Front.Comput.Sci.0.0
DOIDOI

article

Scenario-based verification in presence of variability using a synchronous approach

Jean-Vivien Millo[1], **Frederic Mallet**[1], **Anthony Coadou**[2], **S Ramesh**[2]

1 INRIA/ I3S/ UNS, Sophia-Antipolis, France
2 Global General Motors R&D, India Science Lab, GM Technical Center India, Bangalore, India

Abstract This paper presents a new model of scenarios, dedicated to the specification and verification of system behaviours in the context of Software Product Lines. We draw our inspiration from some techniques that are mostly used in the hardware community, and we show how they could be applied to the verification of software components. We point out the benefits of synchronous languages and models to bridge the gap between both worlds.

Keywords Esterel, UML Marte , Scenario, Verification, Feature interaction, Variability

1 Introduction

The current development trends focus on developing Software Product Lines (SPL) that involve multiple products sharing some common features. Instead of putting various software components in a library for a possible later reuse, an SPL promotes a predictive and well-defined reuse of those components among a set of final products. Each feature is implemented as a separate component, then enabled according to the needs of a given product. Recent studies have shown that advances in SPL methodologies can yield substantial improvements in development cost, effort and time [44].

With the SPL approach comes the problem of *feature interaction* [16,20]. Let use consider a set of features, each feature can be parametrized according to the scope of the SPL. Even though every feature's behaviour has been checked independently, there is nothing that guarantee that the behaviour of the system running all these features will be correct. Let us consider the following example: the feature *Power lock* of a car system lock or unlock the doors when the key button is pressed. The feature $Theft\ lock$ provides a second lock to the car. $Theft\ lock$ is an optional feature and when present, it can be trigged by pressing a second time on the key button. The unlocking of the the theft lock is subject to a user preference variant. It is trigged either by a single press or a double press of the key button. In its first variant, there is no specification about the order into which the theft lock and the usual lock will be activated. However, the physical locks need to be activated in a specific order. This causes hazardous behaviour.

The purpose of this paper is to present a verification approach to tackle the problem of feature interaction in the context of SPL based development of embedded control software. Due to the strong interest of the embedded control software developer for the family of the synchronous languages [10], we take the assumption that the design language of the verified system is either Esterel [41] or a language that can be translated into Esterel with preservation of the original semantics.

Scenarios [18,32,39] describing interaction of subsystems, external actors and users are intuitive and powerful tools to express complex behaviours of systems, be it software or hardware. A designer can use them to describe either an expected behaviour or a prohibited use-case of a system. Scenarios are commonly stated at the requirements level, and used as a partial description of the expected behaviour to check whether the actual system implementation matches.

Introducing variability in the design model requires to introduce variability in the checked properties as well. For instance, LSC introduces a notion of *liveness*, that is "*the distinction between possible and necessary behaviours*" [18]. In an SPL context, "*possible*" means that there exists a configuration that leads to the expected behaviour, while "*necessary*" shall hold in any case. Some authors have gone one step further [46, 47] to extend scenario models to the SPL case, introducing variation points for describing parametrized behaviours. Our work follows this approach. We define a new model of Sequence Diagrams with Variability (SDv), where

Received month dd, yyyy; accepted month dd, yyyy

E-mail: jean-vivien.millo@inria.fr, mallet@unice.fr, and {anthony.coadou, ramesh.s}@gm.com

the variability is explicitly stated and modelled. Distinguishing signals, which carry the notion of variability, from regular signals makes a worthy information when it comes to the verification phase.

SDv model is based on scenarios that capture the expected or unexpected feature interactions. We also propose a set of transformations to generate Esterel observers from these scenarios. Such observers can be used to verify that a given (Esterel) implementation satisfies the specification described by the scenarios, *i.e.,* the implementation does not produce unexpected interactions. For expected interactions, the verification focuses on proving that there is a way in the implementation to actually produce all the expected interactions. We proceed in three steps. First, we define the *domain model*, the set of concepts that we deem necessary to deal with feature interactions in the context of software product lines and in particular when there are some variability issues. The proposed domain-specific language is called SDv. Second, to implement the proposed language and rather than creating a completely ad-hoc verification environment, we extend the interactions defined in the Unified Modeling Language (UML). UML is general purpose and starting from UML is a way to focus only on the extensions that are needed and to reduce the implementation cost. This also facilitates the integration of our verification technique in a complete design flow in which UML is used by other stakeholders to describe requirements, software components, deployments and so on. Since UML is general-purpose, it lacks some of the key constructs that we need in our language, most notably some constructs about timing information and some aspects specific to the use of synchronous semantics and of Esterel. To support these missing constructs we use a small subset of the UML profile for Modelling and Analysis of Real-Time and Embedded systems (MARTE)[1)] and mainly of its Time Model [6]. MARTE is an extension of UML dedicated to embedded systems. MARTE is well-suited to our application domain that is mainly electronic controllers embedded in automotive applications. UML plus MARTE are enough to make all the semantic distinctions that we require for SDv. Third, the semantics of SDv is given by transforming it into Esterel observers with a first occurrence semantics. This is our main contribution, *i.e.,* provide a language that can capture feature interactions, support variability and that is amenable to an automatic, formal and exhaustive analysis.

Another axis of research consists in looking for possible improvements in the way scenarios are verified. We started with two observations:

1. It appears that many state-of-the-art model-checkers for software struggle to cope with the complexity of an industrial-size project. In our case studies, symbolic model-checking of a whole system remains too time-costly, while exhaustive explorations either never terminate or reach memory bounds (see Section 6).
2. However, model-checking is a widely used and smoothly running technique in the hardware industry for verifying design properties. For instance, it was already used in production a decade ago to verify the correctness of the entire logic of a Pentium 4 processor [33].

So why such a difference? Perhaps hardware-related tools are more mature, hence resulting in an increased robustness. But another part of the answer might be that hardware models allow more "efficient" optimizations and pruning in the decision process.

As a consequence we propose to use some verification techniques that are mostly used in the hardware community, and to show how they could be applied to the verification of such software components; here, synchronous languages and models [10] can be considered as the bridge between both worlds. The representation of the system as logical circuits benefit from the massive parallelism and the various possible logical optimizations; the state space explosion can be efficiently limited by pruning unnecessary branches [38].

Section 3 presents the SDv model and its intuitive semantics. Section 2 ... In Section 5, we present the transformation rules to generate an Esterel observer from an SDv scenario. Section 6 presents our validation approach and the benefits of using hardware optimization and verification tools. Section 7 introduces an experimental use-case. We briefly discuss some possible improvements and future works in Section 8.

1.1 Related works

Our work is at the crossing of four domains: SPL, verification, scenario, and synchronous languages. We think that these four keywords forms a unique identifier of our work. However, the association of two or three of these keywords refers to existing works that we are going to detail here:

1.1.1 Scenario and SPL

Message Sequence Charts (MSC) [39], UML Sequence Diagrams [32], and Live Sequence Charts (LSC) [18] have been widely studied in the literature and recent extensions adapt existing construct to the notion of variability [12, 46, 47].

1.1.2 Verification of SPL

There is a growing interest of the SPL community for the verification approach [17, 23, 27, 30, 31]. In this scope, Greenyer et al. [22] are applying a verification approach to the feature interaction problem where a preliminary representation of the behaviour of the system is expressed using scenarios. On contrary, in our approach, the property is expressed with scenario.

Some other works intends to provide a framework to manage [20] and detect [16] feature interaction. In the later, the approach is product centric and so it is not scalable.

[1)] http://www.omgmarte.org

1.1.3 Verification of synchronous design using scenario

The proposed approach takes its roots in André's seminal works on SyncCharts [2] and Synchronous Interface Behaviours [3, 7], where two Esterel-related formalisms can be used as design model and scenario model respectively.

2 Preliminaries

2.1 A tour of Esterel

Esterel is a member of the class of synchronous reactive languages [9, 10, 24]. These languages consider an explicit division of (logical) time into discrete instants, and make the global assumptions that the amount of processing contained in a so-called reaction will complete inside such an instant. This assumption is met in hardware by establishing the proper clock speed so that all electric fronts have become stable, in general-purpose software be decreeing the end of the instant once very instructions of the current reaction have been performed (run to completion), in real-time software by guaranteeing worst-case reaction time (WCRT) bounds. The synchronous assumption in turns guarantee that all behaviours in the same reaction can be approximated to occur "in no time"(simultaneously), which allows a powerful and simple programming style where instants and scheduling schemes are in the hand of the application designer (instead of, say, system OS engineers).

Synchronous reactive systems can further be divided into declarative ones, such as Lustre/SCADE [10] and Signal/Polychrony [10], promoting a streaming data-flow style, and imperative ones, such as Esterel or the graphical SyncCharts version, with explicit control-flow mode structure as hierarchical and/OR automata (with sequence, if-then-else alternative, and parallel compositions). Other current synchronous efforts include Quartz [40] and SynchronousC [42] , and several others as well.

Constructive causality forms an issue specific to Esterel, raised from instantaneous propagation of internal signals. Consider the program fragment `present S then emit S`; it is non-deterministic, as signal `S` can be determined as present or absent in a consistent way. But one has to guess the value of `S`, then check that it leads to no contradiction, rather than propagate valid information from emissions to receptions. The fact that Esterel allows reaction to absence (`present S else P` makes things more complex, and a full-fledge theory had to be developed (with in retrospect some influence back to synchronous circuit design). As a result, many compiling schemes for the language go beyond simple transformation into low-level object code, as causality checking is included on purpose into those simulation schemes so that compiled programs are guaranteed free from causal paradoxes, which may otherwise lead to deadlocks and errors at runtime. Dedicated compilation techniques are describes in [36].

2.2 Software product line, feature, and variability constraints

Software Product Line (SPL) is a software development framework to jointly design a family of closely related software products in an efficient and cost-effective manner. The development process is sliced in two activities: domain engineering and product engineering [35]. The domain engineering activity consists in developing the unitary blocks of software (called the features) that will be assembled later to derive products; this second activity is the product engineering. The difficulties in domain engineering is to develop a feature while considering the relations and interactions with the other features. The SPL based approach aims at facilitating the product engineering activity. In many cases, the effective construction of a product (compile and link the feature source codes together) is automated from the selection of features [26].

