Efficient Consistency Checking of Scenario-Based Product-Line Specifications

Joel Greenyer and Amir Molzam Sharifloo
Dependable Evolvable Pervasive Software Engineering,
Dipartimento di Elettronica e Informazione,
Politecnico di Milano, Italy
{*greenyer*|*molzam*}@*elet.polimi.it*

Maxime Cordy* and Patrick Heymans
PReCISE Research Center,
Faculty of Computer Science,
University of Namur, Belgium
{*mcr*|*phe*}@*info.fundp.ac.be*

***Abstract*—Modern technical systems typically consist of multiple components and must provide many functions that are realized by the complex interaction of these components. Moreover, very often not only a single product, but a whole product line with different compositions of components and functions must be developed. To cope with this complexity, it is important that engineers have intuitive, but precise means for specifying the requirements for these systems and have tools for automatically finding inconsistencies within the requirements, because these could lead to costly iterations in the later development. We propose a technique for the scenario-based specification of component interactions based on Modal Sequence Diagrams. Moreover, we developed an efficient technique for automatically finding inconsistencies in the scenario-based specification of many variants at once by exploiting recent advances in the model-checking of product lines. Our evaluation shows benefits of this technique over performing individual consistency checking of each variant specification.**

***Keywords*-scenario-based specification; product lines; feature compositions; consistency**

I. Introduction

Modern technical systems in areas like transportation or production, but also information systems, typically consist of many components that provide many functions by their interaction. These interactions are sometimes safety-critical and must satisfy complex protocol specifications.

Moreover, often today not only a single product, but a whole *product line*, i.e., many variants of a product, must be developed. Doing this individually is often impractical, so the goal of *product line engineering* [1] is to consider all the variants together throughout the whole development process. One widespread approach to organize a product line is to structure the sets of components and functions that may or may not be present in different variants into *features*.

Capturing a precise specification of a product line, however, is a major requirements engineering challenge. Not only complex interactions and many product variants must be specified, but the behavioral requirements may also imply dependencies and conflicts among features. The requirements have to be carefully revised to ensure that the features can be consistently combined. If inconsistencies remain undetected, desired product variants may not be realizable without costly iterations.

As an example we consider a simplified specification of an autonomous rail vehicle, inspired by the RailCab project at the University of Paderborn[1]. The RailCab track system is divided in sections. For every variant of the system we specify that the vehicles, called *RailCabs*, must request a *switch controller* the permission to enter a switch. The switch controller must then acknowledge or deny such a request. Now there shall be different variants. In one, the switch controller lets only one RailCab pass at a time. In another variant, two RailCabs shall be able to coordinate for a joint entry. There shall be also a third variant where both functions are present. However, suppose that the requirements for the joint entry strictly require the switch control to grant two RailCabs the permission to enter, but the blocking feature allows this under no circumstances—In this case, the product line specification is *inconsistent*.

The contribution of this paper is twofold. First, we propose a scenario-based approach for the intuitive, but precise specification of product lines. Second, we present a novel technique for efficiently consistency checking the specification of all the variants in a product. We provide an implementation and evaluate the applicability of our approach. Also, we present experiment results that evaluate the performance of implementation alternatives.

We propose to specify product lines using a combination of Modal Sequence Diagrams (MSDs) and feature diagrams [2], [3]. MSDs are a flexible variant of Live Sequence Charts (LSCs) [4], proposed by Harel and Maoz [5]. They are an intuitive, visual language for specifying sequences of message that *may*, *must*, or *must not* occur in a system.

The advantage of this approach is that it allows the requirements engineers to focus on the requirements during one particular scenario in the system at a time. This is a natural way to conceive and communicate requirements. Moreover, the behavioral aspects for each feature can be specified separately. For a particular product, the overall specification can then be composed by simply forming the union of the MSDs corresponding to the selected features.

* FNRS Research Fellow

[1]Neue Bahntechnik Paderborn/RailCab, http://www-nbp.upb.de

Thus, no elaborate feature composition mechanisms are required.

In a scenario-based approach, however, contradictions among the different scenarios may be easily introduced. To address this problem, we present a technique for the efficient consistency checking of the specifications for all the variants in a product line. Inspired by an earlier approach by Harel et al. [6], we formulate the consistency checking problem as a model-checking problem. The novelties that we introduce are that we consider (1) which feature an MSD belongs to, and (2) which valid feature combinations are implied by the feature diagram. Then, we employ a recently developed model-checking technique for product lines [7], [8]. If the specification is inconsistent, a counter-example is generated that helps the engineer understand the inconsistency among the features and MSDs.

If the MSD specification is consistent, our technique can even help the requirements engineer in refining the specification so that a state-based implementation for the components can be derived for every product. This, however, remains an outlook of this paper.

Our approach only supports specifications of *static* systems, which consist of a fixed set of components. However, it can also be of use when developing *dynamic* systems, like the RailCab system, where objects may be added or removed, or change their communication relationships. Then our approach can be used to analyze static situations that can occur, for example two RailCabs that approach a switch.

This paper is structured as follows. We introduce the foundations in Sect. II and present our scenario-based product line specification approach in Sect. III. We then explain the consistency checking technique in Sect. IV. In Sect. V, we present an evaluation of our approach and discuss related work in Sect. VI. Last, we conclude and present an outlook in Sect. VII.

II. Foundations

Our approach relies on feature diagrams and MSDs as well as a behavioral modeling and model-checking technique for product lines. This section briefly recalls these concepts.

A. Representing Variability with Feature Diagrams

A popular approach to describe commonality and variability in product lines is to use *feature diagrams*. A feature diagram is essentially a hierarchical decomposition of features. Nodes in the diagram are features and edges specify how features decompose into child features. A parent-child relationship can have different types, which constrain the valid combinations of features that can make up a product. The usual decomposition types are *AND*, *OR*, and *XOR*. An *AND* decomposition means that when the parent feature is present in the product, so must be all the children, except those explicitly labelled as optional[2]. An *OR* (resp. *XOR*) relationship implies that at least (resp. exactly) one child feature must be present when the parent feature is. There also exist cross-tree relationships among the features: the presence of one feature may *require* or *exclude* the presence of another feature. Additionally, we can define arbitrary Boolean constraints over the set of features. We stick to the formal semantics extensively defined in [3].

