From Feature Models to Variation Representation in MSCs*

María Victoria Cengarle[1], Peter Graubmann[2], and Stefan Wagner[1]

[1] Technische Universität München
{cengarle|wagnerst}@in.tum.de
[2] Siemens AG
Peter.Graubmann@siemens.com

Abstract. This paper discusses variation representation in MSCs for their use in the conception and development of system families. For the system family to be, a feature model and a variation point modelling within use cases are assumed given. The language of MSCs is extended with two macrooperators and one operator that allow the representation of variabilities. The macrooperators are to be resolved in terms of a given configuration. Both a grammar and a diagrammatic representation of these operators are given, and their semantics is stated in informal terms.

1 Introduction

The complexity of modern undertakings in software intensive realms demands a discipline on the right level of abstraction. Configuration management of products varying only in more or less peripheral aspects deserves a chapter of its own. So, within software engineering the concept of system families (or product lines) has attained a noticeable status. They are a salient exponent of the praised principle of reuse—from reuse of software requirements through reuse of software code. Software reuse and particularly system families improve productivity and quality; see [5].

At the same time, the Unified Modelling Language (UML) has found broad acceptance both in industry and academia. The UML is more properly a set of languages that allow the specification of software systems from different points of view. In the last revision of UML (see [15]), a dialect of Message Sequence Charts (MSC, see [8]) replaced the quite inexpressive language of interactions of past versions. This was the right choice since MSCs, independently, had gained many supporters particularly in industry.

Surprisingly, however, neither UML interactions nor MSCs offer means for the concise representation of variabilities of a system family. The management of variabilities doubtless is of vital importance and, considering the acceptance of UML and MSCs in industrial scale developments, the extension of the languages enabling configuration management is imperious.

* This work was partly sponsored by the BMBF within the Eureka $\Sigma!$ 2023 Programme, by the ITEA projects ip0004 *CAFÉ* and ip02009 *FAMILIES*, and by the DFG project *InTime*.

The present work reports the syntax and informal semantics of a new extension to the MSC language that enables the specification of variability, and that can be seamless translated into UML interactions. This extension consists, roughly speaking, of facultative portions of MSC diagrams that are parametric and can be instantiated or even removed according to the configuration chosen. The proposed extensions have emerged from the exhaustive case study reported in [16].

Outline. In Sect. 2, we introduce the concepts of feature, variation point and variant, their relationship, and their purpose. Sect. 3 reports how to diagrammatically represent variabilities in MSCs. Sect. 4 presents excerpts of a non-trivial example. Finally, in Sect. 5, we discuss some ongoing investigations on how to formalise the general process of analysing commonalities and variabilities within a system family, and conclude with an outlook to future research.

2 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 [12].

We follow [10] and regard a system family as a collection of software products that are similar in some important respect and have varying features as e.g. versions with different levels of security; see also [14]. Thus in this context the concept of 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 feature(s). So for instance a feature of a car is the car radio, that is, a radio and optionally a media reproductor, either a cassette player or a CD player, whose playing may be interrupted by RDS (radio data system) traffic broadcasts.

There exist methodical approaches such as FODA [11] and FORM [12] 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. We assume given the feature model, 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.

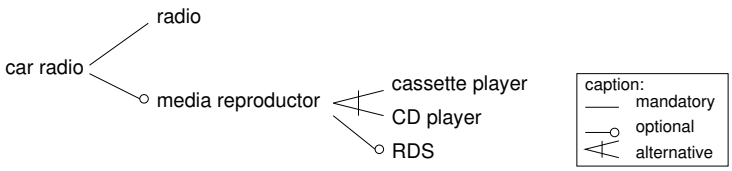


Fig. 1. And/or tree for the car radio

Features and variability are also present in use cases, but a separate feature model constitutes a sine qua non. The need for a feature model has been pointed out in [7]:

Use cases are user oriented, with the objective of determining the requirements of the system family, whereas features are reuser oriented with the objective of organising the results of a commonality and variability analysis of the system family.

In the use cases of the 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 [2,4,9]. For the purposes of this article, we understand use cases as able to describe every one of the different uses of the system, and in particular containing all layers of the system (e.g., 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 variantion 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.[3] Mandatory features are insofar of no great interest, since they must occur in the respective use cases as any other system characteristic. The structure of variation points is described by the grammar in Tab. 1.

For the reuser 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 these two models.

