Semantics of UML 2.0 Interactions with Variabilities [1]

María Victoria Cengarle [a], Peter Graubmann [b] and Stefan Wagner [a]

[a] *Institut für Informatik*
Technische Universität München
Garching b. München, Germany

[b] *Siemens AG*
Corporate Technology
München, Germany

Abstract

Means for the representation of variability in UML 2.0 interactions, as presented in a previous work, are further formalised and given a mathematically formal semantics. In this way, UML 2.0 interactions can be used in the conception and development of system families within domain and application engineering tasks. Following the transition from domain to application engineering as a configuration endeavour, resolution of the variability according to a given configuration is captured by a denotational semantics for plain interactions extended to the features for the specification of variability. An example based on a previous case study explicates the semantics hereby defined.

Key words: UML interactions, variability, system families, product lines, formal semantics

1 Introduction

System family development is characterised in the large by its focus on variability. This concept prevails throughout all engineering phases in both the domain and the application engineering tasks of a system family. For many of the thereby necessary steps, a thorough and detailed description of component behaviour is indispensable.

UML interactions as defined by the upcoming UML 2.0 standard [13], and the cognate Message Sequence Charts (as standardised by the ITU-T [8]), have proven to be a highly useful means for behaviour descriptions, well accepted and widely used in industry as well as in academia. Both language variants however lack the possibility to explicitly describe variability. In this paper, we propose language

[1] This work was partly sponsored by the BMBF within the Eureka Σ! 2023 Programme by the ITEA projects ip0004 *CAFÉ* and ip02009 *FAMILIES*, and by the DFG project *InTime*.

constructs for an adequate language extension and provide a formal semantics for them. We use to this end the formalism of UML interactions; the result, nevertheless, can be easily transferred to Message Sequence Charts (MSCs).

Generally, UML interactions specify message exchange between system and component instances. The messages are ordered along lifelines and standard operators allow the description of more complex behaviour such as parallel or iterative combination of interactions.

To describe variability, the operator variant is introduced that specifies optional behaviour which depends on how the system family is configured. Moreover, we extend the UML interactions with the operator repeat that allows repetition of instances with their complete message exchange pattern; this is useful when the exact number of instances depends on the configuration of the system family. Finally, a variable scoping concept is introduced.

An alternative approach is pursued in [16] where several stereotypes are proposed including ≪optionalLifeline≫ and ≪variant≫ for UML sequence diagrams, with a corresponding UML Profile defined in [17]. In contrast to this approach, we emphasise the parametric nature of variability and systematise its description through operators. KobrA [1], a development process specifically focused on system family development, uses also a stereotype ≪variant≫ but does not describe it as an operator with formal semantics.

Outline

In Sect. 2 the proposed extensions of UML interactions and their usage in product line engineering are described. Sect. 3 introduces the abstract syntax of interactions with variabilities. Afterwards, Sect. 4 equips these extensions with a denotational semantics. Sect. 5 elucidates the meaning of more involved interactions with repetition and variabilities, taken from the case study in [15]. Finally, Sect. 6 concludes by discussing advantages of our proposal and hints at possible directions of future work.

2 Interactions with Variabilities

We first introduce our proposed extensions to UML interactions in informal terms and describe our understanding of fundamental concepts, such as features or variation points, and the relations between them.

2.1 From Features to Variation Points

A *feature* is an essential aspect or characteristic of a system in a domain. Features can be described as distinctively identifiable abstractions that must be implemented, tested, delivered, and maintained; see [11].

We follow [14] and regard a system family as a collection of software products that are similar in some important respect and have varying features (as, for instance, versions with different levels of security); see also [12]. In this context, the concept of a feature gains a new significance. A feature has a (unique) identifier and a number of associated values, which can be features themselves. A feature can be mandatory, optional, or alternative to other features. So, for instance, a feature of a car may be the car radio, that is, a radio and optionally a media reproducer, either a cassette player or a CD player, whose playing may be interrupted by RDS (radio data system) traffic broadcasts. Note that we deliberately do not demand that a feature must be visible to the user. That is because we want to model also technical features that are hidden inside the product.

There exist methodical approaches such as FODA [10] and FORM [11] that help, on the one hand, to identify the commonalities and variabilities within the system and, on the other, to organise several features into an and/or tree. In this paper, we assume the feature model was built by any of these or other means. The and/or tree associated with the car radio example of above is depicted in Fig. 1.

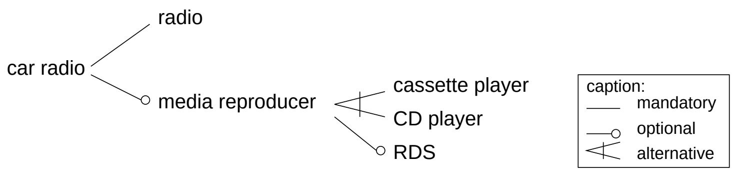


Figure 1. And/or tree for the car radio

Features and variability are also present in use cases, but we particularly argue for the necessity of a separate feature model. The need for a feature model has also been stressed in [7]: Use cases are user oriented, with the objective of determining the requirements of the system family, whereas features are re-user oriented with the objective of organising the results of a commonality and variability analysis of the system family.

In the use cases of a system, variability is mirrored by *variation points*. These are locations within a use case where a variation occurs, and the variation is captured in one or more *variants* depending on the feature to be chosen; see [3,6,9]. For the purposes of this paper, we understand use cases as able to describe the various system usages completely, including in particular all system layers (for instance, low-level redundancy).

Variation points and variants have a (unique) identifier. A non-mandatory feature is mapped to a variant of zero or more variation points (of one or more variation points if the feature is a leaf of the and/or tree), and each variant is associated with at least one non-mandatory feature. Variants and features that are biunivocal can be homonymous. Mandatory features are insofar of no great interest, since they must occur in the respective use cases as any other system characteristic.