[2]Optional features are not used in the example appearing in this paper

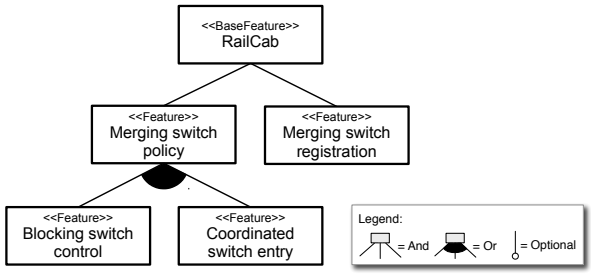

Figure 1. The feature diagram for the RailCab example

Figure 1 shows the feature diagram of our RailCab example. The root feature RailCab is always mandatory. It has two compulsory child features, namely Merging switch policy and Merging switch registration. The former has two additional child features with an OR relationship. Altogether, the diagram thus defines three product variants.

B. Scenario-Based Modeling with MSDs

MSDs were proposed by Harel and Maoz as a formal interpretation of UML sequence diagrams, based on the concepts of LSCs [5]. In the following, we explain the basics of MSDs using our running example, and detail the interpretation of MSDs that we consider in our approach.

An MSD specification consists of a set of MSDs. An MSD can be *existential* or *universal*. Existential diagrams specify sequences of events that must be possible to occur in the system. Universal diagrams specify requirements that must be satisfied by all sequences of events that occur. We focus on universal MSDs, but we explain in the outlook how our approach can also be extended to support existential MSDs.

Each lifeline in an MSD represents an object in an *object system* that consists of *environment objects* and *system objects*. The set of system objects is called the *system*; the set of environment objects is called the *environment*.

The objects can interchange messages. In this paper, we consider only *synchronous* messages where the sending and receiving of the message is a single event. Our approach can, however, be easily extended to support asynchronous communication. We call the sending and receiving of a message a *message event* or simply *event*.

The messages in a universal MSD can have a *temperature* and an *execution kind*. The temperature can be either *hot* or *cold*; the execution kind can be either *monitored* or *executed*. These attributes encode safety and liveness requirements

for events at some point during a scenario. Intuitively, a monitored message says that something may be observed whereas an executed message says that something must eventually happen (liveness). A cold message says that also something else may happen whereas a hot message says that only this event and no event that we expect at another point in the scenario must occur (safety). This interpretation is more versatile than the original definition [5] where the temperature alone reflects both the safety and liveness aspect.

More precisely, the semantics of these messages is as follows: An event can be *unified* with a message in an MSD iff the event name equals the message name and the sending and the receiving objects are represented by the sending resp. receiving lifelines of the message. When an event occurs in the system that can be unified with the first message in an MSD, an *active copy* of the MSD or *active MSD* is created. (We consider that an MSD has only one first message.) As further events occur that can be unified with the subsequent messages in the diagram, the active MSD progresses. This progress is captured by the *cut*, which marks for every lifeline the locations where the messages are attached that were unified with the message events. If the cut reaches the end of an active MSD, the active copy is terminated.

If the cut is in front of a message on its sending and receiving lifeline, the message is *enabled*. If a hot message is enabled, the cut is also *hot*. Otherwise the cut is *cold*. Similarly, if an executed message is enabled, the cut is also *executed*. Otherwise the cut is *monitored*. We also call an enabled executed message an *active* message.

A *safety violation* occurs iff 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, it is called a *cold violation*. Safety violations must never happen, while cold violations may occur and result in terminating the active copy of the MSD. If the cut is executed, this means that the active MSD must progress and it is a *liveness violation* if an active MSD never terminates or progresses to a monitored cut.

Figure 2 shows an MSD. Cold messages are blue, hot messages are red; Monitored messages have a dashed arrow, executed messages have a solid arrow. For clarity, the temperature and execution kind are shown by labels (h/c,m/e). The dashed horizontal lines in the MSD RC1RequestEnterAtEndOfTrackSection also show the reachable cuts and their temperature and execution kind. Intuitively, this MSD expresses the following requirements. We consider a scenario where two RailCabs move along their current track sections and approach a merging switch (see the sketch in Fig. 2. At some point the RailCab rc1 is notified that it reaches the end of the current track section. This is modeled as the message `endOfTS` sent between the environment and the RailCab rc1. Now the RailCab rc1 must send `requestEnter` to the switch control sc, which must reply with `enterAllowed`. These two messages must be sent before the RailCab reaches a point where it is possible for the last time to safely break before entering the switch (modeled by the message `lastBrake`).

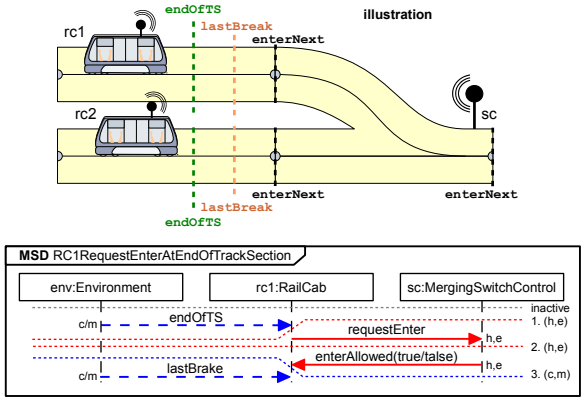

Figure 2. The MSD RC1RequestEnterAtEndOfTrackSection from the RailCab example

We assume that the system is always fast enough to send any finite number of messages before the next environment event occurs. An infinite sequence of message events is called a *run* of the system and its environment. A run *satisfies* an MSD specification consisting of a set of universal MSDs if it does not lead to a safety or liveness violation in any MSD. (Multiple MSDs may be active at the same time.) We say that an MSD specification is *consistent* or *realizable* iff it is possible for the system objects to react to every possible sequence of environment events so that the resulting run satisfies the MSD specification.

