Incrementally Synthesizing Controllers from Scenario-Based Product Line Specifications

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

Christian Brenner
Software Engineering Group, Heinz Nixdorf Institute
University of Paderborn, Germany
cbr@uni-paderborn.de

Maxime Cordy*
PReCISE Research Center
University of Namur, Belgium
mcr@info.fundp.ac.be

Patrick Heymans
PReCISE Research Center
University of Namur, Belgium
phe@info.fundp.ac.be

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

ABSTRACT

Many software-intensive systems consist of components that interact to fulfill complex functionality. Moreover, often many variants of such systems have to be designed at once. This adds complexity to the design task. Recently, we proposed a scenario-based approach to design product lines, which combines feature diagrams and Modal Sequence Diagrams. We proposed a consistency-checking technique based on a dedicated product line model checker. One limitation of this technique is that it is incomplete, i.e., it may fail to show the consistency of some consistent specifications. In this paper we propose a new game-based approach that overcomes this incompleteness and, in addition, automatically synthesizes controllers for the consistent product specifications. We exploit the fact that many variants are similar and efficiently synthesize product controllers incrementally. We provide a prototype tool and evaluate the efficiency of the approach.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specifications; D.2.4 [**Software Engineering**]: Software/Program Verification

General Terms

Synthesis, Software product lines

Keywords

Message sequence diagrams, Controller synthesis, Features

*FNRS Research Fellow

1. INTRODUCTION

Nowadays, software is part of a wide variety of safety-critical systems, including cars, planes, and medical devices, which typically consist of many components that provide functionality through their interaction. These interactions must often satisfy complex specifications, and the failure to comply may have tragic consequences.

Moreover, often today engineers do not only develop a single system, but rather build several *variants* (also called *products*) of the same system—a *Product Line* (PL). The differences between the variants (*variability*) are commonly organized in terms of *features*, which are functions or components that may or may not be present in a given variant. As more features are added to the PL, the number of possible variants grows exponentially, in the worst case.

Managing the variability and ensuring that each product will correctly satisfy its specification is a major challenge in PL engineering. In the first place this requires a precise specification of each product. This implies that (a) the requirements of each feature are precisely specified, and that (b) there are mechanisms to compose the specification of a product from the specifications of its constituent features.

Recently [14], we proposed to specify PLs using feature diagrams (FDs) [21] to model variability, and Modal Sequence Diagrams (MSDs) [17] to specify the behavior of features.

The use of MSDs has several advantages. First, it is an intuitive, yet precise visual formalism that allows engineers to specify sequences of interactions that may, must, or must not happen in a system. This also fits most development processes that propose a scenario-based approach during the early design. Second, with MSDs engineers can describe behavioral aspects of features that can extend or restrict the behavior specified for other features. Third, MSD specifications for a product can be naturally composed by forming the union of the MSDs specifying its constituent features.

A specification of allowed and forbidden behavior, however, can be *unrealizable*, i.e., it may be impossible to build a product that is able to react to all possible sequences of environment events in such a way that its specification is always satisfied. For the development of a PL this can require costly iterations or in the end many desired products may be unrealizable or intrinsically flawed.

In our recent work [14], we therefore developed a technique for checking the realizability of MSD PL specifications. By using a dedicated PL model-checker [9], we can detect which feature combinations are unrealizable. One limitation of this technique is that, although it is capable to detect inconsistencies, it is unable to derive an implementation of a consistent specification. Moreover, our previous technique is incomplete: it may report some realizable product specifications to be unrealizable.

In this paper, we overcome the above limitations and extend our scenario-based PL specification approach with a novel consistency-checking and controller synthesis technique. In particular, this technique introduces the following three key innovations:

(1) The technique is based on an *on-the-fly game-solving algorithm* for *synthesizing controllers* from the MSD specifications of products in a PL. This has three advantages. First and foremost, instead of the yes/no answer generated by a model-checker, the synthesis algorithm will produce for each product a controller that shows which sequences of actions the system can do to implement the specification. Second, the algorithm is complete as it never returns false negatives. Third, being on-the-fly, the game-solving algorithm is more efficient than the model-checking algorithm. It typically only explores parts of the state space described by an MSD specification.

(2) We apply the synthesis *incrementally* and *specifically optimized for PLs described by FDs*. Typically many products in a PL are very similar, i.e., they share many common features and consequently share many common MSDs. Therefore, if a controller for one product could be successfully synthesized, we can more efficiently, based on this controller, synthesize a controller for a similar product. We propose a strategy to systematically derive similar products from FDs to exploit the benefit of the incremental synthesis.

(3) We support MSD specifications that not only specify *requirements* of how the system components must react to environment events, but also specify *environment assumptions*. That way engineers can specify what can or cannot happen in the environment, or how the environment again reacts to actions of the system. This is essential, *e.g.*, in mechatronic systems. As a running example we will consider a production robot for which it is important to assume that a robot arm, if ordered to move to a press, will eventually arrive at the press. Without these assumptions it would be impossible for the system to fulfill its requirements.

We prototypically implemented this technique in ScenarioTools, our new Eclipse/UML-based modeling, simulation, and synthesis tool suite for MSD specifications.

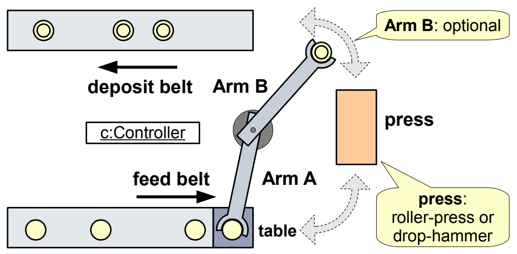
This paper is structured as follows. We explain the foundations and introduce our running example in Sect. 2. The incremental synthesis algorithm is described in Sect. 3. In Sect. 4, we introduce the systematic strategy to derive products from FDs. We overview our tool implementation and show evaluation results in Sect. 5, discuss related work in Sect. 6, and finally conclude in Sect. 7.

2. FOUNDATIONS

In this section we introduce the foundations of this paper, FDs and MSDs. As a running example, we consider a simple PL specification of a production cell, inspired by a case study from Lewerentz *et al.* [22].

2.1 Example

Figure 1 shows an illustration of the production cell. It consists of a production robot with two arms. One arm, *Arm A*, picks up metal blanks that arrive from a *feed belt* on a *table* and places them into a *press*. The press presses the blanks into plates. The other arm, *Arm B*, picks up the pressed plates and places them on a *deposit belt*, where they are transported off again. The production cell is controlled by a central controller. The requirements for the controller and assumptions about the physical behavior of the system are described informally below the sketch (R1-4, A1-6).



Arm A:

R1) When a blank plate arrives at the table, Arm A must pick it up, move it to the press, and release it into the press. The arm then has to move back to the table, where it must arrive before the next blank arrives.

A1) If Arm A is ordered to move to the press, it will eventually arrive at the press

A2) If Arm A is ordered to move to the table, it will eventually arrive at the table

A3) If Arm A picks up a blank, moves to the press, and returns to the table, the next blank will not arrive before Arm A has returned to the table.