Each feature satisfies a specific end-user requirement. In the scope of this work, we consider that a feature can have (configuration) parameters that are set once and remain constant throughout the execution. For example the feature *gear box* can be set to automatic of manual. The features are usually organized in a feature diagram (with a tree structure) that expresses all the relationships of dependency and exclusion between them. A product of the family is a subset of features that satisfies the constraints expressed in the feature diagram. The feature diagram in Figure 1 focuses on the entry control domain of an automotive product line. The features with a filled circles on the top are mandatory while the ones with an empty circle are optional. The features *Gearbox, Door lock, door unlock, Alarm, Last door closed lock,* and *Anti-lockout* have each a single parameter that can take a value from two possible alternative choices. Similarly to the features, the possible values of the parameters are subject to restrictions such as exclusion or dependencies as it is expressed with the cross tree arrows. Some details about the behaviour of the features are given throughout the paper when required.

The number of products that one can derive from a given SPL usually scales from thousands to billions or more [21] thus the existing analysis and verification techniques cannot be applied iteratively on every product of the family. The trend tends to adapt these techniques to feature-based approaches where every feature is analysed independently and then the per feature analysis results are combined to derive a conclusion on the entire SPL [17, 22, 31]. Here, an intermediate step is considered where we analyse the interactions of limited groups of features with a similar goal than the integration testing activity.

2.3 Marte and CCSL

The UML Profile for Modeling and Analysis of Real-Time and Embedded systems [34] (MARTE), adopted in November 2009, has introduced a *Time model* [6] that extends the informal *Simple Time* of The Unified Modeling Language (UML 2.x).

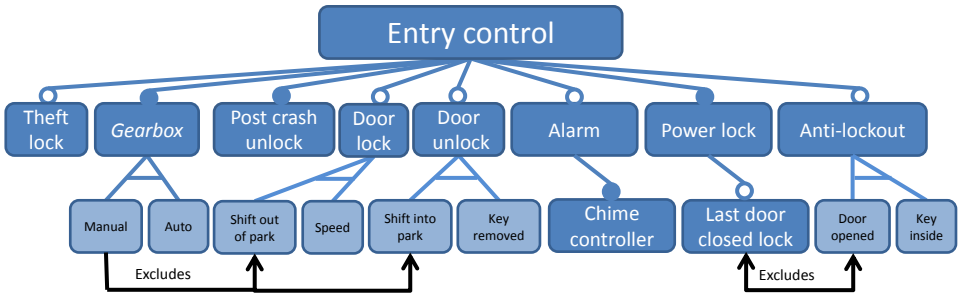


Fig. 1: An example of a feature diagram from the automotive industry.

This time model is general enough to support different forms of time (discrete or dense, chronometric or logical). Its so-called *clocks* allow enforcing as well as observing the occurrences of events and the behavior of annotated UML elements. The time model comes with a companion language called *Clock Constraint Specification Language* (CCSL) [4] and defined in an annex of the MARTE specification.

The notion of multiform logical time has first been used in the theory of Synchronous languages [10] and its polychronous extensions [28]. The use of tagged systems to capture and compare models of computations was advocated by [29]. CCSL provides a concrete syntax to make the polychronous clocks first-class citizens of UML-like models.

A *clock* c is a totally ordered set of *instants*, $\mathcal{I}_c$. In the following, i and j are instants. A *time structure* is a set of clocks C and a set of relations on instants $\mathcal{I} = \bigcup_{c \in C} \mathcal{I}_c$.
CCSL considers two kinds of relations: *causal* and *temporal* ones. The basic causal relation is causality/*dependency*, a binary relation on $\mathcal{I}$: $\preccurlyeq \subset \mathcal{I} \times \mathcal{I}$. $i \preccurlyeq j$ means i causes j or j depends on i. $\preccurlyeq$ is a pre-order on $\mathcal{I}$, *i.e.,* it is reflexive and transitive. The basic temporal relations are *precedence* ($\prec$), *coincidence* ($\equiv$), and *exclusion* ($\#$), three binary relations on $\mathcal{I}$. For any pair of instants $(i, j) \in \mathcal{I} \times \mathcal{I}$ in a time structure, $i \prec j$ means that the only acceptable execution traces are those where i occurs strictly before j (i precedes j). $\prec$ is transitive and asymmetric (reflexive and antisymmetric). $i \equiv j$ imposes instants i and j to be coincident, *i.e.,* they must occur at the same execution step, both of them or none of them. $\equiv$ is an equivalence relation, *i.e.,* it is reflexive, symmetric and transitive. $i \# j$ forbids the coincidence of the two instants, *i.e.,* they cannot occur at the same execution step. $\#$ is irreflexive and symmetric. A consistency rule is enforced between causal and temporal relations. $i \preccurlyeq j$ can be refined either as $i \prec j$ or $i \equiv j$, but j can never precede i.

For SDv, we use the clocks as triggers. We also use the basic CCSL relations (within UML constraints) to tell apart the different relations that we have defined, *i.e.,* Synchronization ($\equiv$), Dependency ($\preccurlyeq$), StrictDependency ($\prec$).

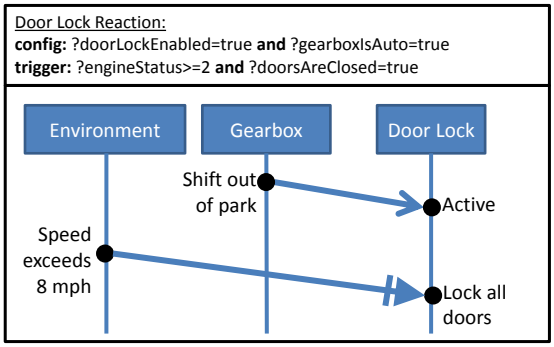


Fig. 2: Basic example of an SDv chart.

3 Sequence Diagram with Variability

SDv is a two-level model of scenarios where the variability is considered as a first class citizen. At low-level, a *chart* depicts the interactions between the features of the system; they are represented as *actors*. At a higher-level, a *graph* (similar to MSG [39]) specifies how to combine several charts. In this section, we explain the main constructs and illustrate them with a few examples.

3.1 SDv chart

3.1.1 Configuration and trigger

A simple example of SDv is given in Figure 2. The chart is called *Door Lock Reaction*. It is applicable for all configurations such that *door lock* feature is enabled and the Gearbox is configured to automatic. Notice that such a configuration expression can only test signals propagating the values of the parameters (including the presence or absence of features) related to the variability of the system; these parameters have constant valuation during the execution and so have the associated signals. If this condition is not matched, then the scenario is vacuously true (this case will be considered differently from the completion of the scenario).

When the configuration holds, the scenario is not necessarily immediately activated since one might want to wait for

the system to be in a specific state. Hence the scenario starts as soon as its trigger turns true. In the case of Figure 2, the engine has to be running and all the doors should be closed. On contrary to configuration, the trigger is testing the value of signals relating the state of the system that evolve during the execution. It is not allow to use neither configuration parameters nor observed signals in the trigger expression.

One should notify the specific syntax of the configuration and the trigger expressions that matches the implementation language (here Esterel). The signals used in these expressions refers to existing signals of the implementation.

Such configuration expressions are used in many other constructs of SDv to specify variant behaviours such as the event, the agent, the hot and cold conditions (see Figure 5) or the branching structure of the graph (see Figure 7).

3.1.2 Event and precedence

The main component of a chart is the description involving interactions among a set of actors. For instance, the chart in Figure 2 involves three actors: Environment, Gearbox, and Door Lock. Gearbox and Door lock are features of the system (Figure 1). Each actor has its own time-line, which enforces a sequential order on all the *events*, indicated by the dots on each time-line. An event is a signal emitted by an actor. SDv , besides event occurrences, specifies an order between pairs of events from different time-lines, indicated by arrows: the source of an arrow precedes the target.

In SDv , an event corresponds to the emission of a signal by an actor. In contrast, MSC and LSC both use *message passing* primitives between actors instead of events. Wired communications in automotive systems actually use broadcasting of signals over a given network, as defined in the synchronous paradigm; hence we chose to refine our model to match this approach.

The example in Figure 2 describes a behaviour of an automatic lock system. In such a system, the Door Lock feature is activated once the shift-out-of-park event occurs, indicated in the chart by the causal arrow between these two events. Similarly, all the doors are locked after the speed exceeds 8 mph. The latter is a strict precedence relation (indicated by the crossed out arrow) and the locking event happens *at least* one cycle *after* meeting the threshold (no simultaneity).

3.1.3 Repetition

An annotation of the form $[m..M]$ adjacent to an event specifies that the event shall be repeated at least m times and at most M times, with $0 \leq m \leq M$. M is finite. The default annotation is $[1:1]$. When repeated events are involved in precedence relations, the last occurrence of the source event has to precede the first occurrence of the target event.

Figure 3 describes the behaviour of an alarm controller. The described behaviour holds for all configuration (**config:***[always]*) and trigs immediately (**trigger:***[always]*).

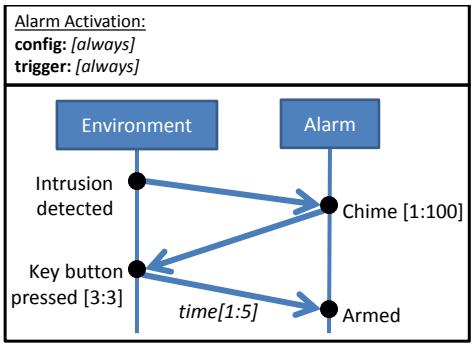


Fig. 3: Example of an SDv chart with repetitive events and timed precedence.

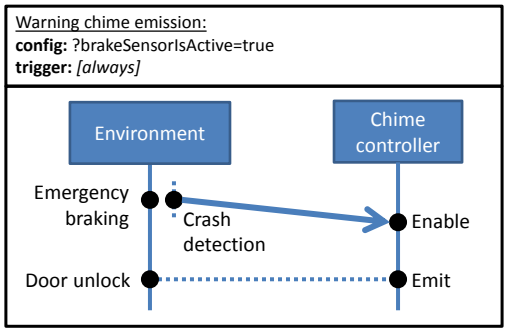


Fig. 4: Example of an SDv chart with parallel and synchronous events.

