On checking delta-oriented product lines of statecharts

On checking delta-oriented product lines of statecharts ☆

Michael Lienhardt[a], Ferruccio Damiani[a,*], Lorenzo Testa[a], Gianluca Turin[a]

[a] *University of Torino, Italy*
(mlienhar@di.unito.it, ferruccio.damiani@unito.it, {lorenzo.testa, gianluca.turin}@edu.unito.it)

Abstract

A Software Product Line (SPL) is a set of programs, called variants, which are generated from a common artifact base. Delta-Oriented Programming (DOP) is a flexible approach to implement SPLs. In this article, we provide a foundation for rigorous development of delta-oriented product lines of statecharts. We introduce a core language for statecharts, we define DOP on top of it, we present an analysis ensuring that a product line is well-formed (i.e., all variants can be generated and are well-formed statecharts), and we illustrate how an implementation of the analysis has been applied to an industrial case study.

Keywords:
Core calculus, Delta-oriented programming, Software product line, Software product line analysis, Statechart

1. Introduction

A *Software Product Line* (SPL) is a set of programs, called *variants*, which are generated from a common artifact base and have well documented variability [12]. *Delta-Oriented Programming* (DOP) [30, 31, 9] [6, Sect. 6.6.1] is a flexible approach to implement SPLs. A delta-oriented SPL consists of a *feature model*, an *artifact base*, and *configuration knowledge*. The feature model provides an abstract description of variants in terms of *features*—each feature represents an abstract description of functionality and each variant is identified by a set of features, called a *product*. The artifact base provides language dependent artifacts that are used to build the variants—it consists of a *base program* (that might be empty or incomplete) and of a set of *delta modules* (*deltas* for short), which are containers of modifications to a program. For example: for Java programs, a delta can add, remove or modify classes and interfaces; for statechart programs, a delta can add, remove or modify states and transitions. Configuration knowledge connects the features in the feature model with the artifacts in the artifact base by associating to each delta an *activation condition* over the features and specifying an *application ordering* between deltas. Once a user selects a product, the corresponding variant is derived by applying the deltas with a satisfied activation condition to the base program according to the application ordering. Thus configuration knowledge defines a mapping from products to variants, and DOP supports the automatic generation of variants based on a selection of features.

The toolchain of the HyVar project [3] supports the development of delta-oriented SPLs where the variants are statecharts [21] expressed in the format supported by Yakindu Statechart Tools [5]. A Yakindu statechart comprises: an *interface definition part*, which declares the elements (e.g., events and typed operations) used by the statetechart to interact with the external environment; and a *state definition part*, which defines the structure of the statechart (i.e., a hierarchical state machine that can use the elements declared in the interface definition part). This toolchain—which provides automatic derivation of a statechart

☆This work has been partially supported by project HyVar (`www.hyvar-project.eu`, this project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 644298), by ICT COST Action IC1402 ARVI (`www.cost-arvi.eu`), and by Ateneo/CSP project RunVar (`runvar-project.di.unito.it`).

*Corresponding Author.

variant as well as C/C++ and Java code generation, linking to external code artifacts, and compilation—has been used to develop product lines of car embedded software systems [3]. The toolchain provides support for guaranteeing that all the statechart variants can be generated and are well formed. According to:

- delta-oriented programming, the generation of a variant fails when trying to apply a delta that contains an operation that cannot be executed (e.g., for an SPL of YAKINDU statecharts, trying to add an already existing event to interface definition part, or trying to remove or to modify a non existing state in the state definition part);
- YAKINDU STATECHART TOOLS, a statechart is well formed if the interface definition part is well-formed (e.g., there are no duplicated declarations) and the state definition part is well-formed, that is: (i) there are no structural flaws (like, e.g., a dangling transition); (ii) all the elements used to interact with the external environment are declared in the interface definition part; and (iii) the use of each of these elements is correct with respect to its declaration (e.g., each operation is used according to the type declared for it).

In this paper we provide a formal account of the SPL analysis technique implemented in the toolchain.

The preliminary version of the HyVar toolchain [10] did not provide any support for guaranteeing that all the variants can be generated and are well formed (i.e., each variant had to be generated in isolation and checked with YAKINDU STATECHART TOOLS). When, in the context of the HyVar project, we faced the problem of enhancing the preliminary version of the HyVar toolchain by adding support for an SPL analysis that automatically checks that all the variants can be generated and are well formed, we decided to first provide a formal account of the analysis. Namely, inspired by our recently proposed type checking approach for delta-oriented SPLs of Java-like programs [14], we:

1. defined FEATHERWEIGHT STATECHART LANGUAGE (FSL), a core textual language that captures the key ingredients of YAKINDU statecharts;[1]
2. formalized for FSL, by a means of a set of typing rules, the well-formedness checks supported by YAKINDU STATECHART TOOLS;
3. defined FEATHERWEIGHT DELTA STATECHART LANGUAGE (FΔSL), a core textual language for delta-oriented SPLs where variants are written in FSL, that captures the key ingredients of the delta operations on YAKINDU statecharts supported by the HyVar toolchain; and
4. defined a suitable version of the checking approach of [14] that applies to the formalization in points 1, 2 and 3 above.

The paper is organized as follows. Section 2 introduces FSL. Section 3 introduces FΔSL. Section 4 presents the checking approach. Section 5 illustrates how the implementation of our analysis integrated in the HyVar toolchain has been applied on the HyVar case study. Section 6 discusses related work and Section 7 concludes.

2. FSL: a featherweight language for statecharts

YAKINDU STATECHART TOOLS [5] is an integrated modeling environment based on the Eclipse IDE [1] for the construction and simulation of UML state machines (statecharts, for short) [4]. YAKINDU is able to automatically translate statecharts into C, C++ or Java source code, and allows to interface the generated code with external libraries. To support its capabilities of interfacing statecharts with external libraries, YAKINDU structures its statecharts in two parts: the *interface definition* part, where the user must declare all the elements (e.g., events and operations) used by the statechart to communicate with the external environment; and the *state definition* part, where the statechart itself is constructed.

[1] Much as Featherweight Java [22] captures the key ingredients of class-based object-oriented programming.

Interface definition part

```
interface Clock:
  event mode
  event set
  operation tick(): boolean
  operation nextSecond(): void

interface Display:
  operation hour(): void
  operation min(): void

interface Set:
  operation nextHour(): void
  operation nextMin(): void
```

State definition part

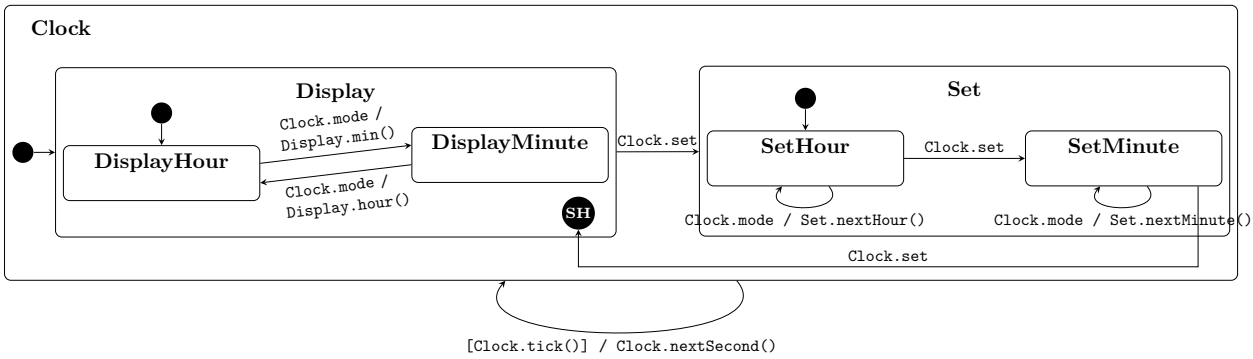


Figure 1: YAKINDU statechart describing a clock: interface definition part (top) and state definition part (bottom)

2.1. Running example: a YAKINDU *statechart describing a clock*

We illustrate the structure of a YAKINDU statechart with a simple clock example described in Figure 1.[2] The top part of the Figure consists of the interface definition part of the statechart, and is structured into three interfaces corresponding to different aspects of the clock functionality. Two kinds of elements are declared in these interfaces: *events* (declared with the keyword **event**) correspond to an external signal (possibly triggered when pressing a button); *operations* (declared with the keyword **operation**) correspond to external functions that can be called from within the statechart. The `Clock` interface declares all the elements used for the basic features of the clock: the `mode` event is used to change the display mode of the clock, while the `set` event is used to set the time of the clock; the `click` operation does not take any argument and returns **`true`** each time a second has passed and is used to make the statechart increase the time counter of the clock at each passing second using the `nextSecond` operation (which increases the time counter of the clock). The `Display` interface declares all the operations related to the clock's display: the operation `hour` changes the display of the clock to display hours and days, while the `min` operation sets the display to show minutes and hours. Finally, the `Set` interface declares all the operations related to setting the clock's time: the operation `NextHour` increments the hour counter of the clock, while the `NextMin` operation increments the minute counter of the clock.

All these events and operations declared in the interface definition part are assembled in a consistent workflow in the state definition part presented in the bottom part of Figure 1. The statechart is modeled with one *composite* main state, called Clock, containing two substates: the state Display corresponds to the

[2] In this example, as well as in the following formalization of the analysis, we do not consider for simplicity some aspects of the YAKINDU statechart model, like concurrency and in/out events. Indeed these elements do not add any value to the formalization and would only clutter the presentation. However, as pointed out in Section 5, in the implementation of our analysis integrated it in the HyVar Toolchain these elements are considered.

clock only displaying time information, while in the state Set the semantics of the events changes to allow to update the clock's time. The black circle and arrow pointing to the Display state show that it is the initial state of Clock: when switched on, the clock starts by simply showing the time. The Display state is also composite and allows to switch between two display modes: the initial DisplayHour state corresponds to the clock only displaying the hours and days, while the DisplayMinutes state corresponds to the clock displaying minutes and hours. The transition that allows to change from the DisplayHour state to the DisplayMinute state is annotated with `Clock.mode/Display.min()`, which means that this transition is taken upon the reception of the `Clock.mode` event (i.e., the `mode` event of the `Clock` interface), which also triggers the execution of the `Display.min` operation (that implements the actual update of the clock's display to the minutes and hours mode). Reciprocally, the annotation on the transition from the DisplayMinute state to the DisplayHour state means that this transition is taken upon the reception of the `Clock.mode` event, and triggers the execution of the `Display.hour` operation (that implements the actual update of the clock's display to the hours and days mode). As shown in the state definition part, it is possible to go from the Display state (or any of its inner states) to the Set state with the `Clock.set` event. Setting the time of the clock starts with setting its hour (with the SetHour state) and switched to setting the minutes upon the reception of the `Clock.set` event. Actually changing the time is done with the `Clock.mode` event that triggers the execution of the `Set.NextHour` operation in the SetHour state (thus incrementing the hour counter of the clock), and that triggers the execution of the `Set.NextMinute` operation in the SetMinute state (thus incrementing the minute counter of the clock). One exits the SetMinute state with the `Clock.set` event and goes back in the Display state, more precisely in the *shallow history* special state SH of Display. This special SH state is actually a variable that stores the last active state inside Display: e.g., if the clock was in the DisplayMinute mode before starting to modify its time, it will go back to that state after setting the time is finished. In UML and in YAKINDU, there are two kind of history: *shallow* and *deep* histories. Both are variables that store the last active state inside a composite state, but while the shallow history only stores the state directly accessible from the current state, the deep history stores the whole path to the last active state deep inside the state hierarchy. For instance, suppose the state DisplayHour is active in Figure 1: the shallow history of the state Clock would be Display, while the deep history would be DisplayHour.

Finally, let us consider the transition looping on the Clock state: this transition is annotated with `[Clock.tick()]/Clock.nextSecond()`, which means that every time the `Clock.tick` operation returns **true** (i.e., every second), the operation `Clock.nextSecond` is executed, thus incrementing the second counter of the clock.

2.2. FSL *syntax*

The FSL language provides a formalization of the core constructs of the YAKINDU statechart model. Following YAKINDU's approach, the FSL syntax is structured in two parts, as shown in the following:

$$P \quad ::= \quad ID\ SD \qquad \text{FSL program}$$

In this syntax, the P entry models a statechart and consists of two elements: ID models the interface definition part of the statechart, while SD models its state definition part (i.e., a state, possibly composite). We give in the following the abstract syntax of ID (the interface definition part) and of SD (the state definition part). Following [22], we use the overline notation for (possibly empty) sequences of elements: for instance $\overline{SD}$ stands for a sequence of state declarations. Moreover, when no confusion may arise, we identify sequences of distinct elements with sets. The empty sequence is denoted by $\emptyset$.

2.2.1. Interface definition part syntax

The abstract syntax for the FSL interface definition part follows the YAKINDU syntax:

ID	::=	$\overline{I}$	Interface Definition Part
I	::=	**interface** `I` : $\overline{D}$	Interface
D	::=	**event** `e` | **operation** `op`($\overline{T}$) : T	Declaration
T	::=	**integer** | **boolean** | **string** | **void**	Type

The interface definition part *ID* comprises a sequence of interface declarations, where interfaces have a name and a set of declared operations and events. Event declarations have the form **event** `e` where `e` is the declared event. Operation declarations have the form **operation** $\texttt{op}(\overline{T}) : T'$ where `op` is the name of the operation, $\overline{T}$ is the list of parameter types of the operation, and T' is its return type. In this core calculus, like in YAKINDU, we consider four types: **integer**, **boolean**, **string** and **void**.

2.2.2. State definition part syntax

The abstract syntax for the state definition part of the FSL language is as follows:

SD	::=	$\texttt{s}(\overline{TD}) \mid \texttt{s}(\overline{SD}, \overline{TD}, \texttt{s}')$	State (State Definition Part)
TD	::=	$\texttt{t}(event\ [guard]\ /\ action, dest)$	Transition
event	::=	`I.e` | **always**	Event
guard	::=	*exp* | **true**	Guard
action	::=	*exp* | **none**	Action
exp	::=	$\texttt{I.op}(\overline{exp})$	Expression
dest	::=	`s` | `s.SH` | `s.DH`	Destination

The state definition part *SD* comprises either a basic state or a composite state. A *basic state* is a term $\texttt{s}(\overline{TD})$ where `s` is the name of the state and $\overline{TD}$ is the set of outgoing transitions from state `s`. A *composite state* is a term $\texttt{s}(\overline{SD}, \overline{TD}, \texttt{s}')$ where $\overline{SD}$ is the set of inner states, $\overline{TD}$ is the set of outgoing transitions from that state, and $\texttt{s}'$ is the initial inner state of `s`. A *transition TD* is a term $\texttt{t}(event\ [guard]\ /\ action, dest)$ where `t` is the name of the transition, *event* is the event necessary to fire the transition, *guard* is its guard expression, evaluated every time `t` may be triggered (the transition is fired if *guard* evaluates to **true**), *action* is the action of the transition and is executed only when `t` is fired; and *dest* is the destination of the transition. An *event* is either an event declared in an interface `I.e`, or the special event **always** that always happens. A *guard* is either an expression *exp*, or the special guard **true** that always evaluates to **true**. An *action* is either an expression *exp*, or the special action **none** that represents the absence of an action. An expression *exp* is a call to an interface's operation. Finally, a *destination dest* is either a state name `s`, or the reference to an history: `s.SH` for the shallow history of the state `s` or `s.DH` for the deep history of the state `s`.

The special event **always**, the special guard **true** and the special action **none** can be encoded by the expressions `Special.always`, `Special.true()` and `Special.none()`, respectively—where `Special` is the distinguished name of a suitable interface that is assumed to be implicitly declared. Therefore, without loss of generality, in the following we will consider the following simplified syntax for transitions:

TD	::=	$\texttt{t}(\texttt{I.e}\ [exp]\ /\ exp, dest)$	Transition

Moreover, since no confusion may arise, in the code and the graphical representations in the examples we omit the special event **always**, the special guard **true** and the special action **none**. E.g., we write:

- $\texttt{t}(\texttt{I.e}, dest)$ as short for $\texttt{t}(\texttt{I.e}\ [\textbf{true}]\ /\ \textbf{none}, dest)$,
- $\texttt{t}(\texttt{I.e}\ /\ exp, dest)$ as short for $\texttt{t}(\texttt{I.e}\ [\textbf{true}]\ /\ exp, dest)$, and
- $\texttt{t}([exp]\ /\ exp, dest)$ as short for $\texttt{t}(\textbf{always}\ [exp]\ /\ exp, dest)$.

2.2.3. Running example: a FSL *statechart describing a clock*

We illustrate the FSL language by giving, in Figure 2, the FSL representation of the YAKINDU statechart described in Section 2.1. According to the FSL syntax, the statechart has a textual representation that is structured in two parts: $ID_{\texttt{clock}}$ models the interface definition part of the statechart, while SD_{Clock} models its state definition part. As FSL's syntax for the interface definition part is identical to YAKINDU's, the interface definition part of Figure 1 and $ID_{\texttt{clock}}$ are identical. Following the structure of the state definition part in Figure 1, SD_{Clock} consists of the composite state named `Clock` that contains two states

Simple Clock = $ID_{\texttt{clock}}$ SD_{Clock}

$ID_{\texttt{clock}}$ =

interface Clock:
event mode
event set
operation tick(): boolean
operation nextSecond(): void

interface Display:
operation hour(): void
operation min(): void

interface Set:
operation nextHour(): void
operation nextMin(): void

$$
\begin{array}{lcl}
SD_{\text{Clock}} & = & \mathsf{Clock}(SD_{\text{Display}}\ SD_{\text{Set}},\ \mathsf{t}_0([\mathsf{Clock.tick()}]\ /\ \mathsf{Clock.nextSecond()}, \mathsf{Clock}),\ \mathsf{Display}) \\
SD_{\text{Display}} & = & \mathsf{Display}(SD_{\text{DisplayHour}}\ SD_{\text{DisplayMinute}},\ \mathsf{t}_1(\mathsf{Clock.set}, \mathsf{Set}),\ \mathsf{DisplayHour}) \\
SD_{\text{DisplayHour}} & = & \mathsf{DisplayHour}(\mathsf{t}_2(\mathsf{Clock.mode}\ /\ \mathsf{Display.min()}, \mathsf{DisplayMinute})) \\
SD_{\text{DisplayMinute}} & = & \mathsf{DisplayMinute}(\mathsf{t}_3(\mathsf{Clock.mode}\ /\ \mathsf{Display.hour()}, \mathsf{DisplayHour})) \\
SD_{\text{Set}} & = & \mathsf{Set}(SD_{\text{SetHour}}\ SD_{\text{SetMinute}},\ \emptyset,\ \mathsf{SetHour}) \\
SD_{\text{SetHour}} & = & \mathsf{SetHour}(\mathsf{t}_4(\mathsf{Clock.set}, \mathsf{SetMinute})\ \mathsf{t}_5(\mathsf{Clock.mode}\ /\ \mathsf{Set.nextHour()}, \mathsf{SetHour})) \\
SD_{\text{SetMinute}} & = & \mathsf{SetMinute}(\mathsf{t}_6(\mathsf{Clock.set}, \mathsf{Display}.\text{SH})\ \mathsf{t}_7(\mathsf{Clock.mode}\ /\ \mathsf{Set.nextMin()}, \mathsf{SetMinute}))
\end{array}
$$

Figure 2: FSL representation of the YAKINDU statechart given in Figure 1

SD_{Display} and SD_{Set}, one looping transition $\mathtt{t}_0$ annotated with `[Clock.tick()]/Clock.nextSecond()`, and whose initial state is `Display`. The SD_{Display} state, named `Display`, is composite with two inner states $SD_{\text{DisplayHour}}$ (which is the initial state of `Display`) and $SD_{\text{DisplayMinute}}$, and one transition to the `Set` state triggered by the `Clock.set` event. Both $SD_{\text{DisplayHour}}$ and $SD_{\text{DisplayMinute}}$ are basic states, respectively named `DisplayHour` and `DisplayMinute` and respectively having a transition to each other triggered with the `Clock.mode` event and switching the clock's interface. The composite SD_{Set} state, named `Set`, contains two substates SD_{SetHour} which is the initial state of `Set`) and $SD_{\text{SetMinute}}$ and no transition. The basic state SD_{SetHour}, named `SetHour`, contains two transitions: $\mathtt{t}_4$, triggered with the `Clock.set` event, goes to the state `SetMinute`, while $\mathtt{t}_5$ triggered with the `Clock.mode` event loops on the `SetHour` state, invoking the `Set.nextHour` operation. Finally, the basic state $SD_{\text{SetMinute}}$, named `SetMinute`, contains two transitions: $\mathtt{t}_6$, triggered with the `Clock.set` event, goes to the shallow history of the `Display` state, while $\mathtt{t}_7$ triggered with the `Clock.mode` event loops on the `SetMinute` state, invoking the `Set.nextMin` operation.