Press (roller-press or drop-hammer press)**:**

R2) When Arm A releases the blank into the press, the press must press. The next blank must not be released before the pressing finished.

A4) The press finishes pressing before Arm A returns to the table.

Press (drop-hammer press)**:**

R3) After Arm A arrives at the press and before Arm A left the critical area, the press must not have started pressing.

A5) If Arm A is ordered to move to the table, it will leave the critical area before arriving at the table

A6) If Arm A is ordered to move to the table before the press is ordered to press, Arm A will leave the critical area before the press will have started pressing.

Arm B:

R4) After the press has finished pressing, Arm B must pick up the pressed plate and move to the deposit belt... [further specification omitted for brevity]

Figure 1: Sketch and informal requirements for the production cell PL

We consider the following variability: (1) Arm B is optional; pressed plates can also be picked up by workers. (2) The press can be of two different kinds. Either the press is a roller press, where the plates are pressed between two cylinders, or the press is a drop-hammer press where plates are formed by a falling hammer. In the latter case, Arm A may be damaged if the pressing starts before the arm has left a critical area.

2.2 Feature Diagrams

The variability of the production cell example can be described by the FD in Fig. 2.

A *feature* commonly designates a unit of difference between products that appears natural to stakeholders or engineers. For example, a feature may model an optional component, or aspects that crosscut many components. There often exist dependencies between features: some are exclusive alternatives, others must co-exist, etc. FDs are commonly used as a means of representing these interdependencies [5].

FDs are tree-like structures that organize features in a hierarchy. Nodes represent features, and edges model parent-

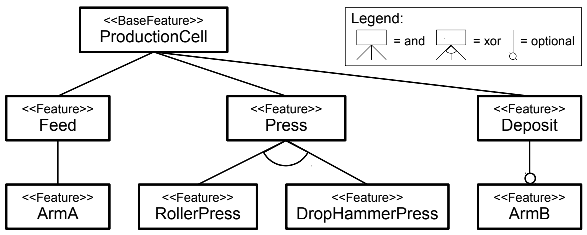


Figure 2: The feature diagram of the production cell

child relationships that specify how a feature (the "parent") is decomposed into sub-features (the "children"). There usually exist three decomposition types: AND, XOR, and OR. An AND decomposition means that if the parent feature is enabled in some product, then all of its children must be too. The only exception to this rule is the optional child-feature, which may or may not be present when its parent is enabled. A XOR (resp. OR) decomposition implies that exactly (resp. at least) one child feature must be enabled when the parent is. Whatever the decomposition type, a child feature cannot be enabled if its parent is not. In addition to the hierarchy, one may specify cross-tree constraints like *feature 'a' requires feature 'b'* and *feature 'c' excludes feature 'd'*. More generally, one may define any arbitrary Boolean constraint over the set of features, but we do not consider this in this paper. As Schobbens *et al.* [25], we define the semantics of an FD as the set of the valid products, *i.e.* the combinations of features that satisfy all the constraints imposed by the FD.

2.3 MSD Specifications of PLs

MSDs are a formal interpretation of UML sequence diagrams proposed by Harel and Maoz [17]. An MSD specification describes the valid interaction behavior of components, that, more generally, we call *objects*. The set of interacting objects is called the *object system*. We consider *open* systems where the objects are either *environment objects* or *system objects*. The set of environment objects is called the *environment*; the set of system objects is called the *system*.

We consider MSD specifications that not only formulate requirements on the system, but also formulate assumptions on how the environment behaves [13]. The requirements and assumptions are two sets of MSDs that, together with a definition of the object system, form an *MSD specification*.

According to our PL specification scheme [14], MSDs can be associated to features. Figure 3 for example shows one assumption and one requirement MSD associated to feature ArmA of the production cell. The requirement and assumption MSDs of a specification for a product are obtained by the union of the requirement resp. assumption MSDs of the features comprising the product. For simplicity, we assume that there is one definition of the object system for all features and products. If an object is not part of a particular feature or product, for example Arm A is optional in the production cell, there is simply no interaction with this object. This is similar to the notion of a global world model in [26].

MSDs can be *existential* or *universal*. Existential diagrams specify sequences of interactions that must be possible. Universal MSDs describe properties that must be satisfied by every sequence of interactions. We only consider universal MSDs in this paper. Existential MSDs are currently not supported by our synthesis algorithm; an extension is planned as future work.

2.3.1 Objects, Lifelines, and Messages

Each lifeline in an MSD represents an object in the object system. The objects can exchange messages; a *message* has a name and sending and receiving objects. For brevity, we only consider *synchronous* messages, where the sending and receiving of a message are a single *event*, also called *message event*. Our approach can also be applied if messages are asynchronous.

In our specification approach, we model the object system by a UML composite structure diagram (CSD). The top-right of Fig. 3 shows the CSD for the production cell. The objects are represented by *roles* in the CSD. System object roles have a rectangular shape; the roles of environment objects have a cloud-like shape. The connectors illustrate which objects exchange messages. The roles are typed by classes (shown on the top left of Fig. 3) where operations define which messages an instance can receive.

The messages in a universal MSD have a *temperature* and an *execution kind*. The temperature can be either *hot* or *cold*. The execution kind can be either *executed* or *monitored*. Figure 3 shows two MSDs. The temperature and execution kind is annotated by labels (c,m), (c,e), (h,m), (h,e) next to the messages. In addition, the arrows of hot messages are colored red; the arrows of cold messages are colored blue. Monitored messages have a dashed arrow; the arrows of executed messages have a solid line.

The temperature and execution kind of messages encode safety resp. liveness properties (see bottom of Fig. 3). Intuitively, a monitored message may happen, whereas an executed message must eventually happen. A hot message must not be "violated", but a cold message may be "violated"; we explain this concept in more detail shortly.

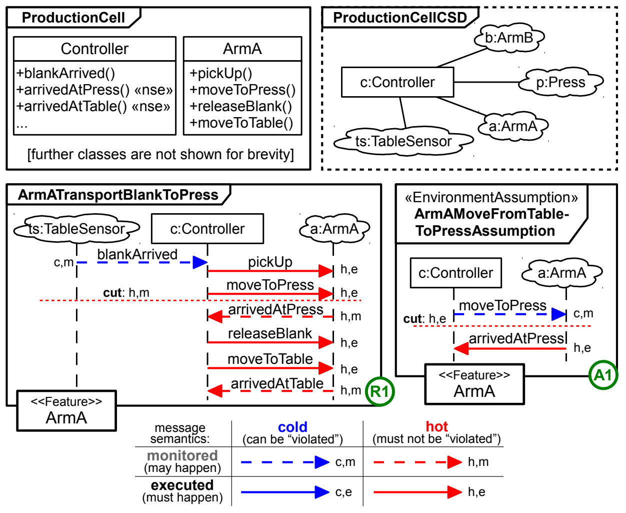


Figure 3: Two MSDs from the specification of feature Arm A

Let us consider as an example the MSD ArmATransportBlankToPress in Fig. 3. It models the requirement R1 described informally in Fig. 1. The first message says that a blank may arrive on the table (`blankArrived`). The second message says that, after the blank arrived, Arm A must pick up the blank (`pickUp`). This message is *executed*, which means that this must eventually happen. Also the message is *hot*, which means that before this happens no other message described in the MSD must be sent, e.g., it is forbidden for another blank to arrive. Similarly, the next message says that next the Arm A must move to the press (`moveToPress`). The fourth message models that next Arm A arrives at the press (`arrivedAtPress`). The message is *hot* and *monitored*. Being monitored, the message says that it is fine if this event never occurs. Being hot, however, the message says that before the event occurs, no other event in the diagram may occur. In short, the rest of the MSD specifies that after Arm A arrives at the press, it must release the blank, move back to the table where it may eventually arrive.

