PEDECIBA Informática
Instituto de Computación – Facultad de Ingeniería
Universidad de la República
Montevideo, Uruguay

Tesis de Maestría
en Informática

On UML statecharts with variabilities

Pedro Andrés Vilanova Guerra

2012

Bentancourt Alves, Martín
Diseño topológico de redes. Caso de estudio: The augmentation Steiner two-node survivable network problem
ISSN 0797-6410
Tesis de Maestría en Informática
Reporte Técnico RT 12-08
PEDECIBA
Instituto de Computación – Facultad de Ingeniería
Universidad de la República.
Montevideo, Uruguay, diciembre de 2011

PEDECIBA Informática
Universidad de la República
Montevideo, Uruguay

On UML Statecharts with Variabilities

Pedro Andrés Vilanova Guerra
Montevideo, Diciembre de 2011

Trabajo de tesis presentado para la obtención
del grado de Magister en Informática
de la Universidad de la República en el programa de
Maestría del área Informática del PEDECIBA

Director de tesis: Dra. Nora Szasz

Tribunal: Dra. Claudia Pons - Revisora
Dr. Ariel Sabiguero
Dra. Cristina Cornes

Resumen

El uso de métodos formales para el diseño de software contribuye a la confiabilidad y robustez del sistema a construir. A medida que los sistemas se vuelven complejos, el enfoque formal es esencial, debido a que permite la demostrabilidad y verificabilidad del diseño. El diseño formal es un proceso que comienza con la etapa de especificación, en la cual el sistema es definido utilizando un lenguaje de modelado; luego la etapa de verificación, en la cual el sistema es analizado mediante un enfoque de corrección basado en pruebas formales utilizando herramientas matemáticas y, por último, la etapa de implementación, en la cual la especificación se convierte en código ejecutable. El Lenguaje de Modelado Unificado (UML por sus siglas en inglés) es un lenguaje específico ampliamente utilizado en la industria y la academia. Desafortunadamente, carece de una semántica formal que permita el desarrollo de modelos utilizando un enfoque de corrección basado en pruebas formales.

Este trabajo se centra en la especificación formal de familias de sistemas, y, en particular, en la semántica de maquinas de estados de UML (UML Statecharts) con variabilidades y sus aplicaciones a líneas de productos de software. La principal contribución es la definición de un formalismo que permite modelar el comportamiento de una familia de sistemas. Tal comportamiento se describe utilizando UML Statecharts en combinación con Diagramas de funcionalidades (Feature Diagrams), con el fin de representar las funcionalidades comunes y variantes de una familia. Para ello se define una relación de orden entre los UML Statecharts, que representa el hecho de que un statechart posee una estructura más rica que otro. Luego se define con precisión la forma de combinar diferentes extensiones de un mismo statechart. Utilizando estos conceptos, es posible definir el efecto que cada funcionalidad tiene en los productos en los cuales se encuentra presente. Estas definiciones proporcionan una forma muy simple de obtener la especificación del comportamiento de un producto de la línea como la combinación de los UML Statecharts que implementan todas las funcionalidades presentes en un producto en particular. Mas aún, se prueba que la relación de extensión propuesta constituye un refinamiento de comportamiento.

El presente enfoque se compara con el estado del arte y se estudia su aplicación práctica con el fin de visualizar sus beneficios y posibles debilidades. Adicionalmente, con el fin de comprobar la adecuacin de la propuesta, una gran parte de las ideas fueron implementadas en un prototipo utilizando Prolog.

Palabras clave: Máquinas de estado UML, Semántica formal, Modelado formal de variabilidades, Máquinas de estado con variabilidades, Líneas de productos de software.

Abstract

The use of formal methods for software design contributes to the reliability and robustness of the system being constructed. As systems become more complex, the formal approach is essential, as it enables provability and verifiability of the design. Formal design is a process starting with the specification stage, in which the system is defined using a modeling language; it continues with the verification stage, in which the system is analyzed in a provable-correct approach using mathematical tools and finally, the implementation stage, in which the specification is converted into code. The Unified Modeling Language (UML) is a specification language widely used by the industry and the academia. Unfortunately, it lacks a formal semantics for the development of provably-correct models.

This work concerns with the formal specification of families of systems, and in particular with the semantics of UML Statecharts with Variabilities and its applications to Software Product Lines. The main contribution is the definition of a formalism which allows to model the behavior of an entire family of systems. Such behavior is described using UML Statecharts in combination with Feature Diagrams in order to represent the common and variant functionalities of the family. Using an order relation among statecharts, which represents when an UML statechart has a richer structure than another one, it is possible to precisely define how to combine different extensions of the same UML statechart into an integral new one. With these notions, it is possible to describe the effect that each feature has on the products in which it is present. These definitions provide a very simple way to obtain the specification of the behavior of a product of the line as the combination of the statecharts that implement all the features present in that particular product. Moreover, a proof that the proposed extension relation constitutes a behavioral refinement is given. The present approach is also compared with related work and its practical application is studied in order to visualize its benefits and possible weaknesses. Additionally, in order to check the adequacy of the present approach, most of the ideas were implemented in a prototype using Prolog.

Summarizing, this thesis contributes to the formalization of concepts widely used in practice as well as its implementation in modeling and formal reasoning tools.

Key words: UML Statecharts, Formal semantics, Formal variability modeling, Statecharts with variabilities, Software product lines.

Contents

1 Introduction **1**
- 1.1 Contribution of this Work . . . 2
- 1.2 Related Work . . . 3
- 1.3 Outline of this Thesis . . . 4

2 Background **5**
- 2.1 Model-Driven Development . . . 5
- 2.2 Unified Modeling Language . . . 6
  - 2.2.1 UML Statecharts . . . 7
- 2.3 Formal Methods . . . 8
- 2.4 Formal Semantics Approaches of UML Statecharts . . . 8
  - 2.4.1 The Mathematical Approach . . . 9
  - 2.4.2 Rationale of the Transition Based Formalism . . . 10
- 2.5 Software Product Lines . . . 11
  - 2.5.1 Variability Modeling . . . 11
  - 2.5.2 Product derivation . . . 12

3 Syntax and Semantics of UML Statecharts **13**
- 3.1 Syntax . . . 13
- 3.2 Statechart Configurations . . . 15
- 3.3 Computing the Next State . . . 16
- 3.4 Structured Operational Semantics . . . 17
- 3.5 Complete Semantics . . . 19

4 Extension and Union of UML Statecharts **21**
- 4.1 Extension Relation . . . 21
  - 4.1.1 Extension functions . . . 21
- 4.2 UML Statecharts Union . . . 23
  - 4.2.1 Properties of the Union . . . 28
  - 4.2.2 Example . . . 29

5 Variability Modeling **31**
- 5.1 Feature Diagrams . . . 31
  - 5.1.1 Feature Diagrams Syntax . . . 31
  - 5.1.2 Feature Diagram Configurations . . . 33
- 5.2 UML Statecharts with Variabilities . . . 34
  - 5.2.1 Set of UML Statecharts with Variabilities SC* . . . 34
  - 5.2.2 Product Configuration . . . 34

6 Behavioral Refinements **37**
6.1 Extension Relation and Configurations 37
6.1.1 Monotonicity of next Function 40
6.2 Extension Relation as a Behavioral Refinement 43
6.3 On the Actions Generated by the SO Semantics 46

7 The SC* Modeler **51**
7.1 Modeling Approach 51
7.2 Architecture 52
7.2.1 User Interface 53
7.2.2 Metamodel 54
7.2.3 Inference Engine 56

8 Case study **59**
8.1 Problem Description 59
8.2 Use Cases 59
8.3 Formal Description 61
8.3.1 Features 61
8.3.2 Statecharts 62
8.3.3 Feature Impact Description 63
8.3.4 Statechart with Variabilities 66
8.3.5 Examples of Products 66

9 Conclusions and Further Work **71**
9.1 Summary and Conclusions 71
9.2 Further work 72
9.2.1 UML enrichments 73
9.2.2 Relations with Lattice Theory 74
9.2.3 SC* Modeler and Integrated Development Environments 74

Chapter 1

Introduction

Software reusability has become a key challenge for the software industry. Software reuse is the systematic (in contrast to ad hoc) use of existing software assets to construct new software. Although the idea of software reuse has been applied and practiced since programming began, the concept of systematic reuse, as a field of study in software engineering, was introduced by Doug McIlroy in 1969 [Mci69]. McIlroy envisioned the construction of complex systems using parameterized families of software components, able to satisfy the needs of any type of user. Later, David Parnas [Par72] developed the information hiding principle and the idea of program families, which became the engineering foundation for reuse based application development. Soon after, extensive research efforts on software reuse followed, with the result of the development of a great diversity of techniques. All of them includes some sort of abstraction (components seen as a "black box"), selection (catalogues of components), specialization (configuration of components through parameters) and/or integration (combination of components), as pointed out in [Kru92, FK05] and references therein.

Nowadays, reusability is of wide interest because of the need of software engineers to construct highly complex systems in a more reliable, cheaper and timely way [MC08, FK05, MMM95, PD93]. As pointed out in [FK05], a crucial breakthrough in software reuse is the concept of Software Product Lines (SPLs). The key idea is that most software systems are not entirely new, in contrast, they are variants of systems that have already been built. Essentially, a SPL consists of a family of systems that share functionality and satisfy, in general, the needs of a particular user [Gom05, CN02]. Recently, the Software Engineering Institute (SEI) [SEI11a] developed a framework called Product Line Engineering (PLE) for reuse-based development of a family of closely related applications [CN02]. The main goal of PLE is to achieve software reuse in a strategic, prescribed way, while using a managed set of features [JKB08].

Software PLE and Model-Driven Software Development (MDD) are two recent trends that have been drawing increased attention from the software development community. Essentially, MDD [BG05] is a software engineering methodology based on models of the system to be constructed and the evolution of those models, in order to perform an incremental development. Models allow to directly capture the needs of the stakeholders, and abstract from specific implementation details, more amenable to analysis. In MDD, models are source artifacts and they are used for automated analysis and code generation. Generative software development [Cza98] and related approaches have been the integration of PLE and MDD [CAK+05]. While MDD allows to represent different aspects of a SPL more abstractly, PLE provides a well defined scope, which puts the development and selection of appropriate modeling languages on a firm basis. Moreover, automated analysis and code generation permits the automatization of product line member creation.

Feature modeling is a technique for representing the commonalities and variabilities

among a set of related systems in a concise way. Usually, Feature Diagrams (FD) are used to document this, using hierarchies of features that describe different types of variability [KCH+90, Kan10, CHE05a, CHE05b]. A particular technique of Model-Driven Development of Product Lines, based on model templates is proposed in [Cza05]. A feature-based model template consists of feature models and annotated models implementing the features. The annotations refer to the features and can have the form of any notation defined using the Meta-Object Facility (MOF) [omg06], such as Unified Modeling Language.

The Unified Modeling Language (UML) has become the industry and academic standard for system specification. UML Statecharts are one of the most important constituents of UML models, since they are widely used for modeling the reactive behavior of systems. The fact that the semantics of UML is only informally described leads to many ambiguities and inconsistencies, which renders difficult or impossible the application of a formal design process.

The subject of this thesis is the formal specification of families of systems that share some functionality. In particular, it investigates an expressive modeling formalism for specifying Software Product Lines, based on a formal approach of UML state machines, in conjunction with Feature Diagrams. By achieving this, the expressivity of UML Statecharts can be augmented with the explicit handling of behavior refinement. A complete semantics will allow the incorporation of the resulting proposal in Model-Driven Development methods.

1.1 Contribution of this Work

The primary goal of this thesis is to specify the behavior of a Software Product Line (SPL) using formal methods. It constitutes an advance towards a complete formal semantics of UML Statecharts with Variabilities in a SPL context. The main contribution is the formulation of a notion of refinement based on UML Statecharts, which can be integrated into software engineering tools. This could provide solid formal background that helps to bridge the gap between the needs of modern software development techniques for a formal specification language and UML, which is mainly described using natural language. Applying formal rigour to the precise semantics of these models and languages allows to automate the steps needed to transform models, as well as to trace and analyze those transformations. Each model should be based on a formalism, in order to rigorously define its syntax and semantics. Unfortunately, UML's lack of rigour can lead to ambiguous, imprecise and contradictive specifications.

In this work, Feature Diagrams (FDs) are used to represent the common and variant functionality of a family of products, presenting a formal syntax for Feature Diagrams and its configurations. On the other hand, based on von der Beeck's [vdB02] UML Statecharts abstract syntax, an order relation is defined among the set of UML Statecharts, which represents when an UML statechart has a richer structure than another one. This relation sets the basis for the definition of behavior refinement of a UML statechart. Then, an operation able to combine different extensions of a given statechart is defined. With these notions, and given the description of a family of products as a FD, an UML Statechart with Variabilities is defined as a function that associates each feature of the FD with a statechart. The mapping must comply with the hierarchical structure and the feature restrictions, i.e., the more features a product has, the richer the statechart that models it must be. In this way, it is possible to describe the effect that each feature has on the products in which it is present. This definition provides a very simple and flexible way to obtain the specification of the behavior of any configuration of the product line as the combination of the statecharts that implement all the features present in that product. We consider both the extension relation and the definition of UML statechart with variabilities authentic contributions. The first one, because it provides an ordering among the abstract

syntax defined in [vdB02], which proved to be a semantic preserving one, in the sense of a properly defined structural operational (SO) semantics. The second one, because it allows to integrate parts of two different modeling languanges, FDs and UML Statecharts, through its abstract syntax.

Then, based on the extension relation, the concept of behavioral refinement of a UML statechart is explored. In contrast to most of the related work, in which there are no exhaustive proofs, it is proven that it is possible to extend a statechart without loosing any behavior, in the sense of the SO semantics proposed in [vdB02]. That is, when a statechart is extended, it is still possible to perform the same semantic transitions on it as before. Therefore, the extension relation can be considered as a behavioral refinement, as it preserves the behavior, but adding new functionalities. Moreover, it is also proved that the set of actions generated by the SO semantics of a given statechart is preserved with any possible extension of it. We consider this also as a contribution because, until the date this work was finished, there were no concrete formal refinement patterns for UML Statecharts, with the sole exception of [MNB04], which uses a different priority mechanism for the transition firing from UML's.

Finally, the practical point of view of this proposal is addressed, through the construction of a tool prototype called "SC* Modeler". This prototype was constructed basically for checking the adequacy of the definitions and for exploring the possibilities of a future software engineering tool. Using this tool, a case study for a product line of microwave oven systems, adapted from a classic SPL modeling book [Gom05], was implemented.

1.2 Related Work

Variability modeling is a domain specific modeling technique, that is becoming more integrated into traditional software engineering. Unfortunately, it is not integrated to UML. Software Product Line Engineering with UML received a lot of attention in recent years, but most of these works "only concern variability in UML static models and few works concern behavioral models" [CABA09]. In a formal setting, few citations are relevant. In [CGW05] a formal semantics of UML interactions with variabilities is given, and in [ZHJ04a, ZHJ04c, ZHJ04b, CABA09] UML sequence diagrams with variabilities are formalized using an algebraic framework for synthesizing flat statecharts from the sequence diagrams. In [GL10], the authors define functions which map UML Statechart components to functionalities of a Feature Diagram. Then, the behavior of a product line is obtained essentially by a selection process, and not a combination of statecharts, in contrast to the present work. In [FG08], the authors define a general labeled transition system framework for describing families of products. Using this framework, products can be derived, but this approach is not intended to be used to describe product families, but rather "to give basic modeling concepts on which verification activities can be carried out".

Although UML Statecharts refinement has already been investigated in a formal setting [GL10], [MNB04] and [SK10], there is still a lack of a formal definition of *behavioral* refinement for UML Statecharts. The UML 2.0 documents use the term refinement (and its counterpart, abstraction) without a specific definition or meaning [Gro05]. Sun Meng et al. [MNB04] propose refinement patterns similar to the ones presented in this work. Their semantics is based on Lattela et.al. [LMM99], which uses a priority mechanism (of conflicting transitions) which is the opposite as required in the UML specification [Gro05]. Moreover, the refinement laws are abstract and elementary, and there are no proofs regarding the resulting behavior once the rules are applied.

The closest work to the present one, in terms of the results obtained, is the recent paper by Schönborn et al. [SK10], based on the semantics presented in [FS06]. Regarding the UML coverage of the semantics, they do not model history pseudostates, and according to them it is a "harder challenge" to implement, compared to other missing UML features like

fork and join states. Moreover, they do not model entry and exit actions. Regarding the refinement patterns, the authors allow the removal of behavior from a statechart, contrary to the present work. With respect to the actions generated by the refined statecharts, which can be a "data-refinement" of the original actions, these are not treated in that work.

1.3 Outline of this Thesis

This thesis is structured as follows: In chapter 2, the general context of this work is presented. A brief description of the current state-of-the-art in formal approaches for UML Statecharts is given, together with the concept of Software Product Lines and its relation with Feature Diagrams as a tool for modeling the variability of a family of systems. In chapter 3, the syntax and semantics of UML Statecharts are presented. The work contained in this chapter is based on [vdB02]. In chapter 4, an extension relation is defined between UML Statecharts, in order to give a formal definition of when a given statechart has a richer behavior than another one. Then, it is shown how to combine different extensions of a given statechart into an integral new one. In chapter 5, main concepts and definitions of formal variability modeling are presented. The syntax and configurations of Feature Diagrams and the definition of UML Statechart with variabilities is given. An early version of the work contained in this chapter and the previous one is published in [SV08]. In chapter 6, it is proved that the extension relation can indeed be considered as a refinement, in the sense that it preserves the semantic transitions defined in the semantics. Moreover, a theorem that proves that the set of possible actions generated by a statechart are preserved by the refinement is given. The work contained in this chapter is published in [SV10]. In chapter 7, the description of a prototype implementation of the ideas introduced in this work is presented. In chapter 8, a case study for a product line of microwave oven systems is presented is order to analyze and validate the proposal. Conclusions, related work, and future research are given in chapter 9.