UseCase	::=	*UseCaseName Actors Dependencies Description*
	...	
Description	::=	*VariationPointList* : *Text*
VariationPointList	::=	*VariationPoint VariationPointList*
	|	ϵ
VariationPoint	::=	*VariationPointName* (*VariantList*)
VariantList	::=	*VariantName* , *VariantList*
	|	*VariantName*

Table 1. Abstract syntax of use cases: variation point fragment

3 From Variation Points to MSCs

A variant is reflected in some way or another in the MSCs specifying the interactions of the system, and we speak of *variant occurrences*. The structure of variant occurrences is described by the attribute grammar in Tab. 2 (cf. [3]). Therein, *Basic* ranges over the basic interactions, and if an *ActualParameter* is given then its value is within

[3] We conjecture that a clean use case model allows a 1:1 relationship between variants and features that are leaves of the and/or tree, and that features that are inner nodes of the and/or tree must not be translated into a variant of the use case model.

the expected ones for the paired *FormalParameter* in the context of the corresponding *VariantName*.[4]

Interaction	::=	*Basic*
	|	*CombinedFragment*
CombinedFragment	::=	...
	|	variant(*VariantName ArgumentList*, *Interaction*)
		{*allowedValues*(*VariantName.name*, *ArgumentList.list*)}
ArgumentList	::=	(*NonEmptyArgumentList*)
		{*ArgumentList.list* := *NonEmptyArgumentList.list*)}
	|	ϵ
		{*ArgumentList.list* := nil)}
NonEmptyArgumentList	::=	*Argument* , *NonEmptyArgumentList*
		{ *NonEmptyArgumentList.list* :=
		[(*Argument.name*,*Argument.value*) |
		$NonEmptyArgumentList_1.list$] }
	|	*Argument*
		{ *NonEmptyArgumentList.list* :=
		[(*Argument.name*,*Argument.value*)] }
Argument	::=	*FormalParameter* : *ActualParameter*
		{ *Argument.name* := *FormalParameter.name*;
		Argument.value := *ActuaParameter.value* }
	|	*FormalParameter*
		{ *Argument.name* := *FormalParameter.name*;
		Argument.value := null }

Table 2. Abstract syntax of interactions: variant occurrence fragment

The syntax of variant occurrences extends that of MSCs. These, 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 (i.e., lifelines). In the second case, by adding, removing or reordering interactions (i.e., messages and signals) and by adding, removing or changing conditions. The result must of course be a valid MSC.

We purposedly do not resort to MSC 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 e.g. which branch to follow, a configuration is the choice within several alternative characteristics at the time the software is deployed, and with this respect no dynamic change can take place. In this way, thus, we keep concerns separated. The

[4] Alternatively, one may choose to derive parameters from the feature model using Boolean functions or dependent types. We prefer the above introduced approach since it better fits into the concept of MSCs.

operator variant$(--,-)$ is more precisely a metaoperator, or a macro, that must be statically resolved in order to obtain a plain MSC.

A diagrammatic notation is proposed in Fig. 2. Therein, the regions enclosed by dashed frames are the variant occurrences. They are labelled by the variant name and its list of parameters as given by the grammar in Tab. 2; the third argument of a variant occurrence, i.e. the interaction, is precisely the diagram below the label and within the dashed frame. Notice that

- 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 variation point instance 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 variation for the vertical dimension is chosen.