More specifically, the message semantics is the following. A message event can be *unified* with a message in the MSD if the event name equals the message name and the sending and receiving objects of the message event are represented by the sending and receiving lifelines of the diagram message. When an event occurs that can be unified with the first message of an MSD, an *active copy* of that MSD is created, also called *active MSD*. (We assume that an MSD has only one first message.) As further events occur that can be unified with the subsequent messages in the MSD, the active MSD progresses. This progress is captured by the *cut* which marks for every lifeline the sent/received messages. If the cut reaches the end of an active MSD, it terminates.

If the cut is in front of a message on its sending and receiving lifelines this message is *enabled*. If a hot message is enabled, the cut is *hot*, otherwise the cut is *cold*. Similarly, if an executed message is enabled, the cut is *executed*, otherwise the cut is *monitored*. The dashed horizontal line in the MSD ArmATransportBlankToPress, see Fig. 3, represents the cut after the occurrence of the message events `blank-Arrived`, `pickUp`, and `moveToPress`. Now the hot, monitored message `arrivedAtPress` is enabled. Since no other message is enabled, the cut is also monitored and executed. There can be multiple MSDs active at the same time. For example Fig. 3 also shows the MSD ArmAMoveFromTableTo-PressAssumption and a cut that an active copy of that MSD would be in after the same sequence of messages.

A *safety violation* occurs in an active MSD if in a hot cut a message event occurs that can be unified with a message in the MSD that is not currently enabled. If this happens in a cold cut, this is called a *cold violation*. Safety violations are forbidden to occur. Cold violations are allowed to occur, but lead to the termination of the active MSD. An active MSD must not remain forever in an executed cut, otherwise this is a *liveness violation*.

2.3.2 Environment Assumptions

Since the environment is uncontrollable, we must assume that environment events can occur in an arbitrary order. For the active MSD ArmATransportBlankToPress with the cut as shown in Fig. 3, waiting for the environment event `arrived-AtPress`, this means that this MSD can easily be violated by the environment: instead of arriving at the press, the Arm A could arrive at the table (`arrivedAtTable`) or the next blank could arrive (`blankArrived`).

Often requirements cannot be fulfilled by a system without making certain assumptions on what will or will not happen in the environment [29]. In the case above, we want to specify that if Arm A moves to the press and returns to the table, blanks will not arrive before Arm A has returned to the table (see A3 in Fig. 1). Furthermore, if Arm A is ordered to move to the press/table, we assume that it will eventually arrive at the press/table (see A1 and A2 in Fig. 1). We also imply that if Arm A is ordered to move to the press/table it will not arrive at the table/press instead.

In order to specify such environment assumptions, we introduced *assumption MSDs* [13]. Fig. 3 shows for example the assumption MSD ArmAMoveFromTableToPressAssumption, which describes that if Arm A is ordered to move to the press, it will also eventually arrive at the press (A1). Assumptions A2 and A3 can also be modeled with assumption MSDs, which we omit for brevity. Technically, assumption MSDs are annotated with a stereotype ⟨⟨Environment-Assumption⟩⟩.

Another, novel extension to MSD specifications that we introduce with this paper is that we distinguish environment events which can occur *spontaneously* from those that only occur *in reaction* to other events in the system and/or environment. We call these events *non-spontaneous* environment events. Non-spontaneous messages can only occur if there is an executed message enabled in an active assumption MSD that can be unified with this event. This is called the *non-spontaneous events assumption*. This concept allows us to further constrain the description of meaningful environment behavior.

Technically, to model this, we annotate the operations that type non-sponaneous environment messages with the stereotype ⟨⟨nse⟩⟩ on operations, see Fig. 3.

2.3.3 Realizability, Consistent Executablility

In the following, we give definitions for different forms of *realizability* of MSD specifications with environment assumptions and non-spontaneous environment events.

We consider reactive systems that continuously react to environment events. We call an infinite sequence of environment and system events a *run*. A run is *accepted* by an MSD if it does not lead to a safety or liveness violation of the MSD. A run *satisfies* an MSD specification iff (1) it is accepted by all requirement MSDs or not accepted by at least one assumption MSD and (2) it does not contain an infinite sequence of system events, i.e., the system must periodically listen for environment events.

We assume that the behavior of the system and the environment can be described by finite state controllers that can be composed in such a way that the environment never blocks system messages and will always eventually send a next environment event. The composition of a system controller together with an environment controller *satisfies* an MSD specification iff every run produced by the composed controllers satisfies the MSD specification.

We make the general assumption that the system is always fast enough to send any (finite) number of system messages before the next environment event occurs; this setting is typically assumed for LSC/MSD specifications [18, 17]. A system controller is called an *implementation* of an MSD specification if the controller formed by the composition of the system controller and any possible environment controller respecting the non-spontaneous events assumption satisfies the MSD specification. An MSD specification is *realizable* or, synonymously, *consistent*, if an implementation exists.

The above definition of realizability also considers that the system can send messages that are not currently active (enabled executed in at least one active MSD). This, however, is usually not the behavior that is desired by the engineers, since no MSD states that these messages should occur [13]. To consider systems that send only active events, we introduce a more restrictive definition. We call a system controller that only sends events that correspond to active events a *consistently executing controller*. We call an MSD specification *consistently executable* if there exists a consistently executing implementation.

The synthesis technique presented in the next section is able to check the consistency and consistent executability of MSD specifications and synthesize respective implementations if they exist.

Given that a feature adds one or more MSDs to the specification, it may happen that the specifications of two features are consistent when considered separately, but inconsistent when combined together. Conversely, the addition of a third feature may also resolve the conflict arisen from the interaction of the former two features, for example because it introduces assumption MSDs. Thanks to its capability of identifying inconsistent feature combinations, our synthesis algorithm is therefore apt to detect *feature interactions*.

3. ON-THE-FLY SYNTHESIS

Our previous consistency-checking approach [14] employs model-checking to check the consistent executability of the MSD specifications of products. Roughly, the approach checks for an MSD specification (containing only requirement MSDs) if always in reaction to a next possible environment event the system can send a sequence of messages so that no safety violation occurs and no active messages remain (and so never a liveness violation occurs).