There are two interpretations for MSDs. The *invariant* interpretation allows for multiple active copies of the same MSD. This may happen if the initial sequence of messages occurs again later in the MSD. In our approach, however, we only support the *iterative* interpretation, where no second active copy is allowed, which makes a formal analysis easier.

Messages can also have parameters of certain types. A message event then carries according values for each parameter. Here we only consider messages that can have a Boolean parameter. A message in an MSD can specify either a literal value for the message, i.e., *true* or *false*, or it can specify no particular value. Then we write *true/false*. In the parametrized case, an event can be *unified* with a message in the MSD if the message in the MSD specifies no value or a value that equals the value carried by the event.

An MSD can also contain *forbidden* messages in a designated fragment labeled forbidden, appended after the actual end of the MSD. Forbidden messages have a temperature, i.e., they can be hot or cold. While there exists an active copy of an MSD, no events that can be unified with a hot forbidden message specified in this MSD are allowed to occur, otherwise this is also a *safety violation*. A message

event that can be unified with a cold forbidden message is allowed, but leads to a cold violation.

Harel and Marelly defined an executable semantics for the LSCs, called the *play-out* algorithm [9], that was later also defined for MSDs [10]. The basic principle is that if an environment event occurs and this results in one or more active MSDs with active (enabled executed) system messages, then the algorithm non-deterministically chooses to send a corresponding message if that will not lead to a safety violation in another active MSD. The algorithm will repeat sending active system messages until no active MSDs or only active MSDs with monitored cuts remain. Then the algorithm will wait for the next environment event, etc.

If the MSD specification is inconsistent, this implies that there exists a sequence of environment events that will lead the play-out algorithm to a situation where it is "stuck", i.e., there are active messages, but they would all lead to safety violations. Such a situation can, however, also occur if the specification is consistent. That is because the play-out algorithm will often make non-deterministic choices without "looking ahead" if they guarantee it not to get stuck later.

We call the possible executions of the play-out algorithm also the *play-out semantics* of an MSD specification. The valid executions of the play-out algorithm are usually only a subset of all the runs that satisfy an MSD specification, which we also call its *general semantics*.

C. Efficient Model-Checking of Product Lines

Our checking procedure is founded on *Featured Transition System* (FTS), a formalism recently introduced by Classen *et al.* for modeling the behavior of product lines [7], [8]. In a nutshell, an FTS is a usual transition system where transitions are annotated with constraints over a set of features from an attached feature model. A product can execute a transition iff its set of features satisfies the associated constraints. Supported by efficient algorithms [7], [8], [11], FTS is a promising approach for verifying product lines.

A key advantage of FTS is to include an explicit notion of feature. FTS model-checking thus allows us to identify *all* the product variants that do not satisfy an intended property. In our context, we can pinpoint exactly the combinations of features for which the combination of MSDs is inconsistent.

As a fundamental formalism, FTS can be hardly used by engineers. It is often preferable to use a high-level language on top of it. Classen *et al.* [8] recently extended NuSMV, an industry-strength model-checker, with efficient FTS-based algorithms. In SMV, NuSMV's input language, one declares variables over finite domains, and describes how the value of each individual variable evolves. Thereby, a transition relation is defined according to the synchronous evolution of all the variables. SMV also provides a construct to restrict the set of authorized transitions. In the rest of this paper, we use SMV as extended in [8].

III. Scenario-Based Specification of Product Lines

In the following, we describe our approach for the scenario-based specification of product lines by a combination of feature diagrams and MSDs. The advantage of our approach is that it allows the requirements engineer to precisely specify the feature-specific behavioral aspects of the system separately for each feature. We call the specifications created for each feature *feature specifications*. Moreover, the MSDs allow for a seamless composition of a complete specification for a particular product variant, without requiring an additional composition or "weaving" mechanism. Also (as explained in Sect. V in more detail), the presented modeling concepts can be entirely realized based on UML and lightweight extensions, so existing modeling tools can be reused for our specification approach.

First, we propose that the features and their valid combinations are modeled as feature diagrams. See Fig. 1, which showed the feature diagram of our RailCab example.

A feature can be associated with a feature specification, which consists of an optional informal description of the requirements and a package that contains the formal specification. The structure of such a package is shown in Fig. 3. The package contains classes and a collaboration. The nodes in the collaboration diagram, called *roles*, describe the objects that are considered in the specification of the particular feature. Here, the roles that represent system objects have a rectangular shape; a role that represents an environment object has a cloud-like shape. Connectors between the roles show which objects interchange messages with each other. The roles are typed over the classes in the package and operations of these classes describe which messages with which parameters an instance can receive.

Each collaboration can contain one or multiple MSDs. Here, the collaboration contains four MSDs that thus make up the behavioral specification of the feature Merging switch registration. The MSD RC1RequestEnterAtEndOfTrackSection was already introduced in Sect.II-B. In this feature, due to the symmetry in the example, the same behavior is again also specified for the second RailCab in the MSD RC2RequestEnterAtEndOfTrackSection. We hide this diagram behind the first here, because it only differs in the lifeline that refers to the RailCab rc2 instead of rc1. In the future, we could also imagine richer constructs that allow us to avoid drawing such redundant diagrams, but this shall not be the focus of this paper.

The two MSDs at the bottom of Fig. 3 say that the RailCab rc1 resp. rc2 must register at the switch control after it is granted the permission to enter the switch and before it effectively enters the switch. Then it must unregister from the switch control after entering the next, subsequent track section. Here again, only the MSD for the first RailCab is shown in the foreground.