The alarm emits chime events at least once and at most hundred times when an intrusion attempt is detected, until the car key is pressed exactly three times. The alarm controller is then armed.

3.1.4 Timed precedence

A precedence between two events can also be labelled by a relative time constraint: the consequence has to occur after the cause, between a minimal and a maximal delay. That is the case, in Figure 3, where the arming of the alarm is supposed to occur from one to five logical instants after the key button is pressed.

3.1.5 Parallel events

The order from top to bottom on a time-line guarantees a strict sequential ordering. However, a vertical dotted segment close to a time-line can be used to specify that some events occur concurrently, as shown in Figure 4: the emergency breaking and the crash detection must happen before the door unlocking (and after any other event earlier on the time-line) but, relatively to each other, their order remains unspecified.

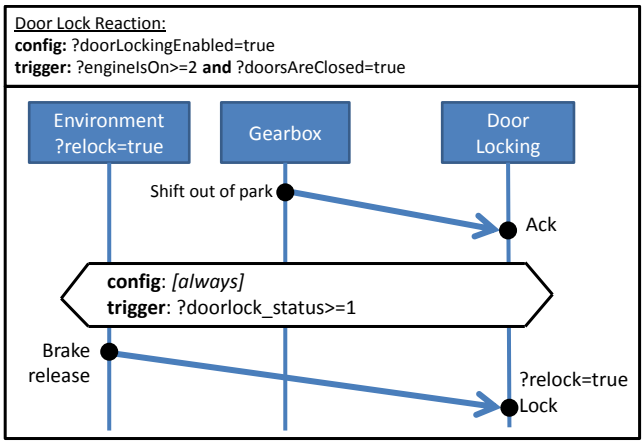


Fig. 5: Another example of an SDv chart where variability is introduced in the chart body.

3.1.6 synchronization

An horizontal dotted line linking two events ensures the synchrony of these events. In the example of Figure 4, the door unlocking and the chime emission must happen exactly at the same instant. If repeated events are synchronized, every instance should be synchronized until the maximum number of occurrence has occurred for one of them. Then, the other can occur freely. The (logical) timed precedence and the synchronization relations are specific elements of SDv .

3.1.7 Multi-line condition and variability in the chart body

A hexagonal box, intersecting all the time-lines, depicts a *condition* which needs to hold for checking the rest of the scenario. A multi-line condition can be *hot* and so is represented with a plain hexagon or *cold* and so is represented with a dotted line hexagon. It also acts as a synchronization point over all the actors, *i.e.* whatever is drawn above (resp. below) a multi-line condition box on a time-line has to occur before (resp. after) it. Like the charts, the multi-line condition uses a configuration and a trigger condition describes the enabling condition for the rest of scenario. When the configuration is evaluated to false, the multi-line condition is ignored. Otherwise, the trigger is evaluated once all actors are synchronized. When the trigger is evaluated to true, the scenario is continued. When the trigger is evaluated to false and the multi-line condition is hot (resp. cold), the chart terminates with an error (reps. a success).

An example is shown in Figure 5. In this example, the second half of the scenario below the multi-line condition box applies to all configurations. At the step after the signal Ack has occurred, the multi-line condition tests whether the status of the feature door lock is active. If it could not be met, then the scenario could have terminated with an error at this point otherwise, the Brake release signal is expected.

An it is shown in Figure 5, a configuration statement can be attached to an actor or an event. The actor or the event are considered in the chart only when the configuration of the system is such that the statement holds.

If an event is disabled, it is not expected to occur and its relations (precedence and synchronization) are ignored but each event at the other end of these relations are still expected to occur. If an actor is disabled, all events on its time-line and their relations are ignored. The multi-line condition remains. In Figure 5, when *relock* is false, neither the break release event nor the lock event are expected.

3.2 SDv graph

Two or more scenarios specified as charts can be composed to form a more complex scenario, which in turn can further be composed. The composition operators are explain below.

Figure 6 describes a complex scenario, composed of two charts or sub-scenarios p and q in parallel, and noted $p||q$, which is composed sequentially with r. $p||q$ terminates when p and q terminates.

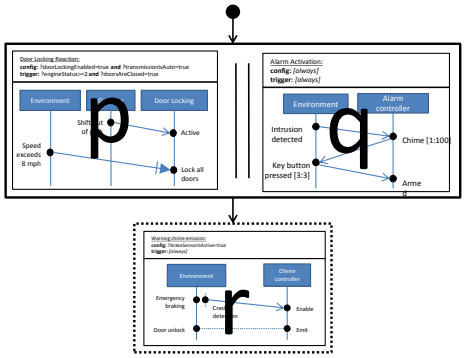


Fig. 6: Parallel, sequence and optional: p and q are in parallel, then optionally in sequence with r

The arrow without the source but with $p||q$ as the target indicates that it is the initial chart. The arrow from $p||q$ to r indicates the sequential relationship. Moreover, the dotted box around r denotes an optional behaviour: An *optional* block may or may not be executed. In opposition to the regular chart, even if it is entered, it may be exited any time without scenario failure.

Figure 7 illustrates the conditional branching. The scenario in this figure starts with the behaviour as specified by p, followed by either q or r. The conditions under which q is evaluated is the label of the edge. Otherwise r is evaluated. The condition is a configuration statement (related to the variability of the system).

Figure 8 illustrates the repeat operator. In this figure, the inner chart p is executed four times.

In Figure 9, another composition operator is illustrated: the chart p is a *pre-scenario*, which acts like a guard to the chart q. p prevents its guarded chart to be executed if it is not matched. The guard may be exited any-time without scenario failure. However, if p holds, then q must hold.

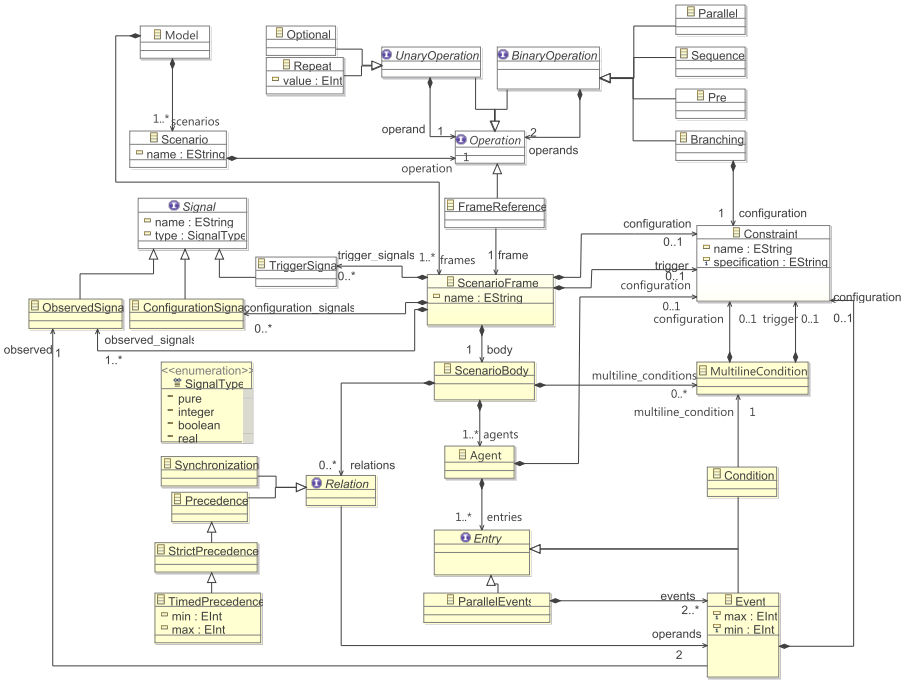


Fig. 10: The metamodel of SDv

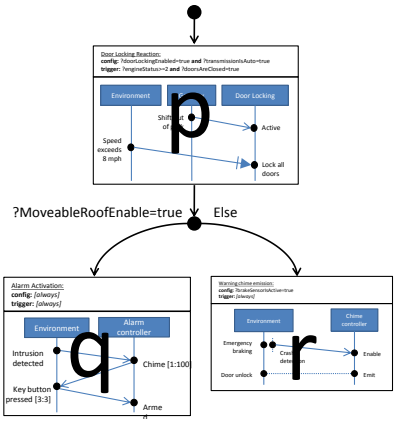


Fig. 7: Branching: p is executed, then either q or r

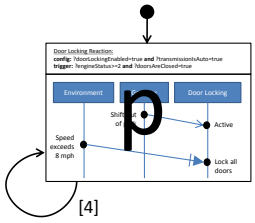


Fig. 8: Repeat: p is executed four times

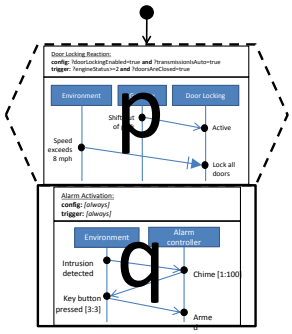


Fig. 9: Pre scenario: q is executed if p succeeds

3.3 SDv metamodel

Thus, in the general case, an SDv behaviour is described with a graph, whose nodes are scenarios, either charts or graphs. The edges between a pair of nodes indicate the causality relation, and ensure a strict evaluation order between the related cause and consequence: the last event of the cause scenario must happen before the consequence scenario can start.

Figure 10 presents the metamodel of SDv. An SDv chart is an instance of ScenarioFrame that contains an instance of ScenarioBody. ScenarioBody contains the elements and relations we have described above (Agent, Event, ParallelEvents, Multi-line Condition, Synchronization, Precedence). An SDv graph is an instance of Operation that could be a tree of operations (Optional, Repeat, Parallel, Sequence, Pre, Branching) where the leafs are instances of FrameReference that references an instance of ScenarioFrame (an SDv Chart).