Chapter 2

Background

In this chapter, main concepts of behavior specification of families of systems are presented, constituting the general context in which the subject of this work is placed. Model-Driven Development is introduced, as a software engineering methodology based on the systematic use of models, capable of reducing development costs. The use of the Unified Modeling Language (UML) is motivated as a modeling language for this purpose, in particular, UML Statecharts for behavioral modeling. Next, a brief description of the current state-of-the-art in formal approaches for UML Statecharts is given and a rationale for the choice of the particular formalization used in this work is also examined. Finally, Software Product Lines are presented together with its relation with Feature Diagrams as a tool for modeling the inherent variability of a family of systems, and a brief description of the current state-of-the-art in formal variability modeling is presented.

2.1 Model-Driven Development

The use of models in engineering and applied sciences is one of the most fundamental technique for addressing inherent complexity. Models provide abstractions of a real-world system that enables the reasoning about that system by ignoring superfluous details while focusing on the relevant ones. Models are used in many ways: predicting the system behavior, reasoning about particular properties, early evaluation about possible system changes, and even communicating important system characteristics. Depending on what is considered relevant, various modeling concepts and notations may be used, in order to provide different "views" of the system. Furthermore, it is often necessary to transform between different views of the system at an equivalent level of abstraction, for example, between a structural view and a behavioral view. In other cases, a transformation converts models between levels of abstraction, usually from a more abstract to a less abstract view, by adding more detail.

Models and model transformations form the basis for a set of software development techniques known as Model-Driven Development (MDD) [BG05]. In that context, models are used to reason about the problem and the solution domain. Applying formal rigour to the semantics of these models, it is possible to define precise rules in order to automate the steps needed to transform one model to another; to trace between model elements and transformations and to analyse relevant characteristics of the models. Each model should be based on a formalism, in order to rigorously define its syntax and semantics. Syntax refers to the way that symbols may be combined to create well-formed sentences in the language, that is, the form and structure of symbols in a given language. Semantics, on the other hand, defines the meaning of syntactically valid strings in a language. Three main semantics approaches can be distinguished: Operational semantics, in which the meaning of a construct is specified by the computation it induces when it is executed

(how to execute); Denotational semantics, in which the meaning is modeled by mathematical objects that represent the effect of executing the constructs (effect of the execution); and Axiomatic semantics, in which the meaning is expressed through properties about the language constructs, expressed with axioms and inference rules from symbolic logic (correctness properties). Visual formalisms are also very useful in order to simplify the communication between different stakeholders involved in a software development process.

One specific implementation of the MDD is the Model-Driven Architecture (MDA), which is supported by the Object Management Group (OMG) [Gro11]. MDA introduces a set of layers that describe different levels of abstraction and uses model transformations as a central element, principally to transform high-level models (such as platform-independent models) toward more implementation-oriented models (platform-specific models). A key characteristic of the MDA approach is to recognize that transformations can be applied to abstract descriptions of some aspect of a system to add more detail, refine that description to be more concrete, or to convert it to another representation. The MDA approach provides an open, neutral basis for system interoperability via OMG's established modeling standards [BG05]. MDA uses the Unified Modeling Language (UML) as its specification language [Gro05]. This language has become the industry and academic standard for system specification.

2.2 Unified Modeling Language

The Unified Modeling Language (UML) is a set of graphical languages for specify and document the artifacts of a software-intensive system. These languages allow specifying a wide variety of aspects of a system, from static structure to dynamic behavior. Structure can be described with static model elements such as classes, relationships, nodes, and components. Behavior describes how the elements within the structure interact over time. Moreover, any UML language can be extended by its own extension mechanisms in order to define domain-specific models.

UML has its roots on object-oriented modeling languages. From the late 1980s, practitioners, faced with a new generation of increasingly complex software systems, began to experiment with alternative approaches to analysis and design. Learning from experience, three clearly prominent methods emerged, each one with its own notation: The Booch method devised by Grady Booch; the Object Modeling Technique (OMT) devised by Jim Rumbaugh; and the Object Oriented Software Engineering (also known as Objectory) devised by Ivar Jacobson. The UML standardization effort started officially in October 1994, with the establishment of a UML consortium, with several partners conformed by leading software industry firms. Finally, UML 1.0 was offered for standardization to the Object Management Group (OMG) in January 1997.

UML 2.0 presents four languages to specify dynamic behavior: Use Cases, State Machines, Activities, and Interactions. Use cases captures the behavior of the system as it appears to an outside user. It specifies functional requirements of the system from the user point of view. State Machines, which are the main concern of this work, specify the dynamic behavior of objects over time. Each object communicates with the rest of the world by detecting events and responding to them. An Activity is a graph of nodes that shows the flow of control and data through the concurrent steps of a computation. Finally, Interactions are used to describe how a set of objects interact with each other in a specific scenario.

UML is defined using a metamodel, that is, a model of the constructs in UML. The metamodel itself is expressed in UML. Each section of the UML specification document contains a diagram showing a portion of the metamodel; a text description of the elements defined in that section, with their attributes and relationships; a list of constraints on elements expressed in natural language and in Object Constraint Language (OCL); and a

text description of the dynamic semantics of the UML constructs defined in the section. According to [RJB04] “The dynamic semantics are therefore informal, but a fully formal description would be both impractical and unreadable by most”. The metamodel is divided into two main packages, structure, which defines the static structure of UML and behavior, which defines the dynamic structure of UML.

Although UML is simple and flexible, it is not the ideal MDA specification language since it is defined using natural language and/or semi-formal language instead of being described in a fully formal way. The lack of a formal setting leads to ambiguous (interpreted in more than one way are possible), imprecise (no clear statement is made), inconsistent (some parts are in contradiction with other ones) and erroneous (it is unclear whether the implicitly given interpretation is the intended one) specifications, as pointed out in the literature [SG98, RW99, FSKdR05, FKS05]. Many of the detected ambiguities and inconsistencies are removed in UML 2.0 [Gro05], but new ones are introduced. Without a rigorous semantic definition, precise model behavior over time is not well defined and automatic code synthesis is not possible to achieve.

This deficiency can be understood in part, because there are thirteen diagram types in UML 2.0. Each notation itself forms a complex language, and the notations are interrelated and interdependent, making the task of providing a unified semantics very challenging [HR00]. These drawbacks are present since the beginning of UML, and promoted extensive academic work in order to solve them [BCR00, BLMF00, CGW05, CK01, Mer02, Krü02, CK04, SZ08, CGW06, BCR, GZK02, Jür02, RCA01, SH05, Stö03, BCD^{+}06]. In general, these works were developed independently, which implies a weakness for achieving a complete and unified semantics. Recently, an heterogeneous approach to the semantics of UML is proposed which deals with the integration of different formalisms. The idea is to define a family of formalisms capturing various UML sublanguages, and morphisms that represent the expected semantic relationships between them [CKTW08]. For the purposes of this work, a survey on the formal semantics approaches for UML Statecharts is given in section 2.4.

2.2.1 UML Statecharts

In several branches of computer science state transition diagrams are commonly used to describe the behavior of real world systems. Those diagrams represent a model of computation called finite state machine, which consists of a finite set of states, an initial state, an input alphabet of events and a transition function which maps current states and event symbols to next states [HU79]. However, as soon as the behavior of the described system becomes complex, the diagram gets unmanageable from the modeler’s point of view (this is usually called state and transition “explosion”).

In the late 80’s, Harel [Har87] developed an effective visual formalism able to express structure by means of hierarchical states. He called this visual formalism “statecharts”. In Harel words, this new formalism “extends conventional state transition diagrams with essentially three elements, dealing, respectively, with the notions of hierarchy, concurrency and communication”. Basically, it allows to describe states in a modular way. In contrast to the usual state transition diagrams which are flat, unstructured and sequential, in the statecharts philosophy each state consists of a hierarchy of states. Behavior is then modeled by the execution of series of actions which are determined by transitions that are triggered by events, one by one being dispatched. Unfortunately, Harel’s extension comes with a price: it is only a visual formalism. Far from being a finished work, statecharts evolved over the years, summing up to at least 20 variants [Bee94, Har97].

UML behavioral state machines, UML Statecharts, are extensively used in the software engineering community for modeling the reactive behavior of object-oriented systems. UML Statecharts were incorporated by the OMG to UML 1.1 in 1997, and it was subject to examination during the UML 1.2 and 1.3 revisions. Moreover, UML Statecharts are one of

the four languages to specify dynamic behavior and serves the foundation for Interactions [Gro05], and was for Activities in UML 1.x.

2.3 Formal Methods

Formal methods is the name usually given to a set of design techniques that rigorously use mathematical models to build software systems. In contrast to other design systems, formal methods use mathematical proofs as a complement to system testing in order to ensure correct behavior. As systems become more complex, the formal approach to system design offers an additional level of confidence. Formal design can be seen as a three step process: First, the formal specification phase. The designer defines a system using a modeling language. This is the process of converting a problem written in natural language into a mathematical notation. Formal modeling languages syntax is rigorously defined, making it possible to distinguish between well-formed formulas and non-well-formed ones. The second phase is the verification: This is the stage in which formal methods differ most from other specification techniques and it involves the use of theorems in order to prove that the system is correct. It is a difficult process, because even a simple system comprises several theorems. In order to accomplish this, almost any formal method uses some sort of automated or semi-automated proving tool. These tools can prove elementary theorems, verify the semantics of the system, or just provide assistance for developing complicated proofs. The third and final phase is the implementation, in which the specification is converted into code. This conversion can be done using code generators in order to obtain provable-correct by construction implementation, which implies a substantial reduction of testing effort. Code generation is the essence of model driven approaches in order to be able to abstract specific implementation details from the designer's mind. Then, the the code generators automatically translate models into implementations.

As mentioned before, the main benefit of formal methods is provability. Furthermore, formality promotes discipline, because it requires thinking in a more systematic way. This discipline can help identifying faulty reasoning in earlier stages of software development. Formal methods also provide a formal and rigorous framework as in other classical (mechanical, civil) engineering processes. The use of formal specification languages, which are generally coupled with a system of logic inference, enables the construction of automatized tools that verify the formal properties of the conceptual models created in the earlier stages.

2.4 Formal Semantics Approaches of UML Statecharts

As mentioned in section 2.2 UML languages are only informally defined. Several efforts have been made in order to give formal semantics to these UML languages. In order to illustrate the landscape of different semantic approaches to UML Statecharts, a succinct categorization based on [CD05] and [DJPV02] is given. Three main broad categories of approaches can be distinguished, according to the underlying formalism: Mathematical Models, Rewriting Systems and Translation Approaches.

1. **Mathematical Models:** This category comprises semantic approaches which are based directly on standard mathematical concepts and notations. The advantage of using a mathematical notation is that it encourages precision and attention to detail, making it more likely that the resulting semantics is complete and unambiguous. In principle, the notation should be accessible to anybody with a standard mathematical background. Four subcategories can be distinguished: Labeled Transition Systems, Abstract State Machines, Petri Nets and other approaches. These approaches will be described in detail in section 2.4.1.

2. **Rewriting Systems:** A rewrite system typically consists of a set of rewrite rules, consisting of a left- and a right-hand side. The execution of a rewrite system involves the repeated application of the rules to some "configuration". In each application, an occurrence of the left-hand side of a rule in the configuration is replaced by the right-hand side. The execution terminates when no matching rule can be found. As a rewriting system based formalism, the authors distinguish between Graph Rewriting and Term Rewriting. Graph rewriting (also called graph transformation) provides a mathematically precise and visual specification technique by combining the advantages of graphs and rules into a single computational paradigm. On the other hand, Term rewriting is a similar concept to graph rewriting, except that the rewrite rules are performed on terms rather than on graphs. An introduction to the subject can be found on [EEPT06] and [BN98].

3. **Translation Approaches:** This category contains approaches which rely on translating a UML state machine into some other formal language, such as a specification language, the input language to a model checker, or a programming language. As a translation based formalism, the authors distinguish between Model Checking Languages and Specification Languages. Model checking is a well-researched dynamic analysis method in which systems are modeled as finite state models. Temporal logic is used to define properties and the models are checked to verify whether these properties hold. This approaches typically transforms UML Statecharts into a language designed for such analysis. It is important to mention that model checking languages are not considered truly formal languages. A good introduction to the subject is [Hol03]. On the other hand, several approaches attempt to inject formalism into UML state machines by translating them into an already formalized specification language, such as Z. An introduction to the subject can be found on [Jac96].

2.4.1 The Mathematical Approach

Since the formalism used in this work belongs to the mathematical approach, a detailed description is presented in this section. As mentioned before, four subcategories can be distinguished:

- **Transition Systems:** In general, a transition system is essentially a structure $(S, \longrightarrow)$ where S is a set of configurations and $\longrightarrow$ is a binary relation on $S \times S$ (called the transition relation). Some examples of transition systems are Labeled Transition System (LTS), Kripke structures and Symbolic Transition Systems. A natural reference is [MP92, MP95]. The formalization used in the present work is included in this category. The most relevant approaches included in this category are [EW00, Esh09], [LMM99, GLM02, MLG06], [Kwo00], [RACH00, CR09] and [vdB02].

- **Abstract State Machines:** Basically, Abstract State Machines (ASMs) are finite sets of transition rules of the form **if** *Condition* **then** *Updates* which transform abstract states. The *Condition* (also called guard) under which a rule is applied is an arbitrary predicate logic formula without free variables, whose interpretation evaluates to true or false. *Updates* is a finite set of assignments of the form $f(p_1, ..., p_n) := p$ where p_i are arguments and p a specified value, which evaluates to v_i and v respectively. The execution is the change in parallel of f at the locations v_i to the value v. The syntax of ASMs is reminiscent of a simple imperative programming language which makes them quite accessible to users with a programming background. A software engineering introduction to ASMs is [BS03]. ASMs can also be considered transition systems [BCR04]. The most relevant approaches included in this category are [CR00, BCR04], [CHS00, SCH02], [JEJ04], [JÖ2, Jür04] and [KG10].

- **Petri Nets:** Petri nets are a well-studied and intuitive formalism that is both graphical and mathematical. Petri nets are a bipartite directed graph consisting of two kinds of nodes and two kinds of arcs. The nodes are places and transitions. The arcs are either input arcs or output arcs.A place is connected to a transition via an input arc. A connection from a transition to a place is established via an output arc. Arcs between the same kinds of nodes are not allowed. The places from which an arc runs to a transition are called the input places of the transition; the places to which arcs run from a transition are called the output places of the transition. A Petri net furthermore consists of tokens represented by dots within places. A transition of a Petri net may fire whenever there is a token at the start of all input arcs; when it fires, it consumes these tokens, and places tokens at the end of all output arcs. The firing is atomic, i.e., a single non-interruptible step. Natural references for Petri nets in software engineering are [Pet81, GV01]. Petri net formalism was introduced in [Pet62]. The most relevant approaches included in this category are [BP01, BP05], [BDM02, MC02], [KP00] and [GZ09].
- **Other:** It is a residual category of approaches that are not included in the preceding ones. Approaches included in this category are [LC08, LC09].

2.4.2 Rationale of the Transition Based Formalism

The explanation of the fundamental reasons for choosing the transition based formalism starts with Gordon Plotkins's [Plo81] words on transition systems: "Very little is required in the way of mathematical background; all that will be involved is symbol-pushing of one kind or another of the sort which will already be familiar to readers with experience of either the non-numerical aspects of programming languages or else formal deductive systems of the kind employed in mathematical logic". Structural Operational Semantics (SOS) was introduced by Plotkin in [Plo81]. A SOS specification defines the behavior of a program in terms of a set of transition relations. SOS specifications take the form of a set of inference rules which define the valid transitions of a composite piece of syntax in terms of the transitions of its components. SOS was intended as being like an abstract machine but without all the complex machinery in the configurations, just the minimum needed to explain the semantical aspects of the programming language constructs. The extra machinery is avoided by the use of the rules, making the exploration of syntactical structure implicit rather than drearily explicit. In an operational semantics the focus are the operations the system can perform. In Plotkin's words: "... it is an operational method of specifying semantics based on syntactic transformations of programs and simple operations on discrete data. The idea is that in general one should be interested in computer systems whether hardware or software and for semantics one thinks of systems whose configurations are a mixture of syntactical objects - the programs and data - ...". In the operational semantics the meaning of a construct is specified by the computation it induces when it is executed on a machine. In particular, it is of interest how the effect of a computation is produced. This leads to an ease on the implementation of an SOS semantics in a rule based programming language, like Prolog.

Von der Beeck's [vdB02] semantics provides a reasonable coverage of UML 1.4 statecharts features, and it is easy to extend both its syntax and semantics. The approach includes full support of the history mechanism (shallow and deep cases) and intralevel transitions which are in general not supported by the rest of the approaches. As will be presented, the definition of the semantics is modularized in two phases as follows: In the first phase an auxiliary UML Statecharts semantics is defined which only deals with processing single input events, but not with sequences of input events. In a second phase this auxiliary semantics is used to define the (complete) UML Statecharts semantics, which is done by processing sequences of input events. This separation already supports modularity. Furthermore, concepts from the SOS approach are used, that is, the auxiliary semantics

is a function on the set of labeled transition systems and where the (semantic) transitions work on single input events. For the second phase, Kripke structures are used as the semantic domain. This selection simplifies the processing of event sequences considerably, since Kripke structures are very appropriate for modeling the fact that the output of one step serves as (part of) the input of the next step. Both phases constitute an operational and modular approach, such that comprehension as well as flexibility (e.g. with respect to subsequent enhancements) are supported without restricting preciseness.

Another advantage of von der Beeck's semantics is that all of the features implemented comply with the UML specification [Gro05], as for example the conflicting transitions execution priority or the maximality principle (in each step of the semantic execution, the maximum number of non conflicting transitions must be taken).

However, many features of UML Statecharts have not been considered in this work: final, junction, choice and terminate pseudostates; deferred, time, and change events; completion transitions; guards, variables, and data dependencies in transitions; creation and destruction of objects; send clauses within actions; and *do* actions. Some of those features can be covered easily, like final pseudostates and *do* actions. In section 9.2.1 a comment on possible extensions, in order to cover a wider range of UML 2.0 statecharts features, will be given.

