L[A_1 \parallel A_2] \downarrow \pi \subseteq L(A_1 \downarrow \pi) \parallel L(A_2 \downarrow \pi) \). The converse can be shown in a similar way. \( \Box \)

**Refinement and Parallel Composition** The definition of parallel composition naturally lends itself to a notion of addition of conformance mappings between design and requirement pairs. Consider FSMr’s \( R_1, R_2 \) corresponding to two features \( f_1, f_2 \). Let \( D_1, D_2 \) be the corresponding FSM’s. Let \( \rho_1^d, \rho_2^d \) be the global predicates of \( R_1, R_2 \), and let \( \rho_1^r, \rho_2^r \) be the global predicates of \( D_1, D_2 \) respectively. Assume that \( D_1 \preceq \Phi_1, R_1 \) and \( D_2 \preceq \Phi_2, R_2 \). Let \( \rho^r = \rho_1^r \land \rho_2^r \land \rho_2^r \) be the global predicate of \( R_1 \parallel R_2 \); likewise, let \( \rho^d = \rho_1^d \land \rho_2^d \land \rho_2^d \) be the global predicate of \( D_1 \parallel D_2 \). We now want to ask if \( D_1 \parallel D_2 \) conforms to \( R_1 \parallel R_2 \). This amounts to computing a conformance mapping between \( D_1 \parallel D_2 \) and \( R_1 \parallel R_2 \) given \( \Phi_1, \Phi_2 \). Consider any valid configuration \( \pi^d \) of \( D_1 \parallel D_2 \). By Lemma 8, we can write \( \pi^d \) as \( \pi_1^d + \pi_2^d \), where \( \pi_1^d, \pi_2^d \) are valid configurations of \( D_1, D_2 \) respectively. Since \( D_1 \preceq \Phi_1, R_1 \) and \( D_2 \preceq \Phi_2, R_2 \), there exists valid configurations \( \pi_1^r \in \Phi_1(\pi_1^d) \) and \( \pi_2^r \in \Phi_2(\pi_2^d) \) in \( R_1, R_2 \) respectively. Given this, the addition of \( \Phi_1, \Phi_2 \) is defined as follows:

**Definition 13 (Addition of conformance mappings)** The addition of conformance mappings \( \Phi_1, \Phi_2 \) is defined to be a mapping \( \Phi = \Phi_1 + \Phi_2 \) as follows.

For every valid configuration \( \pi^d = \pi_1^d + \pi_2^d \) of \( D_1 \parallel D_2 \),

\[
\Phi(\pi^d) = \{ \pi^r \mid \pi^r \text{ is a valid configuration of } R_1 \parallel R_2, \pi^r = \pi_1^r + \pi_2^r \} 
\]

for valid configurations \( \pi_1^r \in \Phi_1(\pi_1^d), \pi_2^r \in \Phi_2(\pi_2^d) \).

**Lemma 14 (Conformance of composition)** Let \( R_1 \) and \( R_2 \) be two FSMr machines corresponding to features \( f_1, f_2 \), and let \( D_1 \) and \( D_2 \) be the corresponding FSMd machines. Let \( D_1 \preceq \Phi, R_1 \) and \( D_2 \preceq \Phi, R_2 \). Let \( \Phi = \Phi_1 + \Phi_2 \) and \( \pi^d \) be a valid configuration of \( D_1 \parallel D_2 \). Then, \( \forall \pi^r \in \Phi(\pi^d), L([D_1 \parallel D_2] \downarrow \pi^d) \subseteq L([R_1 \parallel R_2] \downarrow \pi^r) \).