For the re-user of the system family, every one of these pieces of information is relevant: the feature model, the use case model with its variation points and variants, and the map relating both models. The requirements derived from the features are of equal importance since they form the basis of the system design. However, their relation to the feature model and the variability description in interactions as well as their traceability is not tackled in this paper and left for further research.

2.2 *From Variation Points to Interactions*

A variant is reflected in some way in the interaction specification of the system, and we speak of *variant occurrences*. These extend the UML interactions which, as bidimensional diagrams, may be modified in their horizontal or their vertical dimension. In the first case, the modification is achieved by adding or removing instances (that is, lifelines). In the second case, by adding, removing or reordering interactions (that is, messages and signals) and by adding, removing or changing conditions. The result must of course be a valid interaction.

We purposely do not resort to interaction constructs for parallel or alternative executions (i.e., the operators par resp. alt with its derivative opt), since configuration management is performed at a different level of abstraction. While these operators allow the on-the-fly decision of which branch to follow, we understand a configuration as the choice within several alternative characteristics at the time the software is deployed, and with this respect no dynamic change can take place. Thus, we keep concerns separated. However, for software, variability during runtime is an issue, for instance dynamic loading of new features. This further step of reconfiguration will be dealt with by an upcoming version of the present proposal.

Mainly, the UML interaction language is extended by two operators: the variability operator variant(–,–) and the repetition operator repeat(–)(–, –, –).[2] The variability operator is intended to capture the variability as defined in a feature model. In variant(B,S), the interaction S is associated with an expression over features given by B. If the chosen configuration entails B, then S is included in the configured system. This means, a variant interaction depends on an expression on feature names that indicate the constellation of features that requires this variant.[3]

The diagrammatic notation for the operator variant is shown in Fig. 2, taken from [4]. In it, the regions enclosed by dashed frames represent the variant occurrences. They are labelled by the variant name and its list of parameters; the second argument of a variant occurrence, i.e. the interaction, is precisely the diagram below the label and within the dashed frame. Notice that in Fig. 2

- it is required by the diagram that the message `m1` be sent from object `o1:C1` to object `o2:C2`,
- the message `m2` is only sent if the lifeline corresponding to the object `o3:C3` actually exists, i.e., if the chosen configuration includes that object,
- the message `m3` is only sent if the variant modifying the vertical dimension exists in the chosen configuration, and
- the message `m4` is only sent if both the lifeline for `o3:C3` exists and the variant for the vertical dimension is chosen.

[2] In [4], which is based on the MSC language, we introduced as a further appearance of the variability operator vp(–) for High-Level MSCs (HMSCs); within UML interactions, this operator is subsumed by variant(–,–).

[3] The issue of dependencies among features and the constraints associated with their selection is out of the scope of this work.

So, in the car radio example, there might be an interruption signal sent by the radio antenna to the media reproducer only in the case that such a reproducer as well as the RDS traffic broadcast reception device are present in the chosen variant.

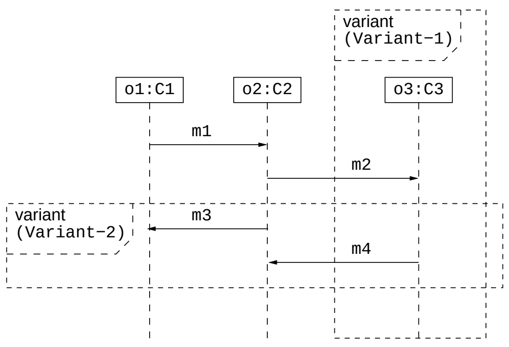


Figure 2. Variant occurrences within an interaction

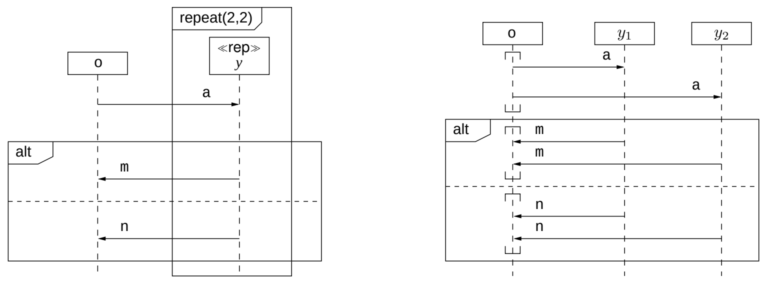
The operator repeat replicates interactions. In repeat$(Y)(m, n, S)$, the set Y contains the lifelines to repeat, m and n are the minimum and maximum number of times the lifelines in Y are to be repeated, and S is the interaction in which the repeated instances occur. As such, repeat resembles the already available loop operator. Whereas loop repeats a message exchange pattern between the same instances, repeat reduplicates the given instances together with their entire message exchange (both, to and from repeated respectively not repeated instances). Thereby, for each message leaving or entering repeat from or to a repeated instance it is assumed that all its repetitions end respectively originate outside the operator in a co-region, that is, sending or reception of all the replica of such a message is done concurrently. The repeated instances are stereotyped as ≪rep≫ in the graphical notation.

Like the loop operator, also the repetition operator allows to specify a lower and a (possibly infinite) upper bound for the number of required repetitions which may, due to variation, depend on a natural number variable. The repetition operator thus allows the introduction of a variable number of instances (which means, a variable number of object or component instantiations) as often necessary when dealing with variation (see for instance the example in Fig. 6 where the variant interaction specifies a varying number of transport vehicles).