2.5 Software Product Lines

According to the Software Engineering Institute [SEI11a], a Software Product Line "is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way. A Software Product Line epitomizes strategic, planned reuse.". One of the major motivations behind Software Product Lines (SPLs) is that they can be more efficient to manage a group of similar software systems as a whole, rather than considering each individual system on its own [Gom05, CN02, SEI11b]. The main goal of SPL engineering is to achieve software reuse in a strategic, prescribed way, while using a managed set of features [JKB08].

SPL engineering introduces two new dimensions into the traditional software engineering approaches: variability modeling and product derivation. Variabilities are characteristics or features that varies from one product to another, whereas product derivation is the process of constructing products from the product line.

2.5.1 Variability Modeling

SPL engineering has to provides a mean to describe the commonalities and variabilities between different products. Usually, Feature Diagrams are used to document this. Feature Diagrams are a family of popular modeling languages used for engineering requirements in SPLs. Feature Diagrams were introduced by Kyo Kang in 1990 as part of the Feature-Oriented Domain Analysis (FODA), which is a method for systematic discovery and exploitation of commonality across related software systems to support software reuse [KCH+90, Kan10].

Although there were initial attempts to develop a formal definition for FD, the first systematic treatment of formal semantics was given by [SHT06, SHTB07]. In that work, Schobbens et al generalizes the various syntaxes found in the literature into a generic construction, called Free Feature Diagrams (FFD). The formal semantics is then defined at FFD level, in which "all formalization choices found a clear answer in the original FODA FD definition". Finally, they showed that the original FODA FD definition suffered no ambiguity problem. Due its importance, a comparison with the present work is given in chapter 5.

In [CHE05a] and [CHE05b], Czarnecki et al. define a new Feature Diagram language to account for staged configuration. They introduce feature cardinality (the number of times a feature can be repeated in a product) in addition to the more usual (group) cardinality. The semantics is defined in a staged process where FDs are translated into an extended abstract syntax, a context-free grammar and an algebra. In [JB08], the authors provide a formalization of dependencies between features and components and propose automated techniques to derive additional information and provide feedback to the user. Other approaches include [dJVI02] which proposes the use of context-free grammars and the feature configurations are defined as the sentences generated by the pertaining grammar. This idea is extended in [Bat05] by combining grammars and propositional logic. Van der Storm [VDS07] maps features to components, which are organized in a dependency tree.

2.5.2 Product derivation

Although variability in software systems is not a new problem, the key difference in a SPL context is that variability involves not only a single product and variability cannot be resolved after the product is delivered to the user. In this context, variability must be specified in the development process and is an integral part of the product line. The "single product" variability is usually called "run time variability", and the SPL variability is usually called "development time variability" [ZHJ04c].

Product derivation support of a particular SPL engineering approach is an important criteria in order to go beyond the description of the PL and serve the propose for resolving variability and obtaining products [CABA09]. Current product derivation approaches can be classified into two main categories: configuration based and transformation based [PKGJ08].

The first approach bases the derivation process on Feature Models (FD). For example, in the FORM approach [KKL+98], Kang et al. define a derivation process which starts with an analysis of commonality among applications in a particular domain. The model constructed during the analysis is called a feature model, which captures mandatory and alternative features. This model is used to define parameterized architectures and reusable components instantiatable during application development. According to the authors "Parameterization of artifacts with features and development of applications through selection of feature sets is a powerful synthesis technique for implementing an effective application generator for a stable domain". This refinement approach was later formalized by Czarnecki [CHE05a, CHE05b, Cza05]. In that work, feature models are mapped to UML activity and class diagrams via annotations, and then an automated configuration process is realized. In [Pre03], the author presents a set of graphic "plug-and-play" specification rules for the construction of statecharts, which allows to combine features based on semantic refinement. In that work, a refined statechart has fewer possible traces and is hence more concrete, in contrast to the present work in where a refined statechart has an augmented behaviour. In [VG07], the authors provide a mapping from the problem domain (modeled as a feature model) to the solution domain.

The second approach consists of transforming core assets rather than configuring them. In general this approach is aligned with MDA standards. Usually, at the requirements level, the product line is modeled in terms of UML use cases and then model transformations relates this requirements to core assets, modeled in terms of UML class and sequence diagrams. The variability is realized via stereotypes, and OCL constraints are used in order to ensure consistency. Finally, a model transformation taking the product line model as its parameter, transforms the core assets into a platform specific model which is finally implemented in the target platform [ZJ06, KMHC05, ZHJ04c].

Chapter 3

Syntax and Semantics of UML Statecharts

In this chapter, main concepts and definitions of UML Statecharts used in this work are presented. These definitions are based on the paper *"A structured operational semantics for UML-statecharts"* by M. von der Beeck [vdB02]. First, the formal syntax of UML Statecharts is presented and the concept of configuration is discussed. Then, given a UML statechart, it is shown how a UML statechart transition modifies the actual configuration of the state. After that, the two phase semantics is presented, first, the auxiliary semantics which only deals with single input events, and second, the complete semantics who deals with sequences of input events.

3.1 Syntax

UML Statecharts (or simply statecharts) constitutes a notation to describe behavioral aspects of a system. As mentioned before, UML Statecharts are a generalization of finite state diagrams. Basically, UML Statecharts consist of *states* and *transitions* between them. The main feature of statecharts is that states can be refined, defining a state hierarchy. The decomposition of a state can be either *sequential* or *parallel*. In the first case, a state is decomposed into a new state automaton (OR state), while in the second case a state is decomposed in two or more automata that can execute concurrently (AND state). Transitions are directed arrows between states. A transition connects a *source state* to a *target state*, and inter-level transitions are allowed. Transitions are labeled by a trigger event, a sequence of actions and the type of history of the target state. There is a *history mechanism* that allows transitions to reenter a sequential state in the last active substate. UML Statecharts follow the *run-to-completion* assumption, that is, "an event occurrence can only be taken from the pool and dispatched if the processing of the previous current occurrence is fully completed" [Gro05].

UML distinguishes between states and pseudostates. The key difference is that pseudostates are never elements of an active state configuration; they mainly serve to model compound transitions or to reduce the graphical complexity. Pseudostates do not have names or associated actions. An example is the history pseudostate.

Let $\mathcal{S}$, $\mathcal{T}$, A, E be countable sets of state names, transition names, actions and events respectively with $\mathsf{E} \subseteq \mathsf{A}$. $\mathcal{S}$ contains an empty name denoted by λ. Events and actions are denoted by $a, b, c, ..$ and sequences of events as well as actions by $\alpha, \beta, \gamma, ...$ The empty sequence is denoted by $()$. The set SC of statecharts is inductively defined by the rules in figure 3.1, together with the functions $\mathsf{name}\colon \mathsf{SC} \to \mathcal{S}$, that is, the name of the statechart, and the predicate $\mathsf{wf\text{-}tran}$ (defined below), which decides if a transition is well formed with

respect to a set of states. Abusing the notation, $\mathsf{name}(s)$ is abbreviated with $\hat{s}$. The same applies to transitions.

$$\frac{\hat{s} \in \mathcal{S} \quad en, ex \in \mathsf{A}^*}{[\hat{s},(en,ex)] \in \mathsf{SC}}\ \mathsf{Basic} \qquad \frac{\begin{array}{l} s_1, .., s_n \in \mathsf{SC},\ \hat{s}_i \neq \hat{s}_j\ \forall_{i \neq j} \\ \hat{s} \in \mathcal{S},\ \hat{s} \neq \hat{s}_i\ \forall_{i=1..n} \\ en, ex \in \mathsf{A}^* \end{array}}{[\hat{s},(s_1,..,s_n),(en,ex)] \in \mathsf{SC}}\ \mathsf{And}$$

$$\frac{\begin{array}{l} s_1, .., s_n \in \mathsf{SC},\ \hat{s}_i \neq \hat{s}_j\ \forall_{i \neq j} \\ \hat{s} \in \mathcal{S},\ \hat{s} \neq \hat{s}_i\ \forall_{i=1..n} \\ T \subseteq \mathsf{T},\ \mathsf{wf\text{-}tran}(\{s_1, .., s_n\}, t)\ \forall t {\in} T \\ \hat{s}_l \in \mathcal{S}. l \in \{1, .., n\} \\ \hat{s}_d \in \mathcal{S}. d \in \{1, .., n\} \\ en, ex \in \mathsf{A}^* \end{array}}{[\hat{s}, (s_1, ..s_n), \hat{s}_d, \hat{s}_l, T, (en, ex)] \in \mathsf{SC}}\ \mathsf{Or}$$

Figure 3.1: Syntax of Statecharts

The rules are explained as follows:

Basic Statecharts: $s = [\hat{s},(en,ex)]$ is a basic statechart with name $\hat{s}$, entry sequence of actions en, exit sequence of actions ex and $\mathsf{type}(s) = \mathsf{basic}$.

And-Statecharts: $s = [\hat{s},(s_1,..,s_n),(en,ex)]$ is an and-statechart with name $\hat{s}$, parallel substates $s_1, .., s_n$, $n > 0$, entry sequence of actions en, exit sequence of actions ex and $\mathsf{type}(s) = \mathsf{and}$.

Or-Statecharts: $s = [\hat{s}, (s_1, .., s_n), \hat{s}_d, \hat{s}_l, T, (en, ex)]$ is an or-statechart with name $\hat{s}$, parallel substates $s_1, .., s_n$, $n > 0$, entry sequence of actions en, exit sequence of actions ex, $\hat{s}_d$ is the name of the default initial substate[1], where $d \in \{1, .., n\}$ is the index of the default initial substate, $\hat{s}_l$ is the name of the current active substate of s, $l \in \{1, .., n\}$ is the index of the current active substate, T is the set of transitions between its substates, s_1 is the default state of s, $\mathsf{type}(s)$=or. The syntax presented in [vdB02] is extended, requiring a default substate in the Or-Statechart. According to the UML superstructure specification [Gro05] , a composite state without initial substate can be considered as an ill-formed model. Then, in this work, an initial substate is required.

The set of transitions T is included in $\mathsf{T} := \mathcal{T} \times \mathcal{S} \times \mathcal{P}(\mathcal{S}) \times \mathsf{E} \times \mathsf{A}^* \times \mathcal{P}(\mathcal{S}) \times \mathcal{S} \times \mathsf{HT}$, where $\mathsf{HT} = \{\mathsf{none}, \mathsf{shallow}, \mathsf{deep}\}$ are the history types.

Further define:

$$\mathsf{SC}_B = \{s {\in} \mathsf{SC} \mid \mathsf{type}(s){=}\mathsf{basic}\}$$

$$\mathsf{SC}_A = \{s {\in} \mathsf{SC} \mid \mathsf{type}(s){=}\mathsf{and}\}$$

$$\mathsf{SC}_O = \{s {\in} \mathsf{SC} \mid \mathsf{type}(s){=}\mathsf{or}\}.$$

Given a statechart s, the following projection functions are defined: $\mathsf{act\text{-}en}(s) := en$ and $\mathsf{act\text{-}ex}(s) := ex$ which are the corresponding entry and exit sequences of actions. Given a

[1]In order to simplify the notation, the name is denoted with the index only, because within a statechart each substate can be uniquely referred to by its name.

statechart transition $t = \langle\hat{t},\hat{s}_i,S_r,e,\alpha,T_d,\hat{s}_j,ht\rangle \in \mathsf{T}$, the following are defined: $\mathsf{name}(t) := \hat{t}$, is the name of the transition t, $\mathsf{sou}(t) := s_i$, $\mathsf{tar}(t) := s_j$, are the source and target states of t respectively, $\mathsf{sou\text{-}r}(t) := S_r$, is the source restriction set, $\mathsf{ev}(t) := e$, is the triggering event of t, $\mathsf{act}(t) := \alpha$, is the action sequence associated to t, $\mathsf{tar\text{-}d}(t) := T_d$, is the target determinator set, $\mathsf{historyType}(t) := ht$, is the history type of t. A transition t uses the history mechanism, if $\mathsf{historyType}(t) \in \{\mathsf{deep}, \mathsf{shallow}\}$. There are two types of history pseudostates defined in UML: shallow and deep history. A shallow history pseudostate is used to represent the most recently active substate. The shallow history pseudostate does not recurse into its substates active state configuration. A deep history pseudostate, in contrast, recurses into the most recently active substates of that substate, if they exist. That is, it "remembers" the active substates along the state hierarchy down to the basic states.

Source restriction and target determinator provide a means for modeling an interlevel transition by a simple transition on the level of the uppermost states the interlevel transition exits and enters. The source and target of the interlevel transition are represented as additional label information by the source restriction and the target determinator.

In the definition of Or-statecharts, the predicate $\mathsf{wf\text{-}tran} \subseteq \mathcal{P}(\mathsf{SC}) \times \mathsf{T}$, defined by mutual recursion, decides if a transition t is well formed with respect to a set of states $s_1,..,s_n$ and it is defined as $\mathsf{wf\text{-}tran}(\{s_1..s_n\},t) \subseteq (\mathsf{sou}(t),\mathsf{tar}(t) \in \{s_1..,s_n\}) \wedge (\mathsf{sou\text{-}r}(t) \in \mathsf{ec\text{-}all}(\mathsf{sou}(t)) \vee \mathsf{sou\text{-}r}(t) = \emptyset) \wedge (\mathsf{tar\text{-}d}(t) \in \mathsf{ec\text{-}all}(\mathsf{tar}(t)) \vee \mathsf{tar\text{-}d}(t) = \emptyset)$. The definition of $\mathsf{ec\text{-}all}$ is postponed to the next section. Please note that the definition of $\mathsf{wf\text{-}tran}$ is well founded.

Note that the definition of SC implies that, within a statechart, each substate can be uniquely referred to by its name. This is an important fact regarding configurations, which is the topic of the next section. From now on, if there is no risk of ambiguity, when denoting a state only the syntactic elements that are relevant in each case are shown. The abbreviation $s_{1..k}$ for $s_1, s_2, .., s_k$ is extensively used. Finally, the substitution notation is used as follows: If t is a term, then $t_{[a/b]}$ is the term which results from replacing all occurrences of a in t by b.

In figure 3.2 a graphical representation of each type of state is shown.

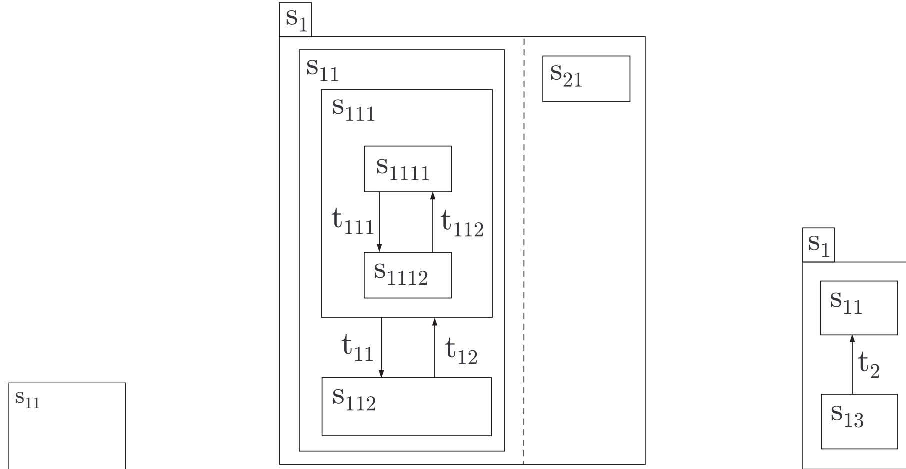


Figure 3.2: Basic-; And-; Or- statechart examples, respectively

3.2 Statechart Configurations

A configuration is a valid global state of a statechart. While the system is in a configuration, events can occur. In response, the system leaves the current configuration by taking a set of

transitions, and enters a new configuration. At the same time, new events can be generated to which the system should respond in the same way. A configurations describes snapshots of a statechart execution.

The function conf: $\mathsf{SC} \rightarrow \mathcal{P}(\mathcal{S})$ gives the current configuration of a statechart s, i.e. the set of the names of all currently active substates within s (also including $\hat{s}$):

$$\begin{array}{ll} \mathsf{conf}([\hat{s}]) & := \{\hat{s}\} \\ \mathsf{conf}([\hat{s},(s_{1..n}),l,T]) & := \{\hat{s}\} \cup \mathsf{conf}(s_l) \\ \mathsf{conf}([\hat{s},(s_{1..n})]) & := \{\hat{s}\} \cup \bigcup_{i=1..n} \mathsf{conf}(s_i) \end{array}$$

The function conf-all: $\mathsf{SC} \rightarrow \mathcal{P}(\mathcal{P}(\mathcal{S}))$ computes the set of all potential configurations of a statechart s (complete or incomplete). By potential, it is implied that not only the current active substate of an or-substate is considered, but all possibilities for choosing such a substate. Note the difference with the definition of conf. The term incomplete denotes a configuration which results from an application of conf-all where the recursion terminates before the basic states are reached. This definition is crucial to handle inter-level transitions just like normal transitions on the level of the uppermost states that the inter-level transition exits and enters.

$$\begin{array}{ll} \mathsf{conf\text{-}all}([\hat{s}]) & := \{\{\hat{s}\}\} \\ \mathsf{conf\text{-}all}([\hat{s},(s_{1..n}),T]) & := \{\{\hat{s}\} \cup c | \exists i \in \{1..n\}.c \in \mathsf{conf\text{-}all}(s_i)\} \cup \{\{\hat{s}\}\} \\ \mathsf{conf\text{-}all}([\hat{s},(s_{1..n})]) & := \{\{\hat{s}\} \cup \bigcup_{i=1..n} c_i | c_i \in \mathsf{conf\text{-}all}(s_i)\} \cup \{\{\hat{s}\}\} \end{array}$$

Incomplete configurations are realized in the second and third cases of the definition by the union with the term $\{\{\hat{s}\}\}$. Note that $\forall s \in \mathsf{SC} : \mathsf{conf}(s) \in \mathsf{conf\text{-}all}(s)$.