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

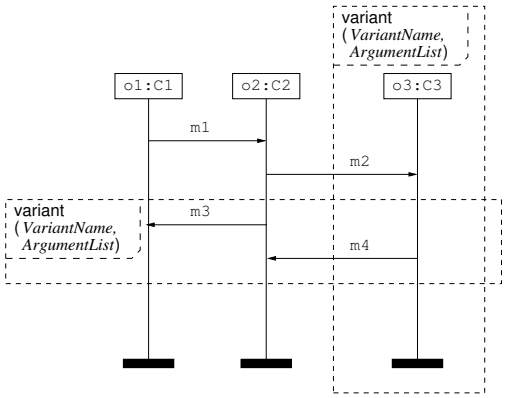


Fig. 2. Variant occurrences within a MSC

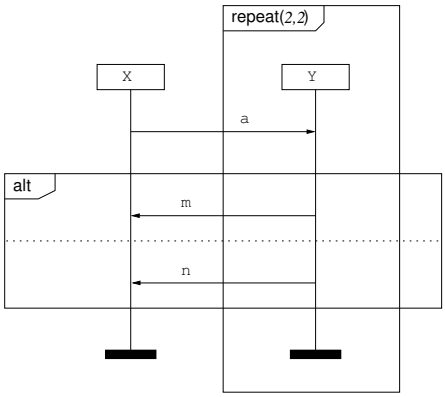
When specifying optional lifelines, it might be the case that the number of (analogous) lifelines is unknown beforehand. This motivated the introduction of a further operator repeat$(-,-,-)$, whose syntax is formalised in the grammar in Tab. 3 extending Tab. 2; there, *Nat* ranges over the natural numbers (including zero). Given an interaction repeat(m, n, S), once the natural numbers m and n are known, the interaction S is copied an arbitrary number of times between m and n. If n is ∞, then S is copied at least m times; if $m > n$ or the choice is to repeat zero times, then the repeat construct

$$\begin{array}{lll} \textit{CombinedFragment} & ::= & \ldots \\ & & \mid \mathsf{repeat}(\textit{Nat}, (\textit{Nat} \mid \infty), \textit{Interaction}) \end{array}$$

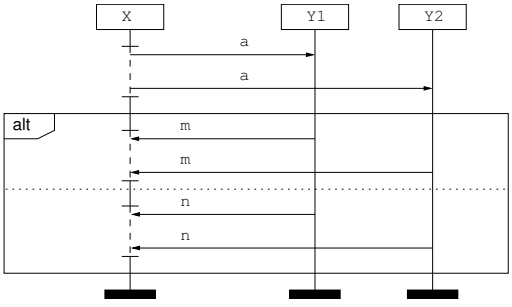
Table 3. Abstract syntax of interactions (contd.): the operator repeat(−, −, −)

is equivalent to skip. W.r.t. lifelines the operator repeat(−, −, −), thus, behaves similar to the operator loop w.r.t. message dispatch/reception.

A simple example of diagrammatic use of repeat(m, n, S) can be found in Fig. 3(a), whose resolution is depicted in Fig. 3(b). A dashed portion on a lifeline, as on the lifeline of object X in Fig. 3(b), expresses that the events occurring on that portion are not ordered.



(a) MSC with repeat, and



(b) its resolution

Fig. 3. Use and resolution of the metaoperator repeat(−, −, −)

Notation. We let repeat(n, S) denote repeat(n, n, S). If two variant occurrences V_1 and V_2 have lists of arguments L_1 and L_2 compatible w.r.t. an interaction S,[5] then we abbreviate variant($V_1\ (L_1)$, variant($V_2\ (L_2), S$)) to variant($V_1(L_1)/V_2(L_2), S$).

High-Level MSCs (HMSCs) allow the abstraction of subgraphs in order to handle complex constellations of interactions. It turns out that an accordingly abstract notation for variabilities within HMSCs sheds light upon the strived abstraction. This considerations gave rise to a notation tailored for HMSCs, in the style of "higher-level" decision nodes, as schematically shown in Fig. 4. The nodes labelled with the keyword vp(*Name*) are the occurrences of the variation points of the use case model; the labels of the outgoing edges are the variant occurrences (without their parameters).

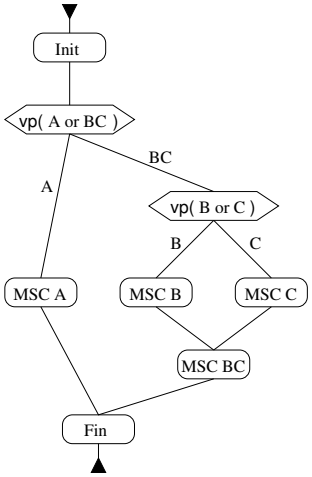


Fig. 4. Variation point occurrences within a HMSC

4 Example

We have carried out a case study that is reported in [16].[6] The needs that emerged trying to use MSCs for system family specification are covered by the constructs introduced above. The case study consisted in the specification and the design of a production system for engine parts or work pieces. These are deburred and washed by machine tools. Autonomous transport vehicles carry the work pieces between machine tools,

[5] L_1 and L_2 are compatible w.r.t. S if $FV(S) \subseteq L_1 \cap L_2$, where $FV(S)$ denotes the set of free variables of S.

[6] The purpose of the case study was twofold: On the one hand, to experiment with the connector construction (see [6]) for the specification of structured interfaces in MSCs, and on the other, to ponder the limitations of MSCs for their use as specification language for system families. The experience with the connector construct is out of the scope of this paper.

from the input storage, and to the output storage. The machine tools may have a buffer for (treated or untreated) work pieces in addition to their workplace.