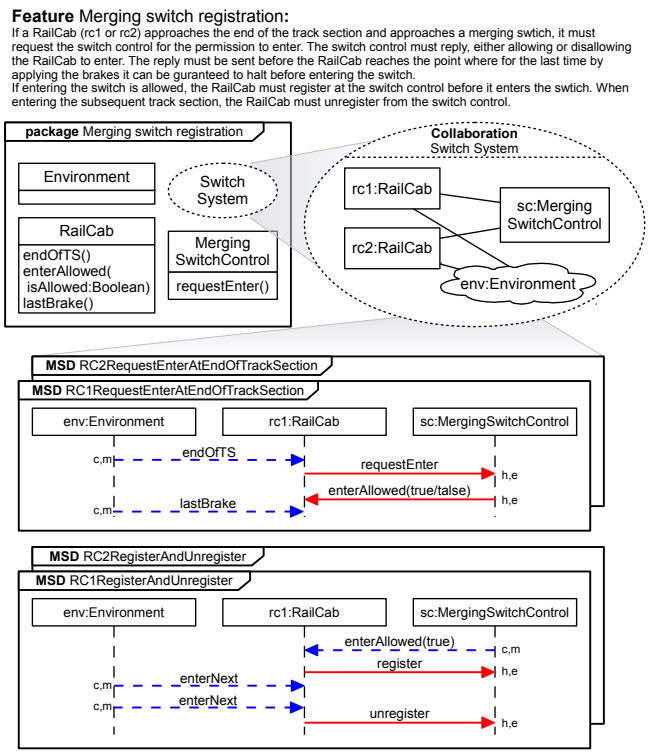

Figure 3. The specification of the feature Merging switch registration

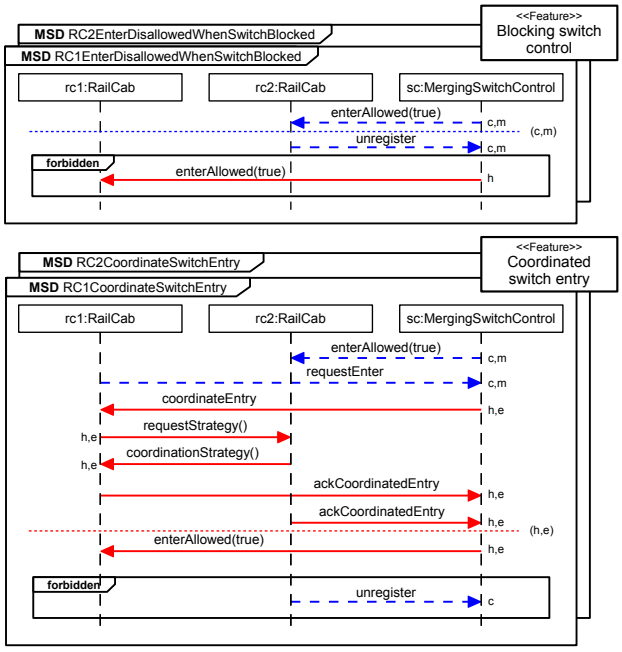

Figure 4. The MSDs for the features Blocking switch control and Coordinated switch entry and the cuts where a safety violation is inevitable

The specifications associated with the features Blocking switch control and Coordinated switch entry follow the same structure, so, for brevity, Fig. 4 only shows the MSDs associated with the features. The behavior of the feature Blocking switch control is specified by the MSDs RC1EnterDisallowedWhenSwitchBlocked and RC2EnterDisallowedWhenSwitchBlocked. They specify that RailCab rc1 resp. rc2 must not be allowed to enter after rc2 resp. rc1 was give the permission to enter the track section and before rc2 resp. rc1 has again unregistered from the switch, i.e., has left the switch.

Finally, the MSDs RC1CoordinateSwitchEntry and RC2CoordinateSwitchEntry specify that if the RailCab rc1 resp. rc2 requests the permission to enter the track section after the opposite RailCab (rc2 resp. rc1) was already given the permission to enter, the switch control can order the RailCabs to coordinate for a joint entry on the switch. To do that, the RailCab requesting the permission to enter must ask the other RailCab for a coordination strategy, in which it prescribes the time and speed at which they can safely pass the switch together. We abstract from the details of such a strategy and, for simplicity, we also do not consider that in a RailCab may also deny a proposed strategy. We assume that they must both acknowledge to the switch control to perform a coordinated entry, and that the switch control must then allow the requesting RailCab to enter.

In this example, there is the following inconsistency. Suppose that the RailCab rc2 was given the permission to enter the track section and has not yet left the track section and unregistered from it. This means that there is an active copy of the MSD RC1EnterDisallowedWhenSwitchBlocked with the cut as illustrated in Fig. 4. At the same time, if the RailCab rc1 requests the permission to enter the switch, this will eventually lead to a situation where in an active copy of RC1CoordinateSwitchEntry the hot and executed message `enterAllowed(true)` is enabled, as also show in Fig. 4. Because the message `enterAllowed(true)` is forbidden in the first MSD, it would be a safety violation to send this message. But not sending this message at all would constitute a liveness violation, because the message is executed in the second MSD. The system could also try to delay sending this message until the second RailCab unregisters from the switch control. Then, however, the environment could meanwhile send `lastBreak`, which would violate the MSD RC1RequestEnterAtEndOfTrackSection, because it would also not have progressed beyond the hot `enterAllowed` message.

The inconsistency could be resolved for example by changing the feature diagram and turning the OR relationship between the child features of Merging switch policy into an XOR relationship. This would then exclude the contradicting combination of features. Also the MSDs could be changed. Changing the hot message `enterAllowed` in RC1CoordinateSwitchEntry to a cold message would for example resolve the problem, but then a coordinated entry would never be allowed. Alternatively, we could add a cold forbidden message to the MSD RC1EnterDisallowed-

WhenSwitchBlocked so that a cold violation terminates the diagram if the switch control asks for a coordinated entry of a RailCab.