The set of possible partial configuratons is further expanded in order to include incomplete parallel configurations. An incomplete parallel configuration is such that it is possible to know the configuration of some of the parallel components of a state, but not all of them. That is,

$$\begin{array}{ll} \mathsf{ec\text{-}all}([\hat{s}]) & := \{\{\hat{s}\}\} \\ \mathsf{ec\text{-}all}([\hat{s},(s_{1..n}),T]) & := \{\{\hat{s}\} \cup c | \exists i \in \{1..n\}.c \in \mathsf{ec\text{-}all}(s_i)\} \cup \{\{\hat{s}\}\} \\ \mathsf{ec\text{-}all}([\hat{s},(s_{1..n})]) & := \{\{\hat{s}\} \cup \bigcup_{i \subseteq \{1,..,n\}} c_i | c_i \in \mathsf{ec\text{-}all}(s_i)\} \cup \{\{\hat{s}\}\} \end{array}$$

The first two cases are analogous to conf-all. The third one allows as a valid configuration a parallel incomplete one, that is, one can "choose" a subset of $\{s_1,..,s_k\}$. For example, let $s = [\hat{s},(s_1,..s_k)]$, where $s_1,...,s_k \in \mathsf{SC}_B$. Then, $\mathsf{ec\text{-}all}(s_i) = \{\{\hat{s}_i\}\}$, $\forall i = 1,..k$ and

$$\mathsf{ec\text{-}all}(s) \supset \{\{\hat{s}_1\},\{\hat{s}_3,\hat{s}_2\},\{\hat{s}_1,\hat{s}_4,\hat{s}_k\}\}$$

The need of this relaxation will become clear later, in section 6.1. Note that, by definition, $\forall s \in \mathsf{SC} : \mathsf{conf\text{-}all}(s) \subseteq \mathsf{ec\text{-}all}(s)$.

3.3 Computing the Next State

If a UML statechart transition is executed, particularly its history type and possibly its target determinator have to be considered. Therefore, the function next is defined, which computes the next state after a UML statechart transition t is executed. Later, this function is used in the SO semantics rule which handles transition execution in an OR statechart.

Given a UML statechart transition t, with target s, history type ht=historyType(t) and target determinator N=tar-d(t) the function next: $\mathsf{HT} \times \mathcal{P}(\mathcal{S}) \times \mathsf{SC} \rightarrow \mathsf{SC}$ computes the UML statechart term $s' = \mathsf{next}(ht, N, s)$ which results after the execution of transition

t.

$$
\begin{array}{lll}
\mathsf{next}(ht, N, [\hat{s}]) & := & [\hat{s}] \\
\mathsf{next}(ht, N, [\hat{s}, (s_{1..n}), l, T]) & := & \left\{ \begin{array}{l} [\hat{s}, (s_{1..n})_{[s_j/\mathsf{next}(ht,N,s_j)]}, j, T] \\ \quad \text{if } (\exists \nu \in N,\ j \in \{1..n\}).\nu = \hat{s}_j \\ \mathsf{next\text{-}stop}(ht, [\hat{s}, (s_{1..n}), l, T]) \text{ otherwise} \end{array} \right. \\
\mathsf{next}(ht, N, [\hat{s}, (s_{1..n})]) & := & [\hat{s}, (\mathsf{next}(ht, N, s_1), .., \mathsf{next}(ht, N, s_n))]
\end{array}
$$

The terms s and $s' = \mathsf{next}(ht, N, s)$ have identical static structure, only the currently active substate information may change. Here it becomes clear why the naming convention is needed: If N contains one of the state names of the substates $s_1, ..., s_n$ then the active state is replaced by s_j and the function is recursively applied. Then, if $N = \mathsf{tar\text{-}d}(t)$, the target determinator information is exploited when zooming into the state hierarchy. Otherwise, i.e. if N does not contain a name ν of one of the state names of the substates $s_{1..k}$, function next-stop is called which uses the history type information to determine currently active substates of a state:

$$
\mathsf{next\text{-}stop}(ht, [\hat{s}, (s_{1..n}), d, l, T]) := \left\{ \begin{array}{ll} [\hat{s}, (s_{1..n}), d, l, T] & \text{if } ht = \mathsf{deep} \\ [\hat{s}, (s_{1..n})_{[s_d/\mathsf{def}(s_d)]}, d, d, T] & \text{if } ht = \mathsf{none} \\ [\hat{s}, (s_{1..n})_{[s_l/\mathsf{def}(s_l)]}, d, l, T] & \text{if } ht = \mathsf{shallow} \end{array} \right.
$$

Note that, if $ht = \mathsf{none}$, then the active state is now the default substate, and the function def is used to initialize the substate s_d.

The function next-stop uses the function def: $\mathsf{SC} \rightarrow \mathsf{SC}$ which defines for an $s \in \mathsf{SC}_O$ that its currently active substate is given by its default substate:

$$
\begin{array}{ll}
\mathsf{def}([\hat{s}]) & := [\hat{s}] \\
\mathsf{def}([\hat{s}, (s_{1..n}), d, l, T]) & := [\hat{s}, (s_{1..n})_{[s_d/\mathsf{def}(s_d)]}, d, d, T] \\
\mathsf{def}([\hat{s}, (s_{1..n})]) & := [\hat{s}, (\mathsf{def}(s_1), ..., \mathsf{def}(s_n))]
\end{array}
$$

When a UML statechart transition t is taken, a set of actions is executed. In general, if a transition from state s_i to s_j with action part α is taken, then the sequence of actions $ex{::}\alpha{::}en$ is executed, with $ex \in \mathsf{exit}(s_i)$, $en \in \mathsf{exit}(s_j)$, where the infix operator :: appends sequences of actions.

$$
\begin{array}{ll}
\mathsf{exit}([\hat{s}, (en, ex)]) & := \{ex\} \\
\mathsf{exit}([\hat{s}, (s_{1..k}), l, T, (en, ex)]) & := \{ex'{::}ex \mid ex' \in \mathsf{exit}(s_l)\} \\
\mathsf{exit}([\hat{s}, (s_{1..k}), (en, ex)]) & := \{m_1{::}..{::}m_k{::}ex \mid \exists p{:}\{1..k\} \leftrightarrow \{1..k\}.\forall_i\ m_i \in \mathsf{exit}(s_{p(i)})\}
\end{array}
$$

$$
\begin{array}{ll}
\mathsf{entry}([\hat{s}, (en, ex)]) & := \{en\} \\
\mathsf{entry}([\hat{s}, (s_{1..k}), l, T, (en, ex)]) & := \{en{::}en' \mid en' \in \mathsf{entry}(s_l)\} \\
\mathsf{entry}([\hat{s}, (s_{1..k}), (en, ex)]) & := \{en{::}m_1{::}..{::}m_k \mid \exists p{:}\{1..k\} \leftrightarrow \{1..k\}.\forall_i\ m_i \in \mathsf{entry}(s_{p(i)})\}
\end{array}
$$

where "$\leftrightarrow$" denotes a bijective function.

3.4 Structured Operational Semantics

Given $s \in \mathsf{SC}$, its structured operational semantics (SO semantics) $[\![s]\!]_{aux}$ is given by a Labelled Transition System $(C, L, \longrightarrow, \mathsf{conf}(s))$ where C is a set of state label sets (configurations), $L \subseteq \mathsf{E} \times \mathsf{A}^* \times \{0, 1\}$ is the set of labels, $\longrightarrow \subseteq C \times L \times C$ is the semantic[2] transition relation, and $\mathsf{conf}(s)$ is the start configuration.

C is defined as $C := \mathsf{conf\text{-}all}(s)$, where L and $\longrightarrow$ are defined by the rules in figure 3.3. We note $c \xrightarrow{e}{}^{f}_{\alpha} c'$ for $(c, (e, \alpha, f), c') \in \longrightarrow$, and $c \not\xrightarrow{e}{}^{f}$ for $\nexists c', \alpha. c \xrightarrow{e}_{\alpha} c'$.

[2]We use the term semantic to differentiate the semantic transition relation from the syntactic element UML transition.

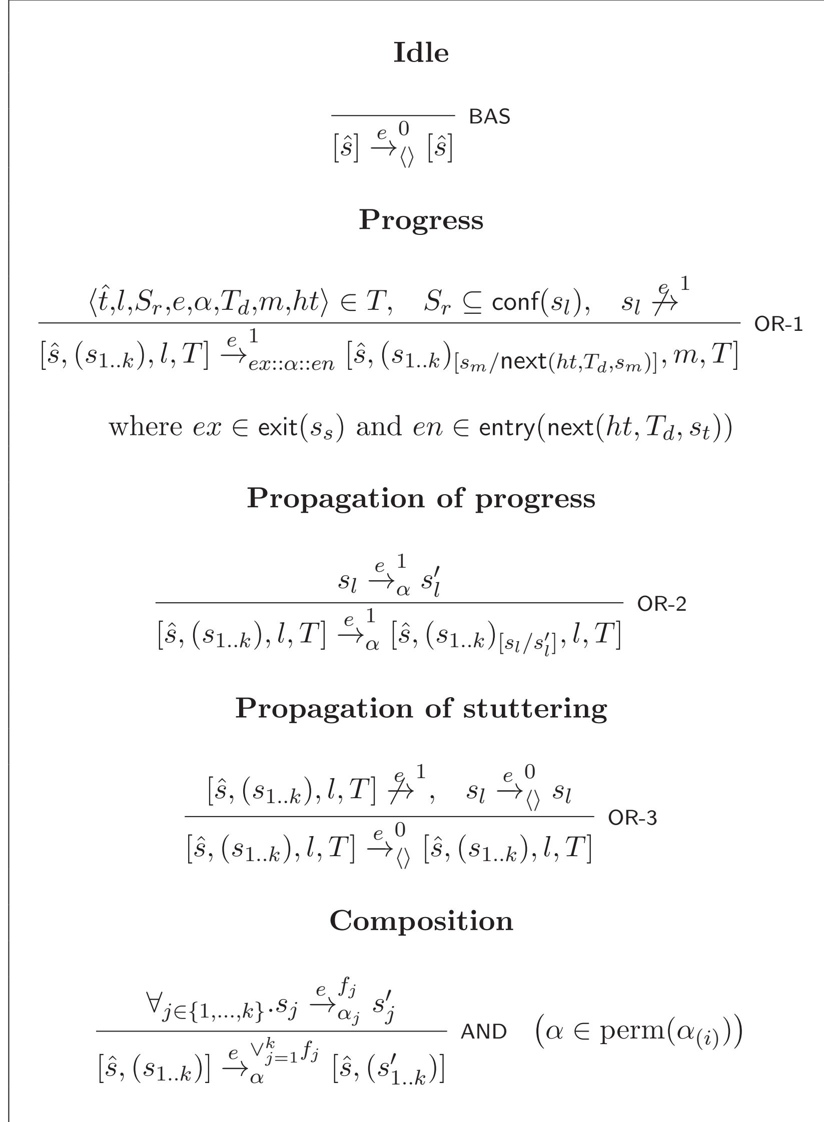


Figure 3.3: Rules of the SO semantics

The stuttering flag f is used to reflect the priority mechanism for statecharts transitions. It also allows the semantics to fullfill the maximality condition of UML statecharts, that is, a maximal number of non conflicting statechart transitions is taken when a semantic transition is performed. The stuttering flag f can take the values 0 or 1. If f=0, then no statechart transition is taken, only the event e is consumed. If f=1 a statechart transition is taken. The flag is needed to assure that idle steps can only occur if no non-idle step is possible. A key role is played by the stuttering step ($f = 0$), since when no UML statechart transition can be taken, a stuttering step (loop) can be done.

Let us explain the OR rules:

OR-1: This rule models a semantic transition from a statechart transition. Given a statechart transition of an OR state with trigger e, then that state can perform a semantic transition with positive flag and trigger e, given that, the source restriction is a complete current configuration of the currently active substate s_l ($S_r \subseteq \mathsf{conf}(s_l)$) and s_l cannot perform a semantic transition with input e and positive flag.

OR-2: This rule propagates the progress of rule OR-1. If the current active substate can perform a semantic transition with positive flag, then the OR state may perform a semantic transition with positive flag.

OR-3: This rule propagates the negative flag. That is, if the current active substate can perform a semantic transition with negative flag, and if the OR state cannot perform a semantic transition with positive flag, then the OR state can perform a semantic transition to itself with positive flag.

3.5 Complete Semantics

As mentioned before, the semantics is divided in two phases, in the first one the auxiliary semantics is defined (section 3.4), and in a second one Kripke structures and the auxiliary semantics are used to define the complete semantics. Given a sequence of input events, the statechart performs a sequence of steps, such that one event of the current sequence of input events is consumed and then a sequence of actions is generated which is added to the remaining sequence of input events. This results in a new sequence of input events which are used in the subsequent step. Kripke structures (a transition system with an initial state) are used in order to model this.

The complete semantics is then, $[\![]\!] : \mathsf{SC} \to \mathcal{K}$, where $\mathcal{K}$ is the set of Kripke structures. Given $s \in \mathsf{SC}$, its complete semantics is given by $[\![s]\!] = (S, st, \Longrightarrow) \in \mathcal{K}$ such that $S = \mathsf{SC} \times A^*$ is the set of Kripke states, $st = (s, \epsilon_0) \in S$ is the start state ($\epsilon_0 \in A^*$) and the transition relation $\Longrightarrow \subseteq S \times S$ is defined by the rule:

$$\frac{s \xrightarrow[\alpha\ \ f]{e} s'}{(s,\epsilon) \Longrightarrow (s',\epsilon'')}\ \mathsf{GI}\ \ (\exists(\epsilon, e, \epsilon') \in \mathsf{sel}, \exists(\alpha, \epsilon', \epsilon'') \in \mathsf{join})$$

where

- $\mathsf{sel} \subseteq A^* \times (A \times A^*)$, given a sequence of events, separates one event from the rest. That is, if $(\epsilon, e, \epsilon') \in \mathsf{sel}$, then it separates e from sequence ϵ and the resulting sequence is ϵ'.
- $\mathsf{join} \subseteq (A^* \times A^*) \times A^*$, composes two sequences of events. That is, if $(\alpha, \epsilon', \epsilon'') \in \mathsf{join}$ then it composes α to the sequence ϵ' and the resulting sequence is ϵ''.

Relations sel and join are defined by the user. This is in accordance to the UML definition which does not define the scheduling strategy of the input event queue.

Chapter 4

Extension and Union of UML Statecharts

In this chapter, an extension relation is defined between UML Statecharts. This extension determines a partial order, which represents when an statechart has a richer structure than another one. Then, given two different extensions of the same statechart, the question on whether there is a way to combine both extensions into an integral new statechart is addressed, showing how this process of combination is formalized. This results into a union operation of UML Statecharts.

4.1 Extension Relation

In this section, a relation $\prec$ between statecharts is defined, such that $s_1 \prec s_2$ (read "s_2 extends s_1") if s_2 enriches states or transitions of s_1 with more complex structures. The definition of the extension relation is given in two stages, first the one-step extension relation $\prec_1$ in figure 4.1.

Basically, it is possible to extend a statechart by either adding a parallel or sequential statechart to it, by adding a new transition between two existing states, or by adding actions in transitions or entry and exit actions. For this last extension, the relation $\triangleleft$ is defined between sequences of actions as the ordinary subsequence relation between elements of A^* ($\alpha \triangleleft \alpha'$ means that α is a subsequence of α'). The subsequence relation is a partial order. We extend subsequence relation to transitions in the following way. Given $t = \langle \hat{t}, j, S, e, \alpha, T, k, ht \rangle \in \mathsf{T}$ and $\tilde{t} = \langle \hat{t}, j, S, e, \tilde{\alpha}, T, k, ht \rangle \in \mathsf{T}$, two statechart transitions which differ only on the generated sequence of actions such that $\alpha \triangleleft \tilde{\alpha}$, then we say that $t \triangleleft \tilde{t}$.

The next step is the definition of $\prec$ as the reflexo-transitive closure of $\prec_1$ which is given in figure 4.2.

In these definitions it is assumed that the well-formedness conditions of the definitions given in section 3 hold whenever a statechart is built.

Finally, note that $(s_{1..k}, \tilde{s})_{[s_t/s'_t]} = (s_{1..k[s_t/s'_t]}, \tilde{s})$ if $t \in \{1..k\}$ and $s_{1..k}, \tilde{s} \in \mathsf{SC}$. If T is a set, $(T \setminus \{t\}) \cup \{t'\}$ is denoted by $T_{[t/t']}$.