If a negative result is returned by the model-checker for a product specification, however, this does not always imply that the MSD specification was not consistently executable. It could also be that, if after an environment event a safety or liveness violation is inevitable, this state could have been avoided by choosing another sequence of system steps in reaction to an earlier environment event. In this case, our previous approach thus returns a false negative.

In the following, we first introduce an example where such a false negative would occur in our previous model-checking approach. We then introduce our new synthesis technique.

3.1 Avoiding False Negatives (Example Cont.)

Figure 4 shows the MSDs for the features Press and DropHammerPress as stated informally in Fig. 1.

The problem with this specification is the following. In combination with the MSD ArmATransportBlankToPress, the MSD PressPlateAfterArmAReleasesBlank specifies that after `releaseBlank`, Arm A must return to the table (`moveToTable`) and the press must press the blank (`press`); the order of these events is not determined.

The MSD ArmALeavesCriticalAreaBeforePressingStarts models the assumption that if first Arm A is ordered to move to the table (`moveToTable`) and after that the press presses the blank (`press`), then Arm A will have left the critical area (`leftCriticalArea`) before the pressing process starts (`pressingStarted`).

If, however, the system chooses, after `releaseBlank`, to first order the press to press (`press`) and then to order Arm A to move to the table (`moveToTable`), it is not guaranteed that Arm A will have left the critical area before the pressing

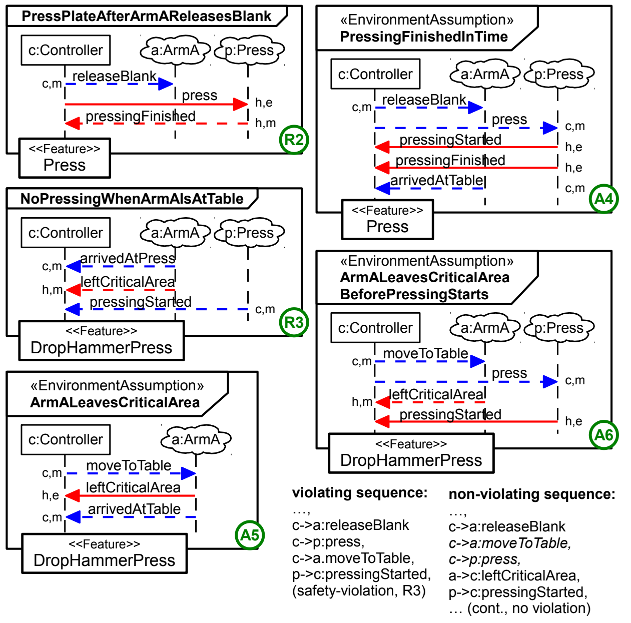


Figure 4: The MSDs specifying the features Press and DropHammerPress

starts. This will violate the requirement MSD NoPressingWhenArmAIsAtTable (which shall prohibit that Arm A is crushed by the drop-hammer). Parts of a violating and a non-violating event sequence, starting with `releaseBlank`, are shown at the bottom of Fig. 4.

In this example, after the "bad choice" of sending `moveToTable` after `press`, the system cannot guarantee to avoid a violation that may occur after a subsequent environment event. Our model-checking-based approach [14] would not be able to find that sending `press` before `moveToTable` is an admissible solution. The reason for this is that model-checking approaches cannot determine whether a violation in reaction to environment events, or *by* an environment event, can be avoided by choosing a particular reaction to an earlier environment event. (See our discussion in [14].)

3.2 On-the-fly Synthesis Algorithm

Instead of viewing the realizability-checking problem as a model-checking problem, we view it as the problem of finding a strategy in an infinite two-player game [1, 6].

In the game, the player is the system and the opponent is the environment. The game structure is a graph where the nodes, also called *states*, are the reachable combinations of cuts of active MSDs. The *initial state* corresponds to no active MSDs. Edges between the states, also called *transitions*, are labeled with the message events appearing in the MSD specification. They represent "moves" by the system or the environment. A move may or may not cause a state change (if not, we have a self-transition). Transitions labeled with system events are called *controllable*; transitions labelled with environment events are called *uncontrollable*.

In our objective to check the consistent realizability of an MSD specification as explained in Sect. 2, we can make the following restrictions in the game graph. First, in states where there are executed cuts in active requirement MSDs, there will only be controllable transitions with events corresponding to active messages. Second, otherwise there will be only uncontrollable transitions with (a) environment events that can occur spontaneously or (b) non-spontaneous events if they are active in assumption MSDs.

For checking the realizability of MSD specifications, we have to check if the system, by choosing controllable transitions in the graph, can guarantee to reach infinitely often a state where the following condition holds: (1) There was no safety violation in any requirements MSD (this information is also retained in the states) *and* (2) there are no active requirement MSDs with executed cuts *or* (3) there was a safety violation of an assumption MSD *or* (4) there is at least one active assumption MSD in an executed cut. Such an objective, where a condition must be satisfied infinitely often, is called a *Büchi objective*. States where the condition is satisfied are called *goal states*.

If there exists a *strategy* for the system to fulfill this objective regardless of the moves of the environment, the MSD specification is consistently executable. A strategy is a function that for every (reachable) state in the game specifies a set of transitions that the system can choose to guarantee the objective. (For states with only uncontrollable outgoing transitions, this set is empty.) From a strategy, we can derive a consistently executable controller for the system.

We developed a new on-the-fly algorithm for finding strategies in Büchi games. *On-the-fly* means that the algorithm will only explore parts of the game graph. The algorithm is based on an on-the-fly algorithm described by David et al. [11, Sect. 6.4.5] (see also Cassez et al. [8]). Our algorithm is novel in two aspects. One novelty is its incremental extension (described in Sect. 3.3). Another novelty is that our algorithm often explores even less of the game graph.

In the scope of this paper, we cannot account for the latter claim in detail. Roughly, however, the difference is as follows. If the algorithm by David et al. finds that from a goal state the system can guarantee reaching another goal state, this goal state is marked *winning*, and so are all other states where the system can guarantee reaching a goal state. If the initial state is winning and all encountered goal states are winning, the algorithm returns true. If, however, it turns out that an encountered goal state is not winning, then first the entire game graph is explored to find all goal states. Second, it is then again checked for all winning goal states if from there it can be guaranteed to reach another winning goal state. Whenever in this process a goal state loses its winning status, this process is repeated. If finally the initial state is winning, the algorithm returns true, otherwise false.

In our variant of the algorithm, instead of marking states winning, we map every state to a set of goal states that the state requires to be also winning in order justify its own winning status. This way, we know which goal states rely on which other goal states. If now a goal state turns out not to be winning, we then know which goal states we have to re-evaluate, i.e., for which goal states we have to find another way of reaching other goal states. In many cases, this map greatly helps to reduce the number of such re-evaluation steps. The worst-case time complexity, however, which is quadratic in the size of the game graph, is not changed.