**Proof.** Given a valid configuration \( \pi^d \) of \( D_1 \parallel D_2 \), we can write it as \( \pi_1^d + \pi_2^d \), where \( \pi_1^d, \pi_2^d \) are respectively valid configurations of \( D_1, D_2 \) (Lemma 8). Since \( D_1 \preceq \Phi, R_1 \) and \( D_2 \preceq \Phi, R_2 \), there exist valid configurations \( \pi_1^r \in \Phi_1(\pi_1^d) \) and \( \pi_2^r \in \Phi_2(\pi_2^d) \) such that \( L(D_1 \downarrow \pi_1^d) \subseteq L(R_1 \downarrow \pi_1^r) \) and \( L(D_2 \downarrow \pi_2^d) \subseteq L(R_2 \downarrow \pi_2^r) \).

Since \( \Phi \) has been computed, for every valid configuration \( \pi^d \) of \( D_1 \parallel D_2 \), there exists some valid configuration \( \pi^r \) of \( R_1 \parallel R_2 \). \( \pi^r \in \Phi(\pi^d) \). As \( \pi^r \) is valid, \( \pi^r \models \rho_1^r \land \rho_1^r \land \rho_2^r \); hence, \( \pi^r \) can be written as \( \pi_1^r + \pi_2^r \), where \( \pi_1^r, \pi_2^r \) are respectively valid configurations of \( R_1, R_2 \) (Lemma 8), and \( \pi_1^r \in \Phi_1(\pi_1^d), \pi_2^r \in \Phi_2(\pi_2^d) \) by definition 13.

\[
L([D_1 \parallel D_2] \downarrow \pi^d) = L(D_1 \downarrow \pi_1^d) \parallel L(D_2 \downarrow \pi_2^d) \text{ by lemma 9.}
\]

Similarly, \( L([R_1 \parallel R_2] \downarrow \pi^r) = L(R_1 \downarrow \pi_1^r) \parallel L(R_2 \downarrow \pi_2^r) \). This along with the observation that \( L(D_1 \downarrow \pi_1^d) \subseteq L(R_1 \downarrow \pi_1^r) \) and \( L(D_2 \downarrow \pi_2^d) \subseteq L(R_2 \downarrow \pi_2^r) \) gives \( L([D_1 \parallel D_2] \downarrow \pi^d) \subseteq L([R_1 \parallel R_2] \downarrow \pi^r) \). \( \Box \)
Considering the example, in the FSMr \(Req_{dl} \parallel Req_{du}\) with \(\rho_r : DL_{Enable} = DU_{Enable} \land Transmission_{dl} = Transmission_{du}\). Any configuration where \(DL_{Enable} = Enable\) but \(DU_{Enable} = Disable\) is invalid. However, \(\Phi(\langle Auto, Speed \rangle)\) contains only configurations where \(DL_{Enable} = Enable\), \(\Psi(\langle Moff, Poff \rangle)\) contains only configurations where \(DU_{Enable} = Disable\) and \(\langle Moff, Poff \rangle\) is a valid configuration of \(Des_{dl} \parallel Des_{du}\). So the design does not conform to the requirement. However, if we make the extra assumption that \(\rho_d : Cp_1 = Moff \land Cp_2 = Poff \iff Cp_3 = Moff \land Cp_4 = Poff\), then \(\langle Auto, Speed \rangle\) and \(\langle Moff, Poff \rangle\) are not compatible anymore and as a result the design conforms to the requirement.

### 3.1 Conformance Checking

Let \(F = \{f_1, ..., f_n\}\) be a set of features and \(\mathcal{F}\) be the complete system comprising the features in \(F\), along with the relations between the features. Let \(R_i\) be the FSMr modeling the expected behavior and variability of \(f_i\), and \(D_i\) the FSMd extracted from the design of \(f_i\). Let \(\rho_{12...n}^d\) and \(\rho_{12...n}^r\) be the compositional predicates for \(R_1 \parallel \cdots \parallel R_n\) and \(D_1 \parallel \cdots \parallel D_n\) respectively. Now we state the variability conformance problem for an SPL as follows: Does there exist a conformance mapping \(\Phi\) such that \(D_1 \parallel \cdots \parallel D_n \leq_{\Phi} R_1 \parallel \cdots \parallel \cdots \parallel R_n\)? A compositional approach to solve the problem is to:

(i) check whether the design of every feature conforms to its requirement using Algorithm 1; (ii) check whether every valid configuration of \(D_1 \parallel \cdots \parallel D_n\) can be mapped to a valid configuration of \(R_1 \parallel \cdots \parallel \cdots \parallel R_n\). This is the conformance condition.