A simple example of the diagrammatic use of repeat can be found in Fig. 3(a), whose resolution is depicted in Fig. 3(b). The portion of the lifeline between square brackets, as on the lifeline of object o in Fig. 3(b), specifies a co-region which expresses that the events occurring inside that portion are not ordered.

With the operators, there are two kinds of variables to be introduced. One comes from the configuration, often indicating the number of entities to be configured. This kind of variables has to be defined in the interaction header. The other kind of variables are the instance variables which are essential for the repetition operator, but show their usefulness also in other contexts. The instance variables are bound to the instances of the interaction either at entering a sequence diagram or a repetition.

To be able to handle these variables, we introduce an explicit scoping concept



(a) Interaction with repeat, and (b) its resolution

Figure 3. Use and resolution of the operator repeat

for interactions. Variables are always bound to a scope, for instance, the repeat operator typically binds instance variables. To be even more flexible we use the operator sd (sequence diagram) to determine the scope of a variable. This is important when an interaction is part of a bigger loop and the interaction should be capable of greater flexibility and involve the same instances in different roles alternatingly. Similar as with the repeat operator we attach in this case the stereotype ≪var≫ to any variable instance.

3 Abstract Syntax

The concrete syntax of UML interactions is extended, as sketched above, with operators for variation and repetition as well as a block construction operator for the definition of local variables. This concrete syntax is mapped in some way to the abstract syntax that is introduced in the present section. This mapping is what could be done by any front-end editor and is outside the scope of this article.

We assume two primitive domains for *messages* $\mathbb{M}$ and *instances* $\mathbb{I}$, and denote by $\mathbb{N}$ the set of natural numbers (including zero). We furthermore assume countable sets of *typed variables* $X = (X_\tau)_{\tau \in T}$, where T denotes a set of types comprising Inst for instances each with an associated classifier inherited from a class diagram, Nat for natural numbers, and possibly further types for integer numbers, boolean values, strings, classifiers (from a class diagram), messages, etc. We let $\mathbb{I}(X)$ denote $\mathbb{I} \cup X_{\mathit{Inst}}$, and $\mathbb{N}(X)$ denote $\mathbb{N} \cup X_{\mathit{Nat}}$.

An *event* e is either of the form $\mathsf{snd}(s, r, m)$ or of the form $\mathsf{rcv}(s, r, m)$, representing the dispatch and the arrival, respectively, of message $m \in \mathbb{M}$ from *sender* instance $s \in \mathbb{I}(X)$ to *receiver* instance $r \in \mathbb{I}(X)$. By $\mathbb{E}(X)$ we denote the set of events over variables in X, by $\mathbb{E}$ the set of ground events (i. e., with no occurrences of variables).

The abstract syntax of interactions, as given by the grammar in Tab. 1, is the one of [5] enriched with a repetition construct and variants, and including variables with scoping. *Event* ranges over $\mathbb{E}(X)$, *Nat* over $\mathbb{N}$, *Messages* over subsets of $\mathbb{M}$, *Instance* over X_{Inst}, *Times* over $\mathbb{N}(X)$, and *Name* over finite strings of symbols of a given alphabet.

In addition, skip denotes the empty interaction. The operator opt($-$) is de-

$$
\begin{array}{rcl}
\mathit{Interaction} & ::= & \mathit{Event} \\
& | & \mathit{CombinedFragment} \\
\mathit{CombinedFragment} & ::= & \mathsf{sd}(\{\mathit{Instances}\})(\mathit{Interaction}) \\
& | & \mathsf{strict}(\mathit{Interaction}, \mathit{Interaction}) \\
& | & \mathsf{seq}(\mathit{Interaction}, \mathit{Interaction}) \\
& | & \mathsf{par}(\mathit{Interaction}, \mathit{Interaction}) \\
& | & \mathsf{loop}(\mathit{Nat}, (\mathit{Nat} \mid \infty), \mathit{Interaction}) \\
& | & \mathsf{ignore}(\mathit{Messages}, \mathit{Interaction}) \\
& | & \mathsf{alt}(\mathit{Interaction}, \mathit{Interaction}) \\
& | & \mathsf{neg}(\mathit{Interaction}) \\
& | & \mathsf{assert}(\mathit{Interaction}) \\
& | & \mathsf{repeat}(\{\mathit{Instances}\})(\mathit{Times}, (\mathit{Times} \mid \infty), \mathit{Interaction}) \\
& | & \mathsf{variant}(\mathit{BExp}, \mathit{Interaction}) \\
\mathit{Instances} & ::= & \mathit{Instance}, \mathit{Instances} \\
& | & \mathit{Instance} \\
\mathit{BExp} & ::= & \mathit{BExp} \wedge \mathit{BExp} \mid \mathit{BExp} \vee \mathit{BExp} \\
& | & \neg \mathit{BExp} \qquad \mid (\mathit{BExp}) \\
& | & \mathit{Name}
\end{array}
$$

Table 1
Abstract syntax of interactions with variabilities (fragment)

fined by $\mathsf{opt}(S) = \mathsf{alt}(\mathsf{skip}, S)$, the operator $\mathsf{consider}(-,-)$ by $\mathsf{consider}(M, S) = \mathsf{ignore}(\mathbb{M}(X) \setminus M, S)$. For metaterms involving the operators loop and repeat, we may use $\overline{n}$ to denote a natural number or infinity. Other interaction operators of UML 2.0 as break and critical, as well as message parameters and conditions, are not considered in this work.

Variants of an unconfigured specification are to be ignored if discarded by a chosen configuration. That is, given a configuration $\mathcal{C}$, any term of the form $\mathsf{variant}(B,S)$ and in whichever context it appears, is equivalent to either S if the configuration satisfies the boolean expression B, or to skip otherwise.