Remark 1. *The extension relation $\prec$ defines a partial order on the set SC, up to the default initial substate and the current active of an Or-statechart.*

4.1.1 Extension functions

The rules in figure 4.1 can be written as (partial) functions that, receiving a statechart and the elements to extend it, returns the extended statechart whenever possible. This will be

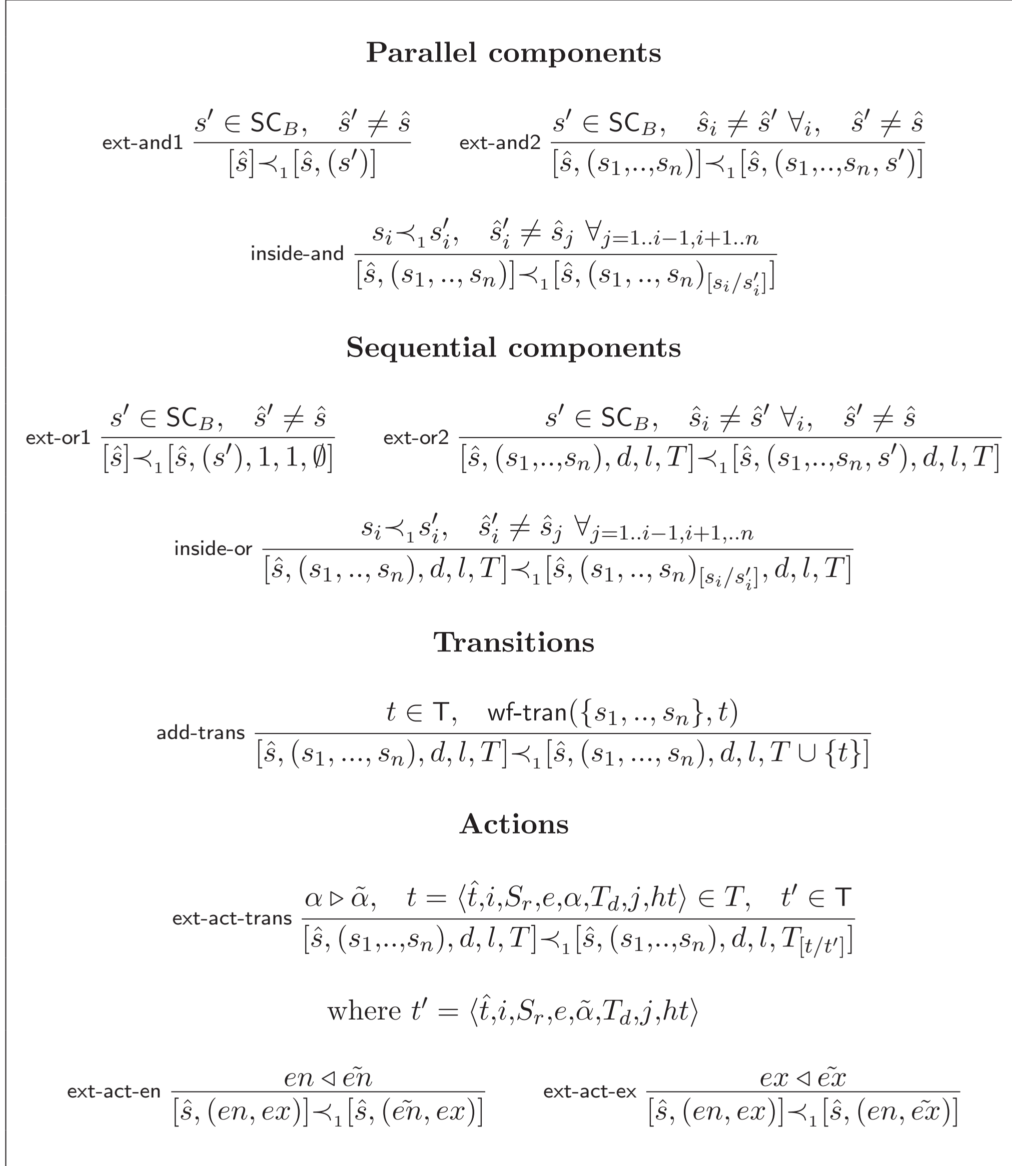


Figure 4.1: One-Step Extension Relation $\prec_1$

Figure 4.2: Extension Relation $\prec$

used in the case study of Chapter 8 where we use the extension relation as a specification tool. The following functions are defined [1]:

[1] $\hookrightarrow$ denotes a partial function

- $\mathsf{ext\text{-}and} : (\mathsf{SC}_B \cup \mathsf{SC}_A) \times \mathsf{SC}_B \hookrightarrow \mathsf{SC}_A$,

$$\begin{array}{lll} \mathsf{ext\text{-}and}([\hat{s}], s') & := [\hat{s}, (s')] & \text{if } \hat{s}' \neq \hat{s} \\ \mathsf{ext\text{-}and}([\hat{s}, (s_1,..,s_n)], s') & := [\hat{s}, (s_1,..,s_n, s')] & \text{if } \hat{s}_i \neq \hat{s}' \ \forall_i \ \wedge \hat{s}' \neq \hat{s} \end{array}$$

- $\mathsf{inside\text{-}and} : \mathsf{SC}_A \times \mathsf{SC} \times \mathsf{SC} \hookrightarrow \mathsf{SC}_A$,

$$\begin{array}{ll} \mathsf{inside\text{-}and}([\hat{s}, (s_1, .., s_n)], s_i, s_i') & := [\hat{s}, (s_1, .., s_n)_{[s_i/s_i']}] \\ & \quad \text{if } s_i \prec_1 s_i' \ \wedge \hat{s}_i' \neq \hat{s}_j \ \forall_{j=1..i-1,i+1..n} \end{array}$$

- $\mathsf{ext\text{-}or} : (\mathsf{SC}_B \cup \mathsf{SC}_O) \times \mathsf{SC}_B \hookrightarrow \mathsf{SC}_O$,

$$\begin{array}{lll} \mathsf{ext\text{-}or}([\hat{s}], s') & := [\hat{s}, (s'), 1, 1, \emptyset] & \text{if } \hat{s}' \neq \hat{s} \\ \mathsf{ext\text{-}or}([\hat{s}, (s_1,..,s_n), d, l, T], s') & := [\hat{s}, (s_1,..,s_n, s'), d, l, T] & \text{if } \hat{s}_i \neq \hat{s}' \ \forall_i \ \wedge \hat{s}' \neq \hat{s} \end{array}$$

- $\mathsf{inside\text{-}or} : \mathsf{SC}_O \times \mathsf{SC} \times \mathsf{SC} \hookrightarrow \mathsf{SC}_O$,

$$\begin{array}{ll} \mathsf{inside\text{-}or}([\hat{s}, (s_1, .., s_n), d, l, T], s_i, s_i') & := [\hat{s}, (s_1, .., s_n)_{[s_i/s_i']}, d, l, T] \\ & \quad \text{if } s_i \prec_1 s_i' \ \wedge \hat{s}_i' \neq \hat{s}_j \ \forall_{j=1..i-1,i+1..n} \end{array}$$

- $\mathsf{add\text{-}trans} : \mathsf{SC}_O \times \mathsf{T} \hookrightarrow \mathsf{SC}_O$,

$$\begin{array}{ll} \mathsf{add\text{-}trans}([\hat{s}, (s_1, ..., s_n), d, l, T], t) & := [\hat{s}, (s_1, ..., s_n), d, l, T \cup \{t\}] \\ & \quad \text{if } \mathsf{wf\text{-}tran}(\{s_1, .., s_n\}, t) \end{array}$$

- $\mathsf{ext\text{-}act\text{-}trans\text{-}a} : \mathsf{SC}_O \times \mathsf{T} \times \mathsf{A} \to \mathsf{SC}_O$, Let $t = \langle \hat{t}, i, S_r, e, \alpha, T_d, j, ht \rangle \in T$,

$$\begin{array}{ll} \mathsf{ext\text{-}act\text{-}trans\text{-}a}([\hat{s}, (s_1,..,s_n), d, l, T], t, a) & := [\hat{s}, (s_1,..,s_n), d, l, T_{[t/t']}] \\ & \quad \text{where } t' = \langle \hat{t}, i, S_r, e, \alpha{::}a, T_d, j, ht \rangle \end{array}$$

  equivalently, $\mathsf{ext\text{-}act\text{-}trans\text{-}p} : \mathsf{SC}_O \times \mathsf{T} \times \mathsf{A} \to \mathsf{SC}_O$ prepends the action a to the sequence α.

- $\mathsf{ext\text{-}act\text{-}en\text{-}a} : \mathsf{SC} \times \mathsf{E} \to \mathsf{SC}$,

$$\mathsf{ext\text{-}act\text{-}en\text{-}a}([\hat{s}, (en, ex)], e) \quad := [\hat{s}, (en{::}e, ex)]$$

  equivalently, $\mathsf{ext\text{-}act\text{-}en\text{-}p} : \mathsf{SC} \times \mathsf{E} \to \mathsf{SC}$ prepends the event e to the sequence en.

- $\mathsf{ext\text{-}act\text{-}ex\text{-}a} : \mathsf{SC} \times \mathsf{E} \to \mathsf{SC}$,

$$\mathsf{ext\text{-}act\text{-}ex\text{-}a}([\hat{s}, (en, ex)], e) \quad := [\hat{s}, (en, ex{::}e)]$$

  equivalently, $\mathsf{ext\text{-}act\text{-}ex\text{-}p} : \mathsf{SC} \times \mathsf{E} \to \mathsf{SC}$ prepends the event e to the sequence ex.

In figure 4.3 some examples of the extension function are shown.

4.2 UML Statecharts Union

Given a statechart, it can be extended in several ways. The question is whether there is a way to combine different extensions into an integral new statechart. Formally, given $r_1, r_2 \in \mathsf{SC}$ such that

$$\exists s \in \mathsf{SC}.\ s \prec r_1 \wedge s \prec r_2$$

the idea is to define a new statechart $r_1 \cup r_2$ such that

$$r_1 \prec r_1 \cup r_2 \quad \text{and} \quad r_2 \prec r_1 \cup r_2$$

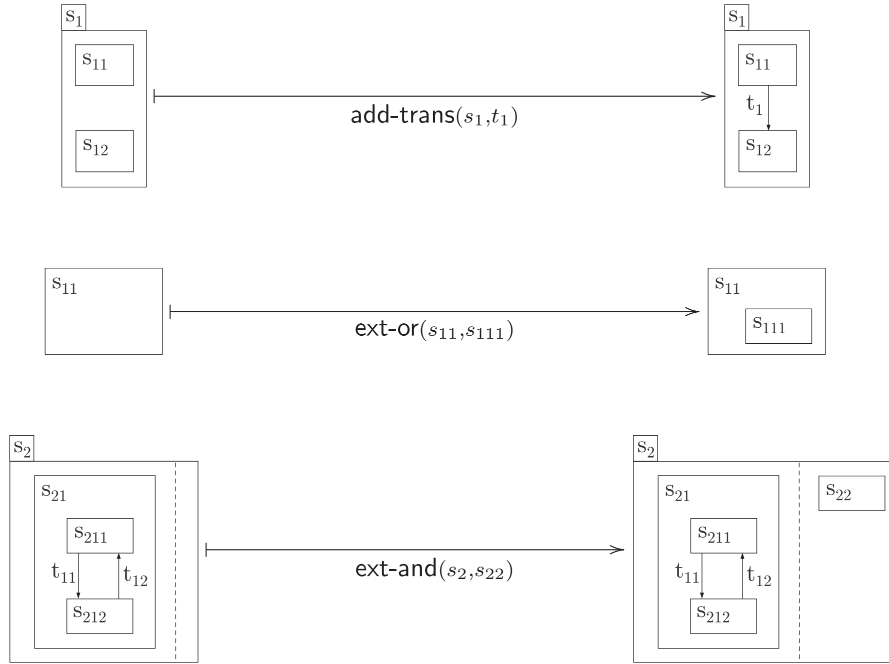


Figure 4.3: add-trans, ext-or and ext-and extension function examples

Moreover, a minimum amount of extending steps as possible is desired, i.e.,

$$\forall r_3 \in \mathsf{SC}.\ (r_1 \prec r_3 \wedge r_2 \prec r_3) \Rightarrow r_1 \cup r_2 \prec r_3$$

It is not possible to define the union of two statecharts in all the cases, even if they are both extensions of the same statechart (consider, for instance, s to be a basic statechart which is extended parallelly on the one hand and sequentially on the other). Therefore, it is needed to handle inconsistent statecharts. For the sake of completeness, a new statechart is added to the syntax defined in section 3.1: $\top$ will stand for the overspecified element of the set SC, as well as for any statechart having it as a component. Regarding the $\prec$ relation, $\top$ is a extension of any statechart. Formally,

$$\forall s \in \mathsf{SC}.\ s \prec \top$$

The definition of $\cup$ is simple but quite extensive, since all the pairs r_1, r_2 such that r_1 and r_2 can be extensions of the same statechart s, according to the definition given in figure 4.1 must be taken into account. It basically consists of carrying out both extensions on the original statechart, whenever this is possible. In this definition, the current active state and the default initial substate of an Or-Statechart play no role because they contain only dynamic information. Note that, given $s_1, s_2 \in \mathsf{SC}$, such that $\mathsf{name}(s_1) = \mathsf{name}(s_2) = \hat{s}$, then there always exists $s \in \mathsf{SC}$ such that $s \prec s_1 \wedge s \prec s_2$, for instance s can be $[\hat{s}, (\langle\rangle, \langle\rangle)]$.

Definition 1. *Let $r_1, r_2 \in SC$ such that $\exists s \in SC.\ s \prec r_1 \wedge s \prec r_2$, then $r_1 \cup r_2$ is defined by induction on $s \prec r_1$.*

Since $\prec$ is the reflexo-transitive closure of $\prec_1$, we consider all the rules of $\prec_1$ plus reflexivity and transitivity.

Extension ext-and1:
Let s=[$\hat{s}$,(en,ex)] and r_1=[$\hat{s}$, (s'), (en, ex)], where s'∈SC_B [2]:

$$\text{ea1a}\ \frac{r_2{=}[\hat{s},(s'')],\quad s''{\in}\mathsf{SC}_B}{r_1{\cup}r_2=[\hat{s},(s',s'')]}\ (\hat{s}'{\neq}\hat{s}'')\qquad \text{ea1b}\ \frac{r_2{=}[\hat{s},(s'')],\quad s''{\in}\mathsf{SC}_B}{r_1{\cup}r_2=[\hat{s},(s'{\cup}s'')]}\ (\hat{s}'{=}\hat{s}'')\qquad \text{ea1c}\ \frac{r_2{=}[\hat{s},(s''),\emptyset],\quad s''{\in}\mathsf{SC}_B}{r_1{\cup}r_2=\top}$$

$$\text{ea1d}\ \frac{r_2{=}[\hat{s},(\tilde{en},ex)],\quad en\lhd\tilde{en}}{r_1{\cup}r_2=[\hat{s},(s'),(\tilde{en},ex)]}\qquad \text{ea1e}\ \frac{r_2{=}[\hat{s},(en,\tilde{ex})],\quad ex\lhd\tilde{ex}}{r_1{\cup}r_2=[\hat{s},(s'),(en,\tilde{ex})]}$$

Extension ext-and2:
Let s=[$\hat{s}$,(s_1,..,s_n),(en,ex)] and r_1=[$\hat{s}$, (s_1,..,s_n, s'), (en, ex)], where s'∈SC_B:

$$\text{ea2a}\ \frac{r_2{=}[\hat{s},(s_1,..,s_n,s'')],\quad s''{\in}\mathsf{SC}_B}{r_1{\cup}r_2=[\hat{s},(s_1,..,s_n,s',s'')]}\ (\hat{s}'{\neq}\hat{s}'')\qquad \text{ea2b}\ \frac{r_2{=}[\hat{s},(s_1,..,s_n,s'')],\quad s''{\in}\mathsf{SC}_B}{r_1{\cup}r_2=[\hat{s},(s_1,..,s_n,s'{\cup}s'')]}\ (\hat{s}'{=}\hat{s}'')$$

$$\text{ea2c}\ \frac{r_2{=}[\hat{s},(s_1,..,s_i,..,s_n)]\quad s_i\prec\tilde{s}_i}{r_1{\cup}r_2=[\hat{s},(s_1,..,\tilde{s}_i,..,s_n,s')]}$$

$$\text{ea2d}\ \frac{r_2{=}[\hat{s},(s_1,..,s_n),(\tilde{en},ex)],\quad en\lhd\tilde{en}}{r_1{\cup}r_2=[\hat{s},(s_1,..,s_n,s'),(\tilde{en},ex)]}\qquad \text{ea2e}\ \frac{r_2{=}[\hat{s},(s_1,..,s_n),(en,\tilde{ex})],\quad ex\lhd\tilde{ex}}{r_1{\cup}r_2=[\hat{s},(s_1,..,s_n,s'),(en,\tilde{ex})]}$$

Extension inside-and:
Let s=[$\hat{s}$, (s_1, .., s_i, .., s_n), (en, ex)] and r_1=[$\hat{s}$, (s_1, .., $\tilde{s}_i$, .., s_n), (en, ex)], where $s_i \prec \tilde{s}_i$

$$\text{iaa}\ \frac{s_i\prec\tilde{s}_i,\quad r_2{=}[\hat{s},(s_1,..,s_n,s')],\quad s'{\in}\mathsf{SC}_B}{r_1{\cup}r_2=[\hat{s},(s_1,..,\tilde{s}_i,..,s_n,s')]}$$

$$\text{iab}\ \frac{s_i\prec\tilde{s}_i,\quad r_2{=}[\hat{s},(s_1,..,\tilde{s}_j,..,s_n)],\quad s_j\prec\tilde{s}_j}{r_1{\cup}r_2=[\hat{s},(s_1,..,\tilde{s}_i{\cup}\tilde{s}_j,..,s_n)]}\ (i=j)$$

$$\text{iac}\ \frac{s_i\prec\tilde{s}_i,\quad r_2{=}[\hat{s},(s_1,..,\tilde{s}_j,..,s_n)],\quad s_j\prec\tilde{s}_j}{r_1{\cup}r_2=[\hat{s},(s_1,..,\tilde{s}_i,..,\tilde{s}_j,..,s_n)]}\ (i\neq j)$$

$$\text{iad}\ \frac{s_i\prec\tilde{s}_i,\quad r_2{=}[\hat{s},(s_1,..,s_n),(\tilde{en},ex)],\quad en\lhd\tilde{en}}{r_1{\cup}r_2=[\hat{s},(s_1,..,\tilde{s}_i,..,s_n),(\tilde{en},ex)]}$$

$$\text{iae}\ \frac{s_i\prec\tilde{s}_i,\quad r_2{=}[\hat{s},(s_1,..,s_n),(en,\tilde{ex})],\quad ex\lhd\tilde{ex}}{r_1{\cup}r_2=[\hat{s},(s_1,..,\tilde{s}_i,..,s_n),(en,\tilde{ex})]}$$

Extension ext-or1:
Let s=[$\hat{s}$,(en,ex)] and r_1=[$\hat{s}$, (s'), ∅, (en, ex)], where s'∈SC_B.