Note that the names of the transitions, which are not present in Figure 1, are immaterial.

2.3. Well-formed FSL *programs*

In FSL like in YAKINDU, it is possible to define structurally erroneous statecharts, e.g., with dangling transitions or annotations using undeclared events or operations. The following well-formedness definition captures all structural errors a statechart may have, and is used in Section 4 to show that no Software Product Line validated by our analysis can generate a statechart with structural errors.

An FSL program P is *well-formed* (i.e., it contains no structural error) iff:

- every interface, event or operation is univocally identified by its name, i.e., for each interface name `I` and event `e` or operation name `op` occurring in P, there is one and only one interface declaration for `I` in P and at most one event or operation declaration for `e` or `t` in `I`.
- every state or transition is univocally identified by its name, i.e., for each state name `s` or transition name `t` occurring in P, there is one and only one state declaration for `s` or a transition declaration for `t` in P, respectively.
- for each composite state `s`, the initial state of `s` is one of its child states.
- every state whose shallow or deep history is the destination of a transition must be a composite state.

$$\text{T:Main}\ \frac{ID \vdash SD}{\vdash ID\ SD} \qquad \text{T:BState}\ \frac{ID \vdash \overline{TD}}{ID \vdash \mathtt{s}(\overline{TD})} \qquad \text{T:CState}\ \frac{ID \vdash \overline{SD} \qquad ID \vdash \overline{TD}}{ID \vdash \mathtt{s}(\overline{SD}, \overline{TD}, \mathtt{s}')}$$

$$\text{T:Transition}\ \frac{ID \vdash exp : \textbf{boolean} \qquad ID \vdash exp' : T \qquad \textbf{interface}\ \mathtt{I} : \overline{D} \in ID \qquad \textbf{event}\ \mathtt{e} \in \overline{D}}{ID \vdash \mathtt{t}(\mathtt{I.e}\ [exp]\ /\ exp', dest)}$$

$$\text{T:Exp}\ \frac{ID \vdash exp_i : T_i \qquad \textbf{interface}\ \mathtt{I} : \overline{D} \in ID \qquad \textbf{operation}\ \mathtt{op}(T_1, \ldots, T_n) : T \in \overline{D}}{ID \vdash \mathtt{I.op}(exp_1, \ldots, exp_n) : T}$$

Figure 3: Typing rules for FSL programs

- P must be well-typed according to the rules in Figure 3—where (in rules (T:BState) and (T:CState)), following [22], given an interface definition part ID and a sequence of transitions $\overline{TD} = TD_1 \cdots TD_n$ ($n \geq 0$), we write $ID \vdash \overline{TD}$ as short for $ID \vdash TD_1 \cdots ID \vdash TD_n$; and similarly (in rule (T:CState)) for sequences of states $\overline{SD}$. Namely, used events and operations must be declared; every guard must be well typed and return a Boolean; and every action must be well typed and can return anything.

3. FΔSL: delta-oriented programming for FSL

In this section we define FΔSL, a textual language for delta-oriented product lines where variants are statecharts written in FSL. This language, which formalizes the approach implemented in the HyVar toolchain, includes delta operations that can modify all elements of an FSL statechart. Our presentation of the FΔSL language is structured as follows: we first present the syntax and illustrate it on the running example (in Section 3.1); we then present its operational semantics, i.e., how a statechart variant can be generated given a FΔSL product line and a set of selected features (in Section 3.2); and finally we define some basic properties and guidelines related to the definition of FΔSL product lines (in Section 3.3).

3.1. FΔSL *Syntax*

A delta-oriented product line is structured in three parts (see the brief description given in Section 1): the feature model, the configuration knowledge and the artifact base. This structure is illustrated in the L entry of the following grammar:

L	::=	$FM \quad CK \quad AB$	Product Line
AB	::=	$P \quad \overline{\Delta}$	Artifact Base
Δ	::=	**delta** $\mathtt{d}$ { $\overline{PO}$ }	Delta
PO	::=	$IO \mid SO$	Program Operation

In this syntax, FM models the feature model that gives the features and products of this product line, while CK models the configuration knowledge that connects the features in the feature model with the artifacts in the artifact base of this product line; and AB is the artifact base, constructed from a base statechart program P (see Section 2.2) and a set of delta Δ. A delta declaration Δ comprises the name $\mathtt{d}$ of the declared delta and a set of *program operations* $\overline{PO}$ describing the transformations performed when the delta is applied to a statechart program. We have two kind of program operations: IO modifies the interface definition part of a statechart, while SO modifies its state definition part (that is a state).

3.1.1. Feature model and configuration knowledge

In this formalization we do not provide a concrete syntax for the feature model and the configuration knowledge of a FΔSL product line, and only focus on their algebraic structure. We first assume that a feature model is a pair $(\mathcal{F}, \Psi)$ giving the set $\mathcal{F}$ of the product line's features and a propositional logic formulas Ψ, where propositional variables are feature names in $\mathcal{F}$, and whose solutions are the products of the product line.[3] The syntax of the propositional logic formula Ψ is as follows:

$$\Psi \quad ::= \quad \textbf{true} \mid \varphi \mid \Psi \Rightarrow \Psi \mid \neg\Psi \mid \Psi \wedge \Psi \mid \Psi \vee \Psi$$

As usual, we say that a propositional formula Ψ is *valid* if it is true for all values of its propositional variables.

Additionally, we assume that a configuration knowledge is a pair $(<, \alpha)$ where: $<$ is a partial order between the delta names of the product line, giving the application order of the different deltas; and α maps all delta names in the product line to propositional formula Ψ, thus giving for each delta which product activates it (i.e., for which products that delta will be applied). Following [14], for a product line L we denote by:

- L.`features` is the set of features $\mathcal{F}$ of L;
- L.`products` is the propositional logic formula Ψ over feature names that specifies the products of L;
- L.`activation` is the mapping α of L's configuration knowledge (a product is generated by applying the activated deltas to the base program)—the names of deltas are unique; and
- $<_L$ is the application order $<$ of L's configuration knowledge (the order in which the activated deltas are applied does not depend on the selected product)—in order to avoid overspecification, the application order may be partial.

In the rest of the document, we assume that also delta names can be used as propositional variables, and denote these extended formulas still by Ψ. We can then define the following formula (that we call the *feature model and delta activations formula*):

$$L.\texttt{FMandDA} \triangleq L.\texttt{products} \wedge \bigwedge_{\texttt{d}} (\texttt{d} \Leftrightarrow L.\texttt{activation}(\texttt{d}))$$

This formula specifies the products of the SPLs and binds each variable `d` to the activation condition of the delta `d` (i.e., it specifies which deltas are activated for each product).[4]

3.1.2. Interface definition part delta operations syntax

The following syntax gives the delta operations that modify the interface definition part of a statechart:

$$\begin{array}{rll} IO & ::= \textbf{adds interface } I \mid \textbf{removes interface } \texttt{I} & \text{Interface Operation} \\ & \quad \mid \textbf{modifies interface } \texttt{I } \{\overline{DO}\} & \\ DO & ::= \textbf{adds } D \mid \textbf{removes } \texttt{op} \mid \textbf{removes } \texttt{e} & \text{Declaration Operation} \end{array}$$

(where the syntax of D has been given at the beginning of Section 2.2.1). The entry IO corresponds to delta operations that manipulate the interface definition part of a FSL statechart. The operation **adds interface** I adds a new interface to the statechart by providing its definition I. The operation **removes interface** `I` removes an interface by providing its name. And the operation **modifies interface** `I` $\{\overline{DO}\}$ modifies the interface `I`, i.e., it executes the operations $\overline{DO}$ inside that interface.

The entry DO introduces four operations to modify the content of an interface. The operation **adds** D adds a new event or operation D to the interface. The operation **removes** `op` removes the operation named `op` from the interface. And the operation **removes** `e` removes the event `e` from it.

[3] See, e.g., [7] for a discussion on other possible representations of a feature model.

[4] The last occurrence of `d` in L.`FMandDA` is not used as a variable: it is used as argument of the map L.`activation` to obtain the activation condition of the delta `d`.

3.1.3. State definition part delta operations syntax

The following syntax gives the different delta operations that manipulate the state definition part of a statechart (that is a state):

SO	::=	**adds** *SD* | **removes** `s` | **modifies** `s` $\{\overline{SO}\}$	State Operation
	|	**initial** `s` | **adds** *TD* | **removes** `t` | **modifies** `t` $\{\overline{TO}\}$	
TO	::=	**destination** *dest* | **event** `I.e` | **guard** *exp* | **action** *exp*	Transition Operation

A state operation *SO* executes inside a composite state and changes its content. The operation **adds** *SD* adds a new inner state by providing its definition *SD*. The operation **removes** `s` removes an inner state by providing its name. The operation **modifies** `s` $\{\overline{SO}\}$ modifies the inner composite state named `s`, i.e., it executes the operations $\overline{SO}$ inside that state. The operation **initial** `s` sets the initial state of the current composite state to be `s`. The operation **adds** *TD* adds a new outgoing transition to the current composite state by providing its definition *TD*. The operation **removes** `t` removes the outgoing transition named `t`. And **modifies** `t` $\{\overline{TO}\}$ modifies the definition of the transition `t` applying to it the operations $\overline{TO}$.

Finally, a transition operation *TO* can either: replace the destination of a transition (with the operation **destination** *dest*); replace its triggering event (with the operation **event** `I.e`); replace the guard of the transition with a new expression *exp* (with the operation **guard** *exp*); or replace its action (with the operation **action** *exp*).

3.1.4. Running example: an FΔSL *product line of clocks*

To illustrate the FΔSL language, we extend, in Figure 4, the FSL clock statechart (described in Section 2.2.3) into an FΔSL product line of clock statecharts. The feature model of this product line is illustrated with both a feature diagram [13] (that organizes the different features of the feature model in a tree structure, where optional features are annotated by the symbol "$\circ$") and the notations introduced in Section 3.1.1. The feature model (in the top left part of Figure 4) has three features: the mandatory feature `Clock`; the optional feature `DisplayMode` that allows to switch between Hour and Minute display mode; and the optional feature `Timer` that adds to the clock a timer functionality. Therefore the product line has the four products in the set:

$$\{\ \{\texttt{Clock}\},\ \{\texttt{Clock},\texttt{DisplayMode}\},\ \{\texttt{Clock},\texttt{DisplayMode},\texttt{Timer}\},\ \{\texttt{Clock},\texttt{Timer}\}\ \}$$

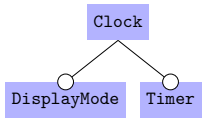
which is the subset of $2^{\{\texttt{Clock},\texttt{DisplayMode},\texttt{Timer}\}}$, described by the proposition formula `Clock`, that contains exactly the sets that contain the feature `Clock`.

The configuration knowledge (in the top right part of Figure 4) gives, using the notation introduced in Section 3.1.1, the activation condition of the four deltas in the product line, and their application order.

Finally, the artifact base of the product line consists of a base statechart (the original running example presented in Figure 2) and the four deltas presented in the bottom part of Figure 4. The deltas dSMode1 and dSMode2 remove the display mode states when the feature `DisplayMode` is not selected: namely, the delta dSMode1 removes the `Display` interface from the interface definition part and the `Display` state from the state definition part of the base statechart, while the delta dSMode2 reintroduces the `Display` state as a basic state. The delta dTimer adds the timer functionality to the base statechart: namely, it adds the `Timer` interface containing the `timer` event and four operations to set the timer and react to the timer's completion, and the composite `Timer` state that contains all the timer functionalities of the clock. The delta dSModeTimer manages the case when the `Timer` feature is selected (i.e., the `Timer` state is added) and the `DisplayMode` feature is not (i.e., the `Display` state is not a composite state anymore).

The new timer functionality is implemented as follows. First, the new interface `Timer`, declaring the operations that are available when the feature `Timer` is selected, is added. Second, the state definition part is modified. As stated with the **modifies** operation at the end of the `dTimer` delta, the new `Timer.timer` event allows to go from the `Display` state into the new `Timer` composite state. The initial state of `Timer` is `SetTimer`, and the purpose of this initial state is to configure the timer, with a looping transition triggered with the `Clock.mode` event that increases the timer counter (with the `Timer.incrTimer` operation). Once the timer is configured, one exits the `SetTimer` state and go into the `RunTimer` state with the `Timer.timer`

Feature Model:



features = {Clock, DisplayMode, Timer}
products = Clock

Configuration Knowledge:

activation(dSMode1) = ¬ DisplayMode
activation(dSMode2) = ¬ DisplayMode
activation(dTimer) = Timer
activation(dSModeTimer) = (¬ DisplayMode) ∧ Timer

dSMode1 < dSMode2 < dTimer < dSModeTimer

Deltas:

```
delta dSMode1 {
 removes interface Display;
 modifies Clock { removes Display; };
}
delta dSMode2 {
 modifies Clock {
  adds Display(t1(Clock.set, Set));
  modifies Set {
   modifies SetMinute {
    modifies t6 { destination Display }
}}}}
```

```
delta dTimer {
 adds interface Timer:
  event timer
  operation incrTimer(): void
  operation startTimer(): void
  operation isTime(): boolean
  operation ring(): void
 modifies Clock { adds Timer(
  SetTimer(t8(Clock.mode/Timer.incrTimer(),SetTimer) t9(Timer.timer/Timer.startTimer(),RunTimer))
  RunTimer(t10([Timer.isTime()] / Timer.ring(), Display.SH)), ∅, SetTimer);
 modifies Display { adds t11(Timer.timer, Timer) }}
}
```

```
delta dSModeTimer {
 modifies Clock {
  modifies Timer {
   modifies RunTimer {
    modifies t10 { destination Display }
}}}}
```

Figure 4: FΔSL product line of clocks (the base program is given in Figure 2)

event: this triggers the timer, which is started with the `Timer.startTimer` operation. Then, when the timer finishes (which is captured by the operation `Timer.isTime` returning **true**), the transition $\mathtt{t}_{10}$ is taken, which plays a ring tone (with the `Timer.ring` operation) and goes back to the shallow history of the `Display` composite state. In case the `DisplayMode` feature is not selected and the `Display` state is not composite, the delta `dSModeTimer` is activated, which changes the destination of the $\mathtt{t}_{10}$ transition to the `Display` state (indeed, `Display` does not have a shallow history when it is a base state).

3.2. FΔSL *operational semantics*

The operational semantics of the FΔSL language is given as a set of small-step reduction rules presented in Figures 5, 6 and 7. This semantics describes how to generate a variant given a specific product (i.e., a set of selected features): consequently, a reduction statement has the form $E_1 \rightarrow_p E_2$ where E_1 is the term to be reduced, p is the product for which the variant is generated, and E_2 is the reduced term.

Many of the reduction rules use two helper functions: *name* returns the name of its parameter (e.g., $name(\textbf{event e}) = \mathtt{e}$ and $name(\mathtt{s}(\overline{TD})) = \mathtt{s}$); and *names* returns the names of all the states, transitions, interfaces, operations and events listed in its (sequence) parameter (e.g., $names(\textbf{event e event e}') = \{\mathtt{e}, \mathtt{e}'\}$, $names(\mathtt{t}(\mathtt{I.e}\ [exp]\ /\ exp, dest)) = \{\mathtt{t}\}$ and $names(\emptyset) = \{\}$). Additionally, while applying the deltas on the base statecharts, we temporarily extend the syntax of a statechart to allow the application of a delta operation on an element of the statechart: for instance, $(SO\ SD)$ corresponds to the application of the SO operation on the state SD.

Figure 5 presents the main reduction rules of the FΔSL semantics. First, rule (R:FM) ensures that the specified product p is a valid product of the product line. By construction, this rule is always the first one to be applied (all the other rules expect the feature model of the product line to be absent, and this rule is the only one that removes it). Rule (R:DApp) looks for a valid delta to be applied to the base statechart (i.e., a delta that is activated by the given product, as stated with $p \vdash \alpha(\mathtt{d})$, and that is first to be applied, as stated with $\forall \mathtt{d}' \in \overline{\Delta}, \mathtt{d}' \not< \mathtt{d}$) and applies it (as stated with the application $(\overline{PO}\ P)$). Rule (R:DDrop) drops the deltas that are not activated by the product (as stated with $p \nvdash \alpha(\mathtt{d})$). Rule (R:CK) returns the generated variant when all the deltas have been applied.

Note that the rule (R:DApp) can be applied several consecutive times, generating a base statechart P of the following form:

$$(PO_1\ \ldots\ PO_n(\ldots(PO_m\ \ldots PO_l\ \ P')\ldots)) \quad \equiv \quad (PO_1\ (PO_2\ (\ldots(PO_l\ \ P')\ldots)))$$

The rest of the rules thus focus on applying in sequence each PO_i on the P' statechart, to obtain the resulting variant that do not contain any delta operation application. Rule (R:IO) applies an interface definition

$$\text{R:FM}\quad \frac{FM = (\mathcal{F}, \Psi) \qquad p \subset \mathcal{F} \qquad p \vdash \Psi}{(FM\ CK\ P\ \overline{\Delta})\ \rightarrow_p\ (CK\ P\ \overline{\Delta})}$$

$$\text{R:DApp}\quad \frac{CK = (<, \alpha) \qquad p \vdash \alpha(\mathtt{d}) \qquad \forall \mathtt{d}' \in \overline{\Delta},\ \mathtt{d}' \not< \mathtt{d}}{(CK\ P\ \textbf{delta}\ \mathtt{d}\ \{\ \overline{PO}\ \}\ \overline{\Delta})\ \rightarrow_p\ (CK\ (\overline{PO}\ P)\ \overline{\Delta})}$$

$$\text{R:DDrop}\quad \frac{CK = (<, \alpha) \qquad p \nvdash \alpha(\mathtt{d})}{(CK\ P\ \textbf{delta}\ \mathtt{d}\ \{\ \overline{PO}\ \}\ \overline{\Delta})\ \rightarrow_p\ (CK\ P\ \overline{\Delta})}$$

$$\text{R:CK}\quad (CK\ P\ \emptyset)\ \rightarrow_p\ P$$

$$\text{R:IO}\quad (IO\ (ID\ SD))\ \rightarrow_p\ ((IO\ ID)\ SD)$$

$$\text{R:SO}\quad \frac{names(SO) \cap names(SD) = \emptyset}{(SO\ (ID\ SD))\ \rightarrow_p\ (ID\ (SO\ SD))}$$

Figure 5: Main reduction rules of the operational semantics of FΔSL

$$\text{R:IAdd}\quad \frac{name(I) \notin names(\overline{I})}{(\textbf{adds interface}\ I\ \ \overline{I})\ \rightarrow_p\ (I\ \overline{I})}$$

$$\text{R:IRem}\quad \frac{name(I) = \mathtt{I}}{(\textbf{removes interface}\ \mathtt{I}\ \ (I\ \overline{I}))\ \rightarrow_p\ \overline{I}}$$

$$\text{R:IMod}\quad (\textbf{modifies interface}\ \mathtt{I}\ \{\overline{DO}\}\ \ (\textbf{interface}\ \mathtt{I} : \overline{D}\ \ \overline{I}))\ \rightarrow_p\ (\textbf{interface}\ \mathtt{I} : (\overline{DO}\ \overline{D})\ \ \overline{I})$$

$$\text{R:DAdd}\quad \frac{name(D) \notin names(\overline{D})}{(\textbf{adds}\ D\ \ \textbf{interface}\ \mathtt{I} : \overline{D})\ \rightarrow_p\ \textbf{interface}\ \mathtt{I} : (D\ \overline{D})}$$

$$\text{R:ORem}\quad \frac{name(D) = \mathtt{op}}{(\textbf{removes op}\ \ \textbf{interface}\ \mathtt{I} : (D\ \overline{D}))\ \rightarrow_p\ \textbf{interface}\ \mathtt{I} : \overline{D}}$$

$$\text{R:ERem}\quad \frac{name(D) = \mathtt{e}}{(\textbf{removes e}\ \ \textbf{interface}\ \mathtt{I} : (D\ \overline{D}))\ \rightarrow_p\ \textbf{interface}\ \mathtt{I} : \overline{D}}$$

Figure 6: Operational semantics of the interface definition part delta operations

part operation on the statechart, by applying it on its interface definition part. And rule (R:SO) applies a state operation on the statechart by applying it on its state definition part. Note however that the rule (R:SO) can be applied only if the names of the states and transitions declared in the delta operation SO do not clash with the names already declared in the statechart (as stated with $names(SO) \cap names(SD) = \emptyset$). This restriction, which reflects the restriction implemented in the toolchain [10], ensures that it never occurs that two different states or transitions in a statechart have the same name.