### 3.2 Checking Conformance Using QBF

We implement the second check using QBF solving. Given FSMd’s \(D_1, \ldots, D_n\) and FSMr’s \(R_1, \ldots, R_n\).

1. Let \(Var(D_i) = \{v_{i1}^d, \ldots, v_{in}^d\}\) be the set of variables of design \(D_i\), and \(Var(R_i) = \{v_{i1}^r, \ldots, v_{im}^r\}\), the set of variables of requirement \(R_i\). Let \(\pi^d : \langle v_{i1}^d = a_1, \ldots, v_{in}^d = a_n \rangle\) be a configuration of \(D_i\). We denote by \(\pi^r_i(x_{i1}, \ldots, x_{in})\) a formula which takes \(n\) values from \(Dom(D_i)\), \(1 \leq i \leq n\) as arguments. If \(\langle v_{i1}^d = a_1, \ldots, v_{in}^d = a_n \rangle\) is a chosen assignment, then \(\pi^d_i(x_{i1}, \ldots, x_{in})\) is the conjunction \(\bigwedge_{j=1}^{n} (x_{ij} = a_j)\).

2. Given \(n\) FSMd’s and \(n\) FSMr’s check if \(D_i\) conforms to \(R_i\) for all \(1 \leq i \leq n\) using Algorithm 1. This gives the map \(\Phi_i\). Assume \(\Phi_i(\pi^d_i) = \{\pi^r_{i1}, \ldots, \pi^r_{im}\}\), where each of \(\pi^r_{i1}, \ldots, \pi^r_{im}\) are configurations of \(R_i\), that have been mapped by \(\Phi_i\) to some configuration \(\pi^d_i\) of \(D_i\).

3. We encode the above conformance mapping using the formula
   \(\Phi_i(x_{i1}, x_{i2}, \ldots, x_{in}) = \bigvee_{j=1}^{m} \pi^r_{ij}(y_{i1}, \ldots, y_{id})\), where \(x_{ij}\) takes values from \(Dom(v_{ij}^d)\), and \(y_{ij}\) from \(Dom(v_{ij}^r)\).

4. Let \(\varphi^d_{i,j} = \rho^d \land \rho_{ij}^d\) and \(\varphi^r_{i,j} = \rho^r \land \rho_{ij}^r\) represent respectively the propositional formulae which ensures consistency of the global predicates.
of \(D_1, D_j\) and \(R_i, R_j\) along with the compositional predicates \(\rho^d\) and \(\rho^r\). Given a set \(S \subseteq \{1, 2, \ldots, n\}\), \(\varphi^d_S\) and \(\varphi^r_S\) can be appropriately written.

The QBF formula for conformance checking is given by

\[
\Psi = \forall x_1 \ldots x_{n}\varphi^d_{1,2,\ldots,n} \Rightarrow \exists y_{11} \ldots y_{nj}(\Phi_1 \land \cdots \land \Phi_n \land \varphi^r_{1,2,\ldots,n})
\]

**Theorem 1.** Given a SPL, let \(f_1, \ldots, f_n\) be the set of features in a chosen product. Let \(D_i, R_i\) be the FSMd and FSMr for feature \(f_i\). Then \(D_1 \parallel \cdots \parallel D_n\) conforms to \(R_1 \parallel \cdots \parallel R_n\) iff \(\Psi \) holds.

