All-At-Once-Synthesis of Controllers from Scenario-Based Product Line Specifications

Maxime Cordy and Patrick Heymans
University of Namur, Belgium
{mcr|phe}@info.fundp.ac.be

Joel Greenyer
Leibniz Universität Hannover, Germany
greenyer@inf.uni-hannover.de

Erika Gressi
Politecnico di Milano, Italy
erika.gressi@mail.polimi.it

Jean-Marc Davril
University of Namur, Belgium
jmdavril@unamur.be

ABSTRACT

Software-intensive systems often consist of multiple components that interact to realize complex requirements. An additional dimension of complexity arises when one designs many variants of a system at once, that is, a software product line (SPL). We propose a scenario-based approach to design SPLs, based on a combination of Modal Sequence Diagrams (MSDs) and a feature model. It consists in associating every MSD to the set of variants that have to satisfy its specification. Variability constitutes a new source of complexity, which can lead to inconsistencies in the specification of one or multiple variants. It is therefore crucial to detect these inconsistencies, and to produce a controller for each variant that makes it behave so that it satisfies its specification. We present a new controller synthesis technique that checks the absence of inconsistencies in all variants at once, thereby more radically exploiting the similarities between them. Our method first translates the MSD specification into a variability-aware Büchi game, and then solves this game for all variants in a single execution. We implemented the approach in ScenarioTools, a software tool which we use to evaluate our algorithms against competing methods.

1. INTRODUCTION

Many software-intensive systems in manufacturing, transportation, or healthcare, consist of multiple components that provide increasingly complex functionalities. Single requirements are often realized by the interaction of several components and, due to concurrent environment events or user inputs, single components must often fulfil several requirements at the same time. In many domains, engineers build several variants (also called *products*) of the same system – a *Software Product Line* (SPL) – in order to satisfy customers' specific needs while maximizing reuse, and thereby reducing development costs and time to market. The differences between the variants (i.e. the variability) are commonly expressed in terms of *features*, i.e. functionalities that may or may not be present in a given variant. A Feature Model (FM) [24] is then built to specify which combinations of features are valid in regard to technical or economical constraints. As more features are added to the SPL, the number of possible variants grows exponentially in the worst case.

Designing a consistent specification for a single software is already a challenging task, but it becomes drastically harder when it comes to SPLs. If inconsistencies remain undetected, they could spread across many products, and a late detection can lead to costly iterations in the development cycle to repair them. Therefore, there is a need for techniques that allow engineers to specify *all* the products they (know they) have to build, detect inconsistencies within their specifications, and derive a controller describing how each product should be implemented.

In this paper, we propose a *Scenario-Based Product Line Specification* (SBPLS) framework that allows engineers to formally specify interactions in product lines of open reactive systems. The framework combines an FM with *Modal Sequence Diagrams* (MSDs), i.e. sequence diagrams with modalities defining scenarios that the system *may* or *must* (not) satisfy [19]. Scenario-based modeling approaches are a natural way for engineers to reason about inter-component behavior during early design. Additionally, MSDs can specify assumptions on what uncontrollable events may or must (not) happen in the environment of certain products [14, 5]. The association with an FM permits to specify functionalities implemented by a single feature [15], or even *feature interactions* that should or should not happen. The latter was not possible in previous SPL specification methods [16, 15]. A product's specification is *inconsistent* (or *unrealizable*) iff there exists a sequence of uncontrollable events that inevitably leads to a situation where the system is forced to violate a safety or liveness requirement.

In the past, we explored two approaches for checking the realizability of an SBPLS. The first transformed the consistency checking into a product-line model-checking problem, and used a dedicated model checker to verify all product specifications at once [16]. This approach, however, was incomplete in that it did not differentiate between (controllable) actions of the system, an (uncontrollable) actions of

SPLC '15 Nashville, TN USA

its environment; due to that, it could yield false positives.

Instead, a *game-based* approach [25] is needed. The game is played by the system against its environment, the former aiming to satisfy requirements and the latter trying to violate them. The specification is realizable iff there exists a *strategy* for the system to always satisfy the requirements regardless of what happens in the environment, provided that the latter respects the assumptions. The strategy can be seen as a *controller* for the system that implements the specification. Discovering a strategy is thus equivalent to synthesizing a controller for the system.

Our second method consisted in using game-solving algorithms to synthesize controllers for all products one at a time [15]. Our algorithms were *on the fly* (i.e. they could achieve the synthesis without exploring every execution of the game) and *incremental* (i.e. they could synthesize a controller for a product more rapidly on the basis of a controller for the previously synthesized product). Although the incrementality already allows the algorithms to partially exploit commonalities between variants, its efficiency depends on the order in which products are synthesized [15].

In this work, we present a novel game-based approach that attempts to synthesize controllers for all products of an SPL at once. The main difference with the previous work is that this new method considers all variants at the same time, thereby exploiting more radically the similarities between the products. However, this is achieved at the cost of keeping track of the variability between products *during* the synthesis. Another difference is that this method does not use on-the-fly game-solving exploration. The starting point of our method is the formal definition of SBPLS. Therein, MSDs are associated to a formula encoding the sets of products for which they are part of the specification. Then, we transform such a specification into a featured game graph whose structure can be modified depending on the features of a considered product. Finally, we propose algorithms that can, from this game graph, determine the consistency of the specification of all products and derive a controller for the consistent ones in a single synthesis run. It makes use of the variability information contained in the game graph to avoid redundant work during the synthesis of the products.

We carried out experiments to evaluate the performance of our new algorithm with respect to (1) a method that synthesizes all the products separately and successively, and (2) our previous on-the-fly, incremental algorithms. The results show that our new algorithm always outperforms the product-by-product method. It can also compete with our previous algorithm, as it sometimes performs better than the latter depending on the case. This indicates that the new approach we propose is viable *per se*, but could be complemented with on-the-fly exploration heuristics to further extend their efficiency.

Structure. We introduce the foundations in Sect. 2. We define SBPLS in Section 3, and introduce the game structure that models the realizabilty-checking problem for an SBPLS in Sect. 4. Section 5 presents the synthesis algorithms. Implementation and evaluation are discussed in Sect. 6. Last, we discuss related work in Sect. 7.

2. FOUNDATIONS

In the following, we introduce the foundations of Feature Models (FMs) and Modal Sequence Diagrams (MSDs).

2.1 Feature Models

Feature Models (FM) are commonly used to specify the variability between the products of an SPL. Basically, a FM is a tree where nodes are features and edges specify how features are decomposed into child features. Each parent-child relationship has a type which constrain the valid sets of features that can be found in products. The usual decomposition types are *AND*, *OR*, and *XOR*, and define that when a parent feature is included, all, at least one, or exactly one, child feature must be included, respectively. One can also add cross-tree constraints to specify additional dependencies between features. These are Boolean formulae $\Phi \in 2^{2^F}$ over the set of features. The semantics of an FM fm, noted $[\![fm]\!]$, is commonly defined as the sets of products that satisfy the decomposition hierarchy and the additional constraints [27].