$$\text{eo1a}\ \frac{r_2{=}[\hat{s},(s'')],\quad s''{\in}\mathsf{SC}_B}{r_1{\cup}r_2=\top}\qquad \text{eo1b}\ \frac{r_2{=}[\hat{s},(s''),\emptyset],\quad s''{\in}\mathsf{SC}_B}{r_1{\cup}r_2=[\hat{s},(s',s''),\emptyset]}\ (\hat{s}'{\neq}\hat{s}'')\qquad \text{eo1c}\ \frac{r_2{=}[\hat{s},(s''),\emptyset],\quad s''{\in}\mathsf{SC}_B}{r_1{\cup}r_2=[\hat{s},(s'{\cup}s''),\emptyset]}\ (\hat{s}'{=}\hat{s}'')$$

$$\text{eo1d}\ \frac{r_2{=}[\hat{s},(s'),\emptyset,(\tilde{en},ex)],\quad en\lhd\tilde{en}}{r_1{\cup}r_2=[\hat{s},(s'),\emptyset,(\tilde{en},ex)]}\qquad \text{eo1e}\ \frac{r_2{=}[\hat{s},(s'),\emptyset,(en,\tilde{ex})],\quad ex\lhd\tilde{ex}}{r_1{\cup}r_2=[\hat{s},(s'),\emptyset,(en,\tilde{ex})]}$$

Extension ext-or2:
Let s=[$\hat{s}$, (s_1,..,s_n), T, (en, ex)] and r_1=[$\hat{s}$, (s_1,..,s_n, s'), T, (en, ex)], where s'∈SC_B.

$$\text{eo2a}\ \frac{r_2{=}[\hat{s},(s_1,..,s_n,s''),T],\quad s''{\in}\mathsf{SC}_B}{r_1{\cup}r_2=[\hat{s},(s_1,..,s_n,s',s''),T]}\ (\hat{s}'{\neq}\hat{s}'')\qquad \text{eo2b}\ \frac{r_2{=}[\hat{s},(s_1,..,s_n,s''),T],\quad s''{\in}\mathsf{SC}_B}{r_1{\cup}r_2=[\hat{s},(s_1,..,s_n,s'{\cup}s''),T]}\ (\hat{s}'{=}\hat{s}'')$$

[2] s and r_1 are hypotesis for all the rules included in this definition, so for the sake of symplicity, we define them once. The same for all the following definitions.

$$\text{eo2c}\ \frac{r_2{=}[\hat{s},(s_1,..,s_i,..,s_n),T],\quad s_i\prec\tilde{s}_i}{r_1{\cup}r_2=[\hat{s},(s_1,..,\tilde{s}_i,..,s_n,s'),T]}\qquad \text{eo2d}\ \frac{r_2{=}[\hat{s},(s_1,..,s_n),T'],\quad T'{=}T\cup\{t\}}{r_1{\cup}r_2=[\hat{s},(s_1,..,s_n,s'),T']}$$

$$\text{eo2e}\ \frac{r_2{=}[\hat{s},(s_1,..,s_n),T'],\quad T'{=}T_{[t/t']},\quad t{\in}T,\quad t'{\in}\mathsf{T},\quad t\lhd t'}{r_1{\cup}r_2=[\hat{s},(s_1,..,s_n,s'),T']}$$

$$\text{eo2f}\ \frac{r_2{=}[\hat{s},(s_1,..,s_n),T,(\tilde{en},ex)],\quad en\lhd\tilde{en}}{r_1{\cup}r_2=[\hat{s},(s_1,..,s_n,s'),T,(\tilde{en},ex)]}\qquad \text{eo2g}\ \frac{r_2{=}[\hat{s},(s_1,..,s_n),T,(en,\tilde{ex})],\quad ex\lhd\tilde{ex}}{r_1{\cup}r_2=[\hat{s},(s_1,..,s_n,s'),T,(en,\tilde{ex})]}$$

Extension inside-or:

Let $s{=}[\hat{s},(s_1,..,s_i,..,s_n),T,(en,ex)]$ and $r_1{=}[\hat{s},(s_1,..,\tilde{s}_i,..,s_n),T,(en,ex)]$ with $s_i\prec\tilde{s}_i$

$$\text{ioa}\ \frac{r_2{=}[\hat{s},(s_1,..,s_n,s'),T],\quad s'{\in}\mathsf{SC}_B}{r_1{\cup}r_2=[\hat{s},(s_1,..,\tilde{s}_i,..,s_n,s'),T]}$$

$$\text{iob}\ \frac{r_2{=}[\hat{s},(s_1,..,\tilde{s}_j,..,s_n),T],\quad s_j\prec\tilde{s}_j}{r_1{\cup}r_2=[\hat{s},(s_1,..,\tilde{s}_i{\cup}\tilde{s}_j,..,s_n),T]}\ (i=j)$$

$$\text{ioc}\ \frac{r_2{=}[\hat{s},(s_1,..,\tilde{s}_j,..,s_n),T],\quad s_j\prec\tilde{s}_j}{r_1{\cup}r_2=[\hat{s},(s_1,..,\tilde{s}_i,..,\tilde{s}_j,..,s_n),T]}\ (i\neq j)$$

$$\text{iod}\ \frac{r_2{=}[\hat{s},(s_1,..,s_n),T'],\quad T'=T\cup\{t\}}{r_1{\cup}r_2=[\hat{s},(s_1,..,\tilde{s}_i,..,s_n),T']}$$

$$\text{ioe}\ \frac{r_2{=}[\hat{s},(s_1,..,s_n),T'],\quad T'{=}T_{[t/t']},\quad t{\in}T,\quad t'{\in}\mathsf{T},\quad t\lhd t'}{r_1{\cup}r_2=[\hat{s},(s_1,..,\tilde{s}_i,..,s_n),T']}$$

$$\text{iof}\ \frac{r_2{=}[\hat{s},(s_1,..,s_n),T,(\tilde{en},ex)],\quad en\lhd\tilde{en}}{r_1{\cup}r_2=[\hat{s},(s_1,..,\tilde{s}_i,..,s_n),T,(\tilde{en},ex)]}\qquad \text{iog}\ \frac{r_2{=}[\hat{s},(s_1,..,s_n),(en,\tilde{ex})],\quad ex\lhd\tilde{ex}}{r_1{\cup}r_2=[\hat{s},(s_1,..,\tilde{s}_i,..,s_n),T,(en,\tilde{ex})]}$$

Extension ext-trans:

Let $s{=}[\hat{s},(s_1,..,s_n),T,(en,ex)]$ and $r_1{=}[\hat{s},(s_1,..,s_n),T',(en,ex)]$ with $T'{=}T\cup\{t\}$

$$\text{eta}\ \frac{r_2{=}[\hat{s},(s_1,..,s_n,s'),T],\quad s'{\in}\mathsf{SC}_B}{r_1{\cup}r_2=[\hat{s},(s_1,..,s_n,s'),T']}\qquad \text{etb}\ \frac{r_2{=}[\hat{s},(s_1,..,\tilde{s}_i,..,s_n),T],\quad s_i\prec\tilde{s}_i}{r_1{\cup}r_2=[\hat{s},(s_1,..,\tilde{s}_i,..,s_n),T']}$$

$$\text{etc}\ \frac{r_2{=}[\hat{s},(s_1,..,s_n),T''],\quad T''{=}T\cup\{t'\}}{r_1{\cup}r_2=[\hat{s},(s_1,..,s_n),T'{\cup}\{t'\}]}\ (\hat{t}\neq\hat{t}')$$

$$\text{etd}\ \frac{r_2{=}[\hat{s},(s_1,..,s_n),T'',(en,ex)],\quad T''{=}T\cup\{t'\}}{r_1{\cup}r_2=\top}\ (\hat{t}=\hat{t}')$$

$$\text{ete}\ \frac{r_2{=}[\hat{s},(s_1,..,s_n),T''],\quad T''{=}T_{[t'/t'']},\quad t'{\in}T,\quad t''{\in}\mathsf{T},\quad t'\lhd t''}{r_1{\cup}r_2=[\hat{s},(s_1,..,s_n),T''{\cup}\{t\}]}$$

$$\text{etf}\ \frac{r_2{=}[\hat{s},(s_1,..,s_n),T,(\tilde{en},ex)],\quad en\lhd\tilde{en}}{r_1{\cup}r_2=[\hat{s},(s_1,..,s_n),T',(\tilde{en},ex)]}\qquad \text{etg}\ \frac{r_2{=}[\hat{s},(s_1,..,s_n),T,(en,\tilde{ex})],\quad ex\lhd\tilde{ex}}{r_1{\cup}r_2=[\hat{s},(s_1,..,s_n),T',(en,\tilde{ex})]}$$

Extension ext-act-trans:

Let $s{=}[\hat{s},(s_1,..,s_n),T,(en,ex)]$ and $r_1{=}[\hat{s},(s_1,..,s_n),T',(en,ex)]$ with $T'{=}T_{[t/t']}$ and $t\lhd t'$

$$\text{eata}\ \frac{r_2{=}[\hat{s},(s_1,..,s_n,s'),T],\quad s'{\in}\mathsf{SC}_B}{r_1{\cup}r_2=[\hat{s},(s_1,..,s_n,s'),T']}\qquad \text{eatb}\ \frac{r_2{=}[\hat{s},(s_1,..,\tilde{s}_i,..,s_n),T],\quad s_i\prec\tilde{s}_i}{r_1{\cup}r_2=[\hat{s},(s_1,..,\tilde{s}_i,..,s_n),T']}$$

$$\text{eatc}\ \frac{r_2{=}[\hat{s},(s_1,..,s_n),T''],\quad T''{=}T\cup\{t''\}}{r_1{\cup}r_2=[\hat{s},(s_1,..,s_n),T'{\cup}\{t''\}]}$$

$$\text{eatd}\ \frac{r_2{=}[\hat{s},(s_1,..,s_n),T''],\quad T''{=}T_{[t''/t''']},\quad t''{\in}T,\quad t'''{\in}\mathsf{T},\quad t''\lhd t'''}{r_1{\cup}r_2=[\hat{s},(s_1,..,s_n,s'),T'_{[t''/t''']}]}\ (\hat{t}\neq\hat{t''})$$

$$\text{eate}\ \frac{r_2{=}[\hat{s},(s_1,..,s_n),T''],\quad T''{=}T_{[t''/t''']},\quad t''{\in}T,\quad t'''{\in}\mathsf{T},\quad t''\lhd t'''}{r_1{\cup}r_2=\top}\ (\hat{t}=\hat{t''})$$

$$\text{eatf}\ \frac{r_2{=}[\hat{s},(s_1,..,s_n),T,(\tilde{en},ex)],\quad en\lhd\tilde{en}}{r_1{\cup}r_2=[\hat{s},(s_1,..,s_n),T',(\tilde{en},ex)]}\qquad \text{eatg}\ \frac{r_2{=}[\hat{s},(s_1,..,s_n),T,(en,\tilde{ex})],\quad ex\lhd\tilde{ex}}{r_1{\cup}r_2=[\hat{s},(s_1,..,s_n),T',(en,\tilde{ex})]}$$

Extension ext-act-en:

Let $s{=}[\hat{s},(en,ex)]$, $r_1{=}[\hat{s},(\tilde{en},ex)]$ with $en\lhd\tilde{en}$ then,

$$\text{eaea}\ \frac{r_2{=}[\hat{s},(s'),(en,ex)],\quad s'{\in}\mathsf{SC}_B}{r_1{\cup}r_2=[\hat{s},(s'),(\tilde{en},ex)]}\qquad \text{eaeb}\ \frac{r_2{=}[\hat{s},(s'),\emptyset,(en,ex)],\quad s'{\in}\mathsf{SC}_B}{r_1{\cup}r_2=[\hat{s},(s'),\emptyset,(\tilde{en},ex)]}\qquad \text{eaec}\ \frac{r_2{=}[\hat{s},(\tilde{en}',ex)]}{r_1{\cup}r_2=[\hat{s},(\tilde{en},ex)]}\ (\tilde{en}'\lhd\tilde{en})$$

$$\text{eaed}\ \frac{r_2{=}[\hat{s},(\tilde{en}',ex)]}{r_1{\cup}r_2=[\hat{s},(\tilde{en}',ex)]}\ (\tilde{en}\lhd\tilde{en}')\qquad \text{eaee}\ \frac{r_2{=}[\hat{s},(en,\tilde{ex})]}{r_1{\cup}r_2=[\hat{s},(\tilde{en},\tilde{ex})]}$$

Let $s{=}[\hat{s},(s_1,..,s_n),(en,ex)]$, $r_1{=}[\hat{s},(s_1,..,s_n),(\tilde{en},ex)]$ with $en\lhd\tilde{en}$ then,

$$\text{eaef}\ \frac{r_2{=}[\hat{s},(s_1,..,s_n,s'),(en,ex)],\quad s'{\in}\mathsf{SC}_B}{r_1{\cup}r_2=[\hat{s},(s_1,..,s_n,s'),(\tilde{en},ex)]}\qquad \text{eaeg}\ \frac{r_2{=}[\hat{s},(s_1,..,\tilde{s}_i,..,s_n),(en,ex)],\quad s_i\prec\tilde{s}_i}{r_1{\cup}r_2=[\hat{s},(s_1,..,\tilde{s}_i,..,s_n),(\tilde{en},ex)]}$$

$$\text{eaeh}\ \frac{r_2{=}[\hat{s},(s_1,..,s_n),(\tilde{en}',ex)]}{r_1{\cup}r_2=[\hat{s},(s_1,..,s_n),(\tilde{en},ex)]}\ (\tilde{en}'\lhd\tilde{en})\qquad \text{eaei}\ \frac{r_2{=}[\hat{s},(s_1,..,s_n),(\tilde{en}',ex)]}{r_1{\cup}r_2=[\hat{s},(s_1,..,s_n),(\tilde{en}',ex)]}\ (\tilde{en}\lhd\tilde{en}')$$

$$\text{eaej}\ \frac{r_2{=}[\hat{s},(s_1,..,s_n),(en,\tilde{ex})],\quad ex\lhd\tilde{ex}}{r_1{\cup}r_2=[\hat{s},(s_1,..,s_n),(\tilde{en},\tilde{ex})]}$$

Let $s{=}[\hat{s},(s_1,..,s_n),T,(en,ex)]$, $r_1{=}[\hat{s},(s_1,..,s_n),T,(\tilde{en},ex)]$ with $en\lhd\tilde{en}$ then,

$$\text{eaek}\ \frac{r_2{=}[\hat{s},(s_1,..,s_n,s'),T,(en,ex)],\quad s'{\in}\mathsf{SC}_B}{r_1{\cup}r_2=[\hat{s},(s_1,..,s_n,s'),T,(\tilde{en},ex)]}\qquad \text{eael}\ \frac{r_2{=}[\hat{s},(s_1,..,\tilde{s}_i,..,s_n),T,(en,ex)],\quad s_i\prec\tilde{s}_i}{r_1{\cup}r_2=[\hat{s},(s_1,..,s'_i,..,s_n),T,(\tilde{en},ex)]}$$

$$\text{eaem}\ \frac{r_2{=}[\hat{s},(s_1,..,s_n),T',(en,ex)],\quad T'{=}T\cup\{t\}}{r_1{\cup}r_2=[\hat{s},(s_1,..,s_n),T',(\tilde{en},ex)]}$$

$$\text{eaen}\ \frac{r_2{=}[\hat{s},(s_1,..,s_n),T',(en,ex)],\quad T'{=}T_{[t/t']},\quad t{\in}T,\quad t'{\in}\mathsf{T},\quad t\lhd t'}{r_1{\cup}r_2=[\hat{s},(s_1,..,s_n),T',(\tilde{en},ex)]}$$

$$\text{eaeo}\ \frac{r_2{=}[\hat{s},(s_1,..,s_n),T,(\tilde{en}',ex)],\quad en\lhd\tilde{en}'}{r_1{\cup}r_2=[\hat{s},(s_1,..,s_n),(\tilde{en},ex)]}\ (\tilde{en}'\lhd\tilde{en})$$

$$\text{eaep}\ \frac{r_2{=}[\hat{s},(s_1,..,s_n),T,(\tilde{en}',ex)],\quad en\lhd\tilde{en}'}{r_1{\cup}r_2=[\hat{s},(s_1,..,s_n),(\tilde{en}',ex)]}\ (\tilde{en}\lhd\tilde{en}')$$

$$\text{eaeq}\ \frac{r_2{=}[\hat{s},(s_1,..,s_n),T,(en,\tilde{ex})],\quad ex\lhd\tilde{ex}}{r_1{\cup}r_2=[\hat{s},(s_1,..,s_n),T,(\tilde{en},\tilde{ex})]}$$

Extension ext-act-ex: analogous to **ext-act-en**.

reflexivity:

$$\text{reflexivity}\ \frac{s=r_1,\quad r_2{\in}\mathsf{SC}}{r_1{\cup}r_2=r_2}\ (s\prec r_2)$$

transitivity:

$$\text{transitivity}\ \frac{s\prec r',\quad r'\prec r_1,\quad r_2{\in}\mathsf{SC}}{r_1{\cup}r_2=r_1{\cup}(r'{\cup}r_2)}\ (s\prec r_2)$$

4.2.1 Properties of the Union

In this section we present four properties of the union. These properties show that indeed it can be considered as a join, in a lattice theory context. Note that, by definition, for all $s \in \mathsf{SC}$, $s \cup s = s$ and for all $s \in \mathsf{SC}$, $s \cup \top = \top$.