The algorithm is shown in pseudo-code in Alg. 1 and 2. The OTFB procedure (On-The-Fly Büchi algorithm) checks if from a given start state, the system can guarantee reaching goal states infinitely often. It relies on the OTFR procedure (On-The-Fly Reachability algorithm), which checks if, from a given start state, the system can guarantee reaching goal states. The global map *reqG* is the aforementioned map. In the global set *goal*, the algorithm stores states that it encounters and identifies as goal states by using the isGoal method. The global set *lose* is used to mark states losing if the algorithm finds that from there the system cannot guarantee reaching another goal state.

Algorithm 1 On-the-fly Algorithm for Büchi Games (Part 1, OTFB procedure)

```
 1: global variables:
 2: Set<State> $goal$;  ▷ goal states added during calls of isGoal
 3: Set<State> $lose$;  ▷ states where Büchi cond. not guaranteed
 4: Map<State, Set<State>> $reqG$;  ▷ states to req. goal states

 5: procedure OTFB($q_0$)
 6:   !OTFR($q_0$) : return $false$;
 7:   Stack<State> $reeval$ = getUndecidedGoals();
 8:   while !$reeval$.isEmpty() do
 9:     State $g$ = $reeval$.pop();
10:     if !OTFR($g$) then
11:       $g == q_0$ : return $false$;
12:       $lose$.add($g$);
13:     end if
14:     $reeval$.isEmpty() : $reeval$ = getUndecidedGoals();
15:   end while
16:   return $true$;
17: end procedure
```

We call OTFB with the initial state of our MSD specification game graph. OTFB then first checks whether from the initial state, the system can guarantee reaching a goal state. If not, it returns false immediately. Then the method getUndecidedGoals computes all goal states which are not winning or losing. A state is winning if it has an entry in the *reqG* map where no goal state in the value set is losing. We call OTFR again on all states that are not winning nor losing as long as such states exist. In every iteration of the loop goal states will be either marked winning or losing. A winning state may become losing, but a losing state remains losing. In this process, the initial state of the game graph may turn out to be losing, and we return false (l. 11), or the initial state is among a set of goal states that remain winning, and we return true.

The procedure OTFR (see Alg. 2) is based on the reachability algorithm described by Cassez et al. [8]. From a given start state, the algorithm performs a depth-first forward exploration, using the *waiting* stack. Already visited states are stored in the *passed* set. The *depend* map stores for every state the transitions by which it was visited.

In the while-loop, after the next transition is popped from the stack, there are the following cases. First, if the target state was not visited yet (l. 27), the transition is added to the *passed* set and the outgoing transitions of the target state will be pushed on the *waiting* stack (l. 46). If, however, the target state is a goal state or is a winning state (has an entry in *reqG*), or is a losing state, instead the transition is pushed on the *waiting* stack again (l. 31). This has the effect that it is popped again in the next iteration.

If a transition is popped that is already in *passed*, the algorithm enters the else branch (l. 36). Here the algorithm performs a *re-evaluation* of the losing resp. winning status of the source state: if the source state was not marked losing before and the state has no outgoing transitions or the system cannot avoid reaching a losing direct successor (de-

Algorithm 2 On-the-fly Algorithm for Büchi Games (Part 2, OTFR procedure)

```
18: procedure OTFR($q_0$)
19:   Set<State> passed;                          ▷ a set of visited states
20:   Stack<Transition> waiting;                  ▷ transitions to be explored
21:   Map<State, Set<Transition>> depend;         ▷ for bwd. reeval.
22:   pushOutTransitions($q_0$, waiting);
23:   while !waiting.isEmpty() $\wedge$ !reqG.containsKey($q_0$) do
24:     Transition $t$ = waiting.pop();
25:     State $q_{src}$ = $t$.getSourceState();
26:     State $q_{tgt}$ = $t$.getTargetState();
27:     if !passed.contains($q_{tgt}$) then       ▷ forward exploration
28:       passed.add($q_{tgt}$);
29:       depend.get($q_{tgt}$).add($t$);
30:       clean(reqG, $q_{tgt}$);    ▷ Remove $q_{tgt}$ entry from reqG if
                   $q_{tgt}$ is mapped to goal states prev. marked losing
31:       if isGoal($q_{tgt}$) $\vee$ reqG.containsKey($q_{tgt}$)
                 $\vee$ lose.contains($q_{tgt}$) then
32:         waiting.push($t$);
33:       else
34:         pushOutTransitions($q_{tgt}$, waiting);
35:       end if
36:     else                                      ▷ re-evaluation
37:       if !lose.contains($q_{src}$) $\wedge$ isLose($q_{src}$) then
38:         lose.add($q_{src}$);
39:         waiting.pushAll(depend.get($q_{src}$));
40:         removeOutTrans($q_{src}$, waiting);   ▷ optimization
41:       else if !reqG.containsKey($q_{tgt}$)    ▷ no entry curr. but
               $\wedge$ update(reqG, $q_{src}$)) then    ▷ new one created
42:         waiting.pushAll(depend.get($q_{src}$));  ▷ for bwd re-eval
43:         removeOutTrans($q_{src}$, waiting);   ▷ optimization
44:       end if
45:       if !reqG.containsKey($q_{tgt}$) $\wedge$ !lose.contains($q_{tgt}$)) then
46:         depend.get($q_{tgt}$).add($t$);
47:       end if
48:     end if
49:   end while
50:   return reqG.containsKey($q_0$);
51: end procedure
```

termined by the method isLose), then the state is marked losing and all the incoming transitions stored on the *depend* map for the source state are scheduled for reevaluation. The same happens if the source state was non-winning initially, but its winning status gets updated (l. 41).

The method update checks for a state whether the system can guarantee to reach a winning successor state and, if so, will create a new *reqG* entry for the state and return true. If a winning successor can be reached via a controllable transition, the *reqG* entry value for the source state will be set to the same value as stored in *reqG* for the target state. If winning successors can be reached via (all) uncontrollable transitions, the *reqG* entry value for the source state will be set to the union of the values stored in *reqG* for all these transitions' target states.

On the correctness of the algorithm: (1) The OTFR procedure always terminates and it returns true iff the start state is winning (i.e., an entry in *reqG* exists or could be created). Our extensions do not change the procedure in principle: the idea of backward-evaluating the losing status of states (l. 37) and the optimizations in lines 40 and 43 are already discussed by Cassez et al. [8]. Instead of marking winning states by inclusion in a winning set, they are keys in the *reqG* map in our case. Since the procedure is not changed in principle, the correctness of the OTFR procedure can be argued along the lines of [8, 11].

(2) The OTFB procedure always terminates and it returns true iff from the start state the system can guarantee to reach goal states infinitely often, which thus means that the MSD specification is realizable. The while-loop terminates for the following reason: Either eventually all goal states are losing, then getUndecidedGoals returns an empty stack. If instead at some point no more goal states are added to *lose*, this means that all states in *reeval* will be winning, and then again getUndecidedGoals returns an empty stack. Since in our case the initial state is a goal state, we can immediately return false if we find that it is losing. Conversely, if finally the initial state is not found losing, it means that it is among a set of winning goal states, from where the system can always guarantee to reach another (winning) goal state.