3.2.1. Interface definition operation semantics

Figure 6 presents the operational semantics of the interface definition part operations. Rule (R:IAdd) adds a new interface to the interface definition part, and can be applied only if the new interface is not already declared. Rule (R:IRem) removes an interface from the interface definition part. Rule (RI:Mod) modifies an interface by applying in sequence all the specified declaration operation on its content (as stated with the application $(\overline{DO}\ \overline{D})$). Rule (R:DAdd) adds a new declaration inside an interface, and can be applied only if the added element is not already declared in this interface. Finally, Rule (R:ORem) removes an operator from an interface, while rule (R:ERem) removes an event from an interface.

3.2.2. State definition part delta operation semantics

Figure 7 presents the operational semantics of the state delta operations. Rule (R:SAdd) adds a new state in a composite state. Note that it is not necessary in this rule to check that the name of the added state do not clash with an existing state, as this was checked in rule (R:SO). Rule (R:SRem) removes a state from a composite state. Rule (R:SInit) changes the initial state of a composite state. And rule (R:SMod)

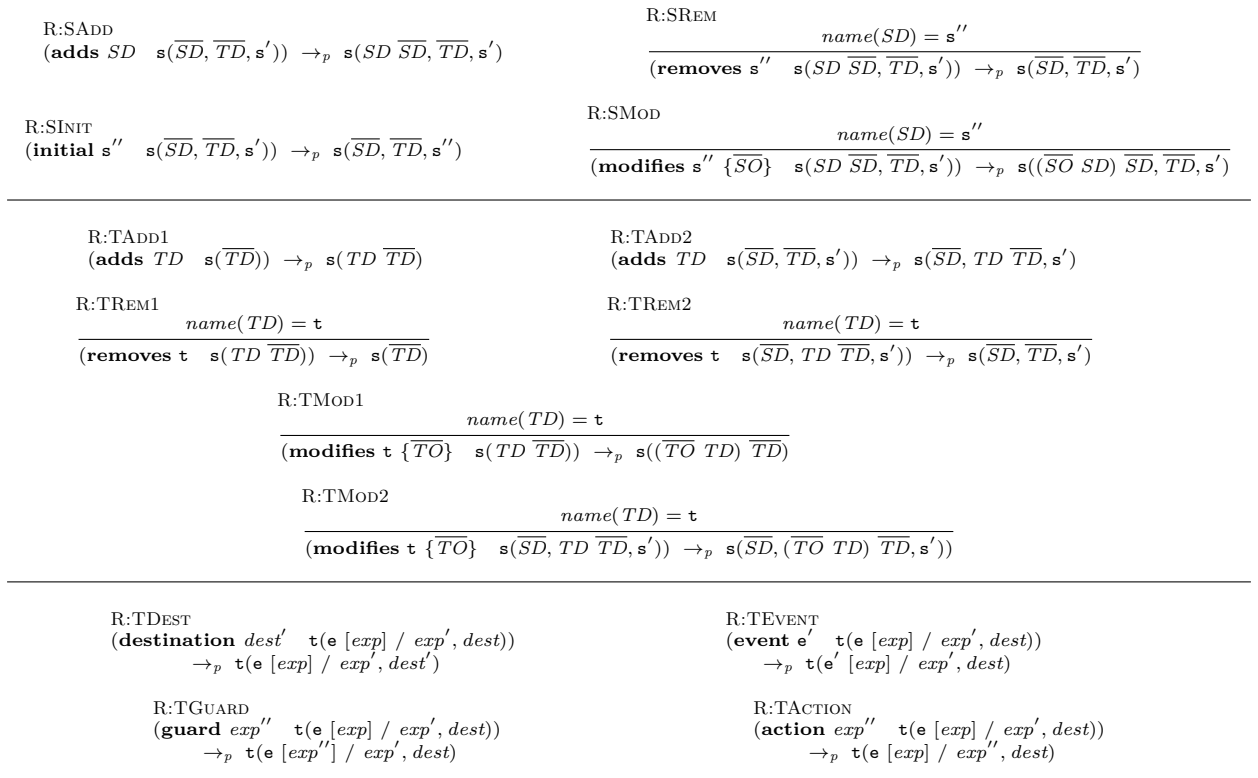


Figure 7: Operational semantics of the state definition part delta operations

modifies an inner state of a composite state, by applying in sequence all the specified state operations on it (as stated with the application $(\overline{SO}\ SD)$).

The six next rules manage the addition, removal and modification of transitions inside a state, and manage these operations in both basic and composite states. Rules (R:TAdd1) and (R:TAdd2) both add a new exiting transition to a basic and a composite state, respectively. Rules (R:TRem1) and (R:TRem2) both remove a transition from a basic and a composite state, respectively. Rules (R:TMod1) and (R:TMod2) both modify a transition, by applying in sequence all the specified transition operations on it (as stated with the application $(\overline{TO}\ TD)$), in a basic and a composite state, respectively.

Finally, the four last rules describe the effect of the transition operation on a transition. Rule (R:TDest) modifies the destination of the transition. Rule (R:TEvent) changes the event triggering the transition. Rule (R:TGuard) modifies the guard of the transition. And rule (R:TAction) modifies the action of the transition.

3.2.3. Running example: generating of a variant of the SPL of clocks

Figure 8 presents the clock statechart generated by the FΔSL product line of clocks, described in Section 3.1.4, by selecting the features `Clock` and `Timer` (i.e., the variant associated to the product $p = \{\texttt{Clock}, \texttt{Timer}\}$). To improve the readability of the statechart, we include, in the bottom part of Figure 8, the graphical representation of the state definition part of the statechart. We briefly describe how this statechart is generated. As p is a valid product of the product line (it contains the feature `Clock`), the reduction rule (R:FM) can be applied. Then, following rules (R:DApp) and (R:DDrop), we can see that all the deltas of the product line are applied to the base statechart, in the following order: first `dSMode1` is applied, then `dSMode2`, followed by `dTimer` and `dSModeTimer`.

Applying the `dSMode1` delta removes the `Display` interface (rule (R:IRem)) and the `Display` state

Timer Clock = $ID_{\texttt{tclock}}$ SD_{tClock}

$ID_{\texttt{tclock}}$ =

interface Clock:
event mode
event set
operation tick(): boolean
operation nextSecond(): void

interface Set:
operation nextHour(): void
operation nextMin(): void

interface Timer:
event timer
operation incrTimer(): void
operation startTimer(): void
operation isTime(): boolean
operation ring(): void

SD_{tClock} = Clock(SD_{Display} SD_{Set} SD_{Timer}, ∅, Display)
SD_{Display} = Display(t_1(Clock.set, Set) t_{11}(Timer.timer, Timer))
SD_{Set} = Set(SD_{SetHour} $SD_{\text{SetMinute}}$, ∅, SetHour)
SD_{SetHour} = SetHour(t_4(Clock.set, SetMinute) t_5(Clock.mode / Set.nextHour(), SetHour))
$SD_{\text{SetMinute}}$ = SetMinute(t_6(Clock.set, Display) t_7(Clock.mode / Set.nextMin(), SetHour))
SD_{Timer} = Timer(SD_{SetTimer} SD_{RunTimer}, ∅, SetTimer)
SD_{SetTimer} = SetTimer(t_8(Clock.mode / Timer.incrTimer(), SetTimer) t_9(Timer.timer / Timer.startTimer(), RunTimer))
SD_{RunTimer} = RunTimer(t_{10}([Timer.isTime()] / Timer.ring(), Display))

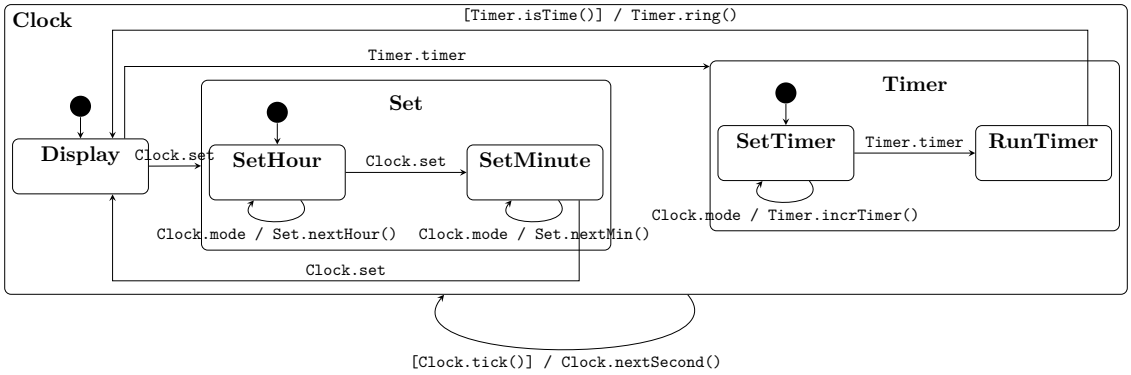


Figure 8: Variant associated to product {Clock, Timer} of the FΔSL product line given in Figure 4

(rule (R:SREM)), which temporarily invalidates it as the initial state of the `Clock` composite state and its shallow history as the destination of the t_6 transition. Then the delta `dSMode2` is applied, which recreate the `Display` state and redirects the destination of the t_6 transition towards `Display` (rules (R:SMOD) and (R:TDEST)). At that point, the statechart is thus well-formed again. Applying the `dTimer` delta adds the `Timer` interface (rule (T:IADD)), the `Timer` state (rule (R:SADD)), and a transition t_{11} going from `Display` to the new `Timer` state (rules (R:SMOD) and (R:TADD1)). Note that the resulting statechart is not well-formed, as the t_{10} transition points to the shallow history of `Display`, which is not a composite state anymore. Finally, the delta `dSModeTimer` changes the destination of the t_{10} transition towards `Display`. The resulting statechart is thus well-formed.

3.3. Pre-well-formed FΔSL *product lines*

In this section we introduce some basic properties of FΔSL product lines that are necessary to prove the correctness of the analysis described in Section 4.

An FSL statechart P is *pre-well-formed* iff for each state name `s` or transition name `t` occurring in P, there is at most one state declaration SD for `s` or transition declaration TD for `t` in P, respectively.

A delta Δ is *pre-well-formed* iff: for each state name `s` occurring in Δ, there is at most one operation for `s` (i.e., **adds** SD with `s` ∈ *names*(SD), **removes** `s` or **modifies** `s`) in Δ; and for each transition name `t` occurring in Δ, there is at most one operation for `t` (i.e., **adds** SD with `t` ∈ *names*(SD), **adds** TD with `t` = *name*(TD), **removes** `t` or **modifies** `t`) in Δ.

An FΔSL product line L is *unambiguous* if and only if: for each of its product, any total ordering of the activated deltas that respects $<_L$ generates the same variant. Under certain conditions [24, 9] this property can be checked efficiently. For instance, if the application ordering is total, then no check at all is needed. Another notable case is when the application ordering is specified by means of a totally ordered partition of the set of deltas.[5] In this case it is enough to check that the deltas in each set S of the partition are pairwise *disjoint* (that is: if one delta adds or removes a state or transition, then the other does not modify the same state or transition; and the two deltas do not modify the initial state of the same composite state, or the destination, event, guard or action of the same transition)–which can be done with linear complexity.

Finally, an FΔSL product line is *pre-well-formed* if and only if: its base program and its deltas are pre-well-formed, and it is unambiguous. For instance, the FΔSL product line of clocks introduced in Section 3.1.4 is pre-well formed.

According the above explanations, the fact that an FΔSL product line is pre-well-formed is straightforwardly checkable in linear time by traversing the product line syntax. In the rest of the document, we always assume that the FSL statecharts and the FΔSL product lines are pre-well-formed.

3.4. FΔSL *programming guidelines*

We conclude the presentation of the FΔSL language with two programming guidelines (first introduced for delta-oriented SPLs of Java-like programs in [14]—see also [15, 16]) that are useful to keep a product line maintainable. Moreover, these two guidelines are necessary to ensure the completeness of our analysis, and so we suppose in the rest of this document that all the FΔSL product lines follow them.

3.4.1. No useless declarations

A *useless declaration* in a product line L is a part of its base program, or a term in an **adds** or **modifies** operation in one of its delta that introduces declarations that are never present in any of the product line's variants (due to some **removes** or **modifies** operation that removes or changes that declaration, or to the delta containing this declaration never being activated). Such useless declarations should be avoided as they clutter a product line's code. Moreover, such declarations could be ill-formed, which would be detected by an analysis tool and could be reported as a possible error in the product line (whereas no variant of the product line would actually contain that error).

We thus have the following guideline:

G1 Ensure that the product line does not contain useless operations.

We now give a formal definition of useless declaration (which makes guideline **G1** automatically checkable). For simplicity, this definition abstracts the base program of a FΔSL product line L as a delta that is always activated and always applied first. Namely, we write `p` the name of the delta corresponding to the base program, and consider that $L.\texttt{activation}(\texttt{p}) = \textbf{true}$ and $\texttt{p} <_L \texttt{d}$ for all delta name `d` in L.

Definition 1 (Useless Declaration in FΔSL)**.** *The declaration, addition or modification of an element N (either an interface* `I`*, an operation* `op`*, an event* `e`*, a state* `s` *or a transition* `t`*) in a delta* `d` *is* useless *if the formula* $(L.\texttt{FMandDA} \wedge \texttt{d}) \Rightarrow \bigvee_{\texttt{d}' \in R} \texttt{d}'$*, where* $R = \{\texttt{d}' \mid \texttt{d}'$ *removes N or modifies N by replacing one of its components (i.e., an initial state, a destination state, an event, a guard, or an action) and* $\texttt{d} <_L \texttt{d}'$*, is valid*$\}$*. A delta* `d` *is* useless *if* $L.\texttt{products} \Rightarrow \neg L.\texttt{activation}(\texttt{d})$ *is valid. An element N in a delta* `d` *is a* useless declaration *iff either* `d` *is useless, either the declaration, addition or modification of N in* `d` *is useless.*

For instance, the FΔSL product line of clocks introduced in Section 3.1.4 does not contain useless operations.

[5] That is, given an ordered sequence of pairwise disjoints sets $S_1, ..., S_n$ such that $\cup_{1 \leq i \leq n} S_i$ is the set of deltas, $\texttt{d} <_L \texttt{d}'$ holds if and only if: $\texttt{d}' \in S_i$, $\texttt{d}' \in S_j$ and $i < j$.

3.4.2. Type uniformity

Type uniformity in a FΔSL product line L corresponds to the fact that two different declarations of the same operation in the same interface (but in two different deltas) must have the same type. This property helps developing and maintaining a product line, as in every parts of the product line, an operation will always have the same type.

The guideline can thus be stated as:

G2 Ensure that the product line is type uniform.

Type-uniformity for FΔSL product lines can be formalized as follows (which makes guideline **G2** automatically checkable):

Definition 2 (Type declaration table, type uniform declaration table and type uniform FΔSL SPL)**.** *A* declaration table *DT is a mapping from pairs* `I.op` *to non-empty sets of type signatures* $(\overline{T}) : T$*. The declaration table of an* FΔSL *product line L contains, for all interfaces* `I` *declared in L and all operations* `op` *declared in this interface, an entry mapping* `I.op` *to all the declared types of that operation in L. A declaration table DT is* type uniform *iff for all* `I.op` *in* dom(*DT*)*, the set DT*(`I.op`) *is a singleton. A* FΔSL *product line L is* type uniform *iff its declaration table is type uniform.*

For instance, the FΔSL product line of clocks introduced in Section 3.1.4 is type uniform.

Type-uniformity allows to modularize the type checking of expressions *exp* in two phases: a first phase, called *partial typing*, that operates on the artifact base only to build the declaration table and to check that the operations are used correctly with respect to their declarations; and a second phase that performs a dependency check, subsumed by a validity analysis similar to the one presented in Section 4.

4. Checking well-formedness of FΔSL product lines

In this section, we present an SPL analysis technique [38] that checks that a FΔSL product line L is *well-formed*, i.e., all its variants can be generated, and all of them are well-formed. This technique is inspired by the technique described in [14] for Java-like programs. It is structured in four different steps. The first and simpler step, called *partial typing*, ensures that L is type uniform and that all the expressions *exp* in L are well typed.[6]

The second step, called *information extraction*, extracts from L the necessary information to perform the rest of the analysis. This step is quite delicate, as many subtle details are important to ensure that L is well-formed, and all of them need to be captured during our extraction.

The third step, called *applicability consistency*, uses the information collected in the previous step to generate a propositional formula that encodes the fact that all the variants of L can be generated. Namely, this generated formula is valid iff all L's variants can be generated.

The fourth and last step, called *dependency consistency*, again uses the information collected in the second step to generate a propositional formula that encodes the fact that all of L's variants that can be generated have *no dependency error*: it ensures for instance that all the operations and events used in the variant's transitions are declared, or that the initial states of all the composite states in the variant do exist and are direct children of their respective composite states. Consider a variant of L: this property, which ensures among other that all used operations are declared, together with the partial typing property, which ensures that all the used operations are used correctly w.r.t. their types, imply that the variant is well-formed.

We present in the rest of this section the different steps of our analysis (in Sections 4.1, 4.2, 4.3 and 4.4, respectively), and conclude with the discussion of its properties (in Section 4.5).

[6] As discussed in [14], it is possible to extend our analysis to also accept non type uniform product lines. However, this would make the formalization of our analysis, as well as its implementation, more complex. Moreover, type uniformity is per se a desiderable property, since it makes an SPL more comprehensible.

4.1. Partial typing

The partial typing part of the analysis checks that all the expressions *exp* in a product line L are well typed. Additionally, partial typing also ensures that the guards in L always return a Boolean.

This property can simply be checked by type-checking every expression in L w.r.t. the declaration table of L. More precisely, we say that a product line L is *partially typed*, written $pt(L)$, iff every transition *TD*, every operation **guard** *exp* that replaces the guard of a transition, and every operation **action** *exp* that changes the action of a transition in L is well typed w.r.t. the following rules (where DT is the declaration table of L):

$$\text{PT:T}\quad \frac{\vdash exp : \textbf{boolean} \qquad \vdash exp' : T}{\vdash \texttt{t(I.e}\ [exp]\ /\ exp', dest)} \qquad \text{PT:G}\quad \frac{\vdash exp : \textbf{boolean}}{\vdash \textbf{guard}\ exp} \qquad \text{PT:A}\quad \frac{\vdash exp : T}{\vdash \textbf{action}\ exp}$$

$$\text{PT:E}\quad \frac{\vdash exp_i : T_i \qquad DT(\texttt{I.op}) = \{(T_1, \ldots, T_n) : T\}}{\vdash \texttt{I.op}(exp_1, \ldots, exp_n) : T}$$

Note that the rule (PT:E) requires that the operation `I.op` has only one type. Hence, to ensure that the analysis performed by the partial typing works as intended, the analysed product line must be type uniform.