4 UML and MARTE for building SDv models

The previous section has discussed the important concepts required to introduce variability in Sequence Diagrams. From an implementation perspective several solutions are available. The first solution is to develop an ad-hoc domain-specific environment for variability, a so-called *domain-specific language*. Such a solution has many advantages since it would allow building a specific environment that makes available all the domain concepts needed to address the issue and only them. It usually leads to very compact and very efficient environment. However, as always, introducing a new tool or environment in a design flow of a big company may be cumbersome and the very least costly since it requires specific training for the staff and building a set of transformation/integration tools to integrate this model with the other tools and models used in the company.

An alternative solution is to use a general-purpose tool (like the Unified Modeling Language—UML) and customize it so that it can be used for our specific case. Using a tool like UML can be very beneficial since lots of engineers are already trained to use it and it covers several aspects of the design flow (like requirement engineering, functional analysis, deployment). One tricky aspect when doing that is to ensure that the same model elements are not used in the different contexts with two different semantics. Another point is that UML is a general-purpose language but it is not intended to cover all usages of all domains. Rather it provides some mechanisms, light-weight extensions like profiling, to extend it when it is required to alter the semantics of some elements or to extend it by adding new concepts.

This section discusses a possible mapping between UML model elements and SDv concepts, when possible. Some aspects were not readily available in UML and required using specific extensions. In particular, since our intention is to generate Esterel observers, the usual run-to-completion event-based semantics of UML may significantly differ from Esterel semantics and from the semantics of SDv discussed in the following section. SDv sometimes uses specific synchronous operations as well as some time-related features. For such constructs, we have used the UML Profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE). MARTE is dedicated to real-time and embedded systems and provides a rich model of time [6] largely inspired from synchronous languages. It is therefore well-adapted to cover some specific synchronous constructs. The purpose here is not to rely heavily on MARTE but to use only the minimum constructs required to cover our needs. MARTE comes with a companion language, called the Clock Constraint Specification Language (CCSL), that is also sparsely used when required. Some previous works has already discussed the way to generate Esterel observers from MARTE/CCSL specifications [5], the work here is entirely different since the primitive constructs of SDv are very different from CCSL constructs.

4.1 UML interactions

It seems quite natural to use UML interactions to model SDv scenarios, however the particular context in which we are requires further comments. This subsection discusses the main concepts of SDv and the way they are encoded with UML counterparts.

Models in SDv (cf. Fig. 2) have a direct equivalent in UML models. A UML model owns some classes that can have a particular behavior. A SDv model owns a set of scenarios and a set of scenario frames. The latter set describes a library of available behaviors, whereas the former set describes one particular way of using those behaviors from the library. Therefore the ScenarioFrame are encoded as UML interactions, whereas the Scenario are encoded as activities.

Scenario describes the ways the different frames are used and composed. Several operations are provided. These operations are encoded as ActivityNodes in UML. We can distinguish two kinds of nodes. The Frame is mainly a CallBehaviorAction that refers to a scenario frame (a UML interaction). All the other operations are activity control nodes. The Sequence is captured by control flows, whereas the Parallel operation is captured by fork and join nodes of activities. Branching, as well as Optional and Repeat are simply captured by decision nodes with adequate guards on the outgoing control flows.

Scenario frames demands a bit more explanations. As discussed before, they are encoded as UML interactions. Such an interaction describes what is observed from the system. Our observers observe both the system and the environment. The SDv scenarios are used to specify what is expected from these interactions between the environment and the system. Observers must not interfere with the observed system, this is particularly easy with Esterel-based systems since the basic communication mechanism is broadcast through signals. Each scenario frame is therefore encoded as a UML interaction, where the life lines represent the SDv **agents** and the basic UML occurrence specifications represents occurrences of **events** observed from either the system or the environment.

Signals are equivalent in SDv and in UML. The distinction between trigger signals, configuration signals and observed signals depends on the context in which they are used. The configuration (resp. trigger) signals are the constrained elements of the configuration (resp. trigger) constraint.

Observed signals are particular cases and demand more explanations. It is important that we know which signals are observed or not. Only the orderings between the observed signals impact the satisfaction or falsification of our observation interaction. Therefore, we impose that the UML interactions must be included within a UMLConsiderFragment. Such fragments in UML explicitly specifies the relevant messages for a given interaction. In our case, we use such fragments to specify which signals are observed. Other signals may be ei-

ther configuration or trigger signals depending on the context in which they are used.

Constraints are also strictly equivalent in SDv and UML. However, we impose that the specification be a UMLOpaqueExpression described with the language Esterel. If so, then the UML specification becomes equivalent to our EsterelExpression.

Events are captured as MessageOccurrenceSpecifications and in UML, represent a message reception as if the signals received through broadcast from the environment and the system were received (but not consumed) by the observing interaction. If the min and max attribute of those events is simply one, then nothing more is required. When those attributes differ from one, then the message occurrence specification must be embedded within a Loop Combined Fragment that specifies how many times this occurrence specification is repeated.

ParallelEvents in SDv means that there is not a strict ordering between two events, which can therefore occur concurrently. This construct is equivalent to UML co-regions. In UML, along a particular lifeline (an agent), the occurrence specifications (the event occurrences) are strictly ordered unless surrounded by a co-region.

Relation clearly depart from messages in UML interactions since they merely represent a constraint on the ordering in which the related events must occur. UML provides a similar construct called GeneralOrdering. The UML GeneralOrdering is equivalent to our strict precedence. This is clearly, one of the specificity of our approach since our UML interactions do not focus on the messages but mostly on the general orderings. Actually, one can consider that the messages are sent (with broadcast) by the environment to the observer. The events are therefore message occurrence specifications but we are not interested in the messages themselves and show only the orderings among the various receptions of messages (called events in SDv). The discussion between the different kinds of relations is a bit subtle and is further discussed in subsection 4.2.

MultilineCond are mainly used to synchronize the different agents (or lifelines). When the condition is reached, then the configuration constraint must be satisfied before proceeding further. This is captured with UML state invariants with a specific semantic interpretation. The scenario completes if at some point, the state invariant becomes true (the configuration is fulfilled) and the trigger occurs. A constraint is applied to the state invariant to encode the trigger.

4.2 Specific synchronous constructs

Some constructs of SDv have no direct equivalent in UML. For such cases we use MARTE stereotypes to build them. This subsection discusses the few such cases.

Synchronization and precedences: the semantic variations amongst the different relations demand using some specific MARTE constructs and more particularly CCSL constraints. CCSL introduces two basic relations between events: coincidesWith and precedes. The former relation directly comes from the synchronous origin of
ccsl and specifies that two events must always occur synchronously. This is the exact intentional semantics of our SDv **synchronization**. The latter relation specifies is an asynchronous operator that specifies that an event precedes another one. More precisely, $a$precedes$b$ means that the i^{th} occurrence of event a must always be observed strictly before the i^{th} occurrence of event b. This is equivalent to our **Strict precedence**. Finally, the disjunction of the two constraints allows building or (weak) **precedence**: the i^{th} occurrence of event a must always be observed either synchronously with or strictly before the i^{th} occurrence of event b. In the three cases, we therefore use UML GeneralOrdering but we add a CCSL constraint on it to distinguish the three cases.

Timed precedences are equivalent to UML duration constraints. However, UML do not specify the time reference used to measure the duration itself. MARTE timed duration constraints are therefore used to introduce such time reference. More specifically, a MARTE timed constraint add an explicit reference to a clock. When applied to duration constraints, this clock is used as a time reference to compute the duration. Such a clock is exactly equivalent to Esterel notion of clock and is therefore particularly well adapted to the situation.

Trigger are used to specify a specific condition on which a scenario frame should execute. In our case, scenario frames are specified as UML interactions and are therefore a specialization of the UMLBehavior metaclass. MARTE provides a specific stereotype, called TimedProcessing that extends the Behavior metaclass with two new properties: start and finish. These properties are events that specify when the behavior starts its execution and when it finishes it. The start property is therefore equivalent to the notion of trigger defined in SDv. In MARTE, the start of a TimedProcessing is a simple event. It can for instance be a TimeEvent whose specification (a UML constraint) specifies exactly when this event occurs.

5 Observers-Based Synchronous Semantics of SDv

Here we give an observer-based semantics of SDv . This semantics associates an *observer* with each scenario (either chart or graph). An observer is an automaton running along with the system that collects the information about its state, its inputs and outputs at every stage, and determines whether the observed behaviour of the system matches the given scenario. The observer automaton emits a single signal *terminated* when the observation matches the scenario. The observer is not modifying the behaviour of the observed system.

A scenario is validated when it occurs as soon as it as been trigged. This is a the first occurrence semantics for which only the first activation of the scenario can lead to success.

Another particularity of the semantics is that the scenario succeed only if the expected events occur. The occurrence of an event when it is not expected cause a failure.

The behaviour of a run of a scenario observer is defined as a sequence of reactions:

$$S_0 \xrightarrow[I_1]{\emptyset} S_1 \xrightarrow[I_2]{\emptyset} \ldots \xrightarrow[I_{n-1}]{\emptyset} S_{n-1} \xrightarrow[I_n]{terminated or \emptyset} 0 \qquad (1)$$

where I_j is the set of observed signals in the observer at the step j. S_{j+1} is the *derivative* of S_j, the new state reached after the j^{th} reaction. 0 denotes the scenario termination. The output of the observer at the last reaction is either *terminated* when the scenario succeeds or $\emptyset$ when it fails. Each reaction consists of a sequence of atomic and instantaneous micro-reactions inferred by the structure of the body of the observer.

The body of the main Esterel module of a scenario is given below. The error signal is required only for internal usage while the *terminated* signal appears in the interface of the main observer.

```
module main_observer:
input all the observed signals;
output terminated;
signal error in
  run observer[...];
end signal
end module
```

The module *observer* is an observer built from an SDv chart of an SDv graph according to the translation rules presented below.