Figure 1 shows the FM of a tea vending machine example. The root feature $VendingMachine$ is mandatory. It has two child features, i.e. an optional feature Cup (which models the capability to provide plastic cups) and a mandatory feature Tea. The latter has three additional child features, such that $Sugar$ is optional whereas $Water$ and $TeaBag$ are mandatory. Finally, $TeaBag$ has two child features ($Green$ and $Lemon$) sharing an OR relationship, meaning that at least one of them must be present in a product. Altogether, this FM defines twelve valid product variants.

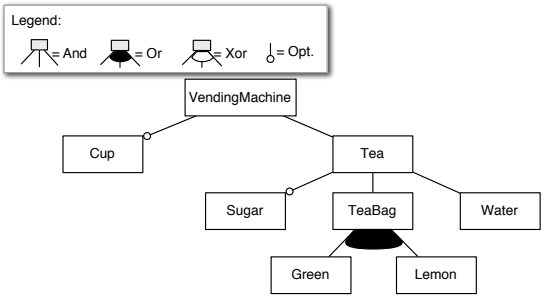


Figure 1: The feature model of the tea vending machine

2.2 Modal Sequence Diagrams

Leaning on Live Sequence Charts [12, 20], Harel and Maoz proposed Modal Sequence Diagrams (MSDs) as a formal interpretation of UML sequence diagrams [19]. MSDs specify the message-based communication behavior of a set of *objects*, which we call *object system*. We consider *open systems* where the object system is partitioned into controllable *system objects* and uncontrollable *environment objects*. We only consider *synchronous* messages, where the sending and receiving of a message together form a single *event*. Messages sent from system objects are called *controllable events* (or system events); those sent from environment objects are called *uncontrollable events* (or environment events). Then a run of the system is a sequence of controllable events and uncontrollable events.

An *MSD specification* consists of a set of MSDs. We consider that all MSDs are *universal*, that is, they specify properties that all runs of the system must satisfy. where MSDs can not only specify *requirements* on how the system objects must react to the environment, but also *assumptions* on what may, will, or will not happen in the environment.

Figure 2 shows the MSD specification of a tea vending machine. In addition to MSDs, we use a composite structure diagram (CSD) to specify the structure of the object system. Our example consists of the vending machine m, its dispensing unit d, and a user u. System objects (e.g. m) have a rectangular shape; environment objects (e.g. u) have a cloud-like shape. The objects are instances of classes that we model in a class diagram (CD) shown on the top left of Figure 2.

An MSD contains lifelines and messages. Messages have a *temperature* (**h**ot or **c**old) and an *execution kind* (**m**onitored or **e**xecuted). In Figure 2, the temperature and execution kind of messages are notated by labels (c,m), (c,e), (h,m), (h,e). The arrows of hot messages are colored red; those of cold messages are blue. Monitored messages are modeled by a dashed arrow; executed ones by a solid arrow.

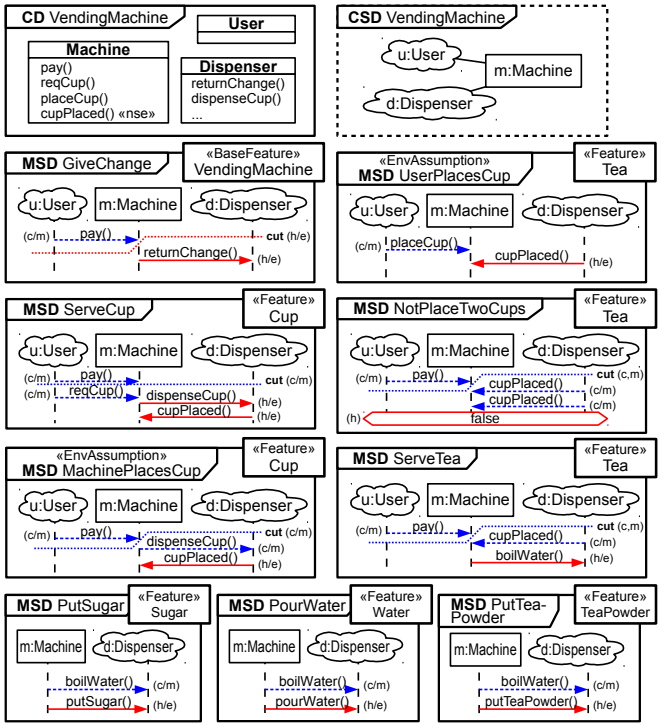


Figure 2: The MSD specification of the tea vending machine

We now detail how temperature and execution kind are used to encode safety resp. liveness properties. When an event corresponding to the first message of an MSD occurs, the MSD becomes *active*. The active MSD progresses as further events that correspond to subsequent messages occur. This progress is represented by the *cut*, which marks messages in the MSD corresponding to events that occurred. In Figure 2, a dashed horizontal line illustrates the cut of several MSDs that were activated by an occurrence of the `pay` event, sent by the user u to the vending machine m. A message in an active MSD is *enabled* when the cut is immediately in front of the message. If a hot message is enabled, the cut is also *hot*; otherwise it is *cold*. If the message is executed, the cut is also *executed*; otherwise it is *monitored*. An active MSD is *violated* when an event occurs and corresponds to a message in the MSD that is not currently enabled. If the cut is hot, this is called a *safety violation*; which must absolutely be avoided. If the cut is cold, the violation is *cold*; meaning that it is allowed to happen and results in a premature termination of the active MSD. Events corresponding to no message in the active MSD do not affect it. If the cut is executed, then it must eventually progress; otherwise there is a *liveness violation* of the MSD. If a cut is monitored, the active MSD is allowed to remain in this cut forever. An active MSD *terminates* when the cut reaches the end of the MSD, which becomes inactive again.

One MSD specification is generally composed of a set of MSDs progressing in parallel. Their current cuts altogether determine which events may or must occur next, that is, according to the enabled messages and their characteristics. This can lead to situations where events strictly required to happen in one MSD is strictly forbidden by another MSD. Harel and Marelly formalised the *play-out* semantics [21]. It specifies that the system and the environment send messages that can activate MSDs or change the current cut of active MSDs, such that (1) the environment can send arbitrary messages, (2) the system can send an infinite number of messages between two successive environment messages, and (3) if there are enabled executed message in any active MSD, the system will send the corresponding messages. These three conditions imply that a new environment message is sent only when either all active MSDs are terminated or all enabled messages are monitored.

A consistent run of the play-out semantics is a run that results in neither a safety violation nor a liveness violation of the specification. Then an MSD specification is consistent iff, given any sequence of environment events, there exists a valid play-out execution of the MSD specification – that is, the system can find a sequence of system events that leads to a consistent run. We define a system (resp. environment) strategy as a mapping from the system (resp. environment) states to the events chosen in these states. Accordingly, we define the synthesis problem as the problem of finding a system strategy that leads to a consistent run regardless of the environment strategy. We also name *controller* a strategy yielded by the synthesis process, as it actually restricts the behavior of the system so that it satisfies the specification. The synthesis problem thus requires to determine if an MSD specification is consistent and, in this case, to return a controller for the system.