The modeling technique presented here allows us to employ a simple mechanism for composing the specification of a product from its single feature specifications. Since every feature specification is a package, we can simply employ the *package merge* mechanism defined in UML2 [12, Sect. 7.3.41] to merge the packages and their contents into a *consolidated product specification package*. Essentially, package merge merges the contents of one or multiple packages into another package. In this process, elements with the same name in the merged packages are mapped to one element with that name in the merging package. This applies to classifiers, such as classes and collaborations, but also operations. The MSDs can simply be composed by forming the union of all feature's MSDs in the consolidated product specification package.

IV. Consistency Checking Scenario-Based Product Line Specifications

We propose an automated method for discovering inconsistencies in MSD product line specifications as described above. For this purpose, we map the specification to an SMV model that encodes the play-out behavior of every product. I.e., for each product, the model describes all the possible reactions of the play-out algorithm to every possible sequence of environment events. We then verify that, in every product variant, the system can always find an admissible sequence of reactions to an environment event.

A. From MSDs to SMV

Our transformation from MSD product line specifications to an SMV models is inspired by Harel *et al.*, who propose a similar translation for LSCs [13]. SMV provides very flexible means for encoding transition relations, which we exploit in our mapping to encode the simultaneous progress of cuts and the activation and termination of MSDs. In the following, we describe the principles of this mapping.

For *each* lifeline in *each* MSD, we define a variable to represent all the reachable cuts of the MSDs. Also we define a variable `event`, which records which message has been sent at each particular step. We restrict the next value of the `event` variable with respect to the current cut: If the system is inactive, then the environment sends a non-deterministically chosen message. While the system is active, i.e., an executed message is enabled, an active message is sent non-deterministically as long as this would not lead to a safety violation in another active MSD. Depending on the message that is sent in a particular step, we progress the lifeline variables. The lifeline variables are also reset when an active MSD terminates or a cold violation occurs. In the special case that the cold violation occurs due to an event unifiable with the first message of an MSD, we accordingly progress the cut beyond the first message.

If a state is reached where there are enabled executed messages, but all corresponding events would lead to a safety violation, the system will not progress. On the contrary, an environment event that leads to a safety violation may occur. Then we set the variable $safetyViolation$ to true.

B. Relating Scenarios with Features

In order to relate a given MSD with the feature that defines it, we also include the notion of feature into the SMV model. As explained by Classen *et al.* [8], the NuSMV extension allows to model features as normal Boolean variables. A product is then modelled by a valuation of these variables. Intuitively, we want to express that the scenario of a given MSD is considered iff the value of its associated feature is true. To do so, we forbid the lifeline variables of an MSD to increase if the corresponding feature value is false.

In the work of Classen *et al.* [8], it was actually not considered that based on a feature diagram only certain combinations of features are valid. Here we only consider such valid combinations and achieve this by deriving Boolean formulae from the feature diagram, which we then include in the SMV model. The translation of these constraints is actually trivial and omitted here.

C. Verification of Consistency

Once all the MSDs have been translated into SMV code and related with their feature, we obtain a formal model describing the play-out behaviour of every product. Still, we have to prove that the specification of each product is consistent. For this purpose, we check the SMV model with the NuSMV extension [8]. This extension allows us to determine the exact set of products (as opposed to only one) that does not satisfy a certain property, expressed *Computation Tree Logic* (CTL) [14]. Intuitively, this logic allows to reason about execution paths and provides ways to specify existential and universal requirements over these paths. For example, the formula $\forall\Box\neg safetyViolation$ means "**For all** paths, the Boolean proposition $safetyViolation$ is **always** false." and the formula $\exists\Diamond\neg system\_is\_active$ expresses that "There **exists** a path where the Boolean proposition *system_is_active* is **eventually** false."

In order to specify that the system is able to react properly each time an environment event occurs, we consider the CTL formula P_1 defined as

$$\forall\Box(envMessage \rightarrow \exists\Box \\ (\neg safetyViolation \wedge \exists\Diamond(\neg system\_is\_active))).$$

Intuitively, this formula says that it must always be the case that if an environment message occurs, the system can find a sequence of system messages that it can send so it eventually reaches a state where there is no active message event and no safety violation occurs (caused by a subsequent environment

event). If the specification of all products satisfy the formula, the model-checker returns `True`. However, if there is at least one combination of features that does not satisfy the formula, the tool returns a Boolean expression defining the set of inconsistent products. In the case of our example, the model-checker returns the formula $\neg(BlockingSwitchControl \wedge CoordinatedSwitchEntry)$, which means that the products with both the features $BlockingSwitchControl$ and $CoordinatedSwitchEntry$ are inconsistent. Together with the formula, the model-checker returns one or several execution traces from which we can understand which sequence of messages leads to a state where a safety violation occurs or the system cannot progress.

D. Discussion

Our approach has a number restrictions. First of all, the SMV model encodes the play-out semantics of the MSD specification. This means that we consider that a system can only send messages that correspond to executed message currently enabled in an active MSD—the system cannot decide to send other messages or not to execute an enabled executed message. This restriction is necessary for our approach to remain feasible. Furthermore, it is a meaningful restriction, because we only consider the execution of messages that are explicitly marked to be executed. Anything else may even be regarded as an unintended behavior by the requirements engineer. However, it makes the approach incomplete, i.e., there may be a system that behaves differently than the play-out algorithm, but implements the specification.

Furthermore, the property P_1 checks that from a state where an environment message occurred, the system can find a valid sequence of system messages in reaction to that event. But if there does not exists such a reaction, it will not consider if it could have avoided that state by choosing another order among steps in a previous sequence of system messages that it sent in reaction to some earlier environment event. This requirement cannot be expressed in CTL, and motivates the need for extending our approach to parity games [15], [16].