In $\mathsf{repeat}(Y)(m, \overline{n}, S)$, "$S$" is the *scope* of the quantifier "$\mathsf{repeat}(Y)$". An occurrence of a variable $y \in X_{\mathit{Inst}}$ is *bound* in an interaction S if either it is the occurrence of y in a quantifier "$\mathsf{repeat}(Y)$" in S, or it lies within the scope of a quantifier "$\mathsf{repeat}(Y)$" in S with $y \in Y$. Otherwise, the occurrence is *free* in S. Free and bound occurrences of variables in a term $\mathsf{sd}(Y)(S)$ are defined likewise. A well-defined interaction has no free occurrences of instance variables.

4 Denotational Semantics

The semantics of an interaction with variabilities and repetitions is based on the semantics of plain interactions; see [5]. The semantics of a plain interaction S states whether a trace t is *positive* or *negative* for the interaction, written $t \models_{\mathrm{p}}$

S and $t \models_{\mathrm{n}} S$, respectively; if t is neither positive nor negative for S, then t is called *inconclusive* for S. The semantics of an extended interaction S depends on a configuration, thus judgements in this setting become $t \models^{\mathcal{C}}_{\mathrm{p}} S$ and $t \models^{\mathcal{C}}_{\mathrm{n}} S$, and denote that a trace t is positive resp. negative for an interaction S configured by $\mathcal{C}$.

The satisfaction relations $\models^{\mathcal{C}}_{\mathrm{p}}$ and $\models^{\mathcal{C}}_{\mathrm{n}}$ are defined the same as $\models_{\mathrm{p}}$ and $\models_{\mathrm{n}}$ for plain interactions; see [5]. For (sub)interactions whose outermost operand is one of variant, repeat or sd, the satisfaction relations are defined as given in Tab. 2, where the satisfaction relation between configurations and Boolean expressions involving names is defined as usual as given in Tab. 3 with *Names*$(\mathcal{C})$ the set of names of the chosen features in the configuration $\mathcal{C}$.

$t \models^{\mathcal{C}}_{\mathrm{p}} \mathsf{variant}(B,S)$	if either $\mathcal{C} \models B$ and $t \models^{\mathcal{C}}_{\mathrm{p}} S$
	or $\mathcal{C} \not\models B$ and $t \models^{\mathcal{C}}_{\mathrm{p}} \mathsf{skip}$
$t \models^{\mathcal{C}}_{\mathrm{p}} \mathsf{repeat}(Y)(m,\overline{n},S)$	if $\exists \sigma : \cup_{i=1}^{n}\{y_i : y \in Y\} \to \mathbb{I} \,.\, \sigma$ is injective and
	$t \models^{\mathcal{C}}_{\mathrm{p}} \mathsf{par}_{i=1}^{\mathcal{C}(m,\overline{n})}(S[y \mapsto \sigma(y_i) : y \in Y])$
$t \models^{\mathcal{C}}_{\mathrm{p}} \mathsf{sd}(Y)(S)$	if $\exists \sigma : Y \to \mathbb{I} \,.\, \sigma$ is injective and
	$t \models^{\mathcal{C}}_{\mathrm{p}} S[y \mapsto \sigma(y) : y \in Y]$
$t \models^{\mathcal{C}}_{\mathrm{n}} \mathsf{variant}(B,S)$	if $t \not\models^{\mathcal{C}}_{\mathrm{p}} \mathsf{variant}(B,S)$
$t \models^{\mathcal{C}}_{\mathrm{n}} \mathsf{repeat}(Y)(m,\overline{n},S)$	if $t \not\models^{\mathcal{C}}_{\mathrm{p}} \mathsf{repeat}(Y)(m,\overline{n},S)$
$t \models^{\mathcal{C}}_{\mathrm{n}} \mathsf{sd}(Y)(S)$	if $t \not\models^{\mathcal{C}}_{\mathrm{p}} \mathsf{sd}(Y)(S)$

Table 2
Semantics of interactions (fragment)

$\mathcal{C} \models B_1 \wedge B_2$	if $\mathcal{C} \models B_1$ and $\mathcal{C} \models B_2$
$\mathcal{C} \models B_1 \vee B_2$	if $\mathcal{C} \models B_1$ or $\mathcal{C} \models B_2$
$\mathcal{C} \models \neg B$	if $\mathcal{C} \not\models B$
$\mathcal{C} \models (B)$	if $\mathcal{C} \models B$
$\mathcal{C} \models \mathit{name}$	if $\mathit{name} \in \mathit{Names}(\mathcal{C})$

Table 3
Satisfaction relation of variant conditions

Upper and lower bounds of a repeat operator, be they variable or not, are disambiguated with a single actual natural number determined by the configuration. A term $\mathsf{repeat}(Y)(m,\overline{n},S)$ becomes equivalent to $\mathsf{par}_{i=1}^{\mathcal{C}(m,\overline{n})} S[y \mapsto y_i : y \in Y]$ where $\mathcal{C}(m,\overline{n})$ is a natural number within the bounds specified for the repetition; if no such natural number exists, then $\mathcal{C}(m,\overline{n}) = 0$ and the repetition is equivalent to skip. Notice that we take advantage of the associativity of the operator par.

Notice moreover that the substitution of actual instances for instance variables occurs each time the repeat (resp. sd) operator is entered. This means, if the repetition is inside a loop construct, then each iteration substitutes the instance variables anew. If this is not the wanted behaviour, then the repeat operator might be placed outside the loop operator. Likewise for the sd operator.