3. SCENARIO-BASED SPL SPECIFICATIONS

Since SPLs commonly capture the variability between their products by using features, we propose to combine the above specification formalism with FMs.

Definition 3.1 (Scenario-Based Product-Line Spec.) *Let M be a set of MSDs and fm be an FM over a set F of features. A, SBPLS is a total function $msdf : M \to 2^{2^F}$ that associates every MSD with a Boolean formula encoding the products in which the MSD is executed. The MSD specification of a product p is the subset of M whose associated formula is satisfied by the product, that is, a set $M' \subseteq M$ such that $m \in M' \Leftrightarrow p \models msdf(m)$.*

Within our SBPLS framework, the specification of a given product comes down to a simple union of all the MSDs that correspond to this product. To synthesize a controller for each product, we can thus apply state-of-the-art synthesis

algorithms on every individual product specification. However, this product-by-product method does not take into account that two products can share commonalities in their specification, *viz.* the MSDs that are part of both specifications. As an alternative, we propose to extend the synthesis algorithms with new heuristics able to exploit commonality to reduce synthesis time. As we will see, instead of a set of controllers our extension returns only one controller annotated with variability information, such that from this controller, we can either derive a controller for a product whose MSD specification is consistent, or perform additional analyses on the featured controller. As for the other products, the synthesis algorithm will detect them as unrealizable.

A particular form of SBPLS consists in associating every MSD with a single feature [16, 15]. In this case, the specification is compositional, that is, features are specified independently from each other. A limitation of this form is that one cannot represent the combined behaviour of features, e.g. to solve feature interactions. This is why we extended the specification language with the capability to associate MSDs to formulae instead of single features.

Figure 2 is an SBPLS of a vending machine SPL. Each MSD is associated to a Boolean formula shown in the top-right of the MSD. In this example, the formulae consist of only one feature; an MSD is thus part of the specification of a given product iff the feature occurring in the associated formula is enabled in the products. This small example already illustrates the complexity of understanding the specification related to multiple products at once. The algorithms and tools we present in the next sections allow engineers to have confidence in their specification and observe its effects on every variant of the future system.

4. FEATURED GAME GRAPH

The synthesis process can be regarded as a game between the system and its environment. The actions they can execute at a given point of time are their respective enabled messages. The winning condition for the system is that it must reach infinitely often a state where no MSD is in an executed cut (henceforth called an accepting state) while avoiding safety violations (represented as a *failure state*). We formalise controller synthesis from an MSD specification via the definition of Büchi game [18].

Definition 4.1 (Büchi Game)
A Büchi Game (BG) is a tuple $bg = (Q, \Sigma, \delta_c, , \delta_u, q_0, A, B)$ *where* Q *is a set of states;* Σ *is the alphabet;* $\delta_c \subseteq Q \times \Sigma \times Q$, *is the set of controllable transitions;* $\delta_u \subseteq Q \times \Sigma \times Q$, *is the set of uncontrollable transitions, with* $\delta_c \cap \delta_u = \emptyset$ *and*

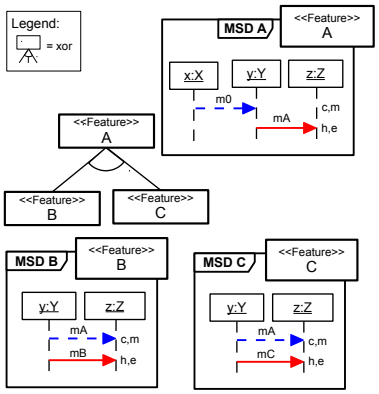
$$\forall q \bullet \exists (q, \sigma, t) \in \delta_c \Leftrightarrow \nexists (q, \sigma', t') \in \delta_u$$

that is, the transitions leaving a given state are all either controllable or uncontrollable; $q_0 \in Q$ *is the initial state;* $A \subset Q$ *is the set of accepting states;* $B \subset Q$ *is the set of failure states, with* $A \cap B = \emptyset$ *and*

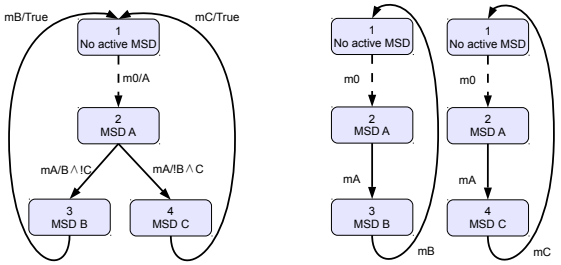
$$\forall b \in B \bullet (b, \sigma, q) \in \delta_c \cup \delta_u \Rightarrow q = b.$$

The semantics of a BG is its set of infinite executions, that is,

$$[\![bg]\!] = \{q_0, \sigma_0, q_1, \cdots \in (Q \times \Sigma)^\omega \mid \forall i \leq 0 \bullet (q_i, \sigma_i, q_{i+1}) \in \delta_c \cup \delta_u\}.$$



(a) An SBPLS.



(b) The corresponding FBG.

(c) Feature cardinalities

Figure 3: An SBPLS transformed into an FBG.

We now give an intuition of how a set of MSDs can be transformed into a BG. The state space Q is defined as the cartesian products of the set of cuts of all the MSDs plus one self-looping failure state, which represents the occurrence of a safety violation. The initial state is the state where all MSDs are inactive. The transitions between states are determined according to the enabled messages. If at least one system message is enabled, the transitions leaving the current state are all controllable and correspond to the sending of an system message. If sending a message results in a safety violation, the corresponding transition targets the failure state. Otherwise, it leads to the state where the MSDs where the message is enabled have advanced to their next cut, while the other MSDs have not moved. If only environment messages are enabled, the outgoing transitions of the current state are uncontrollable. The state reached by such a transition is determined according to the same rules as in the controllable case.

Figure 3 illustrates the FBG formalism. Figure 3a shows the specification of an SPL with two products; Figure 3b shows the corresponding FBG, and Figure 3c depicts the two equivalent projections. We see that the first transition in the FBG, due to feature A and thus common to the two products, is annotated with the formula A. After executing the transition, the FBG reaches a state with two outgoing, exclusive transitions: one labelled with $B \wedge \neg C$ and the other with $\neg B \wedge C$, each of which available to a different product.

To leverage these concepts to SPLs, we first have to extend BG with the capability to express that some MSDs must not be considered for a given product. More precisely, we define that in our new formalism, transitions are associated

to constraints over the set of features, thereby restricting the products able to execute them.

Definition 4.2 (Featured Büchi Game)
A Featured Büchi Game (BG) is a tuple $fbg = (Q, \Sigma, \delta_c, \delta_u, q_0, A, B, fm, \gamma)$ *where* Q, Σ, δ_c, δ_u, q_0, A, *and* B *are defined as in Definition 4.1,* fm *is an FM over a set of features* F, *and* $\gamma : (\delta_c \cup \delta_u) \to 2^{(2^F)}$ *is a total function that associates transitions with a Boolean formula over* F. *For a transition* t *and a product* p, $\gamma(t)$ *is the formula such that* $p \models \gamma(t)$ *if and only if* p *can execute* t.