3.3 Incremental Extension

Transitions that form part of a possible winning strategy for the system are called *winning moves* of the system. If a winning strategy exists, the above algorithm may by chance explore the state graph in such a way that all explored controllable transitions are winning moves. If it explores non-winning moves, *e.g.* by first sending `moveToTable` and then `press` in the production cell example (see Fig. 4), the algorithm will eventually have to backtrack, and will require more time to complete.

Since the products of a PL likely share commonality, our idea is to synthesize controllers *incrementally*, which means helping the synthesis of one controller avoid non-winning moves by remembering winning moves in the controller synthesized previously for a similar product. In the production cell, if a controller was already synthesized for the product with the drop-hammer press and without Arm B, this controller could help avoid the non-winning move during the synthesis of the controller for the product with Arm B.

The incremental extension of the algorithm plugs into the previous synthesis algorithm in two ways. First, during the exploration of a new transition in the game graph, *correspondences* to similar states in a controller for another product (called *base controller* hereafter) are maintained. Intuitively, states are similar if they can be reached by the same event sequence. Formally, let q be a state of the base controller that corresponds to a state q' in the game graph, Σ be the set of message events labeling transitions in the base controller, and Σ' be the set of events specified in the MSD specification. Then, correspondences are formed according to the following rules:

1. If q and q' both have outgoing transitions labeled with the same event, the target states of these transitions are also corresponding. This also applies if these transitions are self-transitions.
2. If an outgoing transition of q is labeled with an event that is not element of Σ', the target state of the transition leaving q also corresponds to q'.
3. If an outgoing transition of q' is labeled with an event that is not element of Σ, the target state of the transition leaving q' also corresponds to q.

Second, the incremental extension plugs into the method pushOutTransitions (l. 22) to modify the order in which transitions are pushed on the *waiting* stack, which determines the order in which they are explored. A transition has *priority* if its source state has a corresponding base controller state with an outgoing transition labeled with the same message event and if it is not safety-violating. pushOutTransitions pushes transitions with priority last onto the stack.

4. SYNTHESIS FOR PRODUCT LINES

In order to synthesize a controller for each product, we need to derive all products from the feature diagram. In order to exploit the benefit of the incremental synthesis, we require a method that determines which controllers should serve as a basis for the synthesis of other controllers.

We propose an automated strategy to enumerate and synthesize all the variants from an FD. It consists in starting from the *backbone*, which is the set of features that appear in every valid product, and then adding more and more features until we obtain a valid product, for which we then synthesize a controller. The synthesis is scheduled according to a top-down exploration of the FD.

Algorithm 3 implements our strategy. We first consider the product that contains the root feature and all the features required by it (line 4). By "required" we mean not only features that the root *directly* requires, but also the features required by those (*e.g.*, their parent). Together, all these features form the backbone. We also compute the set of features that the root excludes. These are dead features, which are part of no valid product since the root is mandatory. As for inclusion, the exclusion is not limited to features directly excluded by the root. Given that *require* and *exclude* are the only cross-tree constraints, the computation of the set of features required/excluded by a given feature comes down to parsing the constraints and exploring the FD.

Once we have computed the backbone, we begin the exploration (lines 6–19). At each iteration, for each feature f newly added, we compute the subsets of its child features that satisfy the decomposition relationship (line 9). Intuitively, each subset represents a choice in the child features to add next. For example, if the decomposition type is XOR then the choices are singletons, one per child feature. In (resp. for) every subset, we also add (resp. compute) the features required (resp. excluded) by the features in the subset. We obtain a set of *choices*, which is a set of features to add and another set of features to exclude. We then pursue the exploration for each of these choices (lines 11–13), subsequently adding more choices to consider.

Each time features are added to the included or excluded feature sets, the algorithm checks that the two sets have an empty intersection (line 8). This ensures that only valid products are considered. Once there is no more choice to make (lines 14–17), that is, every feature is either included or excluded, we synthesize the product defined by the set *Inc* of included features (line 15). The resulting controller will serve as a basis for the next synthesis (line 16).

For the production cell FD in Fig. 2, the top-down strategy first computes the set of features required by the root, that is, {ProductionCell, Feed, ArmA, Press, Deposit }. The algorithm then computes which set of features can be added next. Regarding feature Press, it may choose to add either RollerPress or DropHammerPress. For feature Deposit, it may add ArmB or not; since it is optional, the algorithm excludes it at first. The strategy selects the feature to include in a non-deterministic fashion. If RollerPress is selected, we obtain the product {ProductionCell, Feed, ArmA, Press, Deposit, RollerPress }, for which then a controller is synthesized. The algorithm then backtracks to its unique previous point of choice. As before, it does not include ArmB. Then the algorithm adds DropHammerPress and discards RollerPress. Finally, the algorithm backtracks to its first point of choice, includes ArmB, and repeats the same process.

Algorithm 3 Top-down exploration strategy.

```
 1: procedure TOPDOWN(root)
 2:   Stack stack;
 3:   Map<Set<Feature>, Controller> controllers;
 4:   stack.push({root}.addAll(requires(root)),
        excludes(root), {root}.addAll(requires(root)));
 5:   Controller previous = ⊥
 6:   while !stack.isEmpty() do
 7:     (Set<Feature>, Set<Feature>, Set<Feature>)
          (Inc, Exc, LastInc) = stack.pop();
 8:     if Inc.intersect(Exc).isEmpty() then
 9:       Set<(Set<Feature>, Set<Feature>)>
              Choices = computeChoices(LastInc);
10:       if !Choices.isEmpty() then
11:         for (ToInc,ToExc):Choices do
12:           stack.push(Inc.addAll(ToInc), Exc.addAll(ToExc),
                ToInc.removeAll(Inc));
13:         end for
14:       else
15:         controllers.put(Inc, synthesize(Inc, previous));
16:         previous = controllers.get(Inc);
17:       end if
18:     end if
19:   end while
20: end procedure
```

5. IMPLEMENTATION & EVALUATION

We implemented the on-the-fly synthesis algorithm (cf. Sect. 3), including the incremental variant, and the top-down strategy for product line synthesis (cf. Sec. 4) as part of our Eclipse-based tool suite ScenarioTools. ScenarioTools supports modeling of feature diagrams and MSD specifications by providing suitable UML profiles. This allows engineers to model product lines and system specifications with any Eclipse-based UML editor.

ScenarioTools also supports an interactive simulation of MSD specifications. The simulation and the synthesis both rely on a common implementation of an execution logic which interprets the MSD specification [7]. The tool and the examples presented in this paper can be downloaded[1].

For our production cell example, ScenarioTools successfully synthesizes controllers for all four realizable products. We avoid the false negatives (see Sect. 3.1) as predicted. For this example, the incremental synthesis only shows insignificant improvements. When comparing one single run of the incremental and non-incremental synthesis, the non-incremental synthesis may even be faster. This is because the order in which transitions are pushed on the waiting stack is non-deterministic and the non-incremental synthesis can be lucky and avoid non-winning moves. The incremental synthesis can be unlucky if products are derived in an order where incrementality is not beneficial.