4.1.1. Running example: partial typing of the product line of clocks

Consider the FΔSL product line of clocks described in Section 3.1.4. Any operation in this product line is declared only once: hence this product line is type uniform. Its declaration table DT is as follows:

DT(Clock.tick) = {(∅) → **boolean**}
DT(Clock.nextSecond) = {(∅) → **void**}
DT(Display.hour) = {(∅) → **void**}
DT(Display.min) = {(∅) → **void**}
DT(Set.nextHour) = {(∅) → **void**}
DT(Set.nextMin) = {(∅) → **void**}
DT(Timer.incrTimer) = {(∅) → **void**}
DT(Timer.startTimer) = {(∅) → **void**}
DT(Timer.isTime) = {(∅) → **boolean**}
DT(Timer.ring) = {(∅) → **void**}

To check if the running example is partially typed, let first consider the expressions used in the guards and actions of its transitions. One can note that all these expressions are constituted from a unique call to an operation declared in DT, without any parameter. Hence, these expressions are well typed w.r.t. the rules for partial typing. Finally, the running example contains two guards: `Clock.tick()` on the transition $\texttt{t}_0$ and `Timer.isTime()` on the transition $\texttt{t}_{10}$. Both these operations return a Boolean: we can then conclude that the running example is partially typed.

4.2. Information extraction

The goal of this step is to provide an easy to use and uniform API to access all information related to how a product line L generates its variants. In this API, state, transition, interfaces, events and operations are manipulated in a uniform manner: in the rest of the document we use the term *element* (and the symbol η) to refer to any of these structures, and the term *reference* (and the symbol ρ) to refer to a name of any of these structures.

4.2.1. Structure of the API

The API is structured in three mappings that map references to different kinds of information. The first mapping, called *fadds*, gives information about the declared or added elements. It maps every reference ρ declared in a FΔSL product line L to a set of tuple $(\mathtt{d}, \overline{\rho}, \mathtt{b}, \delta)$, one per element η named ρ in L, where:

- `d` is the delta that adds η. This information is necessary to compute the set of variants in which that element is present.

- $\overline{\rho}$ is the path (i.e., the names of the elements containing η) where `d` adds η. This information is important for two reasons: it allows to ensure that the initial state of a composite state `s` is indeed a direct child of that state; and because removing an element removes all its content, it is necessary to compute the set of variants in which that element is present. As usual, we write $\overline{\rho}'\,\overline{\rho}''$ to denote the concatenation of the paths $\overline{\rho}'$ and $\overline{\rho}''$, and we say that a path $\overline{\rho}'$ is a *prefix* of a path $\overline{\rho}$ to mean that there exists a path $\overline{\rho}''$ such that $\overline{\rho} = \overline{\rho}'\,\overline{\rho}''$.
- $\mathfrak{b}$ is a Boolean stating if η is a composite state or not. This information is important, as shallow and deep histories are valid destination only in composite state. Moreover, state modification operation can only be applied on composite states.
- and δ is the *dependency* of η.

The dependency of an element η is a mapping giving, for every *dependency slot* of η, the set of *dependency elements* that slot has. For instance, every transition has four dependency slots: `event` gives which event is used as trigger by the transition; `guard` gives the set of operations used in the guard expression of the transition; `action` gives the set of operations used in the action expression of the transition; and `exit` gives the name of the state of the destination of the transition, and a Boolean stating if that state must be composite or not (i.e., if the actual destination of the transition is an history of that state). Additionally, a composite state has one dependency slot, `initial`, that gives the name of its initial state, and the path in which that initial state must be. Other kind of elements (like interfaces or events) do not have dependencies. More formally, a dependency slot is an element of the set $\Phi \triangleq \{\texttt{event}, \texttt{guard}, \texttt{action}, \texttt{exit}, \texttt{initial}\}$; and a set of dependency elements, that unifies the different kind of dependencies we just presented, is a set of triplets $(\rho, \omega, \varsigma)$ where:

- ρ is the target of the dependency;
- ω is either the path in which ρ must be declared, or $\bot$ if there is no restriction on where ρ is located;
- and ς is either **true** if ρ must be a composite state, **false** if it must be a basic state, or $\bot$ if it can be any kind of element.

To illustrate our notion of dependency, consider the transition $\mathsf{t}_5(\mathsf{Clock.mode}\ /\ \mathsf{Set.nextHour}(), \mathsf{SetHour})$ of the FSL clock statechart given in Figure 2. Its dependency is the following mapping:

$$[\ \texttt{event} \mapsto \{(\mathsf{Clock.mode}, \mathsf{Clock}, \bot)\}\ ,\ \texttt{guard} \mapsto \emptyset\ ,\ \texttt{action} \mapsto \{(\mathsf{Set.nextHour}, \mathsf{Set}, \bot)\}\ ,\ \texttt{exit} \mapsto \{(\mathsf{SetHour}, \bot, \bot)\}\ ]$$

Moreover, the dependency of the transition $\mathsf{t}_{10}([\mathsf{Timer.isTime}()]\ /\ \mathsf{Timer.ring}(), \mathsf{Display.SH})$ is as follows:

$$[\ \texttt{event} \mapsto \emptyset\ ,\ \texttt{guard} \mapsto \{(\mathsf{Timer.isTime}, \mathsf{Timer}, \bot)\}\ ,\ \texttt{action} \mapsto \{(\mathsf{Timer.ring}, \mathsf{Timer}, \bot)\}\ ,\ \texttt{exit} \mapsto \{(\mathsf{Display}, \bot, \mathbf{true})\}\ ]$$

The second mapping, called *fremoves*, gives information about the removed elements. It maps every reference ρ declared in a FΔSL product line L to a set of pairs $(\mathtt{d}, \overline{\rho})$, one per element η named ρ removed in L, where: `d` is the delta that removes η, and $\overline{\rho}$ is the path followed by `d` to remove η.

Finally, the third mapping, called *fmodifies*, gives information about the modified elements. This mapping is similar to *fadds*: it maps references to a set of tuples $(\mathtt{d}, \overline{\rho}, \varsigma, \delta)$, one per element η named ρ modified in L, where: `d` is a delta that modifies η, $\overline{\rho}$ is the path where `d` modifies η, ς states if η must be a composite state or not, and δ is a *dependency update*, i.e., a mapping that gives which dependency slots of η have been updated during this modification, together with their new values. Note that a dependency update is not a completely new dependency that replaces the old one: indeed some updating operators, like **event** `e`, only changes the value of one dependency slot, the other slots being left unchanged by the modification operation.

$$\text{F:Main}\quad \frac{\vdash P : (fadds_1, fremoves_1, fmodifies_1) \qquad \vdash \Delta_i : (fadds_i, fremoves_i, fmodifies_i)}{\vdash (FM\ CK\ P\ \Delta_2\ \ldots\ \Delta_n) : (\bigcup_i fadds_i, \bigcup_i fremoves_i, \bigcup_i fmodifies_i)}$$

$$\text{F:Base}\quad \frac{\mathtt{p}, \emptyset \vdash ID : fadds_1 \qquad \mathtt{p}, \emptyset \vdash SD : fadds_2}{\vdash ID\ SD : (fadds_1 \cup fadds_2, \emptyset, \emptyset)}$$

$$\text{F:Delta}\quad \frac{\mathtt{d}, \emptyset \vdash PO_i : (fadds_i, fremoves_i, fmodifies_i)}{\vdash \textbf{delta}\ \mathtt{d}\ \{\ PO_1\ \ldots\ PO_n\ \} : (\bigcup_i fadds_i, \bigcup_i fremoves_i, \bigcup_i fmodifies_i)}$$

Figure 9: Main rules of the information extraction algorithm

4.2.2. The information extraction algorithm

The information extraction algorithm is defined as a set of reduction rules that takes as input a FΔSL product line L, and returns a tuple containing the three mappings of the API with all the information related to the added, removed and modified elements in L.

The rules used to extract the three mapping from a L product line are presented in Figures 9, 10 and 11. These rules use three kind of statements. The first kind of statement has the form $\vdash E : (fadds, fremoves, fmodifies)$ and is used when E is a main construction in a FΔSL product line (e.g., a base program, a delta or the product line itself). In that statement, $(fadds, fremoves, fmodifies)$ is the tuple extracted from E that contains the three mappings of our API.

The second kind of statement has the form $\mathtt{d}, \overline{\rho} \vdash E : fadds$ and is used when E is an element (e.g., an interface, a transition). In that statement, $\mathtt{d}$ is the name of the delta adding E (or $\mathtt{p}$ if E is declared in the base program), $\overline{\rho}$ is the path in which E is added and $fadds$ if the extracted mapping containing all the information about the elements declared in E.

Finally, the third kind of statement has the form $\mathtt{d}, \overline{\rho} \vdash E : (fadds, fremoves, fmodifies)$ and is used when E is a delta operation. In that statement, $\mathtt{d}$ is the name of the delta containing the E operation, $\overline{\rho}$ is the path in which E is executed, and $(fadds, fremoves, fmodifies)$ is the tuple extracted from E that contains the three mappings of our API.

Figure 9 presents the main rules of our information extraction algorithm. Rule (F:Main) is the entry point of the algorithm: it takes as input a product line L, extract the mappings from its base program and its deltas, combines them and returns the obtained tuple. Mapping combination, written with the $\cup$ operator, is defined as follows:

$$\mathrm{dom}(\mathtt{M}_1 \cup \mathtt{M}_2) = \mathrm{dom}(\mathtt{M}_1) \cup \mathrm{dom}(\mathtt{M}_2) \qquad \text{and} \qquad (\mathtt{M}_1 \cup \mathtt{M}_2)(\rho) = \begin{cases} \mathtt{M}_1(\rho) & \text{if } \rho \in \mathrm{dom}(\mathtt{M}_1) \setminus \mathrm{dom}(\mathtt{M}_2) \\ \mathtt{M}_2(\rho) & \text{if } \rho \in \mathrm{dom}(\mathtt{M}_2) \setminus \mathrm{dom}(\mathtt{M}_1) \\ \mathtt{M}_1(\rho) \cup \mathtt{M}_2(\rho) & \text{if } \rho \in \mathrm{dom}(\mathtt{M}_1) \cap \mathrm{dom}(\mathtt{M}_2) \end{cases}$$

This definition of mapping combination allows for different deltas to add, remove or modify the same reference, and retain the information related to these operations. Rule (F:Base) constructs the three API mappings from a FSL statechart as follows: its *fadds* mapping is the result of the combination of the information extracted from the interface definition part and the state definition part; the *fremoves* and *fmodifies* mappings of the statechart are empty, as it contains no **removes** nor **modifies** operations. Finally, rule (F:Delta) constructs the three mappings of a delta by combining the mappings obtained from its operations.

Complexity of generating the mappings *fadds*, *fremoves* and *fmodifies* is linear (it's a direct traversal of the SPL syntax), while the complexity and length of the generated formula is polynomial (as we need to consider the interaction between several deltas).

4.2.2.1 Information extraction for the interface definition part and the delta operations. Figure 10 presents how the three API mappings are extracted from the interface definition part of a statechart, and from the delta operations related to this part. Rule (F:D) constructs the *fadds* mapping of an interface definition part by combining the mappings obtained from all the interfaces it contains. Rule (F:I) constructs the

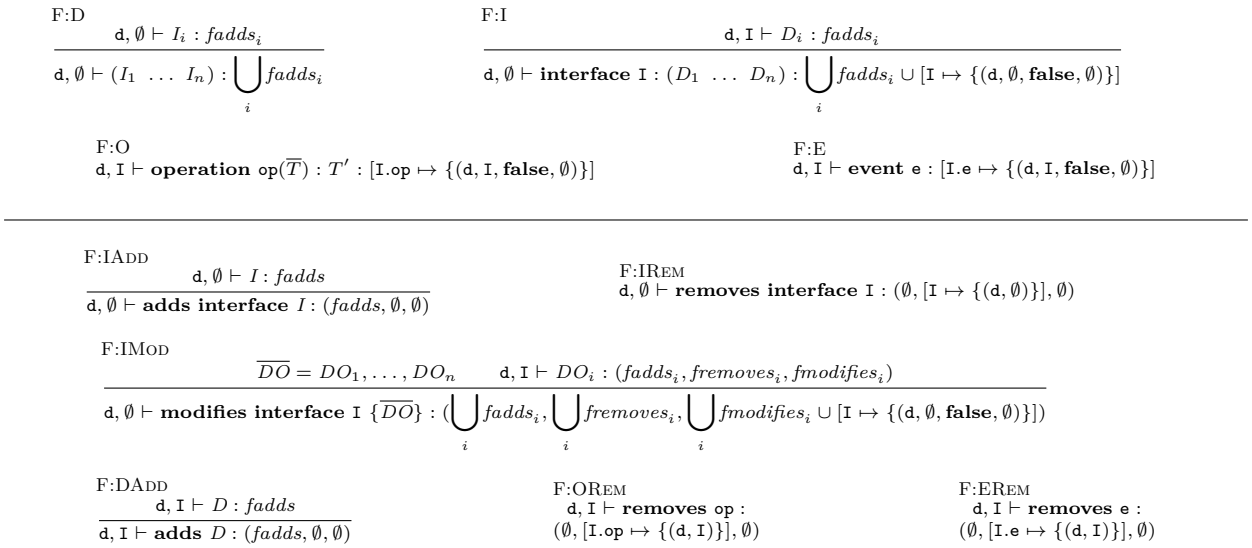


Figure 10: Information extraction for the interface definition part and the delta operations

fadds mapping of an interface `I` by combining the mappings obtained from all the declaration it contains, and adding to the result the information that `I` is declared: `I` is mapped to a set containing one tuple $(\mathtt{d}, \emptyset, \textbf{false}, \emptyset)$ stating that: `I` is added by the delta `d` (given in parameter of this rule), `I` is added at the root level of the statechart (like every interfaces), `I` is not a composite state (it is indeed an interface), and `I` does not have any dependency. Rule (F:O) considers the declaration of an operation `op` in the interface `I` in a delta `d` (given in parameter of the rule), and associates to it the *fadds* mapping with one entry mapping `I.op` to the singleton $\{(\mathtt{d}, \mathtt{I}, \textbf{false}, \emptyset)\}$ stating that: the delta `d` adds the operation `I.op`, this operation is declared inside the interface `I`, this operation is not a composite state, and has no dependency. Rule (F:E) considers the declaration of an event `e` in the interface `I` in a delta `d`, and is almost identical to the rule for operation declarations.

Rule (F:IAdd) constructs the three API mappings for the delta operation that adds an interface: the *fadds* mapping is the one obtained from the added interface, and the two other mappings are empty. Rule (F:IRem) constructs the three API mappings for the delta operation that removes an interface: the *fadds* and *fmodifies* mappings are empty, while the *fremoves* mapping contains one entry for the removed interface, stating that this interface is removed by the delta `d`, and is removed at the root location of the statechart. Rule (F:IMod) constructs the three API mappings for the delta operation that modifies an interface: it collects and combines the mappings obtained by all the operations DO_i that are applied to the interface's content. Moreover, it adds an entry in the *fmodifies* mapping stating that the interface is modified as described in the tuple $(\mathtt{d}, \emptyset, \bot, \emptyset)$: it is modified by the delta `d` at the root location of the statechart, it must not be a composite state (it is an interface) and none of its dependencies are changed.

Rule (F:DAdd) constructs the three API mapping for the delta operation that adds a declaration to an interface: the *fadds* mapping is the one obtained from the added operation, and the two other are empty. Rule (F:ORem) constructs the mappings for the delta operation that removes an operation from an interface: its *fadds* and *fmodifies* mappings are empty, and its *fremoves* mapping has one entry stating that the `I.op` reference, located in the interface `I`, is removed by the delta `d`. Finally, rule (F:ERem) constructs the mappings for the delta operation that removes an event from an interface, and is almost identical to the rule for operation removal.

4.2.2.2 Information extraction for the state definition part and the delta operations. Figure 11 presents how the three API mappings are extracted from the state definition of a statechart, and from the delta operations

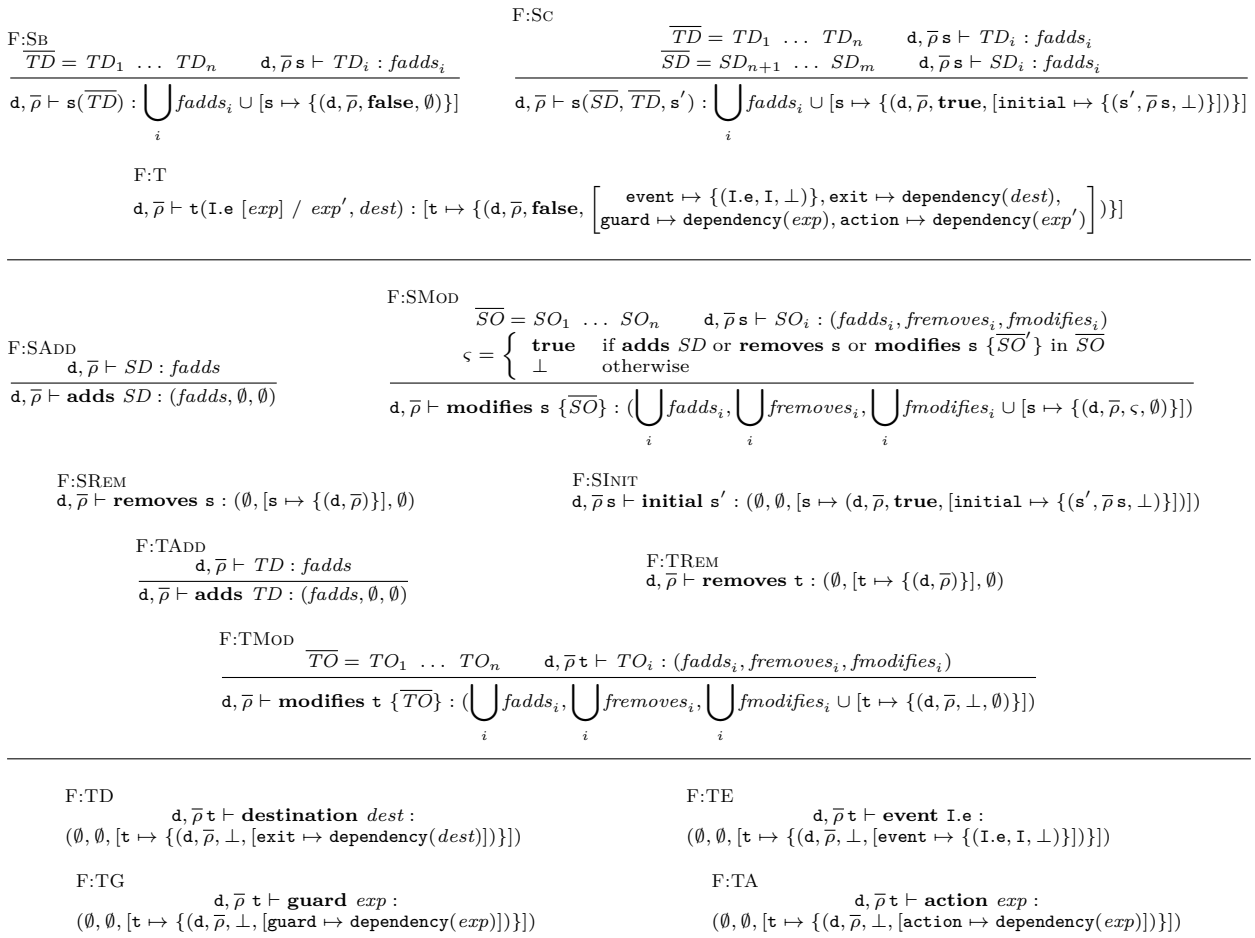


Figure 11: Information extraction for the state definition part and the delta operations

related to this part—the application of the extraction procedure to one of the deltas in the running example is illustrated Section 4.2.3. Rule (F:SB) constructs the *fadds* mapping corresponding to a basic state named `s` and declared in the location $\overline{\rho}$ by the delta `d`: it simply collects the *fadds* mapping obtained from the exiting transitions of that state (stating that these transitions are added by the delta `d` in the location $\overline{\rho}\,\mathtt{s}$) and returns the combination of these mappings, with an additional entry stating that also `s` is added by the delta `d` in the location $\overline{\rho}$, that it is not a composite state and has no dependency. Rule (F:SC) constructs the *fadds* mapping corresponding to a composite state named `s` and declared in the location $\overline{\rho}$ by the delta `d`: it simply collects the *fadds* mapping obtained from the exiting transitions and inner states of that state (stating that these transitions and states are added by the delta `d` in the location $\overline{\rho}\,\mathtt{s}$) and returns the combination of these mappings, with an additional entry stating that also `s` is added by the delta `d` in the location $\overline{\rho}$, that it is a composite state and has one dependency for its initial state. This dependency has thus one entry, the dependency slot `initial` which is mapped to the singleton $\{(\mathtt{s}', \overline{\rho}\,\mathtt{s}, \bot)\}$ encoding the fact that the initial state of `s` is `s'`, that it must be located in the path $\overline{\rho}\,\mathtt{s}$ and that it can be either a basic or composite state. Rule (F:T) constructs the *fadds* mapping corresponding to a transition named `t` and declared in the location $\overline{\rho}$ by the delta `d`: the returned mapping has one entry for `t`, which states that this transition is added by the delta `d` in the location $\overline{\rho}$, that it is not a composite state, and have dependencies for its `event`, `guard`, `action`, and `exit` slots. The `event` slot is mapped to the singleton $\{(\mathtt{I.e}, \mathtt{I}, \bot)\}$ that simply states that the event `I.e` must exist. The values for the other slots are a bit more

complex to construct, and we use the following `dependency` function to define them:

$$\begin{cases} \texttt{dependency}(\texttt{s}) = \{(\texttt{s}, \bot, \bot)\} \\ \texttt{dependency}(\texttt{s.SH}) = \texttt{dependency}(\texttt{s.DH}) = \{(\texttt{s}, \bot, \textbf{true})\} \\ \texttt{dependency}(\texttt{I.op}(exp_1, \ldots, exp_n)) = (\bigcup_i \texttt{dependency}(exp_i)) \cup \{(\texttt{I.op}, \texttt{I}, \bot)\} \end{cases}$$

The dependency corresponding to a simple destination `s` is the name of that state, which can be placed at any location and either basic or composite; the dependency corresponding to a destination with history (either `s.SH` or `s.DH`) is the name of the destination state which must be composite; and the dependency corresponding to an expression *exp* is the collection of all the operations used in that expression.