**Proof.** Given \(D_i \leq_{\Phi} R_i\), assume that \(D_1 \parallel \cdots \parallel D_n\) conforms to \(R_1 \parallel \cdots \parallel R_n\). Then, by definition of conformance, it means that for all valid configurations \(\pi^d\) of \(D_1 \parallel \cdots \parallel D_n\), there exists a valid configuration \(\pi^r\) of \(R_1 \parallel \cdots \parallel R_n\) such that \(L([D_1 \parallel \cdots \parallel D_n] \downarrow \pi^d) \subseteq L([R_1 \parallel \cdots \parallel R_n] \downarrow \pi^r)\). Let \(\Phi\) be the conformance mapping such that \(\pi^r \in \Phi(\pi^d)\).

\(\pi^d\) is a valid configuration of \(D_1 \parallel \cdots \parallel D_n\) implies that \(\pi^d \models \bigwedge_{S \subseteq \{1,2,\ldots,n\}} \rho^d_S\), where \(\rho^d_S\) is the global predicate of \(D_i \parallel \cdots \parallel D_j\), when \(S = \{i_1, \ldots, i_j\}\). Using Lemma 8 repeatedly, we can then say that \(\pi^d = \pi^d_1 + \cdots + \pi^d_n\) for valid configurations \(\pi^d_1\) of \(D_1\). Since \(\pi^r \in \Phi(\pi^d)\), by definition of conformance mappings, \(\pi^r\) must be a valid configuration of \(R_1 \parallel \cdots \parallel R_n\), hence \(\pi^r = \pi^r_1 + \cdots + \pi^r_n\) (Lemma 3), such that \(\pi^r_i \in \Phi(\pi^d_i)\), for valid configurations \(\pi^r_i\) of \(R_i\). \(\pi^r\) is valid means \(\pi^r = \bigwedge_{S \subseteq \{1,2,\ldots,n\}} \rho^r_S\).

Given the above, we show that the QBF \(\Psi\) holds. The LHS of the QBF \(\Psi\) is the formula \(\varphi^d_{1,2,\ldots,n}\), which is the conjunction \(\rho^d_S\) for all subsets \(S\) of \(\{1,2,\ldots,n\}\). The forall quantifier outside would thus evaluate all configurations of \(D_1 \parallel \cdots \parallel D_n\) that satisfy \(\varphi^d_{1,2,\ldots,n}\); that is, which satisfy \(\bigwedge_{S \subseteq \{1,2,\ldots,n\}} \rho^d_S\); hence, all valid configurations of \(D_1 \parallel \cdots \parallel D_n\).

For the QBF to hold good, for all valid configurations of \(D_1 \parallel \cdots \parallel D_n\) that have been evaluated on the LHS, we must find some configuration of \(R_1 \parallel \cdots \parallel R_n\) that satisfies \(\Phi_1 \land \cdots \land \Phi_n \land \varphi^r_{1,2,\ldots,n}\) : (i) any configuration \(\pi\) of \(R_1 \parallel \cdots \parallel R_n\) that satisfies \(\varphi^r_{1,2,\ldots,n}\) would be valid; (ii) further, if it has to satisfy \(\Phi_1 \land \cdots \land \Phi_n\), it must agree with \(\pi^r_i \in \Phi_i(\pi^d_i)\) over \(Var(R_i)\) for all \(1 \leq i \leq n\). By Lemma 8 this means that \(\pi\) can be written as \(\pi^d_1 + \cdots + \pi^d_n\). Thus, for the QBF to hold, we must be able to find for each valid configuration \(\pi^d\) of \(D_1 \parallel \cdots \parallel D_n\), a valid configuration \(\pi^r\) of \(R_1 \parallel \cdots \parallel R_n\) which can be written as \(\pi^r_1 + \cdots + \pi^r_n\), where \(\pi^r_i \in \Phi_i(\pi^d_i)\) for each \(i\). But this is exactly what the mapping \(\Phi\) which checks for conformance of \(D_1 \parallel \cdots \parallel D_n\) with \(R_1 \parallel \cdots \parallel R_n\) does. Since we assume that \(\Phi\) exists, the QBF holds.