For example, if the two products with the drop-hammer press are synthesized consecutively, incrementality is beneficial for the second synthesis run. If however, a product with the drop-hammer press is synthesized after a product with a roller press, the incremental extension of the synthesis algorithm may even force it to pursue non-winning moves. (Here: `moveToTable` after `press`). Comparing 100 synthesis runs of the incremental and non-incremental algorithm, we observe in average 95 states visited by the first, and 96 visited by the second, which is an insignificant improvement. The synthesis takes between 150 and 250 milliseconds.

[1]`http://scenariotools.org`

To further evaluate the benefit of our incremental synthesis approach, we use a technical example (named the "cascading example" hereafter), which we systematically extend to create new examples with larger state spaces.

The structure of our example is presented in Fig. 5. In these specifications, each feature has at most two child features connected with an OR-relationship. To increase the size of the example, we raise the number of features. For $i \geq 1$, there exist only features at depth i in the FD if all at depth $i-2$ already have two child features. Each feature is connected with one MSD named after the feature. In these MSDs, the first message is a cold message and is followed by two hot messages. Only the first message in the MSD of the root feature is an environment message. The first message of a child feature is named like the two hot messages of its parent's MSD. The activation of an MSD of a feature thus triggers two activations of an MSD for each child feature. In addition to this MSD, every feature at depth 3 from the root has an additional MSD that introduces a non-winning move for the system, where the synthesis may run into a "dead end" from which it must backtrack (see Bad3_1 in Fig. 5).

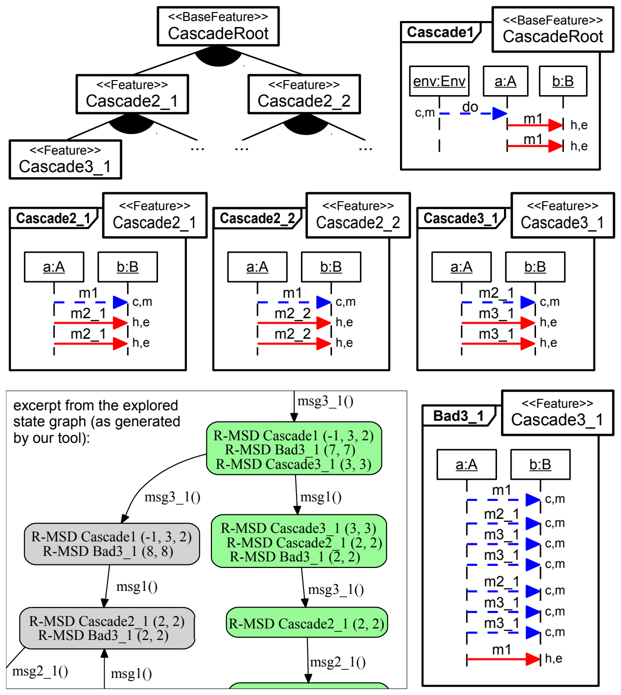


Figure 5: The evaluation example

Bad3_1 introduces a non-winning move as follows. Consider the specification of a product that consists of the features Cascade1, Cascade2_1, and Cascade3_1. Note, that including Cascade3_1, according to what we said above, would imply including Cascade2_2, but we ignore this now. Assume that the following sequence of events occurred: `do`, `m1`, `m2_1`, `m3_1`, `m3_1`, `m2_1`, `m3_1`. Then, the system is in the state shown at the top of the state-graph shown at the bottom in Fig. 5. There are two options for the next event:

m1 A cold violation occurs in Bad3_1, which results in the termination of this scenario. The only sequence of events permitted to occur next is `m3_1`, `m2_1`, `m3_1`, `m3_1`, `m2_1`, `m3_1`, `m3_1`. This sequence will cause a cold violation in Bad3_1, which will result in its termination. This strategy satisfies the MSD specification.

m3_1 We reach the hot cut of Bad3_1 and `m1` must occur, which activates Cascade2_1 and Bad3_1 anew. Then the only admitted sequence of events is `m2_1`, `m3_1`, `m3_1`, `m2_1`, `m3_1`, `m3_1`, which enables again `m1` in Bad3_1. The process is repeated ad infinitum; since there is always an active MSD in an executed cut, this implies a liveness violation.

The non-winning moves shall pose a challenge for the incremental synthesis to avoid some of the backtracking. The examples are designed to be highly challenging even with a small set of features and MSDs, due to the exponentially increasing number of MSD activations. Also, the "dead-ends" from which the synthesis must backtrack may be huge compared to the controller that will finally be produced.

Based on that example, we compare the performance of our incremental algorithm combined with the top-down strategy exploration with respect to synthesizing every product controller from scratch. We test examples with 3 to 15 features and evaluate the scalability of the two approaches with respect to the number of features and MSDs.

We found that the examples are already very challenging for our tool, which was thus far not optimized for performance. Models with more than 9 features were too complex to be considered. To nevertheless provide results, we instead consider a feature diagram with OR child feature relationships, but XOR child feature relationships at level 2. Furthermore, we considered experiments where all feature relationships are XOR.

All benchmarks were run on a Windows PC with a 3,3 GHz Intel Core i5 processor and 8 GB of DDR3 RAM. To avoid variations due to non-determinism in the synthesis algorithm, we repeated each experiment 20 times and computed the average of the synthesis times.

The benchmark results are shown in Table 1. For each experiment, we describe the parent-child relationship, the number of features and products, and for each approach the average number of explored states, the average number of explored losing states, the average synthesis time, and finally the speedup provided by the incremental approach.

We observe that the incremental approach is more efficient in most of the cases. In the OR experiments, the two approaches perform equally well on the 3-feature model. This is expected, since this example does not introduce non-winning moves. However, as the number of products increases, the incremental approach becomes more efficient.

In the XOR experiments the speedup ranges from 1.04 to 2. The only exception is the 15-feature model, for which the non-incremental approach is slightly better. But the results vary. We ran another set of trials for the same model, which had contrary results, in favor of the incremental approach.

Our interpretation of the results is that the incremental algorithm brings improvements for examples where the number of products grows more than linearly with the number of features; for such examples the speedup also increases with the number of features. These conclusions are, however, preliminary, and we plan to run more experiments in the future to support it. There are examples where the improvement of the incremental algorithm is only marginal; in some cases the incrementality also forces the algorithm to pursue non-winning moves first (as above). We will explore smarter ways of deriving products from FDs to avoid this.

Table 1: Synthesis times for the cascading example.