The interaction depicted in Fig. 2 is termed as

$$S = \mathsf{seq}(\mathsf{snd}(\texttt{o1}, \texttt{o2}, \texttt{m1}), \mathsf{seq}(\mathsf{rcv}(\texttt{o1}, \texttt{o2}, \texttt{m1}), S')) \text{ with}$$

$$S' = \mathsf{seq}(S_1, \mathsf{seq}(S_2, S_3))$$

$$S_1 = \mathsf{variant}(\texttt{Variant-1}, S'_1) \qquad S'_1 = \mathsf{seq}(\mathsf{snd}(\texttt{o2}, \texttt{o3}, \texttt{m2}), \mathsf{rcv}(\texttt{o2}, \texttt{o3}, \texttt{m2}))$$

$$S_2 = \mathsf{variant}(\texttt{Variant-2}, S'_2) \qquad S'_2 = \mathsf{seq}(\mathsf{snd}(\texttt{o2}, \texttt{o1}, \texttt{m3}), \mathsf{rcv}(\texttt{o2}, \texttt{o1}, \texttt{m3}))$$

$$S_3 = \mathsf{variant}(\texttt{Variant-1} \text{ and } \texttt{Variant-2}, S'_3)$$

$$S'_3 = \mathsf{seq}(\mathsf{snd}(\texttt{o3}, \texttt{o2}, \texttt{m4}), \mathsf{rcv}(\texttt{o3}, \texttt{o2}, \texttt{m4}))$$

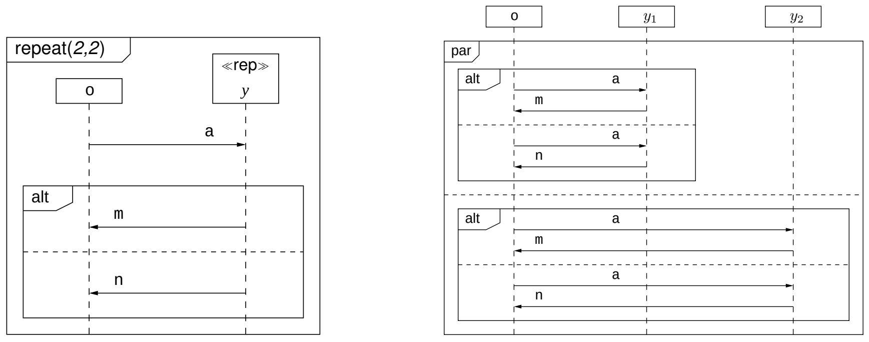
There are four possibilities, according to which one of the possible variants is chosen or removed. Let $\mathcal{C}$ be a configuration for this interaction, let t be a trace. Then t is positive for S configured by $\mathcal{C}$ if it first shows a message m1 sent by o1 and received by o2, (weakly) followed [4] by an interaction t' that is positive for S' likewise configured by $\mathcal{C}$. The trace t' must thus be the (weak) sequential composition of traces t'_1 and t'_2 with t'_1 a positive trace for $S_1 = \mathsf{variant}(\texttt{Variant-1}, S'_1)$ configured by $\mathcal{C}$. Here is where the configuration plays a decisive role: If $\texttt{Variant-1} \in \mathit{Names}(\mathcal{C})$, then t'_1 must be positive for S'_1, otherwise t'_1 must be positive for skip, that is, t'_1 must be empty. What properties has to fulfil the rest t'_2 of the trace t is easy to see, and where it makes a difference whether $\texttt{Variant-1} \in \mathit{Names}(\mathcal{C})$ or not.

Referring back to our example of Fig. 3(a), notice that this interaction corresponds to the term $\mathsf{alt}(\mathsf{repeat}(\{y\})(2, 2, S_{am}), \mathsf{repeat}(\{y\})(2, 2, S_{an}))$, that is, its main operator is an alt and thus both repeated y-instances must choose the same alt-branch. If the repeated instances must decide independently from each other which alt-branch to choose, then the diagram to be specified is the one pictured in Fig. 4(a) that corresponds to the term $\mathsf{repeat}(\{y\})(2, 2, \mathsf{alt}(S_{am}, S_{an}))$. The resolution is shown in Fig. 4(b). It is a simple exercise to demonstrate that this alternative term exhibits the desired behaviour; in order to also graphically visualise it, in the diagram we need to explicitly identify the instances that are to be repeated by means of the stereotype ≪rep≫.

5 Case Study

The developed semantics from above is used in this section on an example from an extensive case study about variability in MSCs [15]. The specification of a holonic

[4] For details about the semantics of weak composition, the reader may consult [5].



(a) Interaction with repeat and alt, and (b) its resolution

Figure 4. Use and resolution of repeat and alt

flow of material in a production system was used as basis for the case study. In this production system, autonomous vehicles (HTFs) transport engine parts between machine tools where they are processed. The basic specification was extended by several variants of both machine tools and transport vehicle mechanisms in order to turn the basic production system into a larger system family. All these variations were incorporated into a single parameterised model that allows variant selection based on system features.

The feature model of the system family is shown in Fig. 5 in shortened form. We only examine the features that are relevant in the following. The holonic transport system (*HTS*) is the root of the feature model. Interesting in the following is the mandatory feature *HTF* that represents the transport vehicles of the system. These vehicles can either follow a *FixedRoute* each or serve on variable routes (*VarRoute*). In case of variable routes we can decide between a central distributor that assigns work orders to HTFs (*HDist*) or a distributed negotiation process between the HTFs (*HNeg*). Furthermore, it is possible to allow a variable number of HTFs in the system (*HVarA*) or to fix the number of vehicles during configuration (*HFixedA*).

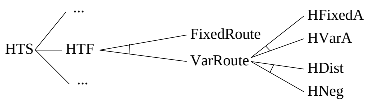


Figure 5. Feature model for the holonic transport system family (fragment)

We present only one exemplary interaction because the whole case study would go beyond the scope of this paper. The interaction is described graphically and in the abstract syntax from Sect. 3, omitting the classes of instances if unambiguous in order to improve readability.