Harel *et al.* show that, if it can be ensured that every execution of play-out avoids safety violation or getting stuck, then statecharts can be transformed from the scenarios for every object in the system [6]. With our approach, we can also prove an according property P_2:

$$\forall\Box(\neg safetyViolation \wedge \forall\Diamond(\neg system\_is\_active)).$$

Note that the previous formula is weaker than this one. When it is satisfied, it means that we can refine the specification so that the other formula is satisfied as well. Inspired by the synthesis method of Harel *et al.*, we can thus provide RE engineers with automated techniques for deriving implementations for product lines. The derivation of state-charts is, however, out of the scope of the current paper.

V. Realization and Evaluation

In this section, we present a tool suite that realizes the presented methodology. We evaluate the applicability of the tool through the RailCab case study. Also, we assess the efficiency of the verification algorithm against the successive verification of the individual products.

A. Implementation

We have implemented our approach through a number of extensions to the Eclipse workbench. MSDs are modeled via a lightweight extension, i.e., profile, of UML. They can be edited via a graphical editor that was implemented as an extension to the TOPCASED UML editor within the ScenarioTools project[3]. We similarly model feature diagrams as a lightweight extension of UML, inspired by [17]. Features are represented by components. A feature component can have a port from where it can reference child feature components by dependencies. The port acts as a group for the child feature dependencies and a particular stereotype allows us to specify whether the child features are in an AND, OR, or XOR relationship.

The input language for NuSMV is a textual language for which we slightly modified an existing XText editor[4]. We could then implement the mapping from the MSDs + feature diagram UML model to SMV via a model-to-model transformation. We specified the mapping by a Triple Graph Grammar, a declarative, rule-based formalism for specifying relations between models. This TGG can be executed by the TGG Interpreter[5].

The transformation result is then fed to NuSMV. The result of model-checking for all valid products is reported as the output. In case there is an inconsistency in any product, a counterexample is generated.

Instructions to download and install our implementation can be found on our website [6].

B. Applicability Evaluation

As a first evaluation, we applied our methodology to model and verify the consistency of the RailCab example. As presented in Section III, the product having all the features is inconsistent, because the MSDs `RC1EnterDisallowedWhenSwitchBlocked` and `RC1CoordinateEntrySwitch` are incompatible.

We modeled this example in our tool, produced the corresponding SMV model, and subsequently fed the model into the NuSMV extension. When we verified the model against the formula P_1, NuSMV identified the inconsistent product and the execution trace illustrating the violation.

[3] http://www.cs.upb.de/index.php?id=scenariotools

[4] http://code.google.com/a/eclipselabs.org/p/nusmv-tools/

[5] http://www.cs.uni-paderborn.de/fachgebiete/fachgebiet-softwaretechnik/forschung/projekte/tgg-interpreter.html

[6] info.fundp.ac.be/fts/implementations/msd2smv

According to this trace, we must avoid the following sequence of events : $env \xrightarrow{endOfTS} rc2$, $rc2 \xrightarrow{requestEnter} sc$, $sc \xrightarrow{enterAllowed} rc2$, $rc2 \xrightarrow{register} sc$, $env \xrightarrow{endOfTS} rc1$. It means that if $rc1$ reaches the track section while $rc2$ is between it and the switch, it is impossible for the system to ensure the satisfaction of all the specifications regardless of the (uncontrolled) environment events.

We decided to change the feature model in order to forbid this combination of features. Once done, NuSMV does not notice a violation of formula P_1 anymore, neither of the formula P_2. Hence, the play-out of the MSD specification for any remaining product will never run into any violation.

C. Performance Evaluation

Although consistency checking our running example takes less than a second, the consistency checking can be very time-consuming. The selection of a specific algorithm and its optimization is thus of utmost importance. As stated by Classen *et al.* [7], [8], there are two methods for model checking an FTS. The first one consists in deriving, from the FTS, the models corresponding to all the valid products and then verifying them individually. The second method relies on the dedicated algorithms proposed by Classen *et al.* [7], [8].

In this evaluation, we compare the performance of both methods through several experiments. More precisely, we compare the time needed by both algorithms to verify the SMV models against formulae P_1 and P_2. To obtain the SMV model related to a particular product, we first remove the declaration of the feature variables in the original model. Then, we replace every feature variable by $true$ if the feature belongs to the considered product, and by $false$ otherwise. In the subsequent evaluations, we do not count the time needed for computing these individual models.

To carry out our experiments, we employ a set of systematically extended example specifications. Each specification in this set has a feature diagram where each feature has at most two child features, connected with an OR-relationship, and there exists only features with distance i from the root if all features with distance $i-2$ already have two child features. Each feature is connected with exactly one MSD. An illustration of the example specification with three features and three MSDs is given in Figure 5. According to the feature diagram, three different products can be derived: {F, F-1}, {F, F-2}, and {F, -1, F-2}. The first message of an MSD 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 an MSD of a child feature is named like the two hot messages of its parent's MSD. This way, one activation of an MSD triggers two activations of an MSD for each child feature.

We extend such a specification by adding two child features and two MSDs. The name of the first and second child feature is formed by appending "-1" resp. "-2" to the name of its parent. The names of the two hot message of the child feature's MSD are extended likewise.

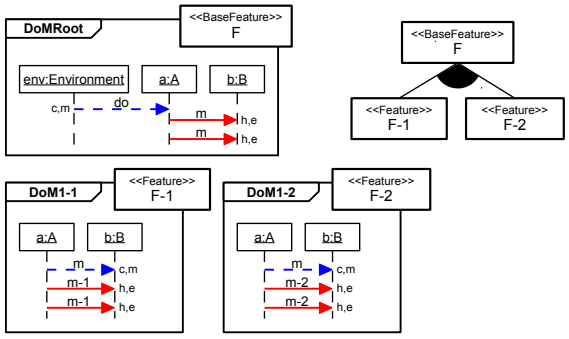

Figure 5. Technical evaluation example with three features

Following this scheme, we created product line specifications for 3, 5, 7, 9, 11, 13, and 15 features. They specify 3, 7, 15, 31, 63, 127, and 255 products. Note that for each additional feature, there is an exponential increase in the number of products and the number of different reachable combinations of cuts.