Rule (F:SAdd) constructs the three API mappings for the delta operation that adds a state: the *fadds* mapping is the one obtained from the added state, and the two other mappings are empty. Rule (F:SMod) constructs the three API mappings for the delta operation that modifies an state: it collects and combines the mappings obtained by all the operations SO_i that are applied to the state's content. Moreover, it adds an entry in the *fmodifies* mapping stating that the state is modified as described in the tuple $(\texttt{d}, \overline{\rho}, \varsigma, \emptyset)$: it is modified by the delta `d` at the location $\overline{\rho}$, it must be a composite state only if some of the operations SO_i manipulate data that are available only in a composite state (like child states), and none of its dependencies are changed. Rule (F:SRem) constructs the three API mappings for the delta operation that removes a state: the *fadds* and *fmodifies* mappings are empty, while the *fremoves* mapping contains one entry for the removed state, stating that this state is removed by the delta `d`, and is removed at the $\overline{\rho}$ location of the statechart. Rule (F:SInit) constructs the three API mappings for the delta operation that changes a state's initial state: the *fadds* and *fremoves* mappings are empty, while the *fmodifies* mapping contains one entry for the modified state, stating that this state is changed by the delta `d`, that it must be a composite state, and that its dependency is changed with a new initial state. Rule (F:TAdd) constructs the three API mappings for the delta operation that adds a transition: the *fadds* mapping is the one obtained from the added transition, and the two other mappings are empty. Rule (F:TRem) constructs the three API mappings for the delta operation that removes a transition: the *fadds* and *fmodifies* mappings are empty, while the *fremoves* mapping contains one entry for the removed transition, stating that this transition is removed by the delta `d`, and is removed at the location $\overline{\rho}$ of the statechart. Rule (F:TMod) constructs the three API mappings for the delta operation that modifies a transition: it collects and combines the mappings obtained by all the operations TO_i that are applied to the transition's content. Moreover, it adds an entry in the *fmodifies* mapping stating that the transition is modified as described in the tuple $(\texttt{d}, \overline{\rho}, \bot, \emptyset)$: it is modified by the delta `d` at the location $\overline{\rho}$, it must not be a composite state (it is a transition) and none of its dependencies are changed by this operation.

Rule (F:TD) constructs the three API mapping for the delta operation that updates the destination of a transition `t`: the *fadds* and *fremoves* mappings are empty, and the *fmodifies* mapping has one entry stating that the transition is modified by the delta `d` at the location $\overline{\rho}$, and that its dependency is updated with a new value for its `exit` dependency slot. Rule (F:TE) constructs the three API mapping for the delta operation that updates the event of a transition `t`: it is very similar to the rule (F:TD), with the difference that the updated dependency slot of the transition's dependency is `event`. Rule (F:TG) constructs the three API mapping for the delta operation that updates the guard of a transition `t`: it is very similar to the rule (F:TD), with the difference that the updated dependency slot of the transition's dependency is `guard`. Finally, rule (F:TA) constructs the three API mapping for the delta operation that updates the action of a transition `t`: it is very similar to the rule (F:TD), with the difference that the updated dependency slot of the transition's dependency is `action`.

4.2.3. Running example: information extraction from the product line of clocks

Consider the running example product line presented in Figure 4. We illustrate the information extraction step of our analysis by giving the three API mapping extracted from the delta `dSMode2`:

```
delta dSMode2 {
 modifies Clock {
  adds Display(t1(Clock.set, Set));
  modifies Set {
   modifies SetMinute {
    modifies t6 { destination Display }
}}}}
```

```
fadds = [
  Display ↦{ (dSMode2, Clock, false, ∅) }
  t1 ↦{ (dSMode2, Clock.Display, false, [ event ↦{(Clock.set, Clock, ⊥)}, guard ↦∅, action ↦∅, exit ↦{(Set,⊥,⊥)}]) }
]
fremoves = ∅
fmodifies = [
  Clock ↦{ (dSMode2, ∅, true, ∅) }
  Set ↦{ (dSMode2, Clock, true, ∅) }
  SetMinute ↦{ (dSMode2, Clock.Set, ⊥, ∅) }
  t6 ↦{ (dSMode2, Clock.Set.SetMinute, ⊥, ∅), (dSMode2, Clock.Set.SetMinute, ⊥, [ exit ↦{(Display,⊥,⊥)} ] ) }
]
```

The *fadds* mapping shows that `dSMode2` adds the basic state `Display` inside the state `Clock`, and that this new state has no dependency. Moreover, this delta adds the transition $\mathtt{t}_1$ in the state `Display` (being itself inside `Clock`), with the event `Clock.set` as trigger and `Set` as destination state (but without any guard nor action). The *fremoves* mapping shows that `dSMode2` does not remove any element. Finally, the *fmodifies* mapping shows that `dSMode2` modifies several elements in the statechart: the `Clock` root state is modified, and must be a composite state (as it must contain the added state `Display` and the modified state `Set`); the `Set` state is modified inside `Clock`, and must be composite (as it must contain the modified state `SetMinute`); the `SetMinute` state is modified inside `Clock.Set`; and the transition $\mathtt{t}_6$ is modified, by a **modifies** operation (which does not perform any operation per se on the transition, but requires that it exists), and by a destination update which changes the destination of the transition to `Display`.

4.3. Applicability consistency

The third step of the analysis constructs a propositional formula, called the *applicability constraint*, that encodes the fact that all the variants of a FΔSL product line can be generated, i.e., that the application of a deltas activated by a product never fails. As described in Section 3.2 (and as implemented in [10]), a delta application can fail for the following reasons: adding an element fails if that element already exists, or if the path in which that element is being added does not exist; removing an element fails if that element does not exist or if it is not in the path in which the operation is performed; and modifying an element fails if it does not exist, or if it is not in the path in which the operation is performed.

All these errors are caused by elements being present or absent when applying the delta. Therefore, we first define formulas stating when each element is present or absent, and then build our applicability constraint on top of them. We define two formulas ensuring the presence of a specific reference and that are used to construct our applicability constraint. The formula $\mathtt{present}(\mathtt{d}, \overline{\rho}, \rho, \varsigma)$ specifies the condition "an element named ρ is present in the path $\overline{\rho}$ when the delta `d` is applied, and its kind *validates* ς (i.e., the element is a composite state when required)". The formula $\mathtt{present}(\mathtt{d}, \overline{\rho}, \rho)$ is similar to the previous one but does not check the kind of the element. We also define a formula ensuring the absence of a specific reference. Namely, the formula $\mathtt{absent}(\mathtt{d}, \rho)$ specifies the condition "the element named ρ is absent when `d` is applied".

The formal definition of these formulas presented below use the helper function `removes` defined as follows:

$$\mathtt{removes}(\overline{\rho}, \rho) \triangleq \{\mathtt{d} \mid \exists \overline{\rho'}\ \rho',\ (\mathtt{d}, \overline{\rho}') \in \mathit{fremoves}(\rho') \wedge (\overline{\rho}'\ \rho') \text{ is a prefix of } (\overline{\rho}\,\rho)\}$$

This function gives the set of deltas that removes the reference ρ located in $\overline{\rho}$: indeed, as removing an element implies removing all its content (as shown in Section 3.2), a delta removes ρ if it directly removes it with the **removes** operation, or if it removes one of its parent. This is stated in our definition with $(\overline{\rho}'\ \rho')$ being a prefix of $(\overline{\rho}\,\rho)$. We can now give formal definition of the presence and absence formulas (to improve readability, we use the wildcard character _ to state that a part of a tuple is irrelevant in the formula's construction):

$$\texttt{present}(\texttt{d}, \overline{\rho}, \rho, \varsigma) \triangleq \bigvee_{\texttt{d}_1 \in S_1} (\texttt{d}_1 \wedge \bigwedge_{\texttt{d}_2 \in S_2(\texttt{d}_1)} \neg \texttt{d}_2) \qquad \text{with} \begin{cases} S_1 = \{\texttt{d}_1 \mid (\texttt{d}_1, \overline{\rho}, \mathfrak{b}, \_) \in \mathit{fadds}(\rho),\ \varsigma \leq \mathfrak{b},\ \texttt{d}_1 <_L \texttt{d}\} \\ S_2(\texttt{d}_1) = \{\texttt{d}_2 \mid \texttt{d}_2 \in \texttt{removes}(\overline{\rho}, \rho),\ \texttt{d}_1 <_L \texttt{d}_2 <_L \texttt{d}\} \end{cases}$$

$$\texttt{present}(\texttt{d}, \overline{\rho}, \rho) \triangleq \bigvee_{\texttt{d}_1 \in S_1} (\texttt{d}_1 \wedge \bigwedge_{\texttt{d}_2 \in S_2(\texttt{d}_1)} \neg \texttt{d}_2) \qquad \text{with} \begin{cases} S_1 = \{\texttt{d}_1 \mid (\texttt{d}_1, \overline{\rho}, \_, \_) \in \mathit{fadds}(\rho),\ \texttt{d}_1 <_L \texttt{d}\} \\ S_2(\texttt{d}_1) = \{\texttt{d}_2 \mid \texttt{d}_2 \in \texttt{removes}(\overline{\rho}, \rho),\ \texttt{d}_1 <_L \texttt{d}_2 <_L \texttt{d}\} \end{cases}$$

$$\texttt{absent}(\texttt{d}, \rho) \triangleq \bigwedge_{(\texttt{d}_1, \overline{\rho}) \in S_1} (\texttt{d}_1 \Rightarrow \bigvee_{\texttt{d}_2 \in S_2(\texttt{d}_1, \overline{\rho})} \texttt{d}_2) \qquad \text{with} \begin{cases} S_1 = \{\texttt{d}_1, \overline{\rho} \mid (\texttt{d}_1, \overline{\rho}, \_, \_) \in \mathit{fadds}(\rho),\ \texttt{d}_1 <_L \texttt{d}\} \\ S_2(\texttt{d}_1, \overline{\rho}) = \{\texttt{d}_2 \mid \texttt{d}_2 \in \texttt{removes}(\overline{\rho}, \rho),\ \texttt{d}_1 <_L \texttt{d}_2 <_L \texttt{d}\} \end{cases}$$

To construct the formula $\texttt{present}(\texttt{d}, \overline{\rho}, \rho, \varsigma)$, we first collect every delta $\texttt{d}_1$, applied before $\texttt{d}$, that adds the reference ρ in the location $\overline{\rho}$ and with a kind $\mathfrak{b}$ that validates ς (we write $\varsigma \leq \mathfrak{b}$ when $\mathfrak{b}$ validates ς, and formally we have $\bot, \textbf{false} \leq \textbf{false}$ and $\bot, \textbf{true} \leq \textbf{true}$). One of these delta must be activated for the formula to be true (as stated with $\bigvee_{\texttt{d}_1} \texttt{d}_1$), but also no other delta can remove ρ before $\texttt{d}$ is applied. Consequently, we also collect every delta $\texttt{d}_2$, applied before $\texttt{d}$ but after $\texttt{d}_1$ that removes ρ (located in $\overline{\rho}$), and ensure that none of them are activated (as stated with $\bigwedge_{\texttt{d}_2} \neg \texttt{d}_2$). The other presence formula is constructed similarly. Finally, to construct the formula $\texttt{absent}(\texttt{d}, \rho)$, we first collect every delta $\texttt{d}_1$, applied before $\texttt{d}$, that adds the reference ρ in any location $\overline{\rho}$: if one of them is activated, then ρ must be removed by any delta $\texttt{d}_2$ before $\texttt{d}$ is applied (as stated with $\texttt{d}_1 \Rightarrow \bigvee_{\texttt{d}_2} \texttt{d}_2$).

We can now define the application constraint, which we structure in three parts, each one responsible to ensure that a specific reason of failure does not occur. The first part contributes to ensure that all **adds** operations succeed during a variant generation: as stated in the beginning of this section, this corresponds to checking that the added element does not already exist, and that the path in which that element is being added does exist. The formula corresponding to that part is written `padds` and is defined as follows:

$$\texttt{padds} \triangleq \bigwedge_{\rho} \bigwedge_{\texttt{d}} \texttt{d} \Rightarrow \texttt{absent}(\texttt{d}, \rho) \qquad \text{with } \rho \in \mathrm{dom}(\mathit{fadds}),\ (\texttt{d}, \_, \_, \_) \in \mathit{fadds}(\rho)$$

This constraint collects every element that is added in the product line (i.e., the references in the domain of *fadds*), and checks that this element is absent before each time it is added by a delta $\texttt{d}$ (if that delta is activated). Note that we do not check in this constraint if the path in which the element is added exists: as the path is traversed by modification operations, its existence is checked with the second part of the application constraint, which ensures that all the **modifies** operations will succeed. This second part, written `pmodifies`, is defined as follows:

$$\texttt{pmodifies} \triangleq \bigwedge_{\rho} \bigwedge_{\texttt{d}} \texttt{d} \Rightarrow \texttt{present}(\texttt{d}, \overline{\rho}, \rho, \varsigma) \qquad \text{with } \rho \in \mathrm{dom}(\mathit{fmodifies}),\ (\texttt{d}, \overline{\rho}, \varsigma, \_) \in \mathit{fmodifies}(\rho)$$

Similarily to `padds`, This constraint collects every element that is modified in the product line (i.e., the references in the domain of *fmodifies*), and checks that this element is present (at the right location and with the right kind) before each time it is modified by a delta $\texttt{d}$ (if that delta is activated). Note also that as before, checking that the path to the modified element exists is not necessary. The third part of the application constraint, written `premoves`, ensures that all the **removes** operations will succeed. Its definition is similar to the other parts of the application constraint:

$$\texttt{premoves} \triangleq \bigwedge_{\rho} \bigwedge_{\texttt{d}} \texttt{d} \Rightarrow \texttt{present}(\texttt{d}, \overline{\rho}, \rho) \qquad \text{with } \rho \in \mathrm{dom}(\mathit{fremoves}),\ (\texttt{d}, \overline{\rho}) \in \mathit{fremoves}(\rho)$$

This constraint simply collects every element that is removed in the product line (i.e., the references in the domain of *fremoves*), and checks that this element is present at the right location before each time it is removed by a delta $\texttt{d}$ (if that delta is activated). Note that as before, checking that the path to the removed element exists is not necessary.

Finally, we now combine all the parts described above to construct the applicability constraint.

Definition 3 (Applicability-consistency)**.** *The* applicability constraint *of a* FΔSL *product line* L*, written* $\texttt{ac}(L)$ *is defined as follows:*

$$\texttt{ac}(L) \triangleq L.\texttt{FMandDA} \Rightarrow (\texttt{padds} \wedge \texttt{pmodifies} \wedge \texttt{premoves})$$

A FΔSL *product line* L *is* applicability-consistent *if the formula* $\texttt{ac}(L)$ *is valid.*

For instance, the FΔSL product line of clocks introduced in Section 3.1.4 is applicability-consistent.

4.4. Dependency consistency

The fourth and last step of the analysis generates a propositional formula, called the *dependency constraint*, that encodes the fact that none of a FΔSL product line's variants that can be generated have a dependency error. In Section 4.2, we gave a precise definition of what a dependency is, and which dependency an element in a FΔSL product line can have: we thus have all the elements to define this dependency constraint.

We construct this constraint in two steps. In the first step, we give a formula stating when a dependency element $(\rho, \omega, \varsigma)$ is solved.[7] This formula, written $\texttt{present}(\rho, \omega, \varsigma)$ is defined as follows:

$$\texttt{present}(\rho,\omega,\varsigma) \triangleq \bigvee_{(\texttt{d}_1,\overline{\rho})\in S_1} (\texttt{d}_1 \wedge \bigwedge_{\texttt{d}_2\in S_2(\texttt{d}_1,\overline{\rho})} \neg\texttt{d}_2)$$

$$\text{with } \begin{cases} S_1 = \{\texttt{d}_1, \overline{\rho} \mid (\texttt{d}_1, \overline{\rho}, \mathfrak{b}, \_) \in \mathit{fadds}(\rho),\ \omega \neq \bot \Rightarrow (\overline{\rho} = \omega),\ \varsigma \leq \mathfrak{b}\} \\ S_2(\texttt{d}_1, \overline{\rho}) = \{\texttt{d}_2 \mid \texttt{d}_2 \in \texttt{removes}(\overline{\rho}, \rho),\ \texttt{d}_1 <_L \texttt{d}_2\} \end{cases}$$

This definition is very similar to the definition of the presence formulas: indeed, a dependency element is solved by the *presence* of a corresponding element. More precisely, in this formula, we first collect every delta $\texttt{d}_1$ that adds the reference ρ in a location $\overline{\rho}$ that validates ω (as stated with $\omega \neq \bot \Rightarrow (\omega = \overline{\rho})$) and with a kind $\mathfrak{b}$ that validates ς (as stated with $\varsigma \leq \mathfrak{b}$). One of these delta must be activated for the dependency to be solved (as stated with $\bigvee_{\texttt{d}_1} \texttt{d}_1$) and no other delta $\texttt{d}_2$ may remove it afterward, so this element that solves the dependency element will be present in the generated variant (as stated with $\bigwedge_{\texttt{d}_2} \neg\texttt{d}_2$).