The variations within this system family are the following. The number of transport vehicles or of machine tools may vary. The sequence of the different treatments on the work pieces may be fixed or dependent on the work piece itself. The transport of work pieces is decided by a negotiation among the vehicles, or each vehicle transports pieces between two machine tools fixed for it. In the first case, the negotiation of orders may be centralised in a distributor, or not. And finally, there may be many different machine tools of the same kind working at for instance different paces. The feature model, with hopefully self explaining feature names, can be found in Fig. 5.

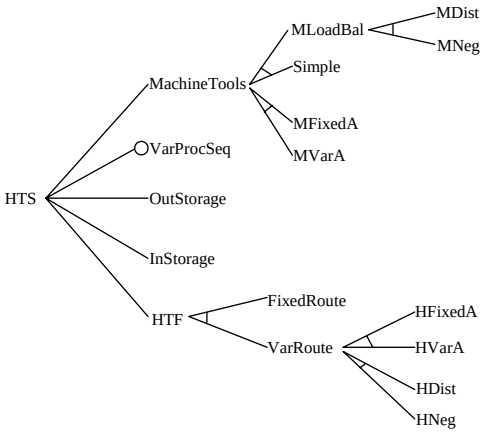


Fig. 5. Case study: feature model

The overall operation of the system family, i.e. including occurrences of variation points and of their variants, can be found in the HMSC depicted in Fig. 6. Variation points are the decision nodes marked with the keyword vp(*Name*), the associated alternative variants label the edges leaving the variation point. If the parent feature of these alternative variants has no further children, then the identifier of this parent feature is chosen to be the *Name* of the variation point in the HMSC; otherwise a new name is introduced as for instance the names HLoadBal and MTrans in Fig. 6. Notice that not all leaves of the feature model label an edge in the HMSC going out of a variation point, and that also some of the inner nodes of the feature model label an edge of the HMSC going out of a variation point. The reason for this is that every variability not necessarily is visible in the most abstract view of a system family, but then can be found in the detailed MSCs.

The expressive power and abbreviation capability of the extensions by means of variant$(--,-)$ and repeat$(-,-,-)$ can be appreciated in the enriched MSCs for the (slightly simplified) negotiation of orders that is reproduced in Fig. 7. In it, the message

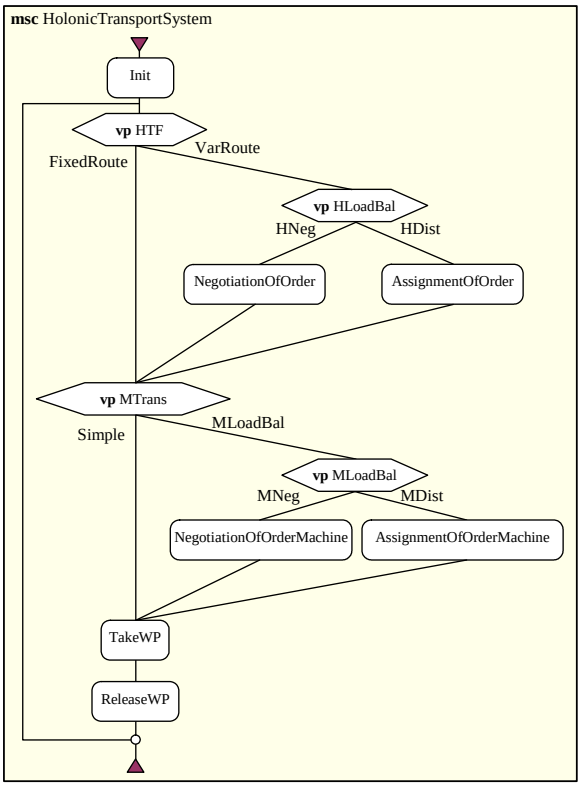


Fig. 6. Case study: high-level MSC

jOrder expresses the requirement of a machine tool or the input storage to be released of a work piece, and is sent to all vehicles. Given that the number of vehicles is greater than or equal to one but unknown beforehand, a parameter is the number *ht* of vehicles that is used to duplicate $ht - 1$ times a lifeline; this is a variation in the horizontal dimension of the diagram. After reception of a jOrder, a job is created within each vehicle. If the route of the vehicles may vary, then the one in fact taking the work piece from its initial position to its destination is the vehicle that offers the best bid for the job. In case the route of the vehicles is fixed, an explicit negotiation for the best offer is not necessary, and the sending of the messages jBid and jEndOfNegotiation is then disabled. This is a variation in the vertical dimension of the diagram.