5.1 The chart observer

The chart observer has two levels: the frame and the body. The frame intends to check the configuration of the chart. When it is correct, it starts waiting for the trigger and launch the body of the observer when it trigs. The Esterel code of the frame is given below. The second output *error* is required for composition only.

```
module frame:
input all the observed signals;
output terminated, aborted;
present <configuration expression> then
  trap trigger in
    loop
      present <trigger expression> then
        exit trigger
      end present;
      pause
    end loop
  end trap;
  run body
end present;
end module
```

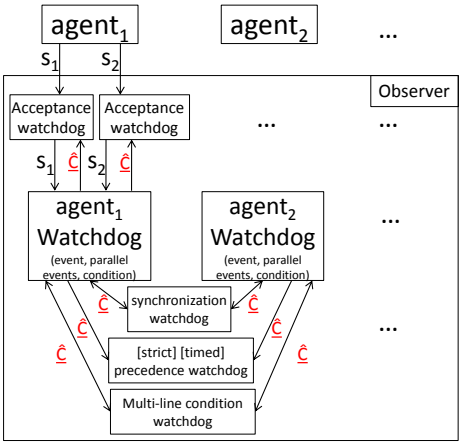


Fig. 11: The internal structure of the body of the chart observer.

The body of the chart observer is a massively parallel Esterel program. Each time-line, each precedence, each multi-line condition, each synchronization is translated in an Esterel module running in parallel to the others. The success of the body is due to the presence of the expected signals in the right order(s) at the right instant(s) but also to the absence of the unexpected signals. The occurrence of an unexpected signal aborts the body. To ensure this property, each signal in input of the observer is caught by an *acceptance watchdog* that either propagates the signal to the observer or aborts the observer if the signal is unexpected at the current instant.

Figure 11 illustrates the internal structure of the body of the chart observer. The signals S_1 and S_2 from the $agent_1$ are caught by an acceptance watchdog. Every agent has its corresponding watchdog which is composed of event watchdogs. The precedence, synchronization, and multi-line condition watchdogs watches their respective relation from the corresponding SDv chart. The "$\hat{\underline{C}}$" symbol indicates an exchange of control signal internal to the observer. The remainder of this section presents the internal structure and behaviour of the body of the observer and its watchdogs.

The agent watchdogs are not implemented as Esterel modules (like the other watchdogs) but as a statement relating the expected behaviour on the corresponding time-line. When every agent's statement terminates, the trap *tterminated* is raised that causes the emission of the output *terminated* followed by the end of the execution. At any moment, the trap *tERROR* can be raised by any of the watchdog that causes the emission of the output *aborted* and the end of the execution.

```
module body:
input all the observed signals;
output terminated, aborted;
signal local control signals in
trap tterminated, tERROR in
  run aw1/accept_wd[...]; exit tERROR||...
  ||
  run pw1/preced_wd[...]; exit tERROR||...
  ||
  run sw1/sync_wd[...]; exit tERROR||...
  ||
  run mw1/mlcond_wd[...]; exit tERROR||...
  ||
  [
    <Agent 1>    ||
    ...          ||
    <Agent n>
  ]; exit tterminated
handle tERROR do  emit aborted;
handle tterminated do present aborted else
    emit terminated end present
end trap
end signal
end module
```

5.1.1 Agent watchdog

follows the content of the agent's time-line. There is three possible actions occurring on the time-line: 1/ an event occurs 2/ parallel events occur 3/ a multi-line condition is crossed.

The statement of any agent watchdog can be generated from the following algorithm:

```
1: present <agent configuration> then
2: for all action a in sequence on the time-line do
3:     if a is an event occurrence then
4:         present <event configuration> then
5:         emit stopAcceptanceWatchdog;
6:         run EventWatchdog[...];
7:         emit startAcceptanceWatchdog
8:         end ;
9:     else if a is a parallel events occurrence then
10:        [
11:        for all event e in parallel do
12:            emit stopAcceptanceWatchdog;
13:            run EventWatchdog[...];
14:            emit startAcceptanceWatchdog
15:        end for
16:        ];
17:    else if a is a multi-line condition crossing then
18:        present <ml condition configuration> then
19:        emit rendezVous;
20:        await immediate notification
21:        end ;
22:    end if
23: end for
24: end present
```

5.1.2 Event watchdog

checks for the occurrence of an event *e* from a specific agent. First, the event watchdog waits for the minimal number of occurrences of the observed signal and then it emits the internal control signal acknowledgement that is expected by the other watchdogs. Then, it wait for additional occurrence of the observed signal until either another signal is received or too much occurrences of the signal occur. The first case is a normal termination the second causes an error aborting the entire observer. An event watchdog is an Esterel module with the following shape. For every event, the watchdog is generated with the appropriate values for *min* and *max*. The second part of the module is generated only when $max > min$. In the following module, every occurrence of the repeat instruction is unrolled in order to generate pure Esterel code. The same is done for every module.

```
module EventWatchdog:
input   observedEvent, any_Other_Event;
output    error, acknowledgement,
    occurred;
await immediate observedEvent; emit
    occurred;
repeat (min-1) times
  await observedEvent; emit occurred
end repeat;
emit acknowledgement;
pause;
%when max>min
weak abort
  repeat (max - min + 1) times
    await immediate observedEvent;
    emit occurred;
    pause
  end repeat;
  emit error
when immediate any_Other_Event;
present observedEvent then emit error end
     present
end module
```

As an illustration of the event watchdog, let us consider the chart of Figure 12. Let us assume that *Producer* emits the signals *A* at the second instant and *B* at the third instant. At the first instant, the event watchdog on *A* will be ready to receive the signal. There is no incoming precedence and the acceptance watchdog is deactivated thanks to the signal *stopAcceptanceWatchdog*.

At the second instant, *A* is received, the *acknowledgement* signal is sent (however no watchdog listen to it) and the observer awaits the possible repetition of *A*

At the third instant, *B* occurs. The waiting on *A* is aborted, the acceptance watchdog on *A* is re-activated (it will be ef-

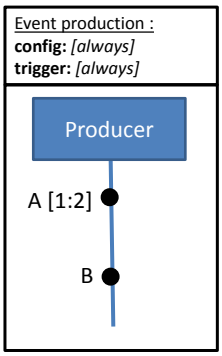


Fig. 12: An occurrence of the signal *B* right after the first occurrence of the signal A will terminate the event watchdog on *A*.

fective at the next instant) and the event watchdog on *A* is terminated. The control is given to the agent watchdog that run the next event watchdog; It is on *B*. So the acceptance watchdog on *B* is deactivated (and thus no error is raised) and the event *B* is caught. The execution is paused.

At the fourth instant, the event watchdog on *B* is terminated, the agent watchdog is terminated and the trap *tterminated* is raised. The scenario succeeds.

5.1.3 Acceptance watchdog

is a special kind of watchdog that does not correspond to any of the SDv operator but guaranty that only the expected signals are received. An acceptance watchdog is the following Esterel module:

```
module AcceptanceWatchdog:
input observed, start, stop;
trap tERROR in
    [
  loop
    abort
      await immediate observed;
      exit tERROR
    when immediate stop;
    await immediate  start;
    pause
  end loop
  ]
end trap
end module
```

5.1.4 Multi-line condition watchdog

expects a *rendez-vous* signal from every concerned agent. Then, it evaluates the condition. If it is met, it notifies the agents observers that the execution can go ahead. If it is violated, it raise either an error to the chart observer in case of an hot condition or a notification of preliminary termination in case of a cold condition. The former cause the emission of the *aborted* signal. This last cause the emission of the *terminated* signal. In both cases, the execution of the chart observer is terminated. An multi-line condition watchdog is the following Esterel module:

```
module multiLineconditionWD :
input isHot;
input rv_1, ... rv_n, condition;
output notify, error, match;
[
  await immediate rv_1
  ||
  ...
  ||
  await immediate rv_n
];
present ?condition then
    emit notify
  else
    present isHot then
      emit error
    else
      emit match
    end present
end present
end module
```

5.1.5 Precedence watchdog

ensures the satisfaction of a single precedence relation. The precedence is satisfied when the last occurrence of the source event occurs before the first of the destination event. A precedence watchdog is the following Esterel module:

```
module PrecedenceWatchdog:
input source_completed, source_occurred,
    dest_occurred;
trap tERROR in
  <weak> abort
    await immediate dest_occurred;
    exit tERROR
  when immediate source_completed;

  await immediate dest_occurred;
  await source_occurred;
  exit tERROR
end trap
end module
```

The source_complemted signal is mapped to the acknowledgement signal emitted in the event watchdog. Similarly, the {source, dest}_occurred signals are mapped to the occurred signals of their respective event watchdogs. The termination of the execution of the watchdog cause the emission of the

aborted signal. The presence of the keyword [weak] change the precedence watchdog in its strict version (See figure 2).

5.1.6 Time precedence watchdog

ensures the satisfaction of a single time precedence. When the source signal is received, a signal called *error* is sustained during *min* instants. Then it is idled during the correct time frame and sustain afterwards. So, an early or (too) late reception of the target signal would be coupled with the presence of the error signal that raises the trap *tERROR* in the observer.

```
module TimedDependenceWatchdog:
input source_completed, source_occurred,
    dest_occurred;
trap tERROR in
  [
  run dw / StrictDependenceWatchdog;
  exit tERROR
  ]||
  [
  signal error in
  await immediate source_completed;
  weak abort
    loop
      abort
        repeat min times
           emit error;pause
        end repeat
        repeat (max-min+1) times
          pause;
        end repeat
        sustain error
      when source_occurred
    end loop
  when dest_occurred;
  present error then
  exit tERROR
  else halt
  end present
end signal
  ]
end trap
end module
```

5.1.7 Synchronization watchdog

ensures the satisfaction of a single synchronization constraint (horizontal dotted line). A synchronization watchdog receives the occurred signal of the pair of watched signals. If they are not simultaneous, an error signal is raised to the observer. An synchronization watchdog is the following Esterel module:

```
module SynchronizationWatchdog:
input left, right;
trap tERROR in
  repeat max times
    abort
      await immediate [left or right];
      exit tERROR
    when immediate [left and right];
    pause
  end repeat;
  halt
end trap
end var
end module
```

5.2 The graph observer

A graph observer is an Esterel module where the only authorized statement are the high level operators of SDv and the calls (run expression) to a chart observer. The called observer(s) can be either chart(s) or graph observer(s). The high level operators can be composed freely with respect to the well-parenthesizing rule.