The second step constructs the main part of the dependency constraint, written `pdeps`, which checks that all the dependencies in a variant are solved:

$$\texttt{pdeps} \triangleq \bigwedge_{\rho} \bigwedge_{\texttt{dep}} \texttt{pdeps}(\rho, \texttt{dep}) \qquad \text{with } \rho \in \mathrm{dom}(\mathit{fadds}),\ \texttt{dep} \in \Phi$$

$$\texttt{pdeps}(\rho,\texttt{dep}) \triangleq \bigwedge_{(\texttt{d}_1,\delta)\in S_1} \texttt{d}_1 \Rightarrow \Big((\bigvee_{\texttt{d}_2\in S_2(\texttt{d}_1)} \texttt{d}_2) \vee (\bigvee_{\texttt{d}_3\in S_3(\texttt{d}_1)} \texttt{d}_3) \vee (\bigwedge_{(\rho',\omega,\varsigma)\in\delta(\texttt{dep})} \texttt{present}(\rho',\omega,\varsigma)) \Big)$$

$$\text{with } \begin{cases} S_1 = \{\texttt{d}_1, \delta \mid (\texttt{d}_1, \overline{\rho}, \_, \delta) \in \mathit{fadds}(\rho) \cup \mathit{fmodifies}(\rho), \texttt{dep} \in \mathrm{dom}(\delta)\} \\ S_2(\texttt{d}_1) = \{\texttt{d}_2 \mid \texttt{d}_2 \in \texttt{removes}(\overline{\rho}, \rho),\ \texttt{d}_1 < \texttt{d}_2\} \\ S_3(\texttt{d}_1) = \{\texttt{d}_3 \mid (\texttt{d}_3, \overline{\rho}, \_, \delta') \in \mathit{fmodifies}(\rho), \texttt{dep} \in \mathrm{dom}(\delta'), \texttt{d}_1 < \texttt{d}_3\} \end{cases}$$

The formula `pdeps` is defined as follows: it collects every reference ρ added in the product line and every dependency slot `dep` that reference can have (in this formula, we over-approximate by collecting all the possible slots), and checks that all the dependency elements in these slots for that reference are solved. This is done with the formula $\texttt{pdeps}(\rho, \texttt{dep})$. This formula collects all the delta $\texttt{d}_1$ that associate a set of dependency elements to the slot `dep` of ρ (as stated with $(\texttt{d}_1, \overline{\rho}, \_, \delta) \in \mathit{fadds}(\rho) \cup \mathit{fmodifies}(\rho) \wedge \texttt{dep} \in \mathrm{dom}(\delta)$). Then these dependency can either be removed by a delta $\texttt{d}_2$ that removes ρ (as stated with $\texttt{d}_2 \in \texttt{removes}(\overline{\rho}, \rho)$), or by another delta $\texttt{d}_3$ that replaces the value associated to that dependency slot (as stated with $(\texttt{d}_3, \overline{\rho}, \_, \delta') \in \mathit{fmodifies}(\rho) \wedge \texttt{dep} \in \mathrm{dom}(\delta')$). Otherwise, if the dependency elements introduced by $\texttt{d}_1$ are never removed, they must be solved (as stated with $\texttt{present}(\rho', \omega, \varsigma)$).

Finally, we now give the complete definition of the dependency constraint.

Definition 4 (Dependency consistency)**.** *The* dependency constaint *of a* FΔSL *product line* L*, written* $\texttt{dc}(L)$ *is defined as follows:*

$$\texttt{dc}(L) \triangleq L.\texttt{FMandDA} \Rightarrow \texttt{pdeps}$$

A FΔSL *product line* L *is* dependency-consistent *if the formula* $\texttt{dc}(L)$ *is valid.*

For instance, the FΔSL product line of clocks introduced in Section 3.1.4 is dependency-consistent.

[7] The notion of dependency element $(\rho, \omega, \varsigma)$ is illustrated in the middle of Section 4.2.1.

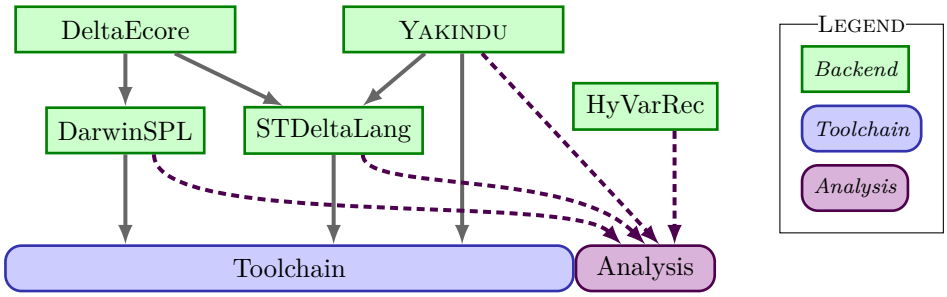


Figure 12: The HyVar production toolchain, which incorporates an implementation of our analysis

4.5. Properties of the analysis

The following theorem states that our analysis is correct and complete: it does check the well-formedness of a FΔSL product line. However, our analysis is still correct but not complete if the Guideline G1 (given in Section 3.4.1) is not followed.

Theorem 1 (Correctness and completeness of FΔSL well-formedness checking)**.** *Let L be a* FΔSL *product line. Consider the properties:*

i. L is partially typed, applicability- and dependency-consistent.

ii. the variants of L can be generated and are well-formed FSL *programs.*

We have (i) implies (ii), and reciprocally (ii) implies (i) if L has no useless declaration.

Proof. See Appendix A. □

For instance, the FΔSL product line of clocks introduced in Section 3.1.4 is well-formed.

5. The HyVar toolchain and case study

In this section we illustrate how the implementation of our analysis integrated in the HyVar toolchain has been applied on the HyVar case study. We first present an overview of the the HyVar toolchain (in Section 5.1) and case study (in Section 5.2). Then we report on the application of the analysis (in Section 5.3).

5.1. The HyVar toolchain

The HyVar toolchain [3] is structured in two parts: the production toolchain, implemented as a set of Eclipse [1] plugins, is used to construct delta-oriented SPLs on YAKINDU statecharts; and the deployment toolchain is used to deliver the different variants of these SPLs to the consumers. We implemented our analysis as an extension of the production toolchain, as shown in Figure 12. The HyVar production toolchain is based on several backends.

YAKINDU *[5].* This module provides the core functionalities to design statecharts, simulate them and generate source code from them. It is important to note that YAKINDU uses the Ecore model [2] to describe its statecharts. YAKINDU statecharts include all the constructs we presented in this paper, like interfaces, states and transitions, but also constructs we did not include in our core calculus, like *region*, *choices*, *synchronization* or *state behavior*. In particular, state behavior is used in the case study illustrated in Section 5.2 and extends basic and composite states with an optional inner behavior of the following syntax:

$$\begin{array}{ll} SB & ::= \mathit{trigger}\ [\mathit{guard}]\ /\ \mathit{action} \\ \mathit{trigger} & ::= \textbf{entry} \mid \textbf{exit} \mid \textbf{always} \mid \textbf{every}\ \mathsf{time} \end{array}$$

The structure of a state behavior *SB* has the same structure as a transition, with a *trigger*, a *guard* and an *action* that is executed when the trigger and the guard are validated. The trigger **entry** is validated when the state becomes activated, **exit** is validated when the state is exited, **always** is continuously validated when the state is activated, and **every** time is validated every time period of time when the state is activated.

DeltaEcore [34]. This module provides a generic framework to define deltas on any Ecore-based model. It provides two main functionalities to the HyVar production toolchain. First, it offers primitives to simply define ad-hoc language to define delta for a specific Ecore-based model. Since the statecharts of YAKINDU are expressed in the Ecore format, it is thus possible to use DeltaEcore to define a language of deltas to manipulate these statecharts.[8] Second, it allows to declare an application order between deltas, and given a set of deltas and a base program, it can apply the given deltas to the base program in a correct order.

DarwinSPL [28]. This module allows to declare DeltaEcore-based SPL by providing: an expressive language to describe feature models, and several configuration files where the user can specify a base program and a set of deltas (written in a DeltaEcore language) used in the SPL, together with their activation condition. Additionally, this module allows to generate variants from valid products: given a valid product, it uses the activation conditions provided by the user to identify which deltas are activated, and uses DeltaEcore to compute the corresponding variant, based on the base program of the SPL.

STDeltaLang. This module implements the DeltaEcore language for YAKINDU statecharts. It supports all the FSL constructs presented in Section 2, like interfaces, states and transitions, but also constructs that are not modeled by FSL, like *regions* and *state specifications* [5]. These elements can thus be added, removed or modified in different ways using the operations declared in this language.

HyVarRec [29]. This module is a solver specialized for detecting anomalies in feature models, like emptiness or false optional, and can also automatically compute a product that validates some user-required properties. We use this module in our toolchain as the backend SAT solver for our analysis.

Incorporating our SPL analysis into the toolchain. We incorporated our analysis into the HyVar toolchain as a new module, that can be called directly from the DarwinSPL interface to check the validity of one of its SPLs. This module is thus based on DarwinSPL, which provides information about the feature model and part of the configuration knowledge (the activation conditions) of an SPL. It moreover relies on the STDeltaLang module to access the content of the deltas and their application order, in order to compute the uniform API described in Section 4.2, and on the YAKINDU module to parse the base program of the SPL, again to compute the uniform API. This implementation extends the core analysis presented in this paper and supports all the delta operations declared in the STDeltaLang module. In particular, state behaviors are managed in the implemented analysis with: the addition of a new dependency slot, called `sb`, that captures all dependencies related to state behaviors; and the extension of partial typing to also check the validity of the expressions contained in state behaviors.

The implementation is structured in three parts, as presented in Section 4. The first part (the partial typing part of the analysis) computes the declaration table of the input SPL, relying on the YAKINDU and STDeltaLang API to parse the different interface definition sections and operations, and then parses all specifications in the SPL to check for type mismatch in operation application. The second part also relies on the YAKINDU and STDeltaLang APIs to extract the uniform API from the base program and the deltas of the SPL. It moreover relies on the DarwinSPL API to obtain the L.`FMandDA` formula. Finally, the third part of the analysis is generic, and only relies on the data generated by the second part to perform the analysis: it translates the generated uniform API into SAT constraints as described in Sections 4.3 and 4.4.

[8] Due to some limitations of DeltaEcore, the deltas supported by the HyVar toolchain do not model nested delta operations and do not model operations that simultaneous add a state together with its outgoing transition set (so, first the state must be added and then each outgoing transition can be added by modifying the just added state).

Additionally, for error reporting, it generates fresh variables set to be equivalent to each important predicate in our translation, like `present(_)`, `absent(_)`, etc.correction Each of these variables is mapped to a message describing their meaning in such a way that, if an error is found during the SAT solving process (carried out by HyVarRec), a human-readable message giving the cause of the error will be shown.

5.2. The HyVar case study

The HyVar use case considers the implementation of software embedded in cars structured in three Electronic Control Units (ECUs). The first ECU, called ECU_A, has 12 features, 12 products and 6 deltas. It is dedicated to the implementation of the core functionalities of the car, like the Emergency Call system, that automatically calls the police in case a car crash occurs. The second ECU, called ECU_B, has 5 features, 6 products and 8 deltas. It is dedicated to core services, like the gear advice or the start/stop functionality. The third and last ECU, called ECU_C, has 9 features, 96 products and 18 deltas. It is dedicated to the GUI of the different services available in the car. Each of these ECUs has a dedicated SPL implemented using the HyVar production toolchain (presented in Section 5.1). In the following, we present the implementation of the ECU_A product line.

The implementation of ECU_A is illustrated in Figures 13, 14 and 15 where, compared to the actual code in the implementation, we adapted some syntax to be closer to the core calculus described in Sections 2 and 3.

Figure 13 (left) describes the feature model of ECU_A by a feature diagram with *cross-tree constraints* (expressed by the propositional formula written just below the feature diagram). One of the main functionality implemented in ECU_A is the emergency call system, that aims to automatically dial emergency numbers in the event of a serious road accident and to wirelessly send impact sensor information and location coordinates to local emergency agencies. Different programs exist such as the eCall/E112 program of the European Union as well as the Russian ERA GLONASS system. This is reflected in the feature model, where an ECU_A is constituted by: a GNSS positioning system, that can either be `GPS` (in Europe) or `Glonass` (in Russia); the core EmergencyCall system, that can either be the European `eCall` or the Russian `ERAGlonass`; and a Language configuration, that can either by `English` or `Russian`. The optional `Diagnostic` feature allows to send additional information during the emergency call, while the optional `ConnectionGateway` corresponds to less relevant functionalities of the ECU. The cross-tree constraints express additional conditions about which feature must be selected together to make sense. For instance, the `ERAGlonass` system can only function in Russia with the `Glonass` positioning system, while the European `eCall` system relies on `GPS` positioning.

Figure 13 (right) describes the configuration knowledge of ECU_A by specifying for each of its six deltas the corresponding activation condition, and by specifying the application order as a total order on a partition of the set of deltas—dGlonass implements the `Glonass` feature by configuring the GNSS system; dCombo allows to have both `GPS` and `Glonass` positioning systems activated in the car; dDiagnostic and dDiagnostic2 implement the `Diagnostic` feature by sending additional data during an emergency call; dRussian and dRussian2 implement both the `ERAGlonass` and `Russian` features by configuring the language of the system, and sending data corresponding to the `ERAGlonass` protocol. The ECU_A product line does not contain any delta for the other features because they are implemented by default in the base statechart of the SPL.

Figure 14 presents the base statechart of the product line. As usual, we provide in this figure the definition section part and the state definition part of the statechart, both in a graphical and textual representation—note that, in YAKINDU, each composite state requires one of its inner states to be specified as final state—in the graphical representation the final state is indicated by "◎", while in the textual representation it is indicated by adding a fourth component to the syntax of composed states, which therefore becomes (cf. Section 2.2.2): $\mathtt{s}(\overline{SD}, \overline{TD}, \mathtt{s}', \mathtt{s}'')$, where $\mathtt{s}'$ is the initial state and $\mathtt{s}''$ is the final state.

The workflow of the ECU_A statechart implements the emergency call system for Europe and is a simple sequence of function calls. First, the state `ATB2Init` calls the `F.ATB2Init()` function, which simply initializes the ECU. Then, the state `SetGnssSystem` calls the `F.SetGnssSystem(1)` function, which defines which positioning system to use: the parameter 1 means `GPS`, 2 means `Glonass`, while 0 means both. Then, the state `SetLanguage` sets which language to use: `"en"` means English, while `"ru"` means Russian. Then, the

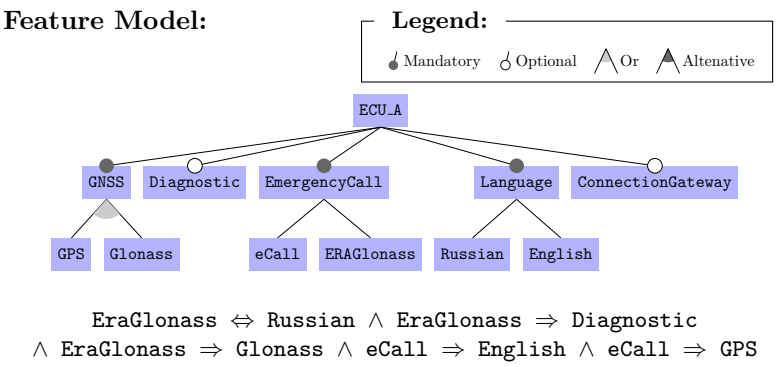


Figure 13: The ECU_A product line: feature model and configuration knowledge

state `eCallMsgInit` is a composite state with only one base state inside, which calls the `F.init_ecallmsg()` function. This function builds the emergency message to send. Then, the state `DataInit` is a composite state with only one base state inside, which calls the `F.init_data()` function. This function builds the additionaly data to add to the message, that includes the position and the damage status of the car. Then, the state `FormatMSD` encodes and sends the message to the authorities. It performs this task with three inner states: `EncodeData` calls `F.encode_message()` which encodes the built message; `PlayPrompt` calls `F.play_ prompt()` which plays a tune to signal to the car passengers that an emergency call is being sent; and `SendMsg` calls `F.send_msg()` which sends the message. Finally, the state `112Call` calls `F.ecall()` which starts a phone call to the authorities.

Figure 15 presents the deltas of the product line. The delta dGlonass selects the `Glonass` positioning system for the emergency call by modifying the behavior of the `SetGnssSystem`: it replaces the parameter of the F.SetGnssSystem function by 2.

The delta dCombo sets the positioning system of the emergency call system, to be able to use both `GPS` and `Glonass`. To do so, it replaces the parameter of the F.SetGnssSystem function by 0.

The deltas dDiagnostic and dDiagnostic2 add more data to the message sent during the emergency call. The delta dDiagnostic adds a new function to the `F` interface, called init_diag_data, responsible to the generation of this additional data. Then, it modifies the eCallMsgInit state by adding a new state inside, called MsgInitDiag, that calls the new function. This new state is inserted inside the statechart's workflow. The delta dDiagnostic2 sets it as the new initial state of eCallMsgInit and adds the outgoing transition to MsgInitDiag; the workflow then proceed as before with the new transition t_9 that continues to the state MsgInit.

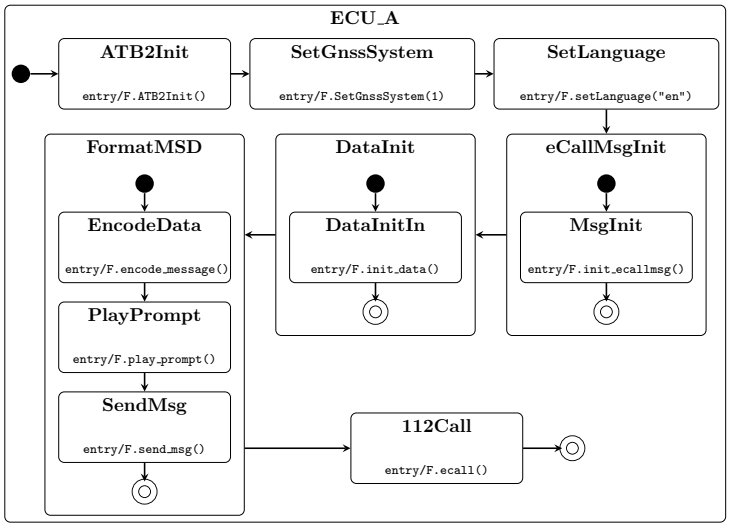
The deltas dRussian and dRussian2 implement both the `ERAGlonass` and `Russian` features. The `ERAGlonass` feature is implemented with the extention of the F interface with two new functions to manage `ERAGlonass` specific data, and two new states `MsgInitAnnex` and `EncodeDataAnnex` to respectively generate and encode the data in the sent message. Indeed, the `ERAGlonass` protocol includes the same data exchange as the European eCall protocol, with some more data to communicate. The state `MsgInitAnnex` is inserted after `MsgInit` in the statechart workflow, and calls init_annex_data() to generate the `ERAGlonass` data. The `MsgInitAnnex` and `EncodeDataAnnex` are respectively inserted within the composite states `eCallMsgInit` and `MsgInitAnnex` by dRussian, that also sets the language in the `SetLanguage` state to `"ru"`. Finally dRussian2 links the new states to the rest of the statechart by adding suitable transitions. It also sets `MsgInitAnnex` as the new final state of `eCallMsgInit`.