The example chosen from the case study is the interaction for the negotiation of an order shown in Fig. 6. It describes a specific scenario possible for the negotiation between a machine tool and the transport vehicles. If the corresponding feature is chosen, the HTFs can bid in a negotiation process for orders from the machine tools. The HTF with the best bid gets the order. The exemplary scenario specified is that the first bid is the best one and the other HTFs only wait until they get the message that the negotiation is over. Furthermore, even if there is no negotiation but fixed routes for the HTFs, there can be different numbers of HTFs. The example was selected because it contains the variant operator in the vertical as well as the horizontal dimension and has also an occurrence of the repeat operator.

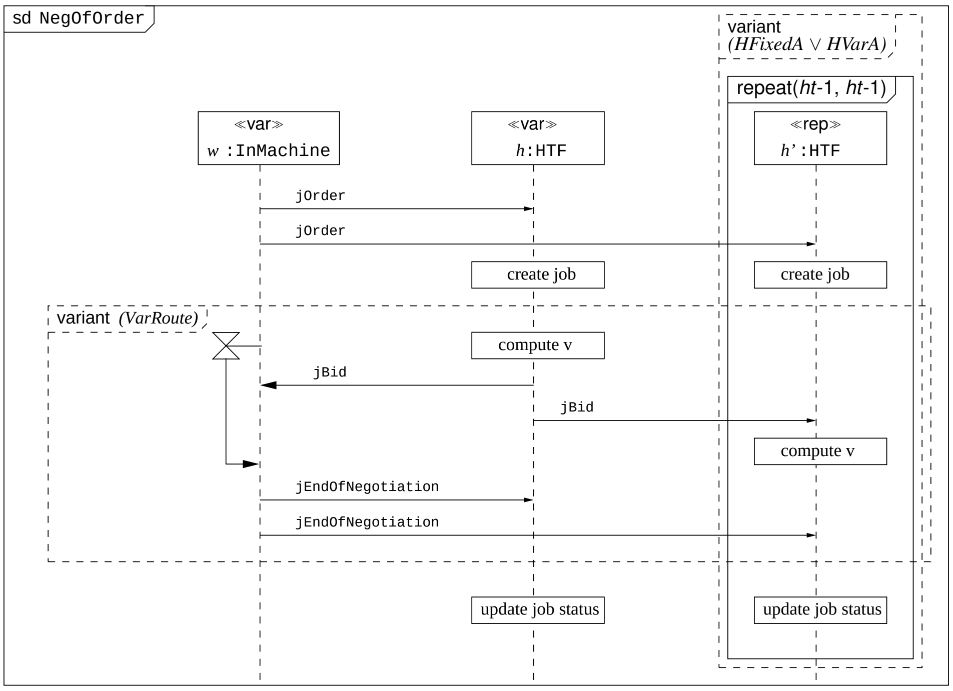


Figure 6. Interaction for the negotiation of an order

The interaction is termed using the abstract syntax of Sect. 3. The convenience of the negation operator ¬ in the variant condition becomes apparent and also the conciseness of the concrete syntax, i.e., of the diagrammatic notation, in comparison with the abstract syntax. Also keep in mind that under normal circumstances a user of the language constructs presented in this paper would not have to deal with the abstract syntax because the translation could be done transparently by an UML tool.

$$S_{NegOfOrder} = \mathsf{sd}(\{w, h\}, \mathsf{seq}(S_{ord}, S_1, S_2))$$
$$S_{ord} = \mathsf{seq}(\mathsf{snd}(w, h, \texttt{jOrder}), \mathsf{rcv}(w, h, \texttt{jOrder}))$$

$$\begin{aligned} S_1 = \mathsf{variant}(&\mathit{VarRoute} \wedge \neg \mathit{HFixedA} \wedge \neg \mathit{HVarA}, \mathsf{seq}(\\ &\mathsf{snd}(h, w, \texttt{jBid}), \mathsf{rcv}(h, w, \texttt{jBid}), \\ &\mathsf{snd}(w, h, \texttt{jEndOfNegotiation}), \\ &\mathsf{rcv}(w, h, \texttt{jEndOfNegotiation}))) \end{aligned}$$

$$S_2 = \mathsf{variant}(\mathit{HFixedA} \vee \mathit{HVarA}, \mathsf{repeat}(\{h'\}, \mathit{ht}-1, \mathit{ht}-1, \mathsf{seq}(\mathsf{snd}(w, h', \mathtt{jOrder}), \mathsf{rcv}(w, h', \mathtt{jOrder}), S_1')))$$

$$\begin{aligned} S_1' = \mathsf{variant}(\mathit{VarRoute}, \mathsf{seq}(&\mathsf{snd}(h, w, \mathtt{jBid}), \mathsf{rcv}(h, w, \mathtt{jBid}),\\ &\mathsf{snd}(h, h', \mathtt{jBid}), \mathsf{rcv}(h, h', \mathtt{jBid}),\\ &\mathsf{snd}(w, h, \mathtt{jEndOfNegotiation}),\\ &\mathsf{rcv}(w, h, \mathtt{jEndOfNegotiation}),\\ &\mathsf{snd}(w, h', \mathtt{jEndOfNegotiation}),\\ &\mathsf{rcv}(w, h', \mathtt{jEndOfNegotiation}))) \end{aligned}$$

The two variables w and h are bound at the top level because the intuition is that the interaction can be executed several times during a system run but most probably with a different machine tool and a different HTF that wins the bidding each time. The binding of the variables using the sd operator allows that.

In the following we analyse two event traces. Given the necessary information from the configuration, we want to evaluate by means of the semantics if the traces are valid for the interaction *NegOfOrder*.