5 Conclusions and Outlook

The contribution of this work can be summarised as follows. We have introduced three metaoperators into the MSC language that act as macros and, once resolved, deliver valid MSCs. In this context, macro resolution is configuration.

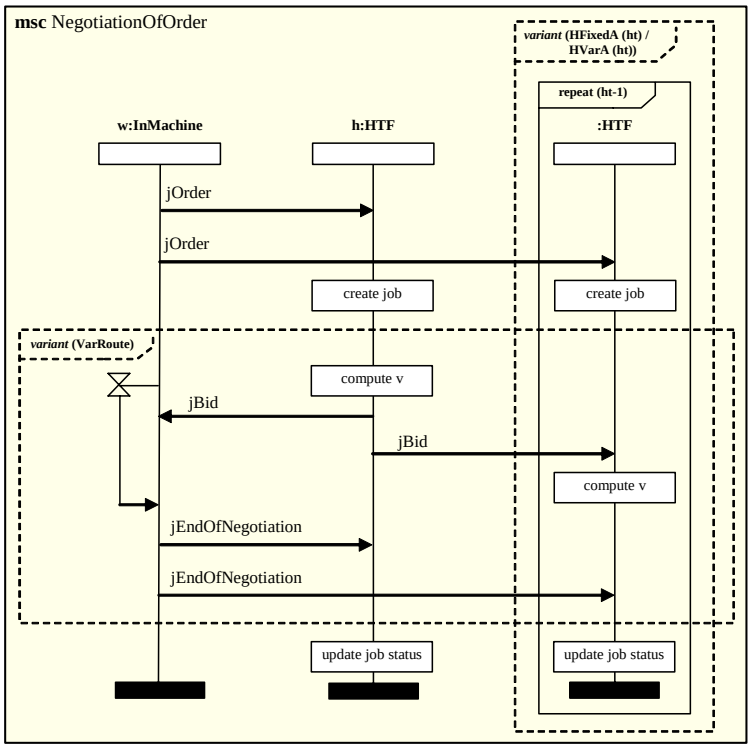


Fig. 7. Case study: MSC for the negotiation of orders

The metaoperator variant(− −, −) manipulates MSCs in their horizontal and vertical dimensions in that it adds, modifies, or removes interactions and instance axes. For copying analogous instance axes, when their exact number is unknown beforehand, one can take advantage of the operator repeat(−, −, −). The metaoperator vp(−) is used in HMSCs and defines high-level decision nodes which are resolved at the time of configuration.

These (meta)operators have been proved to suffice for the purposes of the involved case study reported in [16]. Moreover, due to the fact that a MSC is exactly the union of the above mentioned three dimensions (i.e., vertical and horizontal dimensions plus abstraction), we think that those (meta)operators are indeed enough for variability representation in MSCs.

In the literature, one finds approaches with similar intention:

– The KobrA approach [1] is a development process specifically focused on system development. It contains complete guidance from specification to realisation but does not allow modeling in terms of sequence diagrams.

– In the approach in [13] the special symbol ≪V≫ is used to represent variability in a UML class diagram that serves as domain model. Alternatives can only be expressed in accompanying text.
– In [17] also UML class diagrams are used based on a feature model similar to ours. However, the dependencies from the feature model are expressed as OCL constraints on the class diagram. Also the sterotype ≪optional≫ is introduced to denote optional classes. This was extended in [18] to a UML profile that also includes stereotypes such as ≪optionalLifeline≫ and ≪variant≫ for sequence diagrams but lacks an explicit notion of variation points.

Our approach is an alternative to the last one. We emphasise the parametric nature of variability and systematise the development of MSCs with variations.

The meaning of the extensions introduced in this paper was given in informal terms, a formal semantics must be defined. One possibility is to consider a kind of precompiler, which takes an extended (H)MSC and a configuration, and delivers a (H)MSC without occurrences of the metaoperators. The configuration could be conceived as a subtree of the feature model without alternatives and with optional features possibly removed.

Regarding the whole process, from the analysis of commonalities and variabilities of a system family to be through feature and use case modelling to the specification of MSCs, we feel it could conceptually be carried out as follows. Initially, one is more or less precisely aware of the set of all desired system traces, in any of the members of the system family. Those traces are associated to at least one feature, and the set of all traces can be divided (not partitioned) into subsets associated to the features. This reasoning also helps to produce a feature model.