The converse can be shown in a similar way: that is, if the QBF formula \(\Psi\) holds, then \(D_1 \parallel \cdots \parallel D_n\) will conform to \(R_1 \parallel \cdots \parallel R_n\). \(\square\)

### 4 Implementation and Case Studies

Figure 3 pictorially describes the tool SPLEnD. It takes as input, a pair of xml files corresponding to FSMd, FSMr and outputs a PROMELA file. The
latter is fed to SPIN, which returns the conformance mappings, or declares non-conformance; given the conformance mapping the tool computes a QBF formula $\Psi$ which is fed to CirQit.

We considered two real case studies for our experimentation: Entry Control Product Line, ECPL having 7 features and Banking Software Product Line, BSPL, composed of 25 features. The details of the ECPL and BSPL case studies are given below. The FSMr, FSMd models of each feature contains less than 10 states.

5 ECPL and BSPL

In this section, we describe the two product lines that have been considered in the paper: (i) ECPL and (ii) BSPL.

5.1 ECPL

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 append 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 10 states.

![Fig. 5. Overview of SPLEnD](image-url)
The variability constraints of the ECPL

Figure 6 presents the feature diagram of the ECPL (a la Czarnecki [4]). This diagram presents the variability constraints of the ECPL at the requirement level ($\rho_f$). All the constraints represented by this diagram have to be considered during composition to guarantee the overall consistency of the SPL behavior. The dark gray boxes are features of the ECPL: Power lock, Anti-lockout, Door lock, Door unlock, and Post crash unlock. The light gray boxes are configurations. The black arrow from the “Manual” configuration to the “Shift out of park” configuration and to the “Shift into park” configuration says that if the transmission is manual, the targeted configurations cannot be selected. i.e. In “Manual” configuration, there is no “park” gear.

Fig. 6. The feature diagram of the ECPL.

5.2 BSPL

The Banking Software Product Line (BSPL) consists of 25 behavioral features. The BSPL is used to derived the software for ATM, Bank, Online Banking and Mobile Banking. Figure 7 presents the feature diagram of the BSPL.

Similar to ECPL, we ran Algorithm 1 on all the 25 features of BSPL. In section 4, Figure 10 presents the number of design configurations and execution time of Algorithm 1 for each feature. In the following, we elaborate on the FSMv of 2 features: (i) User Interface and (ii) Withdraw Money. The FSMd/FSMr for all the features has states between 2 and 10 (both inclusive). Figure 8 is the FSMr for feature User Interface, which has $UI$ as an event with global predicate $\rho = \{¬(uip = \text{Disable})\}$. There is only one boolean variable, $Var = \{uip\}$, $uip$ takes values from \{Enable, Disable\}.

Figure 9 is the FSMd for feature User Interface. This FSMd shares the event $UI$ with the FSMr and has global predicate $\rho = \{(\text{type} = 2D \lor \text{type} = \ldots}$
There are two variables, $Var = \{type, graphics\}$, $type$ takes values from $\{2D, 3D\}$, while $graphics$ takes values from $\{Enable, Disable\}$.

The analysis results for the two case studies are summarized in Figures 11 and 10 which gives the times taken by Algorithm 1. The number of product variants and the time taken for Algorithm 1 are very small in both case studies. In the case of ECPL, a bug was found in the feature $Door\ Lock$. In this case, after fixing the bug, for the second step we used SPIN which took 11 seconds. For BSPL, the second step was performed using the QBF approach and CirQit took just 0.005 seconds.