An FBG is equivalent to a set of BGs, i.e. one per valid product. The BG corresponding to a product p is obtained by computing the so-called *projection* of the FBG onto p.

Definition 4.3 (Projection of FBG)
Let $fbg = (Q, \Sigma, \delta_c, \delta_u, q_0, A, B, fm, \gamma)$ *be an FBG, an* $p \in [\![fm]\!]$ *be a product. The projection of* fbg *onto* p *is the BG* $fbg_{|p} = (Q, \Sigma, \delta'_c, \delta'_u, q_0, A, B)$ *where*

$$\delta'_c = \{(q, \sigma, q') \in \delta_c \mid p \models \gamma(q, \sigma, q')\}$$
$$\delta'_u = \{(q, \sigma, q') \in \delta_u \mid p \models \gamma(q, \sigma, q')\}.$$

The semantics of an FBG is defined as a function that associates a valid product of fm to its respective projection.

Definition 4.4 (FBG Semantics)
Let fbg *be an FBG over an FM* fm. *The semantics of* fbg *is a total function* $[\![fbg]\!] : [\![fm]\!] \to 2^{(Q \times \Sigma)^\omega}$ *such that*

$$\forall p \in [\![fm]\!] \bullet [\![fbg]\!](p) = [\![fbg_{|p}]\!].$$

Now that the formal model is defined, we explain how to transform an SBPLS into an FBG. The states of the automaton result from the BG transformation defined above applied to the union of the sets of MSD associated to the features. The idea is that all these MSDs are part of the specification regardless of the considered product, and that features restrict the *activation* of their associated MSDs. The occurrence of a message that triggers no MSD activation results in a single transition in the FBG, which is executable by all products. When some MSDs should be activated (named *candidate* MSDs), multiple transitions are created, namely one per subset of the set of formulae associated to these MSDs. Formally, let M be the set of newly activated MSDs. Then for each $M' \subseteq M$ a transition t is created, targets the state corresponding to the activation of the MSDs in M', and such that $\gamma(t)$ is defined as

$$\gamma(t) = \bigwedge_{m' \in M'} msdf(m') \wedge \bigwedge_{m \in M \setminus M'} \neg msdf(m)$$

where $msdf$ is the SBPLS. Intuitively, $\gamma(t)$ is the conjunction of all the formulae associated to activated MSDs with the negation of all the formulae associated to a candidate MSD that has not been activated. Accordingly, a given product is able to execute only one of these transitions, i.e. the one corresponding to the set of activated MSDs that are part of this product's specification. Once the FBG is built, it acts as the input to our synthesis algorithms with the aim to derive a controller for each product.

Figure 4 shows an excerpt of the BG corresponding to the SBPLS shown in Figure 2. The excerpt starts after the occurrence of `cupPlaced` in `ServeTea`. All the other MSD are

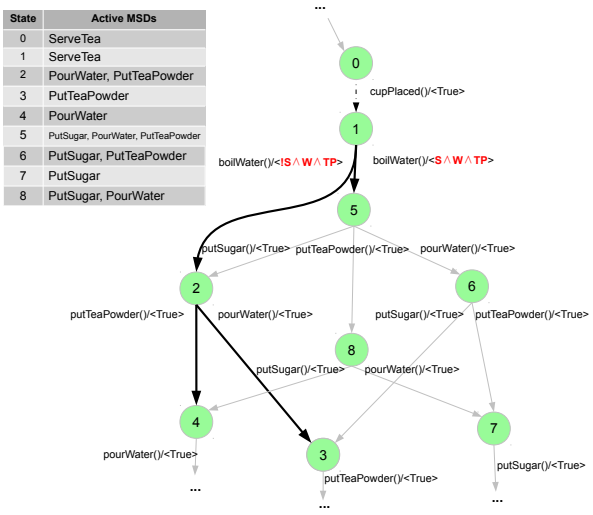


Figure 4: An excerpt of the FBG modeling the tea vending machine SPL.

inactive. Then `m` sends `boilWater` to `d` (see the transition from state 0 to state 1), which can trigger the activation of `PutSugar`, `PourSugar`, and `PutTeaPower` depending on which features are enabled. The transition from state 1 to state 2 illustrates the case where feature *Sugar* is disabled, hence the formula $\neg S \wedge W \wedge TP$ next to the transition. Accordingly, only `PourWater` and `PutTeaPowder` are active in state 2. On the contrary, the transition from state 1 to state 5 models the activation of the three MSDs, and is thus labelled with the formula $S \wedge X \wedge TP$. The other cases have not to be considered, since they correspond to products invalid according to the FM shown in Figure 1.

5. SYNTHESIS ALGORITHM

Now that we have defined FBG and shown how to construct it from an SBPLS, we focus on the synthesis algorithms. We begin by presenting the single-system synthesis algorithms that we then extend to support variability.

5.1 Single-System Synthesis

Cassez *et al.* [7] proposed an algorithm to synthesize a discrete controller from a timed reachability game, itself being an extension of [25]. As we do not consider real time, our starting point is an untimed variant of this algorithm. The objective for the system is to fulfil the so-called *winning condition*, that is, to visit an accepting state infinitely often (which is equivalent to avoiding liveness violations) while avoiding the absorbing failure state. To check this condition, the algorithm computes the set of *winning* states, i.e. the states from which the system can guarantee to reach an accepting state infinitely often. We name *goal* states any state that is both accepting and winning. Thus, an arbitrary state is winning if and only if it can reach a goal state.

Algorithm 1 shows how to compute the winning states. Initially, the set G of goal states is the set of accepting states (Line 1). A first step, encapsulated in function *ReachGoal*, consists in computing the set Win of all states from which the system can guarantee to reach a goal state (Line 2). Then at each iteration (Lines 3–5), we remove from G all goal states that cannot reach some goal state, i.e. that are

Input: $bg = (Q, \Sigma, \delta_c,, \delta_u, q_0, A, B)$.
Output: Win, the set of winning states.

1 $G \leftarrow A$;
2 $Win \leftarrow ReachGoal(bg, G)$;
3 **while** $G \neq G \cap Win$ **do**
4 | $G \leftarrow G \cap Win$;
5 | $Win \leftarrow ReachGoal(bg, G)$;
6 **end**
7 **return** Win

Algorithm 1: Synthesis(bg)

not in Win. We iterate until no more state is removed from G; in this case, from any goal state in G the system can guarantee to reach a goal state (possibly the starting goal state itself), and thus to visit a goal state infinitely often. At that point, the set Win contains only all the states that can reach a goal state. If $q_0 \in Win$ then the specification is consistent as it means that the system can satisfy the winning condition from it. Then, a controller for the system can be obtained by pruning the non-winning states from the BG.