Since NuSMV is a fully symbolic model-checker, the ordering of the variables is an important factor for the performance of the algorithms. NuSMV proposes two modes to control variables ordering. The first method allows one to specify which ordering NuSMV must use throughout the verification. In the second mode, called `dynamic`, the model checker automatically reorders the variables during the execution. While this is usually more efficient than random orderings, it creates a significant overhead that consequently increases the verification time with respect to optimal orderings. During the experiments we carried out, we managed to find out orderings that systematically perform better than the dynamic mode, and to figure out a pattern that, in our examples, ensures an improvement of the efficiency. According to this pattern, a good ordering must satisfy the following two rules. First, the lifeline variables of a given MSDs must be grouped together, and preceded by the feature variables they are associated with. Second, if an MSD activates another then the lifeline variables of the former must be placed after the lifeline variables of the latter. Note that this pattern is also efficient in the SMV model of a particular product. In this case, the feature variables are ignored, since they do not occur in a single-product model. Of course, we could evaluate the efficiency of these orderings through only two examples, and are aware that this is far from sufficient to prove that they are optimal.

All benchmarks were run on a MacBook Pro with a 2.4 GHz Core 2 Duo processor and 4 Gb of RAM. During the experiments, no other application was running so that processor sharing could not influence the verification time. The results are shown in Table I. It provides the verification time for each property, algorithm, and ordering method. First

Table I
VERIFICATION TIMES FOR THE CASCADING EXAMPLE.

	P_1				P_2			
	Ded.		**Enu.**		**Ded.**		**Enu.**	
# Feat.	dyn.	our.	dyn.	our.	dyn.	our.	dyn.	our.
3	0.05	0.01	0.08	0.03	0.05	0.01	0.08	0.03
5	0.21	0.04	0.62	0.11	0.21	0.04	0.64	0.13
7	0.93	0.22	2.64	0.54	1.65	0.32	3.01	0.64
9	6.52	1.75	10.47	3.64	7.16	2.19	11.88	5.06
11	53.88	24.16	131.48	32.53	56.77	45.46	175.12	54.12
13	1631.57	192.68	989.14	216.24	2622.94	330.94	1263.56	431.00
15	7798.22	1510.21	2314.10	1041.50	5364.32	2430.15	2903.26	2700.90

of all, let us note that the ordering pattern we have identified (`our.`) achieves order-of-magnitude improvements over the dynamic ordering. Moreover, this latter ordering becomes particularly inefficient when the number of features reaches 13, especially when it is combined with the dedicated algorithm (**Ded.**). When using our ordering pattern, it turns out that the dedicated algorithm outperforms the enumerative method (**Enum.**) except when P_1 and the 15-feature case are considered. Our theory is that the enumerative approach is more efficient for bigger specifications with a high feature-to-MSD ratio. In practice, however, it is more likely to always have several MSDs per feature and not one, as in this technical example. Thus we expect our approach to be more efficient in most practical cases, but it would require further evaluations to provide evidence for this theory.

Finally, note that the enumerative algorithm does not return a concise formula that identifies the bad products, so the dedicated method also yields improvement in usability.

VI. RELATED WORK

There are many approaches for synthesizing and consistency checking formal scenario specification [6], [15], [16], [18], [19]. Also the relationship between scenarios and goals was studied in the past [20]. However, there are few approaches that consider the formal scenario-based specification of product lines.

Ziadi et al. consider the synthesis from statecharts from sequence diagrams where interaction fragments can be annotated to be active only in certain variants [21]. However, their approach does not support safety and liveness properties as flexibly as ours. They require a high-level diagram that defines a control structure among basic sequence diagrams, and so also contradictions cannot occur in basic scenarios.

Ghezzi and Molzam propose an approach to verify non-functional requirements of software product-lines [22]. They model the system's behavior with sequence diagrams where fragments can be annotated to be active only in certain products. They, however, do not consider that multiple scenarios can be active concurrently.

The relationship between feature models and structural as well as behavioral UML models was studied for example in [23]–[25]. However, here only syntactical consistency relationships among different modeling views are considered.

Harhurin and Hartman propose and approach for modeling and consistency checking families of service-orinented systems [26]. 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 components can be specified, which is crucial for complex interaction protocols, especially if they involve more than two participants.

Lauenroth *et al.* propose an approach where the behavior of features in a product lines is modeled with automata in a FTS-like fashion, i.e. transitions are only enabled when certain features are selected [27]. Furthermore, invariants can be formulated and a SAT-solver is employed to find inconsistent states that do not satisfy the invariants. However, like our FTS, this formalism is not intuitive and suited to be used by engineers during the early design.

VII. CONCLUSION AND OUTLOOK

In this paper, we provide a methodology for the scenario-based specification of product lines, using MSD for the specification of the individual feature behaviors. To verify the consistency of such specifications, we employ recent advances in product lines verification, namely an extension of the industry-strength model-checker NUSMV [8]. We implemented a tool that automatically transforms an MSD product line specification into an SMV model where NUSMV can then pinpoint all the inconsistent combinations of features. We evaluated the applicability of our approach and the performance of different algorithmic options. In particular, we showed that variable ordering is of utmost importance and identified a pattern that, in our experiments, always yielded good results.

In future work, we plan to also support richer MSD language features and to elaborate the automated derivation of statecharts from the specifications. These statecharts could also be checked in combination with existential MSDs

to ensure that particular scenarios are possible to occur. Moreover, we plan to investigate how the principles of FTS model-checking could be applied to synthesis (i.e., parity games), so that we can overcome the current limitations of our model-checking approach. Especially interesting would then be to investigate how to synthesize an implementation for a particular product variant incrementally, assuming an implementation for some other (base) variant already exists. Another challenge is to analyze the consistency of product line specifications for dynamic systems, which could be supported by a combination of synthesis and simulation [28].

ACKNOWLEDGMENT

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.