Let $\mathcal{C}$ be a configuration with $\mathit{HFixedA} \in \mathit{Names}(\mathcal{C})$, $\mathit{VarRoute} \in \mathit{Names}(\mathcal{C})$, and $\mathcal{C}(\mathit{ht}) = 3$. That is, $\mathcal{C}$ sets the number of HTFs to three, and lets them serve on variable routes. Let t_1 be the following trace:

$$\begin{aligned} t_1 = {}&\mathsf{snd}(\mathtt{w3:InMachine}, \mathtt{h1:HTF}, \mathtt{jOrder})\cdot\\ &\mathsf{snd}(\mathtt{w3:InMachine}, \mathtt{h2:HTF}, \mathtt{jOrder})\cdot\\ &\mathsf{snd}(\mathtt{w3:InMachine}, \mathtt{h3:HTF}, \mathtt{jOrder})\cdot\\ &\mathsf{rcv}(\mathtt{w3:InMachine}, \mathtt{h1:HTF}, \mathtt{jOrder})\cdot\\ &\mathsf{rcv}(\mathtt{w3:InMachine}, \mathtt{h3:HTF}, \mathtt{jOrder})\cdot\\ &\mathsf{snd}(\mathtt{h1:HTF}, \mathtt{x:InMachine}, \mathtt{jBid})\cdot\\ &\mathsf{rcv}(\mathtt{w3:InMachine}, \mathtt{h2:HTF}, \mathtt{jOrder})\cdot\\ &\mathsf{rcv}(\mathtt{h1:HTF}, \mathtt{w3:InMachine}, \mathtt{jBid})\cdot\\ &\mathsf{snd}(\mathtt{h1:HTF}, \mathtt{h2:HTF}, \mathtt{jBid}) \cdot \mathsf{rcv}(\mathtt{h1:HTF}, \mathtt{h2:HTF}, \mathtt{jBid})\cdot\\ &\mathsf{snd}(\mathtt{h1:HTF}, \mathtt{h3:HTF}, \mathtt{jBid}) \cdot \mathsf{rcv}(\mathtt{h1:HTF}, \mathtt{h3:HTF}, \mathtt{jBid})\cdot\\ &\mathsf{snd}(\mathtt{w3:InMachine}, \mathtt{h1:HTF}, \mathtt{jEndOfNegotiation})\cdot \end{aligned}$$

$\mathsf{snd}(\mathtt{w3:InMachine}, \mathtt{h3:HTF}, \mathtt{jEndOfNegotiation})\cdot$

$\mathsf{snd}(\mathtt{w3:InMachine}, \mathtt{h2:HTF}, \mathtt{jEndOfNegotiation})\cdot$

$\mathsf{rcv}(\mathtt{w3:InMachine}, \mathtt{h3:HTF}, \mathtt{jEndOfNegotiation})\cdot$

$\mathsf{rcv}(\mathtt{w3:InMachine}, \mathtt{h1:HTF}, \mathtt{jEndOfNegotiation})$

Then, $t_1 \models^{\mathcal{C}}_{\mathrm{n}} S_{\mathit{NegOfOrder}}$, since in t_1 the receipt of message `jEndOfNegotiation` from the `InMachine` to the first repeated instance `h2` is missing.

Let furthermore $\mathcal{C}'$ be a configuration with *HFixedA* $\notin$ *Names*$(\mathcal{C}')$, *HVarA* $\notin$ *Names*$(\mathcal{C}')$, and *VarRoute* $\in$ *Names*$(\mathcal{C}')$, let t_2 be the following trace:

$$
\begin{aligned}
t_2 = {} & \mathsf{snd}(\mathtt{w2:InMachine}, \mathtt{h3:HTF}, \mathtt{jOrder})\cdot \\
& \mathsf{rcv}(\mathtt{w2:InMachine}, \mathtt{h3:HTF}, \mathtt{jOrder})\cdot \\
& \mathsf{snd}(\mathtt{h3:HTF}, \mathtt{w2:InMachine}, \mathtt{jBid})\cdot \\
& \mathsf{rcv}(\mathtt{h3:HTF}, \mathtt{w2:InMachine}, \mathtt{jBid})\cdot \\
& \mathsf{snd}(\mathtt{w2:InMachine}, \mathtt{h3:HTF}, \mathtt{jEndOfNegotiation})\cdot \\
& \mathsf{rcv}(\mathtt{w2:InMachine}, \mathtt{h3:HTF}, \mathtt{jEndOfNegotiation})
\end{aligned}
$$

Then, t_2 is a positive trace for $S_{\mathit{NegOfOrder}}$ configured by $\mathcal{C}'$, i. e., $t_2 \models^{\mathcal{C}'}_{\mathrm{p}} S_{\mathit{NegOfOrder}}$. Notice, however, that $\mathcal{C}'$ is not a valid configuration considering the feature model in Fig. 5. That is, t_2 is not a valid trace of the system family. In other words, feature dependencies play an essential role in the specification of system families. This issue was outside the scope of the present work, the semantics above focus on a different matter. We assume those dependencies checked independently, since this check is on a different layer of abstraction and moreover to be performed in an earlier stage in the development. For a detailed survey, the reader may consult, for instance, [2].

6 Conclusions

The above presented semantics complements and refines the extension to UML 2.0 interactions proposed in [4]. The operator $\mathsf{variant}(B, S)$ specifies an optional interaction S, whose existence in the configured interaction depends on the configuration implying the Boolean expression B. In this context, a Boolean expression B is a name (or feature), a negated Boolean expression, or a conjunction or a disjunction of Boolean expressions. A configuration satisfies a name if the configuration chooses that feature; satisfaction of negation, conjunction and disjunction is defined as usual. The optional interaction S can thus add interactions and lifelines, as shown in the excerpt of Sect. 5 that is extracted from [15].