The *ReachGoal* function differs from a standard reachability procedure in that it must distinguish between controllable and uncontrollable transitions. If the outgoing transitions are controllable, the system can select which one to execute. Therefore, a state q is winning if and only if it has at least one controllable transition leading to a winning state or a goal state, that is,

$$q \in Win \Leftrightarrow \bigvee_{(q,\sigma,q') \in \delta_c} (q' \in Win \cup G).$$

In the uncontrollable case, however, all the outgoing transitions must lead to a winning state or a goal state since the system has no control over their execution:

$$q \in Win \Leftrightarrow \bigwedge_{(q,\sigma,q') \in \delta_u} (q' \in Win \cup G).$$

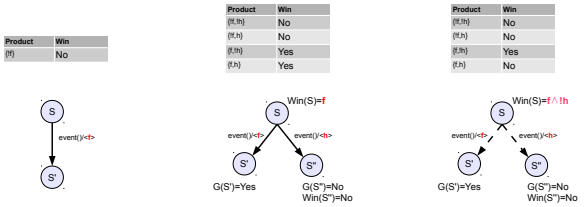
5.2 All-At-Once Synthesis

In [15] we proposed a method that enumerates all variant products from the input FM and then synthesises them one at a time. Our objective is to consider the set of all products at once. To that aim, we turn the above algorithms into *variability aware* extensions and revisit the concepts introduced in the previous section accordingly. The most important definition, which determines whether the specification is consistent, is the winning condition: a state is winning if and only if from this state the system can guarantee to reach an accepting state infinitely often. In FBG, variability impacts the executability of transitions, and thus the reachability between states. Therefore, a state can be winning for only a subset of the valid products, and the set of winning states Win is replaced by a function from Q to 2^{2^F} that associates a given state to a formula encoding the products for which the state is winning. The definition of goal states is modified similarly, since a goal state is an accepting state that is also winning. For recall, Algorithm 1 determines the set of goal states through a greatest fixed-point computation, starting with the set of accepting states and potentially removing states from the G set at each iteration. In our extension (see Algorithm **??**), we define G as a function from A to 2^{2^F} that associates to each accepting

Input: $fbg = (Q, \Sigma, \delta_c,, \delta_u, q_0, A, B, fm, \gamma)$.
Output: Win, the set of winning states.

1 **foreach** $a \in A$ **do**
2 | $G(a) \leftarrow fm$;
3 **end**
4 $Win \leftarrow FReachGoal(fbg, G)$;
5 **while** $\exists a \in A \bullet G(a) \not\Rightarrow Win(a)$ **do**
6 | **foreach** $a \in A$ **do**
7 | | $G(a) \leftarrow Win(a)$;
8 | **end**
9 | $Win \leftarrow FReachGoal(fbg, G)$;
10 **end**
11 **return** Win

Algorithm 2: F-Synthesis(fbg)



(a) No executable transition. (b) Controllable case. (c) Uncontrollable case.

Figure 5: Computation of winning products.

state the formula encoding the set of products for which this state is also winning. We compute it again via fixed-point computation, starting with $G(a) = \top$ for any $a \in A$ (that is, accepting states are goal states for all products). At each iteration, we compute the function Win according to the current function G, and updates G accordingly. The fixed-point is not reached iff after an iteration, an accepting state is a goal state for a smaller set of products.

The procedure to compute the winning function Win is formalised in Algorithm 3. In the initialisation, every state is considered winning for no product (Lines 1–3). At each iteration, we remove a transition from $Waiting$ and analyse it (Line 5). The analysis splits into two parts depending on whether the target state is reached for the first time (Lines 9–16) or not (Lines 17–28). The first part is the *exploration*. The target state is added to the $Visited$ and its outgoing transitions are added to $Waiting$ in order to pursue the exploration further. If the target state is a goal state then the source state is possibly a winning state. Hence, we re-put the transition into the $Waiting$ set in order to trigger the second part of the loop, i.e. the *re-evaluation*. Therein, we determine for which products the source state of the transition is a winning state, depending on whether its outgoing transitions are controllable or uncontrollable. Figure 5 illustrates the winning conditions in both cases. In the controllable case (Lines 18–20), state q is winning for a product p iff there exists a transition from q available to p leading to a state which is a winning state or a goal state for p, that is,

$$p \models Win(q) \Leftrightarrow \exists (q, \sigma, q') \in \delta_c \bullet \\ p \models (\gamma(q, \sigma, q') \wedge (Win(q') \vee G(q'))).$$

If we apply the above formula to the whole set of products,

we obtain that the set of products for which q is winning via q' is encoded by the conjunction of γ with the disjunction of $Win(q')$ and $G(q')$. We thus have

$$Win(q) \Leftrightarrow \bigvee_{(q,\sigma'',q'')\in\delta_c} \big(\gamma(q,\sigma'',q'') \wedge (Win(q'') \vee G(q''))\big).$$

The uncontrollable case (Lines 21–23) is more subtle. A state q is winning for product q iff all the following conditions hold:

1. At least one transition from q is available to p.
2. All the transitions from q *available to* p lead to a state which is a winning state or a goal state for p.

Accordingly, the products for which p is winning are captured by the formula

$$\Big(\bigvee_{(q,\sigma'',q'')\in\delta_u} \gamma(q,\sigma'',q'')\Big) \wedge \bigwedge_{(q,\sigma'',q'')\in\delta_u} \big(\gamma(q,\sigma'',q'') \Rightarrow Win(q'') \vee G(q'')\big).$$

In the second half of the loop of Algorithm 3, we apply the above two formulae to compute the set of products for which a state is winning (Lines 18–23). If this set has changed with respect to prior computations, the algorithm re-evaluates the winning condition for the predecessors of q that it visited earlier (Lines 24–27).

Input: $bg = (Q, \Sigma, \delta_c, \delta_u, q_0, A, B)$, $G \subseteq A$.
Output: Win, the states that can reach a state in G.

```
1  foreach q ∈ Q do
2  |  Win(q) = ⊥;
3  end
4  Visited ← {q0};
5  Waiting ← {(q0, σ, q) ∈ δc ∪ δu};
6  Depend(q0) ← ∅;
7  while Waiting ≠ ∅ do
8  |  Take t = (q, σ, q') ∈ Waiting;
9  |  if q' ∉ Visited then
10 |  |  Visited ← Visited ∪ {q'};
11 |  |  Depend(q') ← {t};
12 |  |  Waiting ← Waiting ∪ {(q', σ, q'') ∈ δc ∪ δu};
13 |  |  if q' ∈ dom(G) then
14 |  |  |  Waiting ← Waiting ∪ {t};
15 |  |  end
16 |  end
17 |  else
18 |  |  if t ∈ δc then
19 |  |  |  Win* ←
   |  |  |  ⋁(q,σ'',q'')∈δc (γ(q, σ'', q'') ∧ (Win(q'') ∨ G(q'')));
20 |  |  end
21 |  |  else
22 |  |  |  Win* ← (⋁(q,σ'',q'')∈δu γ(q, σ'', q'')) ∧
   |  |  |  ⋀(q,σ'',q'')∈δu (γ(q, σ'', q'') ⇒ Win(q'') ∨ G(q''));
23 |  |  end
24 |  |  if Win* ⇏ Win(q) then
25 |  |  |  Win(q) ← Win*;
26 |  |  |  Waiting ← Waiting ∪ Depend(q);
27 |  |  end
28 |  end
29 end
30 return Win
```

Algorithm 3: ReachGoal(bg, G)

After applying the algorithm to compute the winning function, two cases may happen. If the initial state q_0 is not winning for at least one valid product , then the SBPLS is inconsistent for all products and no controller can be synthesized. Otherwise, we can extract from the FBG a controller for any product p satisfying $Win(q_0)$ by (1) removing every state q in the FBG such that p does not satisfy $Win(q)$ as well as its incoming and outgoing transitions; and (2) computing the projection of the resulting FBG.