Using the knowledge of which features are optional and which ones alternative to other ones, the corresponding trace sets are declared variants. In doing so, alternative features are organised around a single variation point, whereas optional features belong each to a variation point of their own.

The (still ideal) traces are then used for two purposes: on the one hand for a use case modelling including variation points, and on the other for a raw specification of MSCs with occurrences of variation points and of variants.

In a next step we will formalise the semantics of the extensions introduced and elaborate on the development process.

References

1. Colin Atkinson, Joachim Bayer, Christian Bunse, Erik Kamsties, Oliver Laitenberger, Roland Laqua, Dirk Muthig, Barbara Paech, Jürgen Wüst, and Jörg Zettel. *Component-based Product Line Engineering with UML*. Component Software Series. Addison-Wesley, 2002.
2. Stan Bühne, Günter Halmans, and Klaus Pohl. Modelling Dependencies between Variation Points in Use Case Diagrams. In *9th International Workshop on Requirements Engineering – foundation for Software Quality (REFSQ'03, Proceedings)*, pages 59–69, 2003.
3. María Victoria Cengarle and Alexander Knapp. UML 2.0 Interactions: Semantics and Refinement. In Jan Jürjens, Eduardo B. Fernández, Robert France, and Bernhard Rumpe, editors, *Workshop on Critical Systems Development with UML (CSDUML'04, Proceedings)*,

pages 85–99. Technical Report TUM-I0415, Institut für Informatik, Technische Universität München, 2004.
4. Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. SEI Series in Software Engineering. Addison-Wesley, 2001.
5. Hassan Gomaa. *Designing Software Product Lines with UML: From Use Cases to Pattern-based Software Architectures*. The Addison-Wesley Object Technology Series. Addison Wesley Professional, 2004.
6. Peter Graubmann. Describing Interactions between MSC Components: The MSC Connectors. *Computer Networks, Special Ed.: ITU-T System Desing Languages (SDL)*, 42(3):323–342, 2003.
7. Martin L. Griss, John Favaro, and Massimo d'Alessandro. Integrating Feature Modeling with the RSEB. In *5th International Conference on Software Reuse (Proceedings)*, pages 76–85, 1998.
8. International Telecommunication Union. ITU-T Recommendation Z.120 (11/99): Message Sequence Chart (MSC). Genève, 2001.
9. Ivar Jacobson, Martin Griss, and Patrik Jonsson. *Software Reuse: Architecture, Process and Organization for Business Success*. ACM Press. Addison-Wesley, 1997.
10. Stanley M. Sutton Jr and Leon J. Osterweil. Product families and process families. In *10th International Software Process Workshop (ISPW'96, Proceedings)*, pages 109–111, June 1996.
11. Kyo Chul Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-021, Carnegie Mellon University, Software Engineering Institute, 1990.
12. Kyo Chul Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Euiseob Shin, and Moonhang Huh. FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures. *Annals of Software Engineering*, 5(1):143–168, 1998.
13. David McComas, Stephen Leake, Michael Stark, Maurizio Morisio, Guilherme Travassos, and Michael White. Addressing Variability in a Guidance, Navigation, and Control Flight Software Product Line. In *Product Line Architecture Workshop at Software Product Line Conference (SPLC1, Proceedings)*, 2000.
14. Elisabetta Di Nitto and Alfonso Fuggetta. Product families: what are the issues? In *10th International Software Process Workshop (ISPW'96, Proceedings)*, pages 51–53, June 1996. `http://www.elet.polimi.it/upload/dinitto/papers/ispw10.ps`.
15. Object Management Group. Unified Modeling Language Specification, Version 2.0 (adopted draft). Technical report, OMG, 2003. `http://www.omg.org/cgi-bin/doc?ptc/2003-08-02`.
16. Stefan Wagner, María Victoria Cengarle, and Peter Graubmann. Modelling System Families with Message Sequence Charts: A Case Study. Technical Report TUM-I0416, Institut für Informatik, Technische Universität München, 2004. `http://wwwbib.informatik.tu-muenchen.de/infberichte/2004/TUM-I0416.pdf`.
17. Tewfik Ziadi, Jean-Marc Jézéquel, and Frédéric Fondement. Product Line Derivation with UML. In *Groningen Workshop on Software Variability Management (Proceedings)*, 2003.
18. Tewfik Ziadi, Jean-Marc Jézéquel, and Frédéric Fondement. Towards a UML Profile for Software Product Lines. In *5th International Workshop on Software Product-Family Engineering (PFE'03, Proceedings)*, volume 3014 of *LNCS*, 2004.