5.3. The application of the analysis

The implementation of the SPL analysis incorporated in the toolchain has been used by the industrial partners of HyVar during the development of the final version of ECU_A (an initial version of ECU_A was developed using a preliminary version of the HyVar toolchain [10] that did not support the SPL analysis) and during the development of ECU_B and ECU_C. According to their feedback the SPL analysis support was very useful: it allowed the developers to ensure that all the variants of the product lines can be generated

interface F:

operation ATB2Init(): void
operation SetGnssSystem(integer): void
operation setLanguage(string): void
operation init_ecallmsg(): void
operation init_data(): void
operation encode_message(): void
operation play_prompt(): void
operation send_msg(): void
operation ecall(): void



$$
\begin{array}{lcl}
SD_{\text{ECU\_A}} & = & \text{ECU\_A}(SD_{\text{Init}}\ SD_{\text{Gnss}}\ SD_{\text{Lang}}\ SD_{\text{eMsgInit}}\ SD_{\text{DataInit}}\ SD_{\text{Format}}\ SD_{\text{Call}},\ \emptyset,\ \text{ATB2Init},\ \text{112Call}) \\
SD_{\text{Init}} & = & \text{ATB2Init}(\text{entry/F.ATB2Init}() \mid \mathsf{t}_1(\mathbf{true},\ \text{SetGnssSystem})) \\
SD_{\text{Gnss}} & = & \text{SetGnssSystem}(\text{entry/F.SetGnssSystem(1)} \mid \mathsf{t}_2(\mathbf{true}, \text{SetLanguage})) \\
SD_{\text{Lang}} & = & \text{SetLanguage}(\text{entry/F.SetLanguage("en")} \mid \mathsf{t}_3(\mathbf{true}, \text{eCallMsgInit})) \\
SD_{\text{eMsgInit}} & = & \text{eCallMsgInit}(SD_{\text{MsgInit}},\ \mathsf{t}_4(\mathbf{true}, \text{DataInit}),\ \text{MsgInit},\ \text{MsgInit}) \\
SD_{\text{MsgInit}} & = & \text{MsgInit}(\text{entry/F.init\_callmsg}()) \\
SD_{\text{DataInit}} & = & \text{DataInit}(SD_{\text{DataInitIn}},\ \mathsf{t}_5(\mathbf{true}, \text{FormatMSD}),\ \text{DataInitIn},\ \text{DataInitIn}) \\
SD_{\text{DataInitIn}} & = & \text{DataInitIn}(\text{entry/F.init\_data}()) \\
SD_{\text{Format}} & = & \text{FormatMSD}(SD_{\text{Encode}}\ SD_{\text{Play}}\ SD_{\text{Send}},\ \mathsf{t}_6(\mathbf{true}, \text{112Call}),\ \text{EncodeData},\ \text{SendMsg}) \\
SD_{\text{Encode}} & = & \text{EncodeData}(\text{entry/F.encode\_data}() \mid \mathsf{t}_7(\mathbf{true}, \text{PlayPrompt})) \\
SD_{\text{Play}} & = & \text{PlayPrompt}(\text{entry/F.play\_prompt}() \mid \mathsf{t}_8(\mathbf{true}, \text{SendMsg})) \\
SD_{\text{Send}} & = & \text{SendMsg}(\text{entry/F.send\_smg}()) \\
SD_{\text{Call}} & = & \text{112Call}(\text{entry/F.ecall}())
\end{array}
$$

Figure 14: The ECU_A product line: base statechart

```
delta dGlonass {
 modifies ECUA {
  modifies SetGnssSystem {
   action entry / F.SetGnssSystem(2)
}}}

delta dCombo {
 modifies ECUA {
  modifies SetGnssSystem {
   action entry / F.SetGnssSystem(0)
}}}

delta dDiagnostic {
 modifies interface F {
  adds init_diag_data(): void
 }
 modifies ECUA {
  modifies eCallMsgInit {
   adds MsgInitDiag (entry / F.init_diag_data())
}}}

delta dDiagnostic2 {
 modifies ECUA {
  modifies eCallMsgInit {
   modifies MsgInitDiag {
    adds t9(true, MsgInit)
   }
   initial MsgInitDiag
}}}
```

```
delta dRussian {
 modifies interface F {
  adds init_annex_data(): void
  adds encode_annex_data(): void
 }
 modifies ECUA {
  modifies eCallMsgInit {
   adds MsgInitAnnex (entry / F.init_annex_data())
  }
  modifies FormatMSD {
   adds EncodeDataAnnex (entry / F.encode_annex_data())
  }
  modifies SetLanguage {
   action entry / F.SetLanguage("ru")
}}}

delta dRussian2 {
 modifies ECUA {
  modifies FormatMSD {
   modifies EncodeData {
    modifies t7 { destination EncodeDataAnnex}
   }
   modifies EncodeDataAnnex {
    adds t11(true, PlayPrompt)
   }
  }
  modifies eCallMsgInit {
   modifies MsgInit {
    adds t10(true, MsgInitAnnex)
   }
   final MsgInitAnnex
}}}
```

Figure 15: The ECU_A product line: its six deltas

and are well-formed YAKINDU statecharts before actually generating any of them, and this was particularly appreciated after the experience of developing the initial version of ECU_A with the preliminary version of the toolchain that did not support the SPL analysis.

We illustrate the error reports that the SPL analysis module shows to the developers by considering three examples that introduce in the implementation of ECU_A three errors that are detected by the partial typing, the application consistency and the dependency consistency phases of the SPL analysis, respectively. Namely, an error that would cause the generation of ill-type typed variants, an error that would cause the generation of some variants to fail, and an error that would causes the generation of incomplete variants. All the three examples consider an erroneous implementation of the delta dCombo—Figure 16 (top) illustrates a screenshot with the correct code of dCombo (cf. the code of dCombo given in Figure 15), while Figures 16 bottom), 17 (top) and 17 (bottom) illustrate the three erroneous versions.

Figure 16 (bottom) illustrates a screenshot with an erroneous version of the delta dCombo where the `F.SetGnssSystem()` operation is invoked (in line 6) with a string parameter, thus causing a partial typing error reported by a pop-up window. The error is due to the fact that, in the interface declaration part of the base statechart, it is specified that `F.SetGnssSystem()` accepts a parameter of integer type.

Figure 17 (top) illustrates a screenshot with an erroneous version of the delta dCombo and the pop-up window reporting an applicability consistency error. The error is due to the fact that line 6 contains a modify operation on the state `MsgInitAnnex` (and inner state of the composite state `eCallMsgInit`) that is added by dRussian, which comes after dCombo in the application order. The error report is structured in three sections presenting: (i) the list of features of a product that cannot be generated; (ii) the list of deltas involved in the generation; and (iii) an applicability dependency that is not satisfied, namely: during the generation of the variant a modification or deletion of state `MsgInitAnnex` would be performed in the delta dGlonass while this state does not exist.

Figure 17 (bottom) illustrates a screenshot with an erroneous version of the delta dCombo and the pop-up window reporting a dependency consistency error. The error is due to fact that line 6 contains the operation invocation `F.init_annex_data()` in the specification of the state `SetGnssSystem()`, while the declaration of the operation `init_annex_data()` is added to the interface `F` by the delta dRussia (which is not always

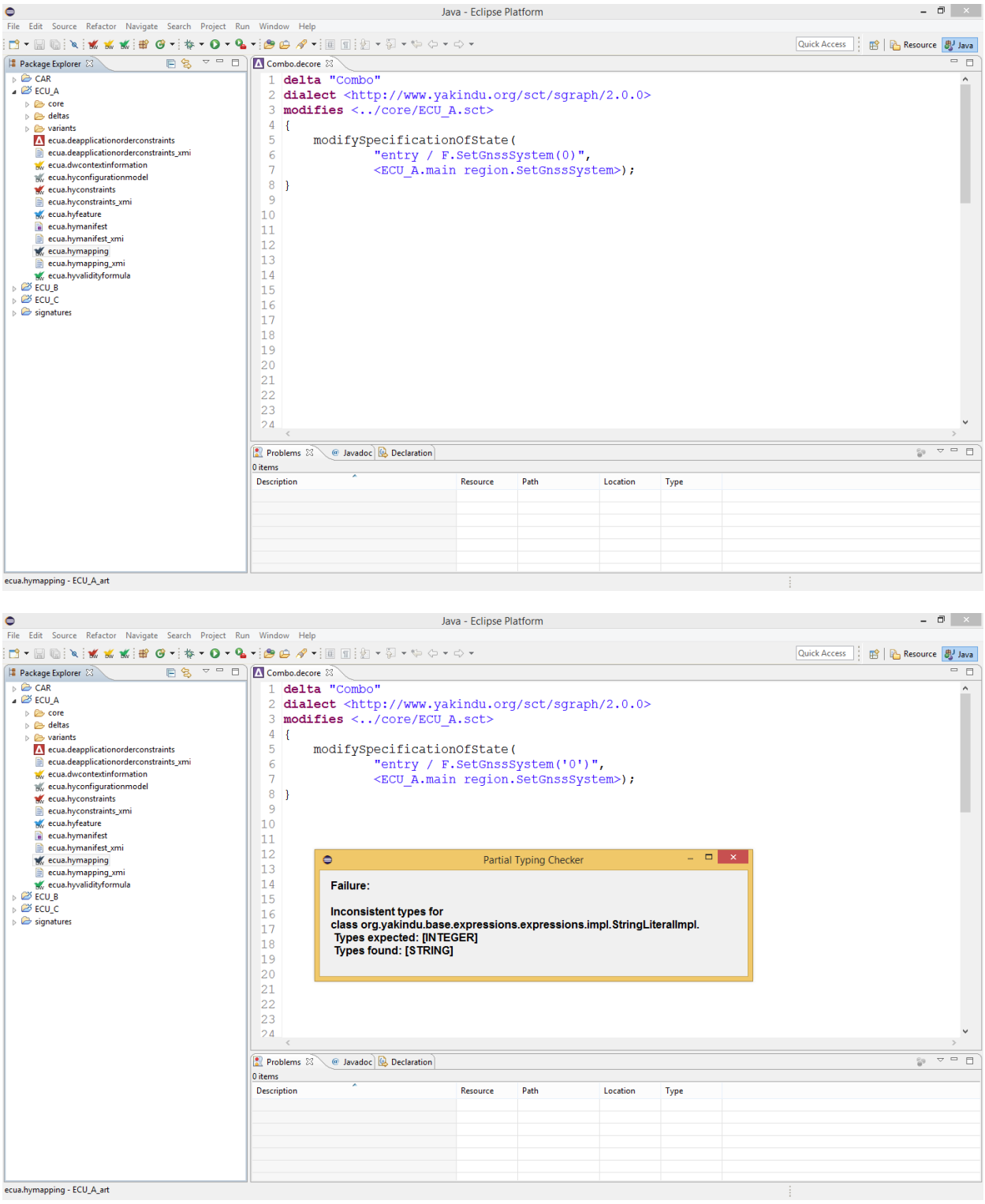


Figure 16: Two versions of dCombo: (top) the correct version; (bottom) an erroneous version where, in line 6, the `F.SetGnssSystem()` operation is invoked with a string parameter—the report pointing out the partial typing error is given by a pop-up window

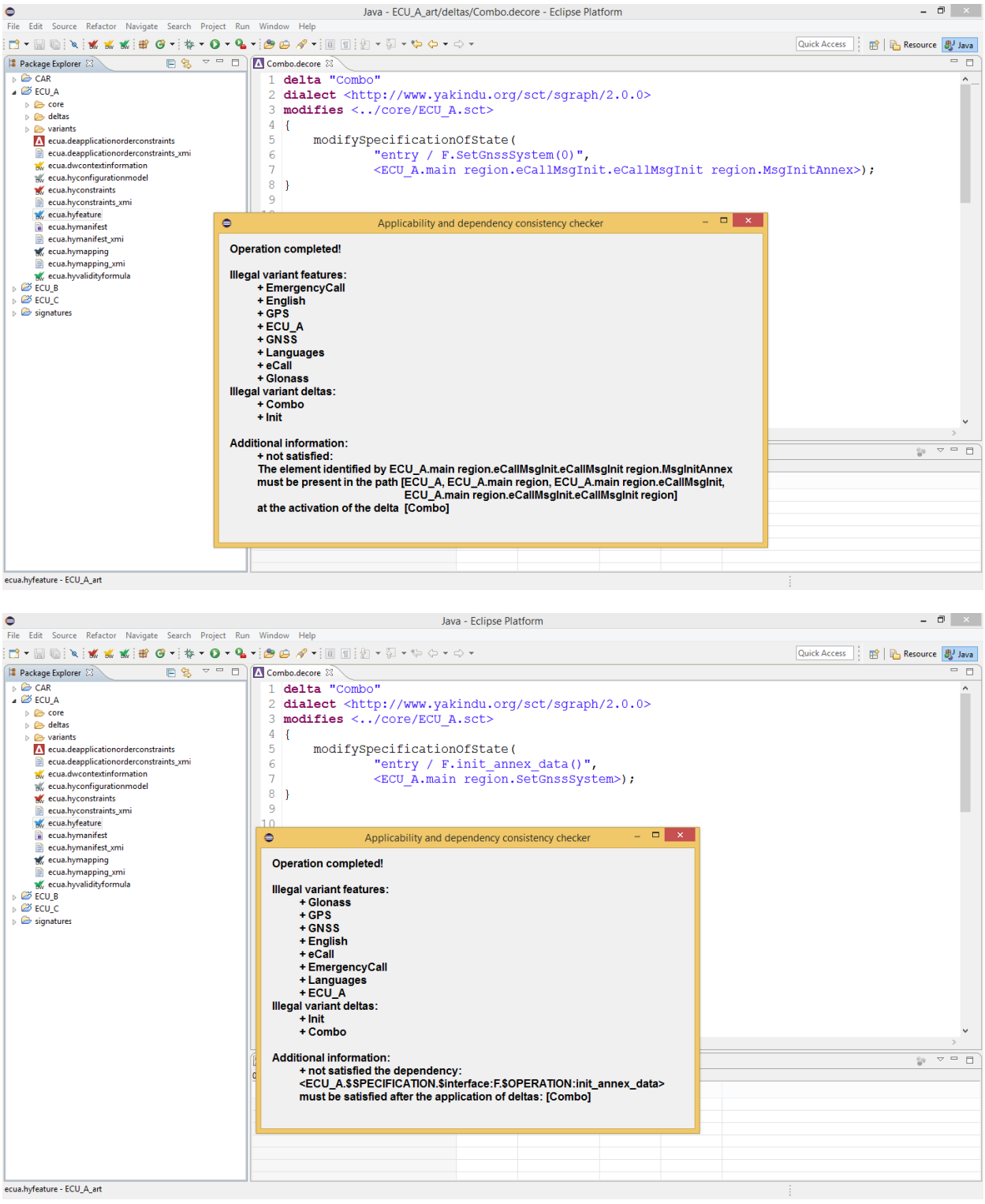


Figure 17: Two erroneous versions of the dCombo delta with their error report: (top) a version containing, in line 6, a modify operation of the state `MsgInitAnnex` (and inner state of the composite state `eCallMsgInit`); (bottom) a version containing, in line 6, the operation invocation `F.init_annex_data()` in the specification of the state `SetGnssSystem()`

activated together with dCombo). The error report is structured in three sections presenting: (i) the list of features of a product that would generate an incomplete variant; (ii) the list of deltas involved in the generation; and (iii) a consistency dependency that is not satisfied, namely: the generated variant would contain an invocation of the operation `F.init_annex_data()` that is not defined.

6. Related work

SPL implementation approaches can be classified into three main categories [33]: annotative, compositional and transformational. *Transformational approaches* express both positive and negative variability: DOP is a transformational approach. *Compositional approaches* express positive variability: the artifacts associated to the selected features are composed to build the corresponding variant. A paradigmatic example of compositional approach is represented by *Feature-Oriented Programming* (FOP) [8] [6, Sect. 6.1]. It can be described as a restriction of DOP where deltas are associated one-to-one with features and have limited expressive power: they can add and modify program elements, however, they cannot remove them (see, e.g., [32] for a detailed comparison between DOP and FOP). *Annotative approaches* express negative variability: all variants are included within the same model (called a 150% model) and each variant is obtained by excluding the artifacts associated to the non-selected features. C preprocessor directives (`#define FEATURE` and `#ifdef FEATURE`) are a paradigmatic example of SPL annotative implementation mechanism.

SPL analysis approaches can be classified into three main categories [38]: product-based, family-based and feature-based. *Product-based* analyses work only on generated variants (or models of variants). *Family-based* analyses work on the artifact base, without generating any variant or model of variant, by exploiting feature model and configuration knowledge to derive results about all variants. *Feature-based* analyses work on the reusable artifacts in the artifact base (base program and deltas in DOP) in isolation, without using feature model and configuration knowledge, to derive results on all variants. We refer to [38] for a survey on SPL analyses.

The Yakindu statecharts language is defined as an ECore metamodel [2] and, in the HyVar toochain, the language of deltas on Yakindu statecharts is defined by DeltaEcore [34], a tool suite which supports developers in defining delta languages for Ecore metamodels. We have designed the core languages FSL and FΔSL, which capture the key ingredients of delta-oriented programming on Yakindu statecharts, in order to exploit them for providing a formal account of an SPL analysis for guaranteeing that all the variants can be generated and are well formed.

Our family-based well-formedness checking mechanism for delta-oriented SPL of FSL statecharts is inspired by our recently proposed type checking approach for delta-oriented SPLs of Java-like programs [14] which, in turn, is inspired by a type checking mechanism for FOP SPLs of Java-like programs, proposed by Delaware et al. [17], that has been partially implemented for the AHEAD Tool Suite [37]. The paper [14] shows how, starting from a set of typing rules for IFJ [9], an imperative version of Featherweight Java [22], it is possible to define a family-based type-checking analysis for SPLs written in IFΔJ, a language for delta-oriented SPLs of IFJ programs. Therefore, in order to use the technique proposed for IFΔJ SPLs to define a family-based well-formedness analysis for SPLs written in FΔSL, we have first formalized the notion of well-formed FSL statechart by a means of a set of typing rules. While defining the analysis for FΔSL SPLs we observed both similarities and differences with respect to the analysis for IFΔJ SPLs. Namely, the structure of the analysis in three main properties, partial typing, applicability consistency and dependency consistency, is the same as for both analysis. However, in many aspects, the structure of a statechart is more complex that the structure of a IFJ program, and so the technique described in this paper reflects this added complexity. First, whereas an IFJ program has only classes and attributes, statecharts have a recursive structure where composite states can contain composite states that can contain states themseves. Due to that added complexity, we needed to introduce the new notion of path to identify where an element is placed in a statechart. Second, the elements of a IFJ program have only one dependency slot: classes depend on their super classes, and fields and methods depend on the types and method they use in their declaration. Due to the fine grain structure of the elements in a statechart, where each part of a transition can be changed, we needed to introduce the notion of dependency slots. Also, because some dependencies,

like a transition toward a shallow history, requires a composite state to be resolved, we needed to introduce the $\mathfrak{b}$ boolean. To make this added complexity more manageable, we introduced the uniform API (*fadds*, *fremoves*, *fmodifies*), that did not exist in [14]. Finally, whereas IFJ includes subtyping, and thus the analysis on IFΔJ SPLs needed to deal with this feature, the analysis presented in this paper did not have to manage this feature.

We are not aware of any other proposals for the analysis of SPLs of statecharts. Instead, several proposals for family-based model checking of on non-hierarchical state machines and transition systems can be found in the literature. These proposals focus either on annotative implementation mechanisms, like *Featured Transition Systems* (FTSs) [11], *Modal Transitions Systems* (MTSs) [19, 36] and *Product Line Labeled Transition Systems* (PL-LTSs) [20], or on the delta-oriented mechanism, like *DeltaCCS* [25].

Both YAKINDU and UML stacharts extend the "classical" notion of statechart proposed by Harel [21]. We are not aware of any formalization of YAKINDU statecharts. The formalization of UML statecharts by von der Beeck [40] provides a textual language for representing statecharts, as done in previous work for classical statecharts [27, 39] as as well as for UML statecharts [23]. It defines *UML-statechart terms* that extend those proposed by Maggiolo Schettini et al. [27] for classical statecharts and models also statecharts constructs (like, e.g., And-states) that are not modeled by our FSL. However, UML-statechart terms do not model the interface definition part, which is a key component of YAKINDU statecharts.

Building on the formalization by von der Beeck [40], an annotative approach and a compositional approach for implementing SPLs of UML statecharts have been proposed by Luna and Gonzales [26] and by Szasz and Vilanova [35], respectively. Luna and Gonzales [26] represent an SPL of UML statecharts by: a feature diagram representing the feature model, an UML-statechart term where each element (i.e., state or transition) is annotated with either the label *optional* or the label *non_optional* (to indicate whether the element is optional), and a mapping form non-mandatory features to optional-annotated elements of the statechart. The variant associated to a product is generated by removing the optional elements that are not associated to any selected feature (an optional element may be associated to more than one optional feature). Since the suppression of elements can return an ill-formed statechart (like, e.g., a statechart containing a dangling transition) the suppression of each element includes a control and rebuilding step (specified by cases) that should restore the well-formedness of the statechart—however, no property of the resulting variant generation algorithm (including whether it is guaranteed to generate a well-formed statechart for each product) is discussed.

Szasz and Vilanova [35] represent an SPL of UML statecharts by: a feature diagram representing the feature model, and a function that associates to each feature a statechart. The variant associated to a given product is then generated by combining the statecharts associated the features in the product by means of suitable statechart combination operations. That statechart combination operations are partial (i.e., the combination of two well-formed statecharts may be undefined). Some properties of the statechart combination operations are proved. However, as clearly stated in the paper, the resulting variant generation algorithm may fail to generated the variant associated to a product. The problem of defining an SPL analysis that guarantees that all the variants can be generated and are well-formed is not considered.

More recently, Wille et al. [41] proposed a variability mining procedure that, given a set S of models (written in a given modeling language, e.g., statecharts) generated by clone-and-own industrial practice [18], semi-automatically identifies variability information (i.e., common an varying parts) on the elements of S, and then extracts from S a delta-oriented SPLs of models. The procedure, which can be applied to different modeling languages, generates a delta language specifically tailored to transforming models in the analyzed modeling language. The procedure is evaluated by two case studies with industrial background that consider a set of *MATLAB/Simulink* models and a set of *Rational Rhapsody* statechart models, respectively. No formal account of the procedure is provided.