For copying analogous lifelines, when their exact number is unknown beforehand, one can take advantage of the operator $\mathsf{repeat}(Y)(m, \overline{n}, S)$. By the time of configuring, the number of lifelines is disambiguated, that is, the instances in Y are repeated $\mathcal{C}(m, \overline{n}) = k$ times, where k is a natural number given by the configuration $\mathcal{C}$, greater than or equal to m, and additionally less than or equal to $\overline{n}$ if the repeat was given a finite upper bound. If no such k exists, e. g. in the case where m and $\overline{n}$ are variables set by the configuration to natural numbers that make the choice of a suitable k impossible, then the term is equivalent to skip. The semantics of repeat$(Y)(\,m, \overline{n}, S)$ is given by the set of traces that positively satisfy the interaction S' obtained from S by duplicating k times the variable lifelines listed in Y and substituting these duplicated instance variables by different actual instances of the appropriate type. In some sense the repeat defines an existential quantifier.

We introduce a further binding operator sd$(Y)(\,S)$ which allows the declaration of instance variables Y local to an interaction S. Semantically, sd$(Y)(\,S)$ is equivalent to repeat$(Y)(\,1, 1, S)$. We do however prefer to define a different operator in order to keep concerns separated.

As already pointed out in [4], these operators have been proven sufficient for the purposes of the involved case study reported in [15]; we nevertheless do not raise the claim that the extensions here presented do suffice to cope with every single variation that a system family may present. The mathematical precise formalisation of the semantics of the newly introduced operators gave rise to slight adjustments as presented in the previous sections.

In this way, we have equipped UML 2.0 interactions with means for the formal yet concise specification of variabilities. In a forthcoming version of the language, we will include a reconfiguration mechanism. Considering the acceptance of UML and MSCs in industrial scale developments, and the imperiousness of configuration management as well as the use of system families (or product lines) in order to cope with larger scale software systems, we are convinced of the usefulness of the approach.

References

[1] Atkinson, C., J. Bayer, C. Bunse, E. Kamsties, O. Laitenberger, R. Laqua, D. Muthig, B. Paech, J. Wüst and J. Zettel, “Component-based Product Line Engineering with UML,” Component Software Series, Addison-Wesley, 2002.

[2] Benavides, D., P. Trinidad and A. Ruiz-Cortés, *Automated Reasoning on Feature Models*, in: *17th Conference on Advanced Information Systems Engineering (CAiSE’05, Proceedings)*, LNCS **3520** (2005), pp. 491–503.

[3] Bühne, S., G. Halmans and K. Pohl, *Modelling Dependencies between Variation Points in Use Case Diagrams*, in: *9th International Workshop on Requirements Engineering – Foundation for Software Quality (REFSQ’03, Proceedings)* (2003), pp. 59–69.

[4] Cengarle, M. V., P. Graubmann and S. Wagner, *From Feature Models to Variation Representation in MSCs*, in: J. Bosch, editor, *2nd Groningen Workshop on Software Variability Management (SVM’04, Proceedings)* (2004), pp. 49–60.

[5] Cengarle, M. V. and A. Knapp, *UML 2.0 Interactions: Semantics and Refinement*, in: J. Jürjens, E. B. Fernández, R. France and B. Rumpe, editors, *Workshop on Critical Systems Development with UML (CSDUML'04, Proceedings)* (2004), pp. 85–99.

[6] Clements, P. and L. Northrop, "Software Product Lines: Practices and Patterns," SEI Series in Software Engineering, Addison-Wesley, 2001.

[7] Griss, M. L., J. Favaro and M. d'Alessandro, *Integrating Feature Modeling with the RSEB*, in: *5th International Conference on Software Reuse (Proceedings)* (1998), pp. 76–85.

[8] International Telecommunication Union, *ITU-T Recommendation Z.120 (11/99): Message Sequence Chart (MSC)* (2001), Genève.

[9] Jacobson, I., M. Griss and P. Jonsson, "Software Reuse: Architecture, Process and Organization for Business Success," ACM Press, Addison-Wesley, 1997.

[10] Kang, K. C., S. G. Cohen, J. A. Hess, W. E. Novak and A. S. Peterson, *Feature-Oriented Domain Analysis (FODA) Feasibility Study*, Technical Report CMU/SEI-90-TR-021, Carnegie Mellon University, Software Engineering Institute (1990).

[11] Kang, K. C., S. Kim, J. Lee, K. Kim, E. Shin and M. Huh, *FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures*, Annals of Software Engineering **5** (1998), pp. 143–168.

[12] Nitto, E. D. and A. Fuggetta, *Product families: what are the issues?*, in: *10th International Software Process Workshop (ISPW'96, Proceedings)* (1996), pp. 51–53.

[13] Object Management Group, *Unified Modeling Language Specification, Version 2.0 (adopted draft)*, Technical report, OMG (2003).

[14] Sutton, S. M., Jr. and L. J. Osterweil, *Product families and process families*, in: *10th International Software Process Workshop (ISPW'96, Proceedings)* (1996), pp. 109–111.

[15] Wagner, S., M. V. Cengarle and P. Graubmann, *Modelling System Families with Message Sequence Charts: A Case Study*, Technical Report TUM-I0416, Institut für Informatik, Technische Universität München (2004).

[16] Ziadi, T., J.-M. Jézéquel and F. Fondement, *Product Line Derivation with UML*, in: *Groningen Workshop on Software Variability Management (Proceedings)* (2003).

[17] Ziadi, T., J.-M. Jézéquel and F. Fondement, *Towards a UML Profile for Software Product Lines*, in: *5th International Workshop on Software Product-Family Engineering (PFE'03, Proceedings)*, LNCS **3014** (2004).