In $Des_{sl}$, the transition from the middle elliptical state to the round state labeled with $Poff : ShiftOutOfPark$ is incorrect; $\Phi((Auto, Poff)) = \emptyset$. Removing this transition fixes the bug.
Fig. 9. FSMd for feature: UserInterface.

| Sr. No. | Features                          | Design Variants | SPIN Time (Sec) |
|---------|-----------------------------------|-----------------|-----------------|
| 1       | UserInterface                     | 6               | 0.002           |
| 2       | CheckingBalance                   | 3               | 0.003           |
| 3       | WithdrawMoney                     | 8               | 0.027           |
| 4       | DepositMoney                      | 2               | 0.002           |
| 5       | PrintingStatement                 | 3               | 0.002           |
| 6       | Login                             | 1               | 0.001           |
| 7       | ATMLogin                          | 1               | 0.001           |
| 8       | ChangeAccountPassword             | 2               | 0.003           |
| 9       | PayBills                          | 2               | 0.003           |
| 10      | PrintingBalanceAfterWithdraw      | 2               | 0.003           |
| 11      | CheckingMoneyExchangeRate         | 2               | 0.003           |
| 12      | MoneyExchange                     | 2               | 0.004           |
| 13      | InternationalTransfer             | 2               | 0.006           |
| 14      | LocalTransferToOtherBank          | 1               | 0.004           |
| 15      | LanguageSelection                 | 2               | 0.001           |
| 16      | MobileTopUp                       | 2               | 0.002           |
| 17      | ChangeMaxLimitForWithdraw         | 1               | 0.003           |
| 18      | LocalTransferToSameBank           | 3               | 0.003           |
| 19      | AddBeneficiary                    | 1               | 0.002           |
| 20      | RemoveBeneficiary                 | 1               | 0.002           |
| 21      | CreateDemandDraft                 | 2               | 0.003           |
| 22      | ChequeClearance                   | 1               | 0.003           |
| 23      | FastWithdrawal                    | 1               | 0.002           |
| 24      | CreditCardPayment                 | 2               | 0.002           |
| 25      | UpdateContactDetails              | 2               | 0.004           |

Fig. 10. Execution time of FSMv-Verifier on Algorithm 1 for BSPL

| Features                  | PL & LDCL | PCU  | DL  | DU  | AL  | TSL  |
|---------------------------|-----------|------|-----|-----|-----|------|
| Design Variants           | 8         | 3    | 4   | 7   | 3   | 8    |
| SPIN Time (Sec)           | 0.436     | 0.031| 0.046| 0.109| 0.015| 0.218|

Fig. 11. Execution time of FSMv-Verifier on Algorithm 1 for ECPL
In the automotive domain, really very large SPLs are constructed [23]. Before undertaking the task of modeling such large examples, in order to quickly determine the scalability of our approach, we generated many random SPLs with 5000 to 25,000 features. Each of the corresponding FSMr/FSMd has two variables (four variants), and 3 to 8 states. Similar to the ECPL and BSPL cases, SPIN took very little time (less than 0.5 seconds) for each (FSMr, FSMd) pair. The composite FSMr/FSMd, and hence the QBF formula $\Psi$ has then 10,000 to 50,000 variables. As we can see from Figure 12, the time taken for the largest example is 196.69 seconds which is quite efficient. Encouraged by this result, we plan to take up the large industrial case studies.

| Variables in FSMr/FSMd | 10000 | 20000 | 30000 | 40000 | 50000 |
|------------------------|-------|-------|-------|-------|-------|
| CirQit 3.1.7 time (Sec)| 4.47  | 25.77 | 65.67 | 119.49| 196.69 |

Fig. 12. Execution time of QBF for Scalability

6 Conclusion

This paper motivated the need for extending the classical design verification problem to evolving SPL in which features and variability information can be added incrementally. The novel aspects of the proposed work are: (i) it verifies that the variability at the design level conforms to that at the requirement level, (ii) it is compositional and (iii) it reduces the conformance checking problem to QBF sat solving. A prototype tool has been implemented and experimented with modest sized examples with encouraging results.