Instead of a controller for an individual product, we can keep a concise representation for the controllers of all products. To that aim, we have to make each state inaccessible to the products for which the state is not winning. This is achieved by replacing the formula labelling any incoming transition of each state q by its conjunction with $Win(q)$. In this case, the products have access only to the states that are winning for them. Also, products for which the specification is inconsistent should not be produced; one can thus add a constraint in the FM that makes these products invalid. This variability-aware controller can be regarded as a featured transition system, i.e. an extension of transition systems that model the behaviour of SPL products [8]. It is therefore possible to further analyse the controller using dedicated model-checking algorithms [8], thereby determining the relevant properties of each product for which the specification is consistent.

6. IMPLEMENTATION AND EVALUATION

We implemented our all-at-once synthesis algorithm as part of our Eclipse-based tool suite ScenarioTools.[1] ScenarioTools supports the modeling of SBPLS through the use of UML profiles. It also supports the synthesis of SBPLS. First, the tool performs an initial exploration of the model to create the states and transitions of the corresponding FBG, including states modeling violations. It then applies the algorithms presented above to determine whether the specification of some variants is not realizable, and extract a controller for the other products, if any. Our project relies on the JDD [2] Java library to encode formulae over the features into a Binary Decision Diagrams (BDD) [1, 6].

We used our tool to verify the consistency of all products of the tea vending machine SPL, and they were all realizable. We also carried out experiments to assess the efficiency of the all-at-once synthesis with respect to (1) the successive synthesis of each product separately, and (2) our previous incremental on-the-fly synthesis algorithm [15]. On-the-fly means that the algorithm may only visit a subset of the game graph to find a system strategy, which can lead to drastic increases in performance [7]. In addition to that, our previous algorithm further facilitates the synthesis by attempting to replicate, during the synthesis of a new product, the strategy used to synthesize the previous product.

All benchmarks were run on a Windows PC with a 2,4 GHz Intel Core 2 Duo processor and 4 GB of RAM. We repeated each experiment 20 times and computed the average of the synthesis times. For our experiments, we use a technical example, which we name the *cascading example*. Its structure is presented in Fig. 6. Each feature in the FM has two child features, connected with an OR-relationship. Each feature is associated to one MSD named after the feature. The first message of an MSD is a cold, monitored message and is followed by one hot, executed message. All messages are controllable by the system except the first message of the

[1] `http://scenariotools.org`
[2] `http://javaddlib.sourceforge.net/jdd/`

MSD of the root feature, which is an environment message. The first message of the MSD of another feature corresponds to the hot message of the MSD of its parent. This way, one activation of an MSD further triggers the activations of MSD of its child features, hence the name *cascading*.

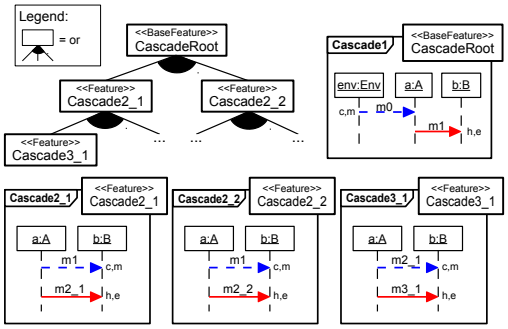
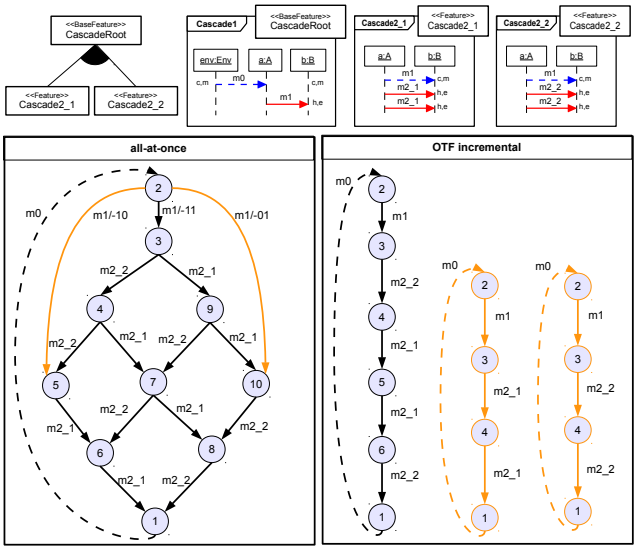


Figure 6: The evaluation example

We derive multiple examples following the previous scheme, thereby creating SBPLS for 3, 5, 7, 9, 11, 13, and 15 features. The valid combinations of those features respectively lead to 3, 7, 15, 31, 63, 127, and 255 products, which represents an exponential increase in both the number of products and the number of FBG states. We also create two other variants of this technical example. The first follows an identical pattern, except that all decomposition types are XOR. This yields SBPLS with a significantly lower number of products (i.e. 2, 3, 4, 5, 6, 7, 8 variants, respectively). The last variant keeps the OR decomposition type, but we intentionally increase the number of alternative paths that an on-the-fly algorithm could avoid. For this purpose, we duplicate the hot message in the MSD of every child feature of the root (e.g. *Cascade*2_1). Thus, after the first hot message is sent, the algorithm can non-deterministically choose between sending the second message or the first message of the newly activated MSDs. This modification drastically increases the number of alternative runs, due to the high number of ways enabled events can interleave.

Table 1 shows how synthesis time increases with respect to the number of features for every example. It provides the parent-child relationship and the number of hot messages in each MSD (e.g. `OR(1)` stands for OR decomposition and one hot message), the number of features involved, the number of valid products, and for each approach the average number of explored states, the average synthesis times, and finally the speedups provided by our all-at-once (**A-a-o)** algorithm wrt. the product-by-product synthesis (**P-b-p**) and our previous on-the-fly, incremental method (**OTF-Inc**).