REFERENCES

[1] K. Pohl, G. Böckle, and F. J. van der Linden, *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005.

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

[3] P.-Y. Schobbens, P. Heymans, and J.-C. Trigaux, "Feature diagrams: A survey and a formal semantics," in *Requirements Engineering, 14th Int. Conf.*, sept. 2006, pp. 139 –148.

[4] W. Damm and D. Harel, "LSCs: Breathing life into message sequence charts," in *Formal Methods in System Design*, vol. 19. Kluwer Academic Publishers, 2001, pp. 45–80.

[5] D. Harel and S. Maoz, "Assert and negate revisited: Modal semantics for UML sequence diagrams," *Software and Systems Modeling (SoSyM)*, vol. 7, no. 2, pp. 237–252, May 2008.

[6] D. Harel, H. Kugler, and A. Pnueli, "Synthesis revisited: Generating statechart models from scenario-based requirements," in *Formal Methods in Software and Systems Modeling*, H.-J. Kreowski, U. Montanari, F. Orejas, G. Rozenberg, and G. Taentzer, Eds., vol. 3393. Springer, 2005, pp. 309–324.

[7] A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, and J.-F. Raskin, "Model checking lots of systems: efficient verification of temporal properties in software product lines," in *Proc. 32nd Int. Conf. on Software Engineering (ICSE'10)*, ser. ICSE'10. ACM, 2010, pp. 335–344.

[8] A. Classen, P. Heymans, P.-Y. Schobbens, and A. Legay, "Symbolic model checking of software product lines," in *Proc. 33rd Int. Conf. on Software Engineering (ICSE'11)*. ACM, 2011, pp. 321–330.

[9] D. Harel and R. Marelly, "Specifying and Executing Behavioral Requirements: The Play-In/Play-Out Approach," *Software and System Modeling (SoSyM)*, vol. 2, p. 2003, 2002.

[10] S. Maoz and D. Harel, "From multi-modal scenarios to code: Compiling lscs into aspectj," in *Proc. Int. Symp. on Foundations of Software Engineering (FSE'05)*, 2006, pp. 219–230.

[11] M. Cordy, A. Classen, G. Perrouin, P. Heymans, P.-Y. Schobbens, and A. Legay, "Simulation relation for software product lines," in *Proc. 34th Int. Conf. on Software Engineering (ICSE'12) (to appear)*. IEEE, 2012.

[12] "UML 2.4.1 Superstructure Specification," August 2011, OMG document `formal/2011-08-06`.

[13] D. Harel, H. Kugler, R. Marelly, and A. Pnueli, "Smart play-out of behavioral requirements," in *Proc. 4th Int. Conf. on Formal Methods in Computer-Aided Design*, ser. FMCAD '02. London, UK: Springer, 2002, pp. 378–398.

[14] E. M. Clarke and E. A. Emerson, "Design and synthesis of synchronization skeletons using branching-time temporal logic," in *Logic of Programs*, ser. LNCS, vol. 131. Springer, 1981, pp. 52–71.

[15] Y. Bontemps and P. Heymans, "From live sequence charts to state machines and back: A guided tour," *Transactions on Software Engineering*, vol. 31, no. 12, pp. 999–1014, 2005.

[16] J. Greenyer, "Scenario-based design of mechatronic systems," Ph.D. dissertation, University of Paderborn, Oct. 2011.

[17] T. Possomps, C. Dony, M. Huchard, and C. Tibermacine, "Design of a UML profile for feature diagrams and its tooling implementation," in *Proc. 23th Int. Conf. on Software Engineering and Knowledge Engineering (SEKE'11)*. Knowledge Systems Institute Graduate School, July 2011, pp. 693–698.

[18] J. Whittle and J. Schumann, "Generating statechart designs from scenarios," in *Proc. 22nd Int. Conf. on Software Engineering (ICSE'00)*, 2000, pp. 314–323.

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

[20] C. Damas, B. Lambeau, and A. van Lamsweerde, "Scenarios, goals, and state machines: a win-win partnership for model synthesis," in *Proc 14th Int. Symp. on Foundations of software engineering*, ser. SIGSOFT '06/FSE-14. New York, NY, USA: ACM, 2006, pp. 197–207.

[21] T. Ziadi, L. Hlout, and J.-M. Jzquel, "Behaviors generation from product lines requirements," in *Proc. UML2004 workshop on Software Architecture Description*, Sep. 2004.

[22] C. Ghezzi and A. Sharifloo, "Verifying non-functional properties of software product lines: Towards an efficient approach using parametric model checking," in *Proc. 15th Int. Software Product Line Conference (SPLC)*, August 2011, pp. 170 –174.

[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*, ser. LNCS, G. Engels, B. Opdyke, D. Schmidt, and F. Weil, Eds. Springer, 2007, vol. 4735, pp. 151–165.

[24] M. Vierhauser, P. Grünbacher, A. Egyed, R. Rabiser, and W. Heider, "Flexible and scalable consistency checking on product line variability models," in *Proc. Int. Conf. on Automated Software Engineering (ASE'10)*. ACM, pp. 63–72.

[25] 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. 12th Int. Conf. on Top productivity through software reuse*, ser. ICSR'11. Berlin, Heidelberg: Springer, 2011, pp. 20–35.

[26] A. Harhurin and J. Hartmann, "Towards consistent specifications of product families," in *Proc. 15th Int. Symp. on Formal Methods*, ser. FM '08. Berlin, Heidelberg: Springer, 2008, pp. 390–405.

[27] K. Lauenroth and K. Pohl, "Dynamic consistency checking of domain requirements in product line engineering," in *International Requirements Engineering, 2008. RE '08. 16th IEEE*, sept. 2008, pp. 193 –202.

[28] J. Frieben and J. Greenyer, "Consistency checking scenario-based specifications of dynamic systems," in *Proc 4th Workshop on Behavioural Modelling – Foundations and Application (BM-FA 2012) (to appear)*. ACM, 2012.