Property 1 (Commutativity of $\cup$)**.** *For all $s, r_1, r_2 \in \mathsf{SC}$ such that $s \prec r_1$ and $s \prec r_2$, $r_1 \cup r_2 = r_2 \cup r_1$, up to the order of the substates.*

Proof sketch. By induction on $\cup$ rules. For each extension, we briefly analyze the rules where union differs from $\top$ and is not trivially commutative by definition. Extension ext-and1: Rule ea1a is commutative up to the order of the substates and in rule ea1b commutativity holds by induction hypothesis. Extension ext-and2: Rule ea2a is commutative up to the order of the substates, in rule ea2b commutativity holds by induction hypothesis, and rule ea2c is symmetric with respect to rule iaa of extension inside-and. Extension inside-and: In rule iab commutativity holds by induction hypothesis, and in rule iac commutativity comes from the fact that $i \neq j$. Extension ext-or1: Rule eo1b is commutative up to the order of the substates and in rule eo1c commutativity holds by induction hypothesis. Extension ext-or2: Rule eo1b is commutative up to the order of the substates, in rule eo1c commutativity holds by induction hypothesis, and rule eo2c is symmetric with respect to rule ioa of extension inside-or, rule eo2d is symmetric with respect to rule eta of extension ext-trans and rule eo2e is symmetric with respect to rule eata of extension ext-act-trans. Extension inside-or: In rule iob commutativity holds by induction hypothesis, rule iod is symmetric with respect to rule etb of extension ext-trans and rule ioe is symmetric with respect to rule eatb of extension ext-act-trans. Extension ext-trans: Rule ete is symmetric with respect to rule eatc of extension ext-act-trans. Extension ext-act-trans: All rules are either symmetric to anther already mentioned, or trivially commutative. Extensions ext-act-en and ext-act-ex are trivially commutative. Finally, extensions reflexivity and transitivity commutes by definition.

□

Property 2 (Associativity of $\cup$)**.** *For all $s, r_1, r_2, r_3 \in \mathsf{SC}$ such that $s \prec r_1$, $s \prec r_2$ and $s \prec r_3$, $(r_1 \cup r_2) \cup r_3 = r_1 \cup (r_2 \cup r_3)$, up to the order of the substates.*

Proof. The proof is straightforward because the order on which the rules are applied does not matter. □

Property 3 (Existence of meets)**.** *For all $s, r_1, r_2 \in \mathsf{SC}$ such that $s \prec r_1$ and $s \prec r_2$, then $r_1 \prec r_1 \cup r_2$ and $r_2 \prec r_1 \cup r_2$.*

Proof sketch. The property holds directly in all cases of the definition of $\cup$ by construction, since $r_1 \cup r_2$ is always defined as an extension of both s_1 and s_2. The only non trivial case is the transitivity one: if $s \prec r' \prec r_1$ and $s \prec r_2$ then $r_1 \cup r_2$ is defined as $r_1 \cup (r' \cup r_2)$. This is well defined, since from $s \prec r'$ and $s \prec r_2$, $r' \cup r_2$ is defined by induction hypothesis, and $r' \prec r' \cup r_2$, $r_2 \prec r' \cup r_2$. Then, again by induction hypothesis, from $r' \prec r_1$, it can be seen that $r_1 \cup (r' \cup r_2)$ is defined and $r_1 \prec r_1 \cup (r' \cup r_2)$, $r_2 \prec r_1 \cup (r' \cup r_2)$. □

Property 4 (Existence of meets)**.** *For all $s, r_1, r_2, r_3 \in \mathsf{SC}$ such that $s \prec r_1$ and $s \prec r_2$, if $(r_1 \prec r_3 \wedge r_2 \prec r_3)$, then $r_1 \cup r_2 \prec r_3$.*

Proof sketch. The property holds directly in all cases of the definition of $\cup$, since the least possible extension to both statecharts is always performed. The only non trivial case is the transitivity one: if $s \prec r' \prec r_1$ and $s \prec r_2$, then $r_1 \cup r_2 = r_1 \cup (r' \cup r_2)$. Let $r_1 \prec r_3$ and $r_2 \prec r_3$. Then, by $r' \prec r_1$ we have $r' \prec r_3$ and hence $r' \cup r_2 \prec r_3$ by $r_2 \prec r_3$ and induction hypothesis. Finally, from $r_1 \prec r_3$ and induction hypothesis, $r_1 \cup (r' \cup r_2) \prec r_3$ is obtained. □

4.2.2 Example

Now, an example of the computation of the union is shown. Given $r_1, r_2 \in \mathsf{SC}$, both extensions of some $s \in \mathsf{SC}$, the method for computing $r_1 \cup r_2$ is to perform single steps of extensions. The transitivity case for the definition of $\cup$ and the properties stated above ensure confluence, i.e., no matter the order in which the extensions are done, it is always possible to obtain a unique "diamond" with s at the top and $r_1 \cup r_2$ at the bottom.

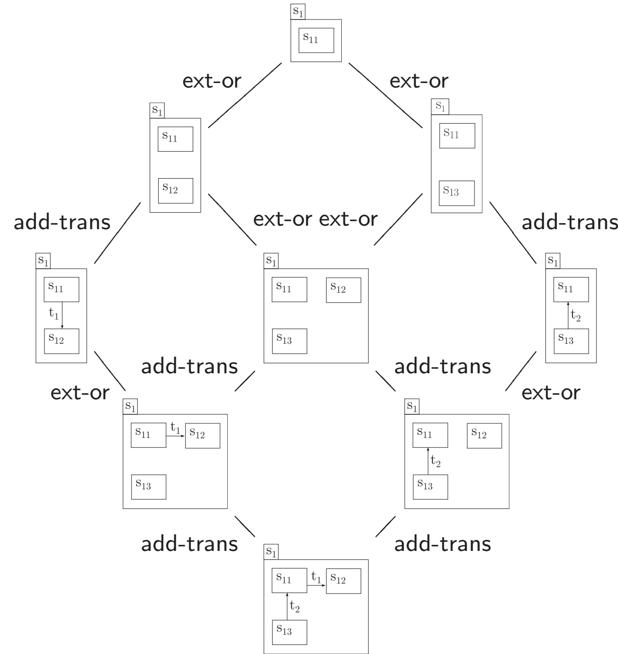


Figure 4.4: Example of the computation of the union

In figure 4.4 the union is computed for two statecharts that are extensions of $[\hat{s}_1, ([\hat{s}_{11}]]), 1, \emptyset, ((), ())]$:

$$r_1 = [\hat{s}_1, ([\hat{s}_{11}], [\hat{s}_{12}]), 1, \{\langle \hat{t_1}, \hat{s}_{11}, \emptyset, e, (), \emptyset, \hat{s}_{12}, \mathsf{none} \rangle\}, ((), ())]$$

and

$$r_2 = [\hat{s}_1, ([\hat{s}_{11}], [\hat{s}_{13}]), 1, \{\langle \hat{t_2}, \hat{s}_{11}, \emptyset, e, (), \emptyset, \hat{s}_{13}, \mathsf{none} \rangle\}, ((), ())]$$

Chapter 5

Variability Modeling

In this chapter, the main concepts and definitions for formal variability modeling are presented. First, we define a formal syntax of Feature Diagrams. We use Feature Diagrams are used in combination with UML Statecharts in order to represent the common and variant functionalities of a family of products. Then, in order to define a particular product of a family, the condept of configuration is introduced. Finally, in order to define the behavior of a whole family, the effect that each feature has on the product in which it is present is determined, raising the definition of UML Statecharts with variabilities.

5.1 Feature Diagrams

Feature Diagrams are used to document features. A feature is a property of a system that directly affects end users, which can be either human or other systems. The main purpose of a Feature Diagram is to define concisely the legal configurations (generally called products) of some (usually software) artefact. In the case of software product lines, the main goal of a feature diagram is to specify commonalities and differences amongst the products of the line. In this context, a feature is a distinctive characteristic of a product.

Czarnecki et al [Cza98, CHE05a] propose three types of features, namely *mandatory*, *optional* and *alternative*. Additionally, there is a *consists of* relation among features, meaning that a feature comprises one or more other features. In this work, the composed feature is called *parent* feature and its components are called *children* or *subfeatures* of the parent feature. Moreover, a set of constraints over features can be defined. A constraint is a proposition over the set of features.

In order to define a particular product of a line a feature diagram can be *configured*, by choosing which features are present in the product, complying with the following interdependency rules: mandatory features must always be present in a product if their parent feature is present, optional features may or may not be present in a product if their parent feature is present, and exactly one of the alternative subfeatures must be present in a product when their parent feature is present.

A Feature Diagram can naturally be represented as a tree, where the nodes represent the features and the arcs represent the *consists of* relation between them.

5.1.1 Feature Diagrams Syntax

Given $\mathcal{F}$ a set of feature names, a Feature Diagram is defined as a 6-tuple

$$\Upsilon = \langle \mathsf{L},\mathsf{N},\mathsf{N_C},\mathsf{R_M},\mathsf{R_O},\mathsf{R_A} \rangle$$

where: $\mathsf{L} \in \mathcal{F}$, is the product line name (the root of the tree[1]); $\mathsf{N} \subseteq \mathcal{F}$ is the set of features, ($\mathsf{L} \notin \mathsf{N}$); $\mathsf{N_C} \subseteq \mathcal{C}$ is the set of constraints over features, where $\mathcal{C}$ are the propositional calculus formulas with variables $f_i \in \mathcal{F}$ and connectives $\wedge$, $\vee$ and $\neg$; $\mathsf{R_M}, \mathsf{R_O} \subseteq \{\mathsf{L}\} \cup \mathsf{N} \times \mathsf{N}$, are the mandatory and optional *consists of* relations respectively; and $\mathsf{R_A} \subseteq \{\mathsf{L}\} \cup \mathsf{N} \times \mathcal{P}(\mathsf{N})$ is the alternative *consists of* relation. In addition, the union of the relations $\mathsf{R_M}$, $\mathsf{R_O}$ and $\mathsf{R_A}$ must constitute a tree with nodes in $\mathcal{F}$ and root L. $\mathsf{FD}_\mathcal{F}$ is the set of feature diagrams with features in $\mathcal{F}$.

Basic functions on Feature Diagrams

First, the projection functions for Feature Diagrams are defined as: M, O and A: $\mathsf{FD}_\mathcal{F} \to \mathcal{F}$ are the set of mandatory, optional and alternative features, defined respectively as

$$\mathsf{M}(\langle \mathsf{L},\mathsf{N},\mathsf{N_C},\mathsf{R_M},\mathsf{R_O},\mathsf{R_A}\rangle) := \{f \in \mathcal{F} \mid \exists \langle f',f\rangle \in \mathsf{R_M}\},$$

$$\mathsf{O}(\langle \mathsf{L},\mathsf{N},\mathsf{N_C},\mathsf{R_M},\mathsf{R_O},\mathsf{R_A}\rangle) := \{f \in \mathcal{F} \mid \exists \langle f',f\rangle \in \mathsf{R_O}\},$$

$$\mathsf{A}(\langle \mathsf{L},\mathsf{N},\mathsf{N_C},\mathsf{R_M},\mathsf{R_O},\mathsf{R_A}\rangle) := \{f \in \mathcal{F} \mid \exists \langle f',A\rangle \in \mathsf{R_A}.\ f \in A\}.$$

For $\Upsilon \in \mathsf{FD}_\mathcal{F}$, the result of applying this function to Υ will be written as M_Υ, O_Υ and A_Υ, i.e., with the feature diagram argument written as a subscript. The same convention also applies to the projection functions L_Υ, N_Υ, $\mathsf{N_C}_\Upsilon$, $\mathsf{R_M}_\Upsilon$, $\mathsf{R_O}_\Upsilon$ and $\mathsf{R_A}_\Upsilon$ that denote the components of a Feature Diagram $\Upsilon = \langle \mathsf{L},\mathsf{N},\mathsf{N_C},\mathsf{R_M},\mathsf{R_O},\mathsf{R_A}\rangle$.

Also, the following functions and relations are defined:

- $\mathsf{chld} \subseteq \mathsf{FD}_\mathcal{F} \times \mathcal{F} \times \mathcal{F}$ is the child relation:

$$\mathsf{chld}_\Upsilon(f',f) \text{ iff } \langle f,f'\rangle \in \mathsf{R_M}_\Upsilon \cup \mathsf{R_O}_\Upsilon \vee \exists A \subseteq \mathsf{N}_\Upsilon.\ (\langle f,A\rangle \in \mathsf{R_A}_\Upsilon \wedge f' \in A)$$

- fts: $\mathsf{FD}_\mathcal{F} \to \mathcal{P}(\mathcal{F})$ defines the set of features of a given Feature Diagram:

$$\mathsf{fts}_\Upsilon := \mathsf{M}_\Upsilon \cup \mathsf{O}_\Upsilon \cup \mathsf{A}_\Upsilon$$

- $\mathsf{subft} \subseteq \mathsf{FD}_\mathcal{F} \times \mathcal{F} \times \mathcal{F}$ is the transitive closure of the subfeature relation in a Feature Diagram:

$$\mathsf{subft}_\Upsilon(f',f) \text{ iff } \mathsf{chld}_\Upsilon(f',f) \ \vee\ \exists f'' \in \mathsf{N}_\Upsilon.\ (\mathsf{chld}_\Upsilon(f',f'') \wedge \mathsf{subft}_\Upsilon(f'',f))$$

- subfts: $\mathsf{FD}_\mathcal{F} \times \mathcal{F} \to \mathcal{P}(\mathcal{F})$ defines the set of subfeatures of a given feature in a Feature Diagram:

$$\mathsf{subfts}_\Upsilon(f) := \{f' \in \mathsf{fts}_\Upsilon \mid \mathsf{subft}_\Upsilon(f',f)\}$$

- Subfts: $\mathsf{FD}_\mathcal{F} \times \mathcal{P}(\mathcal{F}) \to \mathcal{P}(\mathcal{F})$ defines the set of all the subfeatures of the members of a given set of features in a Feature Diagram, including the set itself:

$$\mathsf{Subfts}_\Upsilon(F) := F \cup \bigcup_{f \in F} \mathsf{subfts}_\Upsilon(f)$$

Graphically, optional features are marked with a black dot, and alternatives features with a line across alternative group.

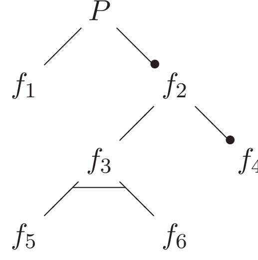
Example

The following diagram
is denoted by

$$\Upsilon_1 = \langle P, \{f_1, f_2, f_3, f_4, f_5, f_6\}, \emptyset, \{\langle P,f_1\rangle, \langle f_2,f_3\rangle\}, \{\langle P,f_2\rangle, \langle f_2,f_4\rangle\}, \{\langle f_3,\{f_5, f_6\}\rangle\}\rangle \in \mathsf{FD}_\mathcal{F},$$

where $\mathcal{F} = \{P, f_1, f_2, f_3, f_4, f_5, f_6\}$.

[1] Following [Cza98], the root of the tree is not a feature but a concept, thus satisfying the condition that every feature has a parent. For the sake of homogeneity, in this work it will be considered as a feature.



5.1.2 Feature Diagram Configurations

Feature Diagrams describe the common and variant functionalities of products in a product line. In order to obtain specific products of a line defined by a Feature Diagram Υ, the possible configurations of Υ are defined as the instances of the tree that are consistent with the relations amongst its features and the constraints of Υ. Configurations are represented as trees of features, where all the features become mandatory. Formally, given a set of feature names $\mathcal{F}$, a configuration of Υ is a 3-tuple

$$\Phi = \langle \mathsf{P},\mathcal{Y},\mathsf{R}\rangle$$

where: $\mathsf{P} \in \mathcal{F}$, is the product name (the root of the tree); $\mathcal{Y} \subseteq \mathcal{F}$, is the set of features of the product ($\mathsf{P} \notin \mathcal{Y}$); and $\mathsf{R} \subseteq \{\mathsf{P}\}\cup\mathcal{Y}\times\mathcal{Y}$ is the *consists of* relation. Additionally, configurations must be trees under the relation R, with root P. $\mathsf{CS}_{\mathcal{F}}$ is the set of configurations in $\mathcal{F}$.

A configuration of a Feature Diagram is determined by the set of optional and alternative features that are selected for the product. The function

$$\mathsf{conf}\colon \mathsf{FD}_{\mathcal{F}} \times \mathcal{P}(\mathcal{F}) \hookrightarrow \mathsf{CS}_{\mathcal{F}},$$

builds a configuration from a Feature Diagram Υ and a set F of chosen features. In fact, for a Feature Diagram $\Upsilon = \langle \mathsf{L},\mathsf{N},\mathsf{N_C},\mathsf{R_M},\mathsf{R_O},\mathsf{R_A}\rangle$ and $C \subseteq \mathcal{F}$, the function $\mathsf{conf}_\Upsilon(C)$ is determined by the elements of C that are optional and alternative features of $\mathcal{F}$.

So, in the definition of conf the features in C that are not in $\mathsf{O}_\Upsilon\cup\mathsf{A}_\Upsilon$ are not taken into account. For the alternative features, exactly one child can be chosen from each parent. So $\mathsf{conf}_\Upsilon(C)$ is defined iff for every chosen feature exactly one of its alternative children is chosen in the result. Additionally, all formulas in $\mathsf{N_C}$ must be satisfied by the configuration.

Basically, the function conf "erases" from Υ all optional and alternative features that are not in C as well as the subtrees that have those features as roots: So, let

$$F' = \mathsf{Subfts}_\Upsilon(\mathsf{O}_\Upsilon\cup\mathsf{A}_\Upsilon\backslash C),$$

$$\mathcal{Y} = \mathsf{M}_\Upsilon\cup\mathsf{O}_\Upsilon\cup\mathsf{A}_\Upsilon\backslash F'$$

and

$$\mathsf{R} = (\mathsf{R_M}_\Upsilon\cup\mathsf{R_O}_\Upsilon\cup\{\langle f,f'\rangle \mid \exists A\subseteq\mathsf{N}.\ \langle f',A\rangle\in\mathsf{R_A}_\Upsilon\wedge f\in A\})\cap(\mathcal{Y}\cup\{P\}\times\mathcal{Y}\cup\{P\})$$

Besides, all the formulae in $\mathsf{N_C}$ must be satisfied for the features present in the configuration. So, if for all $\alpha \in \mathsf{N_C}$, $\mathcal{Y}\cup\mathcal{Y}^\neg \models \alpha$, then

$$\mathsf{conf}_\Upsilon(C){:=}\langle \mathsf{L},\mathcal{Y},\mathsf{R}\rangle$$

otherwise $\mathsf{conf}_\Upsilon(C)$ is undefined (where $\mathcal{Y}^\neg{=}\{\neg f_i \mid f_i{\in}F'\}$).