A graph observer is decomposed in n Esterel modules where n is the number of operations used in the graph. Let assume the graph observer to be a tree of operators where the leafs are calls to chart observers. Every node of the tree is translated into a module and the module of the root node is the main. Whatever is the operator, the generated Esterel module has the same head and tail given below. Only the inner part changes from an operator to another

```
module ComposedObserver:
input all the observed signals;
output terminated, aborted;
signal terminated_s1, terminated_s2 in
signal ERROR_s1, ERROR_s2 in
trap tterminated, tERROR in
  %%%%%%%%%%FAILURE%%%%%%%%%%%%%%%%%%%%
  await [ERROR_s1 or ERROR_s2];
  exit tERROR
  ||
  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  <Operator dependent part>
  ;
  exit tterminated
handle tERROR do
  emit aborted;
handle tterminated do
  present aborted else emit terminated
     end present
end trap
end signal
end signal
end module
```

The signals *terminated_s*2 and *ERROR_s*2 are declared and used only for binary operators (sequence, parallel, branching, pre scenario). In case of optional operation, the *FAILURE* section is omitted because an optional scenario does not raise error. In case of pre scenario, the signal *ERROR_s*1 is not included in the *FAILURE* for similar reason but *ERROR_s*2 is.

5.2.1 The optional statement

In the observer based semantics with a strong hierarchy as it is for SDv, when a module receives an error from a child module, it is supposed to propagate this error to its parent module. But if the child module is running under an optional statement, the error is blocked and the child terminates instantaneously. The following statement is the inner part of the generated Esterel module.

```
run Observer_1[signal terminated_s1/
    terminated, ERROR_s1/aborted];
```

5.2.2 Sequence

The following statement is the inner part of the generated Esterel module. One can see that the translation is quite straightforward because of the strict hierarchy imposed by our semantics.

```
run Observer_1[signal terminated_s1/
    terminated, ERROR_s1/aborted]
;
run Observer_2[signal terminated_s2/
    terminated, ERROR_s2/aborted]
```

5.2.3 The parallel statement

The following statement is the inner part of the generated Esterel module.

```
run Observer_1[signal terminated_s1/
    terminated, ERROR_s1/aborted]
||
run Observer_2[signal terminated_s2/
    terminated, ERROR_s2/aborted]
```

5.2.4 The repeat statement

The following statement is the inner part of the generated Esterel module.

```
repeat m times
  run Observer_1[signal terminated_s1/
      terminated, ERROR_s1/aborted];
  pause
end repeat
```

5.2.5 The branching statement

The following statement is the inner part of the generated Esterel module.

```
present <conditionOfBranching> then
run Observer_1[signal terminated_s1/
    terminated, ERROR_s1/aborted];
else
run Observer_2[signal terminated_s2/
    terminated, ERROR_s2/aborted];
end present
```

5.2.6 The pre-scenario statement

In a pre-scenario, the guard scenario (p) is executed first. If it comes to its correct termination, the guarded scenario (q) follows. Otherwise, q is skipped and the pre-scenario succeeds.

```
run Observer_1[signal terminated_s1/
    terminated, ERROR_s1/aborted];
present terminated_s1 then
  run Observer_2[signal terminated_s2/
      terminated, ERROR_s2/aborted]
end present
```

6 The Verification Flow

The verification flow that we use is illustrated in Figure 13. It takes two kinds of inputs:

1. The system design: In our case, we chose to apply our verification technique to systems designed using Statecharts or discrete-time Simulink, but the overall verification flow could be more general, as far as we can generate an Esterel model from the design that preserves the semantics;
2. A feature model from which we extract the variability constraints related to the observed features and their parameters;
3. The SDv that is extracted from the requirements.

The Esterel synchronous language is used as a pivot between the design language, either Statecharts or Simulink,

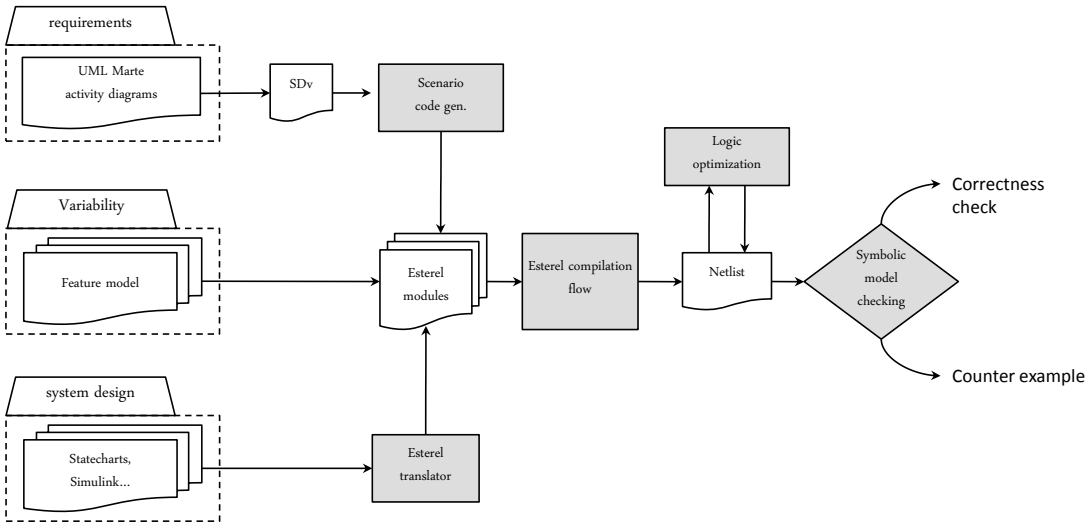


Fig. 13: The verification flow.

the variability constraints, and SDv observers, so that the observed system and its observer can simply be synthesized and linked together. A direct code generation from Statecharts to Esterel might be tedious, so we first transform Statecharts into SyncCharts [2]. Basically, this step consists in introducing a synchronous semantics to the model, but it is also necessary to fix several syntactic issues (see Section 1). Then we use the SyncCharts Compiler Collection [14] to generate the Esterel code. As for discrete-time Simulink, its translation to Lustre has been proposed by Tripakis *et al.* [43]; adapting it to the Esterel case is a purely syntactic matter.

An Esterel compiler, such as the INRIA Compiler [41], synthesizes a BLIF netlist from the Esterel code. This netlist can then be optimized using any off-the-shelf logic optimizer, as those commonly used in the hardware industry. Finally, the scenario can be verified through symbolic model-checking.

6.1 From Statecharts to SyncCharts

Statecharts have been given many different semantics [25, 45]. In our case, we deal with a GM-specific semantics, close to the one of Stateflow. Here we discuss a few translation issues from such Statecharts to SyncCharts, but we do not detail our domain-specific semantics since it is not essential for the overall understanding of the process.

A common feature between our statecharts semantics and the one of Stateflow or SyncCharts is that the evaluation of transition guards is top-down: outgoing transitions of the active macrostate have priority over the inner state transitions.

However, concerning the order in which outgoing transitions of a given state are tested, SyncCharts ensure a strict ordering using explicit priority, whereas our statecharts do not offer any guaranty. Hence we can only ensure that the transformation will preserve the semantics in case of non-overlapping guards. Indeed this is a common design guideline for ensuring a deterministic behaviour[2)]. The problem of determining whether a statechart may suffer from overlapping guards goes beyond the scope of this work, but in our case we solved it through static analysis: if there exists a solution satisfying the conjunction of guards, then we ask the designer to strengthen those guards.

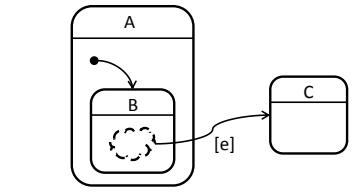
Our statecharts do not have a time model, and their reactions are much less complex than those of general SyncCharts, possibly involving combinatorial cycles or reincarnation. When triggered, a statechart reacts to the input events by taking a single transition per step, which corresponds to an *instant* in the synchronous terminology. Such a behaviour can be obtained in SyncCharts using only strong, non-immediate transitions.

SyncCharts enforce a strict hierarchy of states and transitions, whereas Statecharts allow transitions to cross their parent macrostate boundaries. Hence it is necessary to split the transitions across state boundaries. The solution is inspired from the translation of Esterel traps into SyncCharts, as proposed in Prochnow *et al.* [37]. Figure 14 shows an example of such trap, emulated using immediate weak abortions: the *traphalt* signal may be emitted in order to prevent the termination at low-level if a trap with higher priority is emitted. A similar transformation has been elaborated for incoming transition across boundaries.

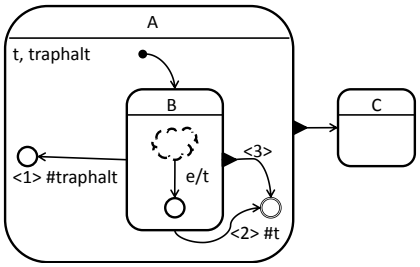
6.2 From UML Marte to SDv

We have implemented the transformation rules described in Section 4 so that a user can capture a SDv specification using a UML tool and then automatically generate Esterel observers.

[2)] In fact the generated sequential code would be obviously *deterministic* in the usual sense. Here we mean that the designer is not necessarily aware of internal choices taken by the code generator, and transition overlappings at the model level might lead to unexpected behaviours from the designer's point of view.



(a) Statechart.



(b) SyncChart.

Fig. 14: Transformation of an outgoing transition.

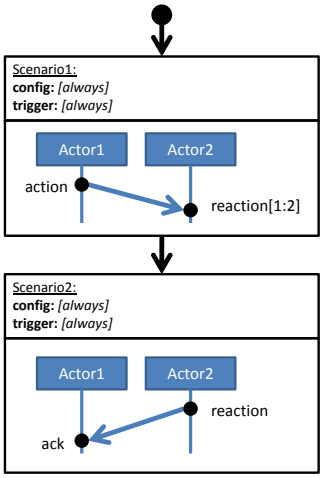


Fig. 15: An example of ambiguous scenario where the final optional signal of the first scenario conflicts with the first signal of the second scenario.