Let us first compare our all-at-once algorithm wrt. the product-by-product method. In the XOR cases, we see that the number of states explored and the average synthesis time using our **A-a-o** are about halved wrt. **P-b-p**. When dealing with OR decompositions, and thus with a steeper increase in the number of products, we observe even more improvements: when synthesizing 255 products, **A-a-o** visits only 573 states in the `OR(1)` case and 70222 states in the `OR(2)` case, whereas **P-b-p** visits 22196 and 169475 states, respectively. This leads to a speedup of 19.97 and 29.21, respectively. More generally, the relative performance of **A-a-o** tends to rise as the number of products increases.



	all-at-once	OTF incremental
States visited for one variant with 3 features	10	6
States visited for the whole PL	10	14

Figure 7: **A-a-o** and **OTF-Inc** applied to the OR (2) case with three features.

We then compare our **A-a-o** algorithm with our previous **OTF-Inc** synthesis. In the `XOR(1)` case the two algorithms perform quite closely. We observe that the all-at-once algorithm still explores less states, but the difference in synthesis time is not substantial. The reason is that **A-a-o** has to deal with an additional overhead, i.e. the management of BDDs use to encode the feature formulae encountered in the FBG. In this case, the overhead is not offset by the heuristics offered by **A-a-o**. Benefits are, however, observed in the `OR(1)` case. In the largest case of 15 features (255 products), **A-a-o** visits only 573 states (against 2718 for **OTF-Inc.**), which allows it to reach a speedup of 1.8. As against **P-p-b**, this speedup also tends to increase with the number of features and products, although not as remarkable as the speedup related to **P-p-b**. Still, this is a noticeable result as it shows that even without being on the fly, our algorithm can outperform an optimised algorithm such as **OTF-Inc**. We notice that for 3, 5, 7, and 9 features in the `OR(2)` case, the **A-a-o** synthesis also still outperforms **OTF-Inc**. However, this is not true for the other cases. As an attempt to explain this phenomenon, we analyzed in depth the behavior of each algorithm when applied to this example.

Figure 7 illustrates the behavior of both algorithms on the `OR(2)` case with three features. Black states and transitions represent the (part of the F)BG related to the product with all features. After `m1` is sent, our **A-a-o** algorithm explores all the six alternative paths leading to state 1 whereas **OTF-Inc** can avoid visiting all of them, and even potentially visit only one. The additional parts that are explored to synthesize the other products are depicted in orange. In the FBG, only two transitions are added to the graph. Conversely, even if it explores only parts of the state space for each variant, **OTF-Inc** needs to explore one more graph for each other variant of the SPL. In the end, it explores more states than **A-a-o**, which is why the latter is faster.

However, when the number of avoidable paths grows, as in the 15-feature case, **OTF-Inc** starts performing better, even

Table 1: Synthesis times for the cascading example with OR and XOR decomposition.

Experiment	#Features	#Products	#States			Time (ms)			Speedup	
			A-a-o	P-b-p	OTF-Inc	A-a-o	P-b-p	OTF-Inc	P-b-p	OTF-Inc
XOR (1)	3	2	4	6	6	12	12	12	1	1
XOR (1)	5	3	6	11	11	21	34	14	1.62	0.67
XOR (1)	7	4	8	16	16	29	41	20	1.41	0.69
XOR (1)	9	5	10	22	22	31	57	25	1.84	0.8
XOR (1)	11	6	12	27	27	32	77	29	2.41	0.9
XOR (1)	13	7	14	33	33	38	81	39	2.13	1.02
XOR (1)	15	8	16	39	39	45	82	46	1.82	1.02
OR (1)	3	3	5	11	10	12	15	11	1.25	0.92
OR (1)	5	7	11	42	32	17	45	15	2.65	0.88
OR (1)	7	15	26	158	86	79	219	96	2.77	1.21
OR (1)	9	31	56	546	218	138	451	197	3.27	1.43
OR (1)	11	63	111	1718	506	219	1459	341	4.88	1.56
OR (1)	13	127	243	6060	1190	508	5862	834	11.54	1.64
OR (1)	15	255	573	22196	2718	1260	25161	2267	19.97	1.8
OR (2)	3	3	10	18	14	43	98	108	2,27	2.51
OR (2)	5	7	29	92	50	197	554	250	2,81	1.27
OR (2)	7	15	83	429	142	383	866	450	2,26	1.18
OR (2)	9	31	245	1921	374	712	2948	805	4,14	1.13
OR (2)	11	63	704	7685	886	1309	7891	1087	6,03	0.83
OR (2)	13	127	2108	35407	2126	7388	47877	5237	6,48	0.7
OR (2)	15	255	7022	169475	4926	23255	679356	13364	29,21	0.57

in the number of explored states. As the number of features grows, the probability to have different but equivalent paths in the game graph also increases. When the number of those alternative paths is high, **OTF-Inc** avoids visiting a large part of the graph, which is more efficient than exploiting the commonalities between the variants. Even if this seems a mitigated result for our new algorithm, this opens interesting perspectives, as we could design an on-the-fly version of our all-at-once algorithm, which would benefit from the two heuristics. This is far from a straightforward task, though, as both the encoding of variability and the on-the-fly exploration brings substantial modifications to the synthesis algorithm. We therefore left that for future work.

7. RELATED WORK

There are many approaches for consistency-checking and synthesizing controllers from scenarios [30, 17, 4, 18, 14, 26]. However, these consider only single systems.

The relationship between FMs and UML models was studied, e.g., in [23, 29, 28]. Ziadi *et al.* synthesize statecharts from sequence diagrams where interaction fragments can be annotated to be active only in certain variants [31]. However, they do not consider that inconsistencies may arise between the scenarios or the features. Ghezzi and Molzam propose a similar formalism, and then verify non-functional requirements using probabilistic model checking [13]. However, they do not consider concurrent scenarios. Shaker *et al.* proposed to model SPL behavior with a combination of FMs and a feature-aware extension of statecharts [28], where features are introduced as a new parallel region, and may change the priorities or the triggering conditions of transitions. In the past, we defined featured transition systems as a variability-aware extension of transition systems [9, 8]. They rely on similar mechanisms we used to encode variability as Boolean formulae in FBRG. We also designed verification algorithms to check properties expressed in temporal logics. None of these methods is equipped with consistency checking or synthesis mechanisms. In particular, in [9, 8] we assumed that the transition system was given a priori; now, we can produce it from a scenario-based specification whose consistency has been previously checked.

Apel *et al.* [2] extended Alloy with collaboration-based design and feature composition. The Alloy Analyzer can then detect structural and semantic dependences between the features. However, Alloy cannot express complex behavioral interactions and detect inconsistencies in them. SPLVerifier [3] is a tool for verifying inconsistencies and harmful feature interactions in C or Java code. Safety properties are inserted into the code in the form of assertions; liveness properties are, however, not supported.

Harhurin and Hartman propose an approach for modeling and consistency-checking families of service-oriented systems [22]. They model possible service compositions and formally specify constraints on the input and output sequences of the ports of a service. Then, combinations of input/output ports that are incompatible in a certain product can be detected by using a theorem prover. In comparison, our approach allows the requirements engineer not only to consider the input/output behavior of a single service, but also the interactions between multiple components.

Most recently, Cohen and Maoz [10] modeled the variability of the different live sequence charts semantics, and built a tool that can perform analyses on these sequence charts whose results change according to the selected semantics.