7. Conclusion

The work reported in this paper originated in the context of the HyVar project, where we faced the problem of enhancing the preliminary version of the HyVar toolchain (that support delta-oriented SPLs of

YAKINDU statecharts) by adding support for an SPL analysis that automatically checks that all the variants can be generated and are well formed. In this paper we present a formal account of the well-formedness SPLs analysis technique that has been integrated into the toolchain, and illustrate how the analysis has been applied to an industrial case study. In future work we would like to further evaluate the implementation by considering other case studies. We also plan to define other static analyses for delta-oriented SPL of YAKINDU statecharts (like, e.g., model checking) and to incorporate them into the HyVar toolchain.

The analysis enforces the type uniformity guideline on the SPL (see Section 3.4.2). According to the feedback received from the HyVar industrial partners, type uniformity helps developing and maintaining an SPL. Moreover, extending the analysis to deal with non type-uniform SPLs would significantly increase the complexity of the analysis. Therefore, we are currently not planning this extension.

We have designed the well-formedness analysis for delta-oriented SPL of statecharts by taking inspiration from our recently proposed type checking approach for delta-oriented SPLs of Java-like programs [14]. Although, as discussed in Section 6, due to the differences between statecharts and Java-like programs the formulation of the analysis was not straightforward, this experience strengthened our belief that the underlying technique is quite flexible (i.e., it can be customized to different kind of variant languages).

Delta-oriented programming (like feature-oriented programming), is based on the idea of defining operations that modify a program written in a given language (the language in which the variants of the SPL are written). Therefore, in order to provide a formal account of delta-oriented programming for a given language it is needed to provide a precise description

- of the syntax of the language of the variants, and
- of the syntax and of the semantics of the delta operations for it.

Similarly, in order to formalize a family-based analysis for delta-oriented SPLs that exploits the technique used in this paper, it is needed to define

- the analysis for the language of the variants, and
- the "lifting" of the analysis to delta-oriented SPLs.

This suggests that, given a formalization of the analysis like the one presented in this paper or in our previous work [14], formalizing the analysis for SPLs where the language of variants is "significantly" different implies redefining almost from scratch not only the formalization of the language of variants (syntax and analysis), but also the "delta part" of the formalization. A quite challenging direction for future work is to investigate whether it is possible to define a generic framework for delta-oriented programming that can be straightforwardly instantiated to different language of variants.

Acknowledgements

We would like to thank the anonymous reviewers for insightful comments and suggestions for improving the presentation.

References

[1] Eclipse IDE. www.eclipse.org/.

[2] Eclipse Modeling Framework (EMF). www.eclipse.org/modeling/emf/.

[3] The HyVar home page. www.hyvar-project.eu.

[4] Uml state machine. www.uml-diagrams.org/state-machine-diagrams.html.

[5] Yakindu statechart tools. www.itemis.com/en/yakindu/state-machine/.

[6] S. Apel, D. Batory, C. Kästner, and G. Saake. *Feature-Oriented Software Product Lines: Concepts and Implementation.* Springer, 2013.

[7] D. Batory. Feature Models, Grammars, and Propositional Formulas. In *Proc. of SPLC 2005*, volume 3714 of *LNCS*, pages 7–20. Springer, 2005.

[8] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. *IEEE Transactions on Software Engineering*, 30:355–371, 2004.

[9] L. Bettini, F. Damiani, and I. Schaefer. Compositional type checking of delta-oriented software product lines. *Acta Informatica*, 50(2):77–122, 2013.

[10] C. Chesta, F. Damiani, L. Dobriakova, M. Guernieri, S. Martini, M. Nieke, V. Rodrigues, and S. Schuster. A toolchain for delta-oriented modeling of software product lines. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications: 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part II*, volume 9953 of *Lecture Notes in Computer Science*, pages 497–511, Cham, 2016. Springer International Publishing.

[11] A. Classen, M. Cordy, P.-Y. Schobbens, P. Heymans, A. Legay, and J.-F. Raskin. Featured Transition Systems: Foundations for Verifying Variability-Intensive Systems and Their Application to LTL Model Checking. *IEEE TSE*, 39(8):1069–1089, 2013.

[12] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison Wesley Longman, 2001.

[13] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. ACM Press/Addison-Wesley Publishing Co., 2000.

[14] F. Damiani and M. Lienhardt. On type checking delta-oriented product lines. In E. Ábrahám and M. Huisman, editors, *Integrated Formal Methods: 12th International Conference, IFM 2016, Reykjavik, Iceland, June 1-5, 2016, Proceedings*, volume 9681 of *Lecture Notes in Computer Science*, pages 47–62, Cham, 2016. Springer International Publishing.

[15] F. Damiani and M. Lienhardt. Refactoring delta-oriented product lines to achieve monotonicity. In *Proceedings 7th International Workshop on Formal Methods and Analysis in Software Product Line Engineering, FMSPLE@ETAPS 2016, Eindhoven, The Netherlands, April 3, 2016.*, volume 206 of *EPTCS*, pages 2–16, 2016.

[16] F. Damiani and M. Lienhardt. Refactoring delta-oriented product lines to enforce guidelines for efficient type-checking. In *Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications - 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part II*, volume 9953 of *Lecture Notes in Computer Science*, pages 579–596, 2016.

[17] B. Delaware, W. R. Cook, and D. S. Batory. Fitting the pieces together: a machine-checked model of safe composition. In *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2009, Amsterdam, The Netherlands, August 24-28, 2009*, pages 243–252. ACM, 2009.

[18] Y. Dubinsky, J. Rubin, T. Berger, S. Duszynski, M. Becker, and K. Czarnecki. An exploratory study of cloning in industrial software product lines. In *Proceedings of the 2013 17th European Conference on Software Maintenance and Reengineering*, CSMR '13, pages 25–34, Washington, DC, USA, 2013. IEEE Computer Society.

[19] D. Fischbein, S. Uchitel, and V. A. Braberman. A Foundation for Behavioural Conformance in Software Product Line Architectures. In *Proceedings of the ISSTA Workshop on Role of Software Architecture for Testing and Analysis (ROSATEA'06)*, pages 39–48. ACM, 2006.

[20] A. Gruler, M. Leucker, and K. D. Scheidemann. Modeling and Model Checking Software Product Lines. In *FMOODS*, volume 5051 of *LNCS*, pages 113–131. Springer, 2008.

[21] D. Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8(3):231 – 274, 1987.

[22] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM TOPLAS*, 23(3):396–450, 2001.

[23] D. Latella, I. Majzik, and M. Massink. Towards a formal operational semantics of uml statechart diagrams. In P. Ciancarini, A. Fantechi, and R. Gorrieri, editors, *Formal Methods for Open Object-Based Distributed Systems*, pages 331–347, Boston, MA, 1999. Springer US.

[24] M. Lienhardt and D. Clarke. Conflict detection in delta-oriented programming. In *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change - 5th International Symposium, ISoLA 2012, Heraklion, Crete, Greece, October 15-18, 2012, Proceedings, Part I*, volume 7609 of *Lecture Notes in Computer Science*, pages 178–192. Springer, 2012.

[25] M. Lochau, S. Mennicke, H. Baller, and L. Ribbeck. Incremental model checking of delta-oriented software product lines. *Journal of Logical and Algebraic Methods in Programming*, 85(1):245 – 267, 2016. Formal Methods for Software Product Line Engineering.

[26] C. Luna and A. Gonzalez. Behavior specification of product lines via feature models and UML statecharts with variabilities. In *XXVII International Conference of the Chilean Computer Science Society (SCCC 2008), 10-14 November 2008, Punta Arenas, Chile*, pages 32–41. IEEE Computer Society, 2008.

[27] A. Maggiolo-Schettini, A. Peron, and S. Tini. Equivalences of statecharts. In U. Montanari and V. Sassone, editors, *CONCUR '96: Concurrency Theory*, pages 687–702, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[28] M. Nieke, G. Engel, and C. Seidl. Darwinspl: An integrated tool suite for modeling evolving context-aware software product lines. In *Proceedings of the Eleventh International Workshop on Variability Modelling of Software-intensive Systems*, VAMOS '17, pages 92–99, New York, NY, USA, 2017. ACM.

[29] M. Nieke, J. Mauro, C. Seidl, and I. C. Yu. User profiles for context-aware reconfiguration in software product lines. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications*, pages 563–578, Cham, 2016. Springer International Publishing.

[30] I. Schaefer. Variability modelling for model-driven development of software product lines. In *Proc. of Intl. Workshop on Variability Modelling of Software-intensive Systems (VaMoS '10)*, volume 37 of *ICB-Research Report*. Universität Duisburg-Essen, 2010.

[31] I. Schaefer, L. Bettini, V. Bono, F. Damiani, and N. Tanzarella. Delta-oriented programming of software product lines. In *Software Product Lines: Going Beyond - 14th International Conference, SPLC 2010, Jeju Island, South Korea, September 13-17, 2010. Proceedings*, volume 6287 of *Lecture Notes in Computer Science*, pages 77–91. Springer, 2010.

[32] I. Schaefer and F. Damiani. Pure delta-oriented programming. In *Proceedings of the Second International Workshop on Feature-Oriented Software Development, FOSD 2010, Eindhoven, Netherlands, October 10, 2010*, pages 49–56. ACM, 2010.
[33] I. Schaefer, R. Rabiser, D. Clarke, L. Bettini, D. Benavides, G. Botterweck, A. Pathak, S. Trujillo, and K. Villela. Software diversity: state of the art and perspectives. *International Journal on Software Tools for Technology Transfer*, 14(5):477–495, 2012.
[34] C. Seidl, I. Schaefer, and U. Aßmann. Deltaecore - A model-based delta language generation framework. In *Modellierung 2014, 19.-21. März 2014, Wien, Österreich*, volume 225 of *LNI*, pages 81–96. GI, 2014.
[35] N. Szasz and P. Vilanova. Statecharts and variabilities. In *Second International Workshop on Variability Modelling of Software-Intensive Systems, Universität Duisburg-Essen, Germany, January 16-18, 2008, Proceedings*, ICB Research Report, pages 131–140, 2008.
[36] M. H. ter Beek, A. Fantechi, S. Gnesi, and F. Mazzanti. Modelling and analysing variability in product families: Model checking of modal transition systems with variability constraints. *Journal of Logical and Algebraic Methods in Programming*, 85(2):287 – 315, 2016.
[37] S. Thaker, D. S. Batory, D. Kitchin, and W. R. Cook. Safe composition of product lines. In *Generative Programming and Component Engineering, 6th International Conference, GPCE 2007, Salzburg, Austria, October 1-3, 2007, Proceedings*, pages 95–104. ACM, 2007.
[38] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake. A classification and survey of analysis strategies for software product lines. *ACM Comput. Surv.*, 2014.
[39] A. C. Uselton and S. A. Smolka. A compositional semantics for statecharts using labeled transition systems. In B. Jonsson and J. Parrow, editors, *CONCUR '94: Concurrency Theory*, pages 2–17, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg.
[40] M. von der Beeck. A structured operational semantics for uml-statecharts. *Software and Systems Modeling*, 1(2):130–141, Dec 2002.
[41] D. Wille, T. Runge, C. Seidl, and S. Schulze. Extractive software product line engineering using model-based delta module generation. In *Proceedings of the Eleventh International Workshop on Variability Modelling of Software-intensive Systems*, VAMOS '17, pages 36–43, New York, NY, USA, 2017. ACM.

Appendix A. Proof of Theorem 1

Preliminary Lemma.

Lemma 1. *For all products of a product line L, there exists one and only one solution of the formula $L.\texttt{FMandDA}$ that entirely describes the construction of this product. More precisely, given a product, there exists one and only one solution σ of $L.\texttt{FMandDA}$ such that: i) all the variables corresponding to features selected for that product are set to* **true** *in σ; ii) all the variables corresponding to deltas activated for the construction of this product's variant are set to* **true** *in σ; and iii) all the other variables in* $\mathrm{dom}(\sigma)$ *are set to* **false** *in σ.*

Proof. Let p denote the considered product. By definition, p validates the formula $L.\texttt{products}$; moreover, we can extend p as follow to construct a solution of $L.\texttt{FMandDA}$:

$$\sigma \triangleq p \cup \Big(\bigcup_{\texttt{d} \in \mathrm{dom}(L.\texttt{activation})} [\texttt{d} \mapsto p(L.\texttt{activation}(\texttt{d}))] \Big)$$

where $p(\Psi)$ replaces all the variables in Ψ by their value in p.

Let now consider a solution σ' of $L.\texttt{FMandDA}$ with $\sigma'(f) = p(f)$ for all $f \in L.\texttt{features}$, and $\mathrm{dom}(\sigma') = L.\texttt{features} \cup \mathrm{dom}(L.\texttt{activation})$. It is easy to see that $\sigma' = \sigma$. □

Applicability Consistency.

Lemma 2. *The product line L is applicability-consistent if and only if all its variants can be generated without error.*

Proof. In case L has no product, this lemma is trivial. Let now consider that the product line has at least one product: we prove the equivalence by proving each implication independently.

⇒. Suppose chosen a specific product of L: by Lemma 1, there exists one solution σ of $L.\texttt{FMandDA}$ that entirely describes the construction of the product. Because $\texttt{ac}(L)$ si valid, σ is thus also a solution of $\texttt{ac}(L)$.

Let us prove this implication by contradiction: suppose that the variant cannot be generated because a delta operation of a delta $\mathtt{d}$ cannot be applied. We prove the property by induction on the problematic delta operation. For simplicity, we only consider operations on states: the cases for operations on interfaces, declarations and transitions are almost identical.

- Case **adds** *SD*. By rules (R:SO) and (R:SAdd), this operation cannot be applied either because the target state is not composite, or because one of the names ρ in *names*(*SD*) is already present in the statechart. The first option is not possible: the operation is contained into a **modifies** operation. By rule (F:SMod) in Figure 11 and the definition of `pmodifies`, the target state must be composite. The second option is not possible either as it is in contradiction with `padds`: indeed this constraint implies $\mathtt{d} \Rightarrow \mathtt{absent}(\mathtt{d}, \rho)$, meaning that ρ cannot be present when $\mathtt{d}$ is applied.
- Case **removes** `s`. By rule (R:SRem), this operation cannot be applied either because the target state is not composite, or because the state `s` is not present at the current location in the statechart. Like in the previous case, the first option is not possible. The second option is not possible either as it is in contradiction with the formula `premoves`: indeed this constraint implies $\mathtt{d} \Rightarrow \mathtt{present}(\mathtt{d}, \overline{\rho}, \mathtt{s})$, meaning that `s` is present in the right location when $\mathtt{d}$ is applied.
- Case **modifies** `s` $\{SO_1 \ \ldots \ SO_n\}$. By rule (R:SMod), this operation can always be applied.
- Case **initial** `s`. By rule (R:SInit), this operation cannot be applied because the state in which this operation is applied is not composite. Like in the **adds** *SD* case, this is not possible.

$\Leftarrow$. Let us prove this implication by contradiction: suppose that all the products can be generated but $\mathtt{ac}(L)$ is not valid. As $\mathtt{ac}(L)$ is not valid, there exists a solution σ of $L.\mathtt{FMandDA}$ which puts to false the right part of the implication in $\mathtt{ac}(L)$. By Lemma 1, σ corresponds to a product of L which, by hypothesis, can be generated. The right part of the implication in $\mathtt{ac}(L)$ can be false for three possible reasons: `padds` or `premoves` or `pmodifies` could be false. Let consider the case where `padds` is false (the other cases being similar). By construction, this means that there exists a product triggering a delta $\mathtt{d}$ such that $\mathtt{absent}(\mathtt{d}, \rho)$ is false for some ρ. By definition of the predicate `absent`, this means that ρ is present when $\mathtt{d}$ is applied during the construction of the product: this is impossible as it would mean that the application of $\mathtt{d}$ would fail (due to rule (R:SAdd) or (R:SMod)). Hence the hypothesis is wrong, and $\mathtt{ac}(L)$ is valid. □

Dependency Consistency.

Lemma 3. *Consider a generated variant P of a product line L: then all elements in P are declared at most once.*

Proof. This is a direct consequence of the semantics of the delta and of the pre-well-formedness property: a delta cannot add an element that is already declared, and when adding a full composite state to a statechart, that state contains no duplicate declaration. Hence, it is not possible to have duplicate declaration in a statechart constructed that way. □

Lemma 4. *Consider that all variants of a product line L can be generated. Then if L does not contain any useless operation, each of its declarations can be found in at least one variant.*

Proof. Let consider one element η in L: this element is declared inside a delta or the base program $\mathtt{d}$. As L follows G1, the formula $P.\mathtt{products} \Rightarrow \neg P.\mathtt{activation}(\mathtt{d})$ is not valid, and so there exists a solution σ that invalidates the formula. Hence, by Lemma 1, σ corresponds to a product p of L that activates $\mathtt{d}$. Moreover, we have that the following constraint is not valid $(P.\mathtt{FMandDA} \wedge \mathtt{d}) \Rightarrow \bigvee_{\mathtt{d}'} \mathtt{d}'$, where $\mathtt{d}'$ removes or modifies η and $\mathtt{d} <_L \mathtt{d}'$. This means that there are no delta after $\mathtt{d}$ that replaces or remove the considered declaration, and so, it is part of the variant corresponding to the product p. □

Lemma 5. *Suppose that L is dependency-consistent and well partially-tyepd. Consider a generated variant P of L. Then P is well-formed.*

Proof. By Lemma 1, the considered variant corresponds to a solution σ of $L.\texttt{FMandDA}$. Hence, by construction and because L is dependency-consistent, σ is also a solution of $\texttt{dc}(L)$. Let now prove that P validates all five properties of statechart well-formedness:

- By Lemma 3, we have that every interface, event or operation is declared at most once. Additionally, let consider that a transition depends on some operation `I.op` (the proof is identical for interfaces and events). By construction of the variant, there exists a delta (or the base program) `d` that added to the variant that transition. Consequently, by construction of the constraint `pdeps` (with rule (F:DAdd) of Figure 10), we have that $\texttt{present}(\texttt{I.op})$ holds, which means that `s` is declared at least once in L.
- As the second point of the statechart well-formedness is identical to the first one but for states and transitions, this point is proven in the same way.
- Let consider a composite state `s` of the variant, and let $\texttt{s}'$ be its initial state. Let $\overline{\rho}$ be the path in which `s` is placed. By construction of the variant, there exists a delta (or the base program) `d` that set in the variant that initial state. Consequently, by construction of the constraint `pdeps` (with rules (F:SAdd) and (F:SInit) of Figure 11), we have that $\texttt{present}(\texttt{s}', \overline{\rho}\ \texttt{s}, \bot)$ holds, which means that $\texttt{s}'$ is a child state of `s`.
- the proof of the fourth point of the statechart well-formedness is similar to the previous one, with rules (F:TAdd) and (F:TD) of Figure 11.
- Let us consider an expression *exp* inside the variant. As the first point is validated, we have that all operations in *exp* are declared. We can then conclude, with the fact that the rules for partially typing ensure that *exp* is well typed.

□

Lemma 6. *Consider that all variants of a type-uniform product line L can be generated and are well-formed. Then if L does not contain any useless operation, it is dependency-consistent and partially-typed.*

Proof. Let prove that the formula `pdeps` is valid. To do so, let consider a element ρ declared inside a delta `d`. By Lemma 4, there exists a variant containing that declaration. Then, as by hypothesis that variant is well-formed, all the dependencies of ρ are satisfied. Hence, the following formula is valid

$$\bigwedge_{(\rho',\omega,\varsigma)} \texttt{present}(\rho',\omega,\varsigma) \qquad \text{with } (\texttt{d},\overline{\rho},\_,\delta) \in \mathit{fadds}(\rho) \cup \mathit{fmodifies}(\rho),\ \texttt{dep} \in \mathrm{dom}(\delta),\ (\rho',\omega,\varsigma) \in \delta(\texttt{dep})$$

This consequently means that the formula `pdeps` is valid, and so L is dependency-consistent.

Similarily, if we consider an expression *exp* inside a delta `d`, there exists a variant that contains this expression. As the variant is well-formed, *exp* is well-typed, and so we can conclude that the rules in Section 4.1 validate *exp*. Hence, L is partially-typed. □

Main Theorem.

Theorem 1. This theorem is a direct consequence of Lemmas 2, 5 and 6. □