It must be noted though that we have mainly worked on the UML abstract syntax to implement this transformation. Even though, the UML provides most of the necessary concepts finding the right graphical tool is a separate issue. Indeed, even though the profiling mechanism promises a lot, the available implementations are far from satisfactory at the moment. A big effort is still required from the UML tool vendors to make the customization as easy as possible for the end users.

6.3 From SDv to an Esterel Observer

We have also implemented the transformation rules described in Section 5 so that any SDv scenario (chart or graph) can be converted into an Esterel observer.

The writing of SDv charts for observation purpose is subject to a limitation that prevent from ambiguous scenarios *i.e.,* the last event on each time-line should have a fix repetition value of the form $[x : x]$. Otherwise, the termination of the chart cannot be ever decided. Let us consider the case of Figure 15, where scenarios 1 and 2 are expected to occur in sequence. The signal *reaction* is the last of scenario 1 and the first of scenario 2. There is no way to decide whether the second occurrence of the signal *reaction* should be collected by scenario 1 or 2.

A more general solution, that we have not yet implemented, would use a back-tracking mechanism or a pipelined observer. We mention these ideas in Section 8.

6.4 Integrating variability constraints in Esterel

The main module is an Esterel module grouping together the design modules running in parallel along with the scenario observer. The list of inputs of the main module contains the signals relating the configurations of the variability of the system. As it is, the Esterel module accepts any combination of values (present/ absent) disregarding of the variability constraints defined in the feature model. We have defined three different ways to express the variability constraints in Esterel:

First, we can use the Esterel constructions expressing the exclusion and dependency between input signals. *a#b* means that the input signals a and b cannot occur simultaneously while a => b means that the presence of the signal a implies the presence of b. These two constructs are expressive enough to represent the mandatory, the optional, the exclusion and the require relations. However they are not adequate to represent or and xor group because at least one of the child feature has to be present when the parent is present but there is nothing in Esterel to force the presence of an input (We let to the ready the task of expressing each relation in Esterel).

Second, we can convert the variability constraints in a propositional formula [8] and add this constraint in the CTL formula in input of the model checker. Let ρ be the propositional formula expressing the variability constraint and ϕ the checked property in CTL. The property $\rho \implies \phi$ limits the validity of the formula to the valid variants of the system

Last, we can write an combinatorial module that takes as input a set of signals S and generate in output values for the configuration signals of the system. The behaviour of the module is such that every possible valuation of the signals of S lead to a valid variant. This last solution need to modify the interface of the main module. The signals of S have to be added to the list of input signals whereas the configuration signals are now declared as local signals.

6.5 The Need of Pre-Model-Checking Optimizations

The fact is that a scenario is used for checking a given trace, or set of traces, but not the whole system behaviour: in most cases, that does not involve each software component, and even not each functionality of a given software component.

As a consequence, a pre-optimization can be used for pruning as many dead or useless[3)] branches as possible. Whereas the generated netlist may be pretty large and complex, optimization techniques allow to reduce massively the problem complexity. On the netlist, just one output has to be considered, the one corresponding to the *terminated* signal, and the other outputs can be ignored. A standard logical optimizer would start from this output, track back the required inputs and latches using a simple dependency analysis, and remove all the dandling nodes that do not fan-out into the output or such latches, thus allowing to only consider the part of the design which is strictly necessary to the very observer. This optimization, which basically consists in a depth-search and produces a functionally equivalent model, is linear in the size of the netlist. Further optimizations and cleaning passes can be applied to the network, for simplifying either the logic (balancing, refactoring, propagating constants, *etc*) or the latches (retiming, merging equivalent registers).

It is noticeable that the variability can be efficiently handled at this point. The actual configuration may vary from a product to another, but we know that parameters are constant for a whole product life, in opposition to regular signals which may vary from an instant to another. Then a first benefit comes from the fact that we introduced a semantic difference between variability and regular signals in Section 3. This information allows register removals and logic minimization using constant propagation if it is taken into account by the tool: once a parameter value is set at the beginning of the scenario, it holds till the end. The second valuable point is that variability constraints can be used by the optimizer for eliminating unreachable states and unnecessary logics. Thus the state space explosion due to the design variability but also to the gathering of the design models with the generated observer can be fairly limited.

6.6 Model-Checking

A scenario is a representation of an expected or an unexpected behaviour. In the first case, the system is supposed to support a run that validate the scenario. In the second, the system should never support that run. The signal *terminated* is emitted when the scenario finish successfully. The translation of the verification objective in CTL [19] is the following:

Expected scenario: $EF\ terminated$
Unexpected scenario: $AG\ !terminated$

The INRIA Compiler [41] is provided with the Esterel verification engine called Xeve [13]. Xeve is a symbolic model checker based on Binary Decision Diagram technology. It takes in input a BLIF netlist along with the description of the relations between the inputs (dependency, exclusion). One can force the values of some inputs to present or absent or let it free. For each output, Xeve is checking whether the signal is 1/ never emitted, 2/ possibly emitted, 3/ possibly not emitted, or 4/ always emitted. Xeve is not doing distinction between the depth (F/G) and the width (A/E) of the execution. In a scenario relating an expected behaviour, the output *terminated* has to be possibly emitted whereas in a scenario relating an unexpected behaviour, the output *terminated* has to be never emitted.

[3)] With respect to the checked scenario.

7 Experiments

7.1 Unit testing

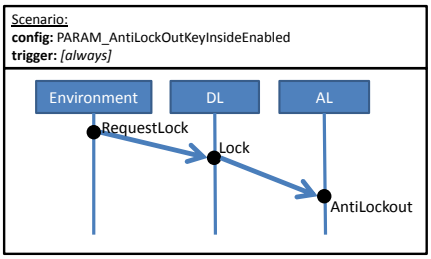
We have validated the correctness of our transformation from SDv to Esterel by performing unit testing of every instruction of the SDv language. For each instruction, we have generated a scenario that mostly use only it and we coupled this scenario with a pseudo-design that either validates or invalidates the scenario. Both cases have been explored for every instruction. We used the verification flow to check that the actual behaviour conforms to the expected one. We have written 11 scenario frames and 17 scenarios (one for each scenario frame plus one for each composition operator). These scenario have been tested using 43 pseudo-designs where 24 conforms to the scenarios and 19 invalidates them. Every run of the verification engine on these examples where fast (only few milliseconds) but the size of the state space where very limited (less than 100 states). We found a case where both *terminated* and *aborted* where emitted simultaneously and corrected it.

7.2 An industrial case study

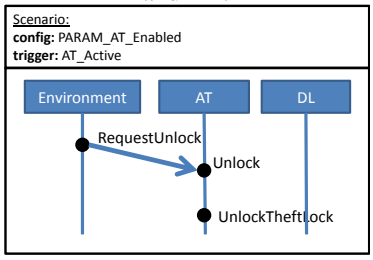
We applied the proposed process to a real case-study from the automotive industry. Among all, we have selected three features that interact strongly and we have defined three scenarios. The first scenario relates an expected behaviour and the two other relate unexpected behaviours. These scenarios describe the interactions between the Environment (the user) and the three controller's features. Door Lock (DL) is the core controller to deal with locking and unlocking actions, Anti-Theft (AT) provides some automatic locking and safety functionalities, and Anti-Lockout (AL) prevents the inadvertent lockout situations where keys are locked in the vehicle. Figure 16 presents the three scenarios. The first relates the main use case of the feature AL. The action *AntiLockOut* unlocks the driver door when a key is detected inside the vehicle.

The two other scenarios focus on the interactions between DL and AT. When AT is enabled, it partially overrides the behaviour of DL. So it is not only about the correct behaviour of AT but also that DL keeps stalling when it does. The second scenario presents a problematic case where the theft lock is unlocked after the normal lock. This case, if it occurs, could physically damage the locks of the cars. Theft lock has to be opened first. The last scenario presents another problematic case. The AT feature has an option called *SinglePress*.

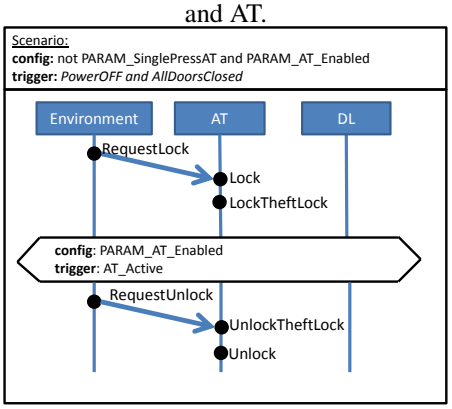
When one want to lock or unlock the theft lock, it could either press twice the key button to lock first the normal lock and then the theft lock or press once the key button to lock both the locks (in the correct order). A similar distinction exists for the unlocking actions. This second option is available only when the parameter *PARAM_SinglePressAT* is active. The scenario describes the expected behaviour of the system when the option *SinglePress* is selected. Since the precondition of the scenario is *notPARAM_SinglePressAT*, it is an unexpected behaviour.



a) Scenario relating the expected interaction between DL and AL.



b) Scenario relating the unexpected interaction between DL and AT.



c) Scenario relating another unexpected interaction between DL and AT.

Fig. 16: Experimental use cases: an entry point controller made of three features (Door Locking, Anti-Theft, Anti-Lockout).

Each feature is made of several concurrent statecharts, interacting with each others: DL has 14 statecharts, AL has 4, AT has 10. The Esterel code generation process translates these three features to about 2,500 lines of code, plus 400 more lines for each observer. For improving the readability of the Esterel code, we also make use of a library of predefined

	Before reduction	After	Gain
Latches	147	83	43.5%
Gates	1,582	568	64.1%
Edges	3,744	1087	71.0%
Depth	49	22	55.1%

Table 1: Netlist optimization

Name	#State space	Exec. time	result
Scenario1	16,213,825	11m 50s	possibly emitted
Scenario2	1,079,425	5s	never emitted
Scenario3	267,265	5s	never emitted

Table 2: Verification results

Esterel modules, such as SR latches or rising/falling edges detectors. So we generate an Esterel module for each statechart, one module for the scenario, plus a top-level module wrapping them all. The top level module is different for every verification effort. For example, when checking the first scenario, the feature AT could be disabled; when checking to two others, the feature AL can be disabled. However, the interfaces of the top level modules are the same. It has 93 inputs, among which 13 parameters specifying the variability and 80 regular signals. The variability constraints has been encoded as signal relation because there was no *or* or *xor* groups to encode. Then the Esterel synthesis flow produces a BLIF netlist. This BLIF netlist is then shrunk down by 60% in 1.27 second[4)], using the Berkeley ABC [11] logic optimizer with a custom optimization script. Table 1 shows a reduction of the size of the complexity of the network from half to the two-thirds for the first top level modules. The two others are optimized with a similar gain.