8. CONCLUSION AND OUTLOOK

We presented a novel synthesis algorithm that is able to produce a controller for every consistent variant of an SPL, exploiting the similarities between them. In the end, one can obtain a specific controller for each consistent product of the SPL, or a *featured* controller that concisely specifies the consistent behaviors of all products. In the latter case, we can rely on efficient SPL model-checking techniques such as [8] to investigate the satisfaction of complex temporal properties by all products. This combination of synthesis and model checking offers interesting application perspectives, as they provide a complete approach ranging from an intuitive modelling language to the verification of critical behavioral requirements. They would require, however, a proper integration of ScenarioTools and an SPL model checker such as ProVeLines [11].

Another important work is to continue improving the practical performance of our algorithm. We will study how to combine it with heuristics to explore the FBRG on the fly to avoid redundant paths in the graph. While this will require a lot of work, we believe combining both approaches can yield drastic reductions in synthesis time. Moreover, we will study the applicability of SPL-specific optimisations designed for model checking [8] to our synthesis methods.

9. REFERENCES

[1] S. B. Akers. Binary decision diagrams. *Computers, IEEE Transactions on*, 100(6):509–516, 1978.

[2] S. Apel, W. Scholz, C. Lengauer, and C. Kastner. Detecting dependences and interactions in feature-oriented design. In *ISSRE '10*, pages 161–170. IEEE Computer Society, 2010.

[3] S. Apel, H. Speidel, P. Wendler, A. von Rhein, and D. Beyer. Feature-interaction detection using feature-aware verification. In *ASE'11*. IEEE, 2011.

[4] Y. Bontemps and P. Heymans. From live sequence charts to state machines and back: A guided tour. *TSE*, 31(12):999–1014, 2005.

[5] C. Brenner, J. Greenyer, and V. Panzica La Manna. The ScenarioTools play-out of modal sequence diagram specifications with environment assumptions. In *GT-VMT'13*, volume 58. EASST, 2013.

[6] R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys (CSUR)*, 24(3):293–318, 1992.

[7] F. Cassez, A. David, E. Fleury, K. G. Larsen, and D. Lime. Efficient on-the-fly algorithms for the analysis of timed games. In M. Abadi and L. de Alfaro, editors, *CONCUR'05*, volume 3653 of *LNCS*, pages 66–80. Springer, August 2005.

[8] A. Classen, M. Cordy, P.-Y. Schobbens, P. Heymans, A. Legay, and J.-F. cois Raskin. Featured transition systems: Foundations for verifying variability-intensive systems and their application to LTL model checking. *TSE*, pages 1069–1089, 2013.

[9] A. Classen, P. Heymans, P.-Y. Schobbens, and A. Legay. Symbolic model checking of software product lines. In *ICSE'11*, pages 321–330. ACM, 2011.

[10] B. Cohen and S. Maoz. Semantically configurable analysis of scenario-based specifications. In *Fundamental Approaches to Software Engineering*, volume 8411 of *Lecture Notes in Computer Science*, pages 185–199. Springer Berlin Heidelberg, 2014.

[11] M. Cordy, P.-Y. Schobbens, P. Heymans, and A. Legay. Provelines: A product-line of verifiers for software product lines. In *SPLC'13*. ACM, 2013.

[12] W. Damm and D. Harel. LSCs: Breathing life into message sequence charts. In *Formal Methods in System Design*, volume 19, pages 45–80. Kluwer Academic, 2001.

[13] C. Ghezzi and A. M. Sharifloo. Model-based verification of quantitative non-functional properties for software product lines. *Information and Software Technology*, 2012.

[14] J. Greenyer. *Scenario-based Design of Mechatronic Systems*. PhD thesis, University of Paderborn, October 2011.

[15] J. Greenyer, C. Brenner, M. Cordy, P. Heymans, and E. Gressi. Incrementally synthesizing controllers from scenario-based product line specifications. In *FSE'13*, ESEC/FSE 2013. ACM, 2013.

[16] J. Greenyer, A. M. Sharifloo, M. Cordy, and P. Heymans. Efficient consistency checking of scenario-based product line specifications. In *RE'12*, pages 161–170. IEEE, 2012.

[17] D. Harel and H. Kugler. Synthesizing state-based object systems from LSC specifications. In *Foundations of Computer Science*, volume 13:1, pages 5–51, 2002.

[18] D. Harel, H. Kugler, and A. Pnueli. Synthesis revisited: Generating statechart models from scenario-based requirements. In *Formal Methods in Software and Systems Modeling*, volume 3393 of *LNCS*, pages 309–324. Springer, 2005.

[19] D. Harel and S. Maoz. Assert and negate revisited: Modal semantics for UML sequence diagrams. *Software and Systems Modeling (SoSyM)*, 7(2):237–252, May 2008.

[20] D. Harel and R. Marelly. *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer, August 2003.

[21] D. Harel and R. Marelly. *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer, August 2003.

[22] A. Harhurin and J. Hartmann. Towards consistent specifications of product families. In *FM 2008: Formal Methods*, volume 5014 of *LNCS*, pages 390–405. Springer, 2008.

[23] P. Jayaraman, J. Whittle, A. Elkhodary, and H. Gomaa. Model composition in product lines and feature interaction detection using critical pair analysis. In *Model Driven Engineering Languages and Systems*, volume 4735 of *LNCS*, pages 151–165. Springer, 2007.

[24] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, 1990.

[25] X. Liu and S. Smolka. Simple linear-time algorithms for minimal fixed points. In K. Larsen, S. Skyum, and G. Winskel, editors, *Automata, Languages and Programming*, volume 1443 of *Lecture Notes in Computer Science*, pages 53–66. Springer Berlin Heidelberg, 1998.

[26] S. Maoz and Y. Sa'ar. Assume-guarantee scenarios: Semantics and synthesis. In R. B. France, J. Kazmeier, R. Breu, and C. Atkinson, editors, *Model Driven Engineering Languages and Systems*, volume 7590 of *LNCS*, pages 335–351. Springer Berlin Heidelberg, 2012.

[27] P.-Y. Schobbens, P. Heymans, J.-C. Trigaux, and Y. Bontemps. Feature Diagrams: A Survey and A Formal Semantics. In *RE'06*, pages 139–148, 2006.

[28] P. Shaker, J. M. Atlee, and S. Wang. A feature-oriented requirements modelling language. In *RE'12*, pages 151–160, 2012.

[29] M. Vierhauser, P. Grünbacher, A. Egyed, and W. Rabiser, R.and Heider. Flexible and scalable consistency checking on product line variability models. In *Proc. of ASE'10*, pages 63–72. ACM, 2010.

[30] J. Whittle and J. Schumann. Generating statechart designs from scenarios. In *ICSE '00*, 2000.

[31] T. Ziadi, L. HÃľlouÃńt, and J.-M. JÃľzÃľquel. Behaviors generation from product lines requirements. In *Proc. of UML'04 Workshop on Software Architecture Description*, 2004.