References

1. Benavides, D., Segura, S., Ruiz-Corts, A.: Automated analysis of feature models 20 years later: a literature review. Information Systems 35(6) (2010)
2. Classen, A., Heymans, P., Schobbens, P.Y., Legay, A.: Symbolic model checking of software product lines. In: Proceedings of ICSE 2011. (2011) 321–330
3. Cordy, M., Classen, A., Perrouin, G., Heymans, P., Schobbens, P.Y., Legay, A.: Simulation relation for software product lines: Foundations for scalable model checking (to appear). In: Proceedings of ICSE 2012. (2012)
4. Czarnecki, K., Eisenecker, U.: Generative Programming: Methods, Tools, and Applications. Addison-Wesley Professional (2000)
5. Metzger, A., Pohl, K.: Variability management in software product line engineering. In: ICSE COMPANION ’07: Companion to the proceedings of the 29th International Conference on Software Engineering. (2007) 186–187
6. Holzmann, G.J.: The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley Professional (2003)
7. Gouliaeva, A., Bacchus, F.: Exploiting qbf duality on a circuit representation. In: Proceedings of AAAI 2010. (2010)
8. Narwane, G.K., Krishna, S.N., Millo, J.V., S.Ramesh: Compositional verification of evolving software product lines. In: http://www.cse.iitb.ac.in/~krishnas/TR2012.pdf. (2012)
9. Larsen, K.G., Nyman, U., Wasowski, A.: Modal i/o automata for interface and product line theories. In: Proceedings of ESOP 2007. (2007) 64–79
10. Raclet, J.B., Caillaud, B., Badouel, E., Legay, A., Benveniste, A., Passerone, R.: Modal interfaces: Unifying interface automata and modal specifications. In: Proceedings of EMSOFT 2009. (2009)
11. Fantechi, A., Gnesi, S.: Formal modeling for product families engineering. In: Proceedings of SPLC 2008. (2008) 193–202
12. Gruler, A., Leucker, M., Scheidemann, K.D.: Calculating and modeling common parts of software product lines. In: Proceedings of SPLC 2008. (2008) 203–212
13. Gomaa, H., Olimpiew, E.: Managing variability in reusable requirement models for software product lines. In: High Confidence Software Reuse in Large Systems. (2008) 182–185
14. Berg, K., Bishop, J., Muthig, D.: Tracing software product line variability: from problem to solution space. In: Proceedings of SAICSIT ’05, (2005) 182–191
15. Metzger, A., Pohl, K., Heymans, P., Schobbens, P.Y., Saval, G.: Disambiguating the documentation of variability in software product lines: A separation of concerns, formalization and automated analysis. In: Proceedings of RE 2007. (2007) 243–253
16. Riebisch, M., Brcau, R.: Optimizing design for variability using traceability links. In: Proceedings of ECBS 2008. (2008) 235–244
17. Krishnamurthi, S., Fisler, K.: Foundations of incremental aspect model-checking. ACM Trans. Softw. Eng. Methodol. 16(2) (2007) 39
18. Liu, J., Basu, S., Lutz, R.: Compositional model checking of software product lines using variation point obligations. Automated Software Engineering 18 (2011) 39–76
19. Cordy, M., Schobbens, P.Y., Heymans, P., Legay, A.: Behavioural modelling and verification of real-time software product lines. In: Proceedings of SPLC 2012. (2012) 66–75
20. Lauenroth, K., Metzger, A., Pohl, K.: Quality assurance in the presence of variability. Technical report, SSE, Institut fur Informatik und Wirtschaftsinformatik, universtitat Duisburg Essen (2011)
21. Gruler, A., Leucker, M., Scheidemann, K.D.: Modeling and model checking software product lines. In: Proceedings of FM00DS 2008. (2008) 113–131
22. Vardi, M.Y., Wolper, P.: An automata-theoretic approach to automatic program verification. In: Proceedings of LICS 1986. (1986) 322–331
23. Flores, R., Krueger, C., Clements, P.: Mega-scale product line engineering at general motors. In: Proceedings of SPLC 2012. (2012) 259–269