Example kind	#Features	#Products	avg(#States) Inc.	avg(#States) N-inc.	avg(#Losing) Inc.	avg(#Losing) N-inc.	Time (ms) Inc.	Time (ms) N-inc.	Speedup
XOR	3	2	14	14	0	0	4	8	2
XOR	5	3	38	40	1	3	13	16	1.23
XOR	7	4	63	70	3	10	19	25	1.31
XOR	9	5	109	113	2	6	29	33	1.13
XOR	11	6	155	164	1	10	38	43	1.13
XOR	13	7	201	203	0	2	43	45	1.04
XOR	15	8	248	256	0	8	54	51	0.94
OR+XOR(level2)	3	3	25	25	0	0	6	6	1
OR+XOR(level2)	5	5	79	96	4	21	22	107	4.86
OR+XOR(level2)	7	8	638	730	470	562	316	360	1.13
OR+XOR(level2)	9	14	4419	5158	3921	4660	3551	3992	1.12
OR+XOR(level2)	11	20	7909	10111	7081	9283	6509	8400	1.29
OR+XOR(level2)	13	34	18811	24118	16925	22232	18555	25016	1.35
OR+XOR(level2)	15	48	41738	55502	38794	52558	43072	57930	1.34

6. RELATED WORK

There are many approaches for consistency-checking and synthesizing controllers from formal scenario specifications [28, 15, 6, 16, 13, 23]. Also the relationship between scenarios and goals was studied in the past [10]. However, there are but a few approaches that consider formal scenario-based specification of PLs.

Ziadi *et al.* synthesize statecharts from sequence diagrams where interaction fragments can be annotated to be active only in certain variants [30]. They thus model variability in an annotative fashion, whereas ours is compositional. Another difference is that they do not consider that inconsistencies may arise between the scenarios or the features.

Ghezzi and Molzam propose an approach to verify non-functional requirements of SPLs [12]. They model the system's behavior with sequence diagrams where fragments can be annotated to be active only in certain products. However, they do not consider concurrent scenarios.

The relationship between FDs and structural as well as behavioral UML models was studied, e.g., in [20, 27, 2, 26]. Shaker *et al.* recently proposed to model PL behavior with a combination of FDs and a feature-aware extension of statecharts [26], where features are introduced as a new parallel region, and may change the priorities or the triggering conditions of transitions. None of these methods is equipped with consistency checking or synthesis mechanisms.

Apel *et al.* [3] extended Alloy with collaboration-based design and feature composition. In their extension, features are defined as refinements of modules. The specification of a feature consists of predicates describing assumptions and assertions. Based on these specifications, the Alloy Analyzer can detect structural and semantic dependences between the features. However, Alloy cannot express complex behavioral interactions and detect inconsistencies in them. SPLVerifier [4] is a feature-aware verification tool for verifying inconsistencies and harmful feature interactions in C or Java code. There, features are specified in isolate modules. 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 [19]. 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 the interactions between multiple components.

7. CONCLUSION

In this paper, we presented a novel technique for synthesizing controllers from scenario-based product line specifications. We introduced a new game-based synthesis algorithm that overcame the limitations of our previous approach. We also provided an extension for the incremental synthesis of controllers, which exploits the similarities between product specifications in PLs. We implemented the approach prototypically. Evaluations show that the incremental variant of the algorithm could outperform the non-incremental variant in many cases. Altogether, we pave the way for the intuitive and rigorous design of PLs of reactive systems.

In the future, we will design new heuristics that further improve the performance of the incremental algorithm. In particular, we will investigate alternative strategies to explore the FD, and thus to schedule the products to synthesize. A new strategy could be to perform a bottom-up exploration, starting from a product including all the features and successively removing features. Other variants of the top-down strategy can be defined by changing how OR-relationships are dealt with. For instance, we can choose the next features to include based on how much the MSDs connected to its features conflict (that is, share events) with the MSDs connected to the already added features.

A limitation of our approach is that it cannot cope with arbitrary Boolean constraints between the features. We can extend our feature-tree exploration algorithm to support any Boolean feature constraints; in this case, however, we have to combine it with backbone computation algorithms [24].

Acknowledgments

This research is funded by the European Commission, Programme IDEAS-ERC, Project 227977 SMScom, and by the Fund for Scientific Research – FNRS in Belgium, Project FC 91490. Christian Brenner is supported by the International Graduate School "Dynamic Intelligent Systems".

8. REFERENCES

[1] M. Abadi, L. Lamport, and P. Wolper. Realizable and unrealizable specifications of reactive systems. In *Proc. of ICALP '89*, volume 372 of *LNCS*, pages 1–17. Springer, 1989.

[2] M. Alférez, R. E. Lopez-Herrejon, A. Moreira, V. Amaral, and A. Egyed. Supporting consistency checking between features and software product line use scenarios. In *Proc. of ICSR'11*, volume 6727 of *LNCS*, pages 20–35, Berlin, Heidelberg, 2011. Springer.

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

[4] S. Apel, H. Speidel, P. Wendler, A. von Rhein, and D. Beyer. Feature-interaction detection using feature-aware verification. In *Proc. of ASE'11*, pages 372–375. IEEE, 2011.

[5] T. Berger, R. Rublack, D. Nair, J. M. Atlee, M. Becker, K. Czarnecki, and A. Wąsowski. A survey of variability modeling in industrial practice. In *Proc. of VaMoS'13*, pages 7:1–7:8. ACM, 2013.

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

[7] C. Brenner, J. Greenyer, and V. Panzica La Manna. The ScenarioTools play-out of modal sequence diagram specifications with environment assumptions. In *Proc. of GT-VMT'13*, 2013. (to appear).

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

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

[10] C. Damas, B. Lambeau, and A. van Lamsweerde. Scenarios, goals, and state machines: a win-win partnership for model synthesis. In *Proc. of FSE'06*, pages 197–207, New York, NY, USA, 2006. ACM.

[11] A. David, G. Behrmann, P. Bulychev, J. Byg, T. Chatain, K. G. Larsen, P. Pettersson, J. I. Rasmussen, J. Srba, W. Yi, K. Y. Joergensen, D. Lime, M. Magnin, O. H. Roux, and L.-. Traonouez. Tools for model-checking timed systems. In *Communicating Embedded Systems – Software and Design*, pages 165–225. ISTE Publ./John Wiley, 2009.

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

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

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

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

[16] 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, pages 309–324. Springer, 2005.

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

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

[19] 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.

[20] 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.

[21] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical report, Software Engineering Institute, Carnegie Mellon University, 1990.

[22] C. Lewerentz and T. Lindner. *KORSO: Methods, Languages, and Tools for the Construction of Correct Software*, volume 1009 of *LNCS*, chapter Case study "production cell": A comparative study in formal specification and verification, pages 388–416. Springer, 2006.

[23] S. Maoz and Y. Sa'ar. Assume-guarantee scenarios: semantics and synthesis. In *Proc. of MODELS'12*, pages 335–351. Springer, 2012.

[24] J. Marques-Silva, M. Janota, and I. Lynce. On computing backbones of propositional theories. In *Proc of ECAI'10*, pages 15–20. IOS Press, 2010.

[25] P.-Y. Schobbens, P. Heymans, and J.-C. Trigaux. Feature diagrams: A survey and a formal semantics. In *Proc. of RE'06*, pages 139–148. IEEE, 2006.

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

[27] 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.

[28] J. Whittle and J. Schumann. Generating statechart designs from scenarios. In *Proc. of ICSE'00*, pages 314–323. ACM, 2000.

[29] P. Zave and M. A. Jackson. Four dark corners of requirements engineering. *ACM Transactions on Software Engineering and Methodology*, 6(1):1–30, 1997.

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