Finally the scenarios are verified using Xeve to determine whether the *terminated* signal can be emitted or not. The execution time and the size of the state space is given in Table 2. One should note that for Scenario 2 and 3, the size of the state space before the reduction made by ABC was respectively 5 billions and 20 billions with an execution time of 1 and 11 hours.

8 Discussion and Further Works

We would like to extend our semantics from first occurrence to any occurrence [45]. However, this improvement need to deal with the problem of "frames interleavings". When the scenario fail, it is not enough to restart it from the beginning but a back tracking mechanism has to be implemented in the observer. For example, if the observer is expecting the sequence of events *aab* but *aaab* comes. While receiving the third signal *a*, the observer fails to match the pattern but should not restart, instead, it should backtrack by a step only,

[4)] average value

as it is implemented in the Aho-Corasik algorithm [1].

We plan to tackle this issue in further works using pipelined observers. The idea has already been used to some extent for synthesizing logical circuits from regular expressions [15]. However, regular expressions consider only one input at a time, while big scenarios that we can consider may have complex reactions involving dozens of signals at a time. In some sense, this idea must be extended from one to n dimensions to solve this problem.

References

1. Alfred V. Aho and Margaret J. Corasick. Efficient string matching: An aid to bibliographic search. *Communications of the ACM*, 18:333–340, June 1975.
2. Charles André. SyncCharts: a Visual Representation of Reactive Behaviors. Technical Report RR 96–56, I3S, Sophia-Antipolis, France, April 1996.
3. Charles André. Synchronous Interface Behavior: Syntax and Semantics. Technical Report RR 00–11, I3S, Sophia-Antipolis, France, December 2000.
4. Charles André. Syntax and semantics of the Clock Constraint Specification Language (CCSL). Research Report 6925, INRIA, May 2009.
5. Charles André and Frédéric Mallet. Specification and verification of time requirements with CCSL and esterel. In Christoph M. Kirsch and Mahmut T. Kandemir, editors, *LCTES*, pages 167–176. ACM, June 2009.
6. Charles André, Frédéric Mallet, and Robert de Simone. Modeling time(s). In Gregor Engels, Bill Opdyke, Douglas C. Schmidt, and Frank Weil, editors, *Model Driven Engineering Languages and Systems*, volume 4735 of *Lecture Notes in Computer Science*, pages 559–573. Springer, 2007.
7. Charles André, Marie-Agnés Péraldi, and Jean-Paul Rigault. Scenario and property checking of real-time systems using a synchronous approach. In *ISORC*, pages 438–444. IEEE Computer Society, 2001.
8. Don Batory. Feature models, grammars, and propositional formulas. In Henk Obbink and Klaus Pohl, editors, *Software Product Lines*, volume 3714 of *Lecture Notes in Computer Science*, pages 7–20. Springer Berlin / Heidelberg, 2005.
9. A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9):1270–1282, 1991.
10. Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, January 2003.
11. Berkeley Logic Synthesis and Verification Group. ABC, release 10216. http://www.eecs.berkeley.edu/ālanmi/abc/.
12. Rodrigo Bonifácio and Paulo Borba. Modeling scenario variability as crosscutting mechanisms. In *Proceedings of the 8th ACM international conference on Aspect-oriented software development*, AOSD '09, pages 125–136, New York, NY, USA, 2009. ACM.
13. Amar Bouali. Xeve: an Esterel Verification Environment (version v1_3). Technical Report RT–0214, INRIA, Sophia-Antipolis, France, December 1997.
14. Julien Boucaron and Daniel Gaffé. SyncCharts Compiler Collection. http://julien.boucaron.free.fr/i3s/.
15. Janusz A. Brzozowski and Ernst Leiss. On equations for regular languages, finite automata, and sequential networks. *Theoretical Computer Science*, 10(1):19–35, 1980.
16. Andreas Classen. Problem-oriented feature interaction detection in software product lines. In *Proceedings of the 9th International Conference on Feature Interactions in Software and Communication Systems (ICFIŠ07)*, September 2007.
17. Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, and Axel Legay. Beyond boolean product-line model checking: Dealing with feature attributes and multi-features. In *Proceeding of the International conference on Software Engineering 2013 (ICSE ï£¡13)*, 2013.
18. Werner Damm and David Harel. LSCs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19:45–80, July 2001.
19. E. Allen Emerson. Temporal and modal logic. In *HANDBOOK OF THEORETICAL COMPUTER SCIENCE*, pages 995–1072. Elsevier, 1995.
20. Roberto Silveira Silva Filho and David F. Redmiles. Managing feature interaction by documenting and enforcing dependencies in software product lines. In Lydie du Bousquet and Jean-Luc Richier, editors, *Feature Interactions in Software and Communication Systems IX, International Co nference on Feature Interactions in Software and Communication Systems, ICFI 2007, 3-5 September 2007, Grenoble, France*, pages 33–48. IOS Press, 2007.
21. Rick Flores, Charles Krueger, and Paul Clements. Mega-scale product line engineering at general motors. In *Proceedings of SPLC 2012*, pages 259–269, 2012.
22. Joel Greenyer, Amir Molzam Sharifloo, Maxime Cordy, and Patrick Heymans. Efficient consistency checking of scenario-based product-line specifications. In *Proceeding of the international conference on Requirement Engineering 2012 (RE'12)*, 2012.
23. Alexander Gruler, Martin Leucker, and Kathrin D. Scheidemann. Modeling and model checking software product lines. In *Proceedings of FMOODS 2008*, pages 113–131, 2008.
24. Nicolas Halbwachs. *Synchronous programming of reactive systems*. Kluwer Academic Pub., 1993.
25. David Harel and Hillel Kugler. The Rhapsody semantics of Statecharts. In *Lecture Notes in Computer Science*, volume 3147, pages 325–354. Springer-Verlag, 2004.
26. Charles W. Krueger. New methods in software product line practice. *Commun. ACM*, 49(12):37–40, 2006.
27. Kim Lauenroth, Klaus Pohl, and Simon Toehning. Model checking of domain artifacts in product line engineering. In *ASE '09: Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 269–280, Washington, DC, USA, 2009. IEEE Computer Society.
28. Paul Le Guernic, Jean-Pierre Talpin, and Jean-Christophe Le Lann.

Polychrony for system design. *Journal of Circuits, Systems, and Computers*, 12(3):261–304, 2003.

29. E. A. Lee and A. L. Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(12):1217–1229, December 1998.
30. Jing Liu, Samik Basu, and Robyn Lutz. Compositional model checking of software product lines using variation point obligations. *Automated Software Engineering*, 18:39–76, 2011. 10.1007/s10515-010-0075-7.
31. Jean-Vivien Millo, S Ramesh, Krishna Sankaranarayanan, and Ganesh K Narwane. Compositional verification of software product line. In *Proceeding of the International conference on integrated Formal Method 2013 (iFM'2013)*, 2013.
32. Object Management Group. OMG Unified Modeling Language (OMG UML), Superstructure, V2.1.2, November 2007.
33. John O'Leary, Xudong Zhao, Rob Gerth, and Carl-Johan H. Seger. Formally verifying ieee compliance of floating-point hardware. *Intel Technology Journal*, Q1'99:1–14, 1999.
34. OMG. *UML Profile for MARTE, v1.0*. Object Management Group, November 2009. formal/2009-11-02.
35. Klaus Pohl, Gï£¡nter Bï£¡ckle, and Frank van der Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer, Berlin Heidelberg New York, August 2005. ISBN: 3-540-24372-0.
36. D. Potop-Butucaru, S.A. Edwards, and G. Berry. *Compiling Esterel*. GeoJournal library. Springer, 2007.
37. Steffen Prochnow, Claus Traulsen, and Reinhard von Hanxleden. Synthesizing safe state machines from Esterel. *Proceedings of the 2006 Conference on Language, Compilers, and Tool Support for Embedded Systems (LCTES'06)*, 41:113–124, June 2006.
38. S. Ramesh and Purandar Bhaduri. Validation of pipelined processor designs using Esterel tools: A case study. In *Proceedings of the 11th International Conference on Computer Aided Verification (CAV'99)*, pages 84–95, London, UK, 1999. Springer-Verlag.
39. Michel A. Reniers. *Message Sequence Chart: Syntax and Semantics*. PhD thesis, Eindhoven University of Technology, 1999.
40. Klaus Schneider. The synchronous programming language quartz. Technical report, Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany, 2009.
41. The Esterel Team. Esterel Compiler 5.92. http://www-sop.inria.fr/esterel-org/filesv5_92/.
42. C. Traulsen, T. Amende, and R. von Hanxleden. Compiling synccharts to synchronous c. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, pages 1–4, 2011.
43. Stavros Tripakis, Christos Sofronis, Paul Caspi, and Adrian Curic. Translating discrete-time simulink to lustre. *ACM Transactions on Embedded Computing Systems*, 4, November 2005.
44. Frank Van der Linden, Klaus Schmid, and Eelco Rommes. *Software Product Lines in Action*. Springer-Verlag, 2007.
45. Michael Von der Beeck. A comparison of Statecharts variants. In *Proceedings of the Third International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 128–148, London, UK, 1994. Springer-Verlag.
46. Stefan Wagner, María Victoria Cengarle, and Peter Graubmann. Modelling System Families with Message Sequence Charts: A Case Study. Technical Report TUM-I0416, Technische Universität München, October 2004.
47. Tewfik Ziadi, Loïc Helouët, and Jean-Marc Jézéquel. Towards a UML profile for software product lines. In *Proceedings of the Fifth International Workshop on Product Family Engineering (PFE-5)*, volume 3014, 2003.