The set of all possible configurations of a Feature Diagram is given by the function $\mathsf{Confs}\colon \mathsf{FD}_{\mathcal{F}} \to \mathcal{P}(\mathsf{CS}_{\mathcal{F}})$, defined as

$$\mathsf{Confs}_\Upsilon := \bigcup_{C\subseteq\mathsf{fts}_\Upsilon} \{\mathsf{conf}_\Upsilon(C)\in\mathsf{CS}_{\mathcal{F}} \mid \mathsf{conf}_\Upsilon(C) \text{ not undefined}\}$$

Example

For the Υ_1 of the last example, some possible configurations are the following:
$\mathsf{conf}_{\Upsilon_1}(\emptyset) = \langle P,\{f_1\},\{\langle P,f_1\rangle\}\rangle$
$\mathsf{conf}_{\Upsilon_1}(\{f_2, f_5\}) = \langle P, \{f_1, f_2, f_3, f_5\}, \{\langle P,f_1\rangle, \langle P,f_2\rangle, \langle f_2,f_3\rangle, \langle f_3,f_5\rangle\}\rangle$
$\mathsf{conf}_{\Upsilon_1}(\{f_2, f_4, f_6\}) = \langle P, \{f_1, f_2, f_3, f_4, f_6\}, \{\langle P,f_1\rangle, \langle P,f_2\rangle, \langle f_2,f_3\rangle, \langle f_2,f_4\rangle, \langle f_3,f_6\rangle\}\rangle$

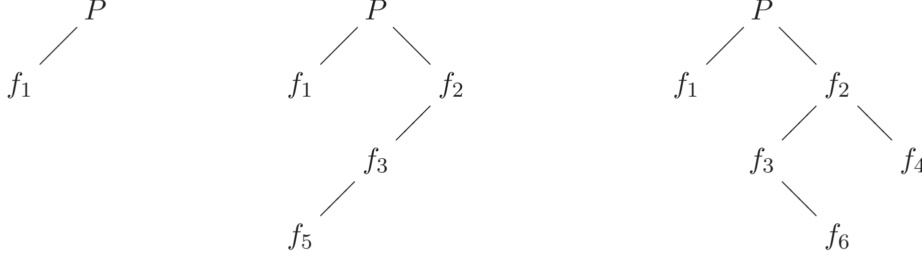


Figure 5.1: FD configurations for Υ_1. Left: $\mathsf{conf}_{\Upsilon_1}(\emptyset)$, Center: $\mathsf{conf}_{\Upsilon_1}(\{f_2, f_5\})$, Right: $\mathsf{conf}_{\Upsilon_1}(\{f_2, f_4, f_6\})$

5.2 UML Statecharts with Variabilities

In section 5.1 Feature Diagrams were introduced as the means to represent the common and variant functionalities of the products of a software product line. Basically, they consist of sets of features organized under certain hierarchy represented as relations. Given a feature diagram Υ, each set of chosen features determines a particular configuration C, which represents the features present in a particular product of the line. Statecharts are used to describe the behavior of a system.

5.2.1 Set of UML Statecharts with Variabilities SC*

In order to define the behavior of a whole product line, it is needed to describe the effect that each feature has on the products in which it is present. For this, the set SC* of statecharts with variabilities is defined:

Definition 2. *Given a Feature Diagram Υ=$\langle L,N,N_C,R_M,R_O,R_A\rangle \in FD$, a SC* for Υ is a function*

$$\Psi\colon N \cup \{L\} \rightarrow SC$$

that associates each feature of Υ with a statechart. In order to guarantee that the hierarchy of features represented by the relations R_M, R_O and R_A is reflected by the statecharts that implement the features, it is further required that:

$$\forall \langle f,f' \rangle \in chld_{\Upsilon}.\ \Psi(f) \prec \Psi(f') \tag{5.1}$$

With restriction 5.1, observe that the image of the set of features fts_{Υ} under Ψ (i.e., $\Psi(\mathsf{fts}_{\Upsilon}) \subseteq \mathcal{P}(\mathsf{SC})$) has the same tree structure as the feature diagram Υ, where the parent-child relation between statecharts is the extension relation $\prec$.

5.2.2 Product Configuration

Given a Feature Diagram Υ=$\langle \mathsf{L,N,N_C,R_M,R_O,R_A} \rangle \in \mathsf{FD}$, a SC* for Υ shows how each feature affects the behavior of the lines. Then, given a configuration $C = \langle \mathsf{P},\mathcal{Y},\mathsf{R} \rangle$ of Υ, in order to obtain the statechart that implements all the features present in $\mathcal{Y}$, the union of all the statecharts in the image of $\mathcal{Y}$ under Ψ (i.e., $\Psi(\mathcal{Y})$) must be taken.

Note that, by the definition of configuration and observation above, if $\langle f,f' \rangle \in \mathsf{chld}_\Upsilon$, then $\Psi(f) \prec \Psi(f')$ and (by the definitions given in 4.2), $\Psi(f)\cup\Psi(f') = \Psi(f')$. So, instead of calculating the union of all the statecharts in $\Psi(\mathcal{Y})$, it is enough to consider the union of the leaves of the tree (i.e., of those features in $\mathcal{Y}$ such that there is no $\langle f,f' \rangle$ in R). It is important to recall that this property greatly simplyfies the calculation of the statechart that implements all the given features.

Taking the example of section 5.1.1, and assuming there exist statecharts $s, s_1, ..s_6$, a possible $\mathsf{SC^*}$ Ψ for Υ_1 could be defined as $\Psi(P){=}s$, $\Psi(f_i){=}s_i$ $(i{=}1,..6)$, and the following relations must hold between the statecharts: $s \prec s_1, s_2$, $s_2 \prec s_3, s_4$, and $s_3 \prec s_5, s_6$ (see figure 5.2).

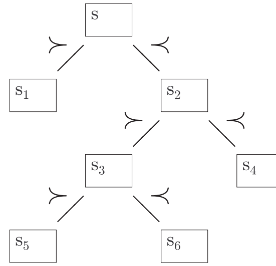


Figure 5.2: SC* example

Then, for the configuration $C_1{=}\mathsf{conf}_{\Upsilon_1}(\{f_2, f_5\})$, $\Psi(C_1)$ is the set $\{s, s_1, s_2, s_3, s_5\}$, with the structure shown in figure 5.3.

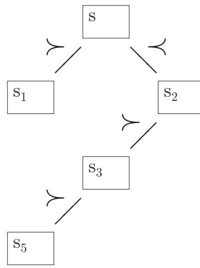


Figure 5.3: Configuration example

The statechart that specifies the product corresponding to the configuration C_1 is obtained by just calculating $s_1 \cup s_5$. A case study can be found on section 8.

Although the union of two statecharts may be overspecified, this is not a problem, because some configurations may produce inconsistent behavior of the system, and it is the designer's responsibility to deal with that fact. To consider this possibility, the following is defined:

Definition 3. *Given* $\Upsilon = \langle L,N,N_C,R_M,R_O,R_A \rangle \in FD$, Ψ *an SC* for* Υ *and a configuration* $C \in Confs_\Upsilon$, Ψ covers C *iff*

$$\bigcup_{f \in fts_C} \Psi(f) \neq \top$$

Definition 4. *Given* $\Upsilon = \langle L, N, N_C, R_M, R_O, R_A \rangle \in FD$ *and* Ψ*, an* SC^* *for* Υ*,* Ψ covers Υ *if* Ψ *covers each possible configuration* C *of* Υ*:*

$$\forall C \in Confs_{\Upsilon}. \bigcup_{f \in fts_C} \Psi(f) \neq \top$$

Finally, a comment regarding the possibility of including the definition of SC* in other FD formalisms found in the literature is given. In [SHT06, SHTB07] the authors define the Free Feature Diagrams (FFDs) as a parametric construction that generalizes the syntax of many FD variants. Each FD approach can be obtained by simply providing values for these parameters. The FFD definition on [SHTB07] can be augmented with the function Ψ: $N \cup r \rightarrow \mathsf{SC}$, where r is the root of the FD and N is the set of nodes[2], which associates each node (feature) of the FD with a statechart. Finally, whenever the restriction 5.1 [3] is included in the definition of valid model for an FD [SHTB07] pg. 466, their work can be extended to include the present approach of SC*.

[2] author's notation
[3] every "child" ("son") node refines its parent

Chapter 6

Behavioral Refinements

In this chapter it is proved that the extension relation can indeed be considered as a refinement, in the sense that it preserves the semantic transitions defined in the SO semantics (section 3.4), obtaining that every reachable semantic state in a statechart is still reachable once the statechart is extended. In order to achieve that, a proof that the extension relation at least preserves the set of all possible configurations of a statechart is given. Once the space of all possible configurations is preserved, a key proof is presented: the function next (which computes the next state after a UML statechart transition is executed) is monotone (isotone or order-preserving) with respect to the extension relation. Moreover, in order to handle the fact that not every extension preserves the semantics, a working definition of *safe* semantic transition is given, in accordance with the UML mechanism for executing transitions. Then, the main result of this work follows: a theorem which states that the extension relation is indeed a behavioral refinement in the sense that the safely extended statecharts preserve all the reachable states of the original one. Finally, a theorem that proves that the set of possible actions generated by a statechart are preserved by the refinement is given.

6.1 Extension Relation and Configurations

In order to prove the main result of this work, that is, that the reachable semantic states of a given UML statechart are preserved by the extension relation, it is needed that the set of all possible configurations of a statechart are maintained. The following lemmas prove that the extension relation preserves the set of all possible configurations of a statechart. Moreover, in this section it will become clear the need of extending the $\mathsf{conf\text{-}all}$ definition in order to include partial parallel configurations.

Lemma 1. *For all* $s, \tilde{s} \in \mathsf{SC}$, *if* $s \prec_1 \tilde{s}$ *then* $\mathsf{conf}(s) \subseteq \mathsf{conf}(\tilde{s})$

Proof. By induction on $s \prec_1 \tilde{s}$. The main idea of the proof is the following. First, assume the form of s, then according to the extension rule applied, deduce the form of $\tilde{s}$. Finally, check that $\mathsf{conf}(s) \subseteq \mathsf{conf}(\tilde{s})$.

- $\mathsf{ext\text{-}and1}$. Assume $s = [\hat{s}]$. Then $[\hat{s}] \prec_1 [\hat{s}, (s')]$, where $s' \in \mathsf{SC}_B$. Finally, $\mathsf{conf}([\hat{s}]) = \{\hat{s}\} \subseteq \mathsf{conf}([\hat{s}, (s')]) = \{\hat{s}, \hat{s}'\}$.
- $\mathsf{ext\text{-}and2}$. Assume $s = [\hat{s}, (s_{1..k})]$. Then $[\hat{s}, (s_{1..k})] \prec_1 [\hat{s}, (s_{1..k}, s')]$, where $s' \in \mathsf{SC}_B$. Finally, $\mathsf{conf}([\hat{s}, (s_{1..k})]) = \{\hat{s}\} \cup \bigcup_{i=1}^{k} \mathsf{conf}(s_i) \subseteq \mathsf{conf}([\hat{s}, (s_{1..k}, s')]) = \{\hat{s}, \hat{s}'\} \cup \bigcup_{i=1}^{k} \mathsf{conf}(s_i)$.
- $\mathsf{inside\text{-}and}$. Assume $s = [\hat{s}, (s_{1..k})]$. Then $[\hat{s}, (s_{1..k})] \prec_1 [\hat{s}, (s_{1..k})_{[s_i/\tilde{s}_i]}]$, where $s_i \prec_1 \tilde{s}_i$. Here the inductive hypothesis is applied, that is, $\mathsf{conf}(s_i) \subseteq \mathsf{conf}(\tilde{s}_i)$ since $s_i \prec_1 \tilde{s}_i$.

Now,

$$\mathsf{conf}([\hat{s},(s_{1..k})]) = \{\hat{s}\} \cup \bigcup_{j=1..i-1,i+1..k} \mathsf{conf}(s_j) \cup \mathsf{conf}(s_i)$$
$$\mathsf{conf}([\hat{s},(s_{1..k})_{[s_i/\tilde{s}_i]}]) = \{\hat{s}\} \cup \bigcup_{j=1..i-1,i+1..k} \mathsf{conf}(s_j) \cup \mathsf{conf}(\tilde{s}_i)$$

Finally, $\mathsf{conf}([\hat{s},(s_{1..k})]) \subseteq \mathsf{conf}([\hat{s},(s_{1..k})_{[s_i/\tilde{s}_i]}])$.

- ext-or1. Assume $s = [\hat{s}]$. Then $[\hat{s}] \prec_1 [\hat{s},(s'),1,1,\emptyset]$, where $s' \in \mathsf{SC}_B$. Finally, $\mathsf{conf}([\hat{s}]) \subseteq \mathsf{conf}([\hat{s},(s'),1,1,\emptyset])$.
- ext-or2. Assume $s = [\hat{s},(s_{1..k}),l,T]$. Then $[\hat{s},(s_{1..k}),l,T] \prec_1 [\hat{s},(s_{1..k},s'),l,T]$, where $s' \in \mathsf{SC}_B$. Finally,

$$\begin{aligned}\mathsf{conf}([\hat{s},(s_{1..k}),l,T]) &= \{\hat{s}\} \cup \mathsf{conf}(s_l)\\ &= \mathsf{conf}([\hat{s},(s_{1..k},s'),l,T])\end{aligned}$$

- inside-or. Assume $s = [\hat{s},(s_{1..k}),l,T]$. Then $[\hat{s},(s_{1..k}),l,T] \prec_1 [\hat{s},(s_{1..k})_{[s_i/\tilde{s}_i]},l,T]$, where $s_i \prec_1 \tilde{s}_i$. Here the inductive hypothesis is applied, that is, $\mathsf{conf}(s_i) \subseteq \mathsf{conf}(\tilde{s}_i)$ since $s_i \prec_1 \tilde{s}_i$. Two cases must be taken into account,
  1. When $i = l$,

  $$\mathsf{conf}([\hat{s},(s_{1..k}),l,T]) = \{\hat{s},\mathsf{conf}(s_i)\}$$
  $$\mathsf{conf}([\hat{s},(s_{1..k})_{[s_i/\tilde{s}_i]},l,T]) = \{\hat{s},\mathsf{conf}(\tilde{s}_i)\}$$

  then $\mathsf{conf}([\hat{s},(s_{1..k}),l,T]) \subseteq \mathsf{conf}([\hat{s},(s_{1..k})_{[s_i/\tilde{s}_i]},l,T])$.
  2. When $i \neq l$,

  $$\begin{aligned}\mathsf{conf}([\hat{s},(s_{1..k}),l,T]) &= \{\hat{s},\mathsf{conf}(s_l)\}\\ &= \mathsf{conf}([\hat{s},(s_{1..k})_{[s_i/\tilde{s}_i]},l,T])\end{aligned}$$

- The cases add-trans, ext-act-en, ext-act-ex and ext-act-trans are trivial because the state structure is not modified, so the current configuration is preserved.

□

Please note that $s \prec_1 \tilde{s}$ not necessarily implies that $\mathsf{conf\text{-}all}(s) \subseteq \mathsf{conf\text{-}all}(\tilde{s})$, because of the rule ext-and2. For example, $s = [\hat{s},(s_1,s_2)]$, where $s_1,s_2 \in \mathsf{SC}_B$. Then, $\mathsf{conf\text{-}all}(s) = \{\{\hat{s}\},\{\hat{s},\hat{s}_1,\hat{s}_2\}\}$. If rule ext-and2 is applied, resulting in $\tilde{s} = [\hat{s},(s_1,s_2,s')]$, where $s' \in \mathsf{SC}_B$, clearly $\mathsf{conf\text{-}all}(\tilde{s}) = \{\{\hat{s}\},\{\hat{s},\hat{s}_1,\hat{s}_2,\hat{s}'\}\} \not\subseteq \mathsf{conf\text{-}all}(s)$. In the next lemma, the extended set of configurations ec-all (defined in section 3.2) is needed.

Lemma 2. *For all $s,\tilde{s} \in SC$, if $s \prec_1 \tilde{s}$ then $\mathsf{ec\text{-}all}(s) \subseteq \mathsf{ec\text{-}all}(\tilde{s})$*

Proof. By induction on $s \prec_1 \tilde{s}$. The main idea of the proof is the following. First, assume the form of s, then according to the extension rule applied, deduce the form of $\tilde{s}$. Finally, check that $\mathsf{ec\text{-}all}(s) \subseteq \mathsf{ec\text{-}all}(\tilde{s})$.

- ext-and1. Assume $s = [\hat{s}]$. Then $[\hat{s}] \prec_1 [\hat{s},(s')]$, where $s' \in \mathsf{SC}_B$. Finally, $\mathsf{ec\text{-}all}([\hat{s}]) = \{\{\hat{s}\}\} \subseteq \mathsf{ec\text{-}all}([\hat{s},(s')]) = \{\{\hat{s},\hat{s}'\},\{\hat{s}\}\}$

- ext-and2. Assume $s = [\hat{s}, (s_{1..k})]$. Then $[\hat{s}, (s_{1..k})] \prec_1 [\hat{s}, (s_{1..k}, s')]$, where $s' \in \mathsf{SC}_B$. Finally,

$$\begin{aligned}
\text{ec-all}([\hat{s}, (s_{1..k})]) &= \{\{\hat{s}\} \cup \bigcup_{i \subseteq \{1,..,k\}} c_i \mid c_i \in \text{ec-all}(s_i)\} \cup \{\{\hat{s}\}\} \\
\text{ec-all}([\hat{s}, (s_{1..k}, s')]) &= \{\{\hat{s}\} \cup \bigcup_{i \subseteq \{1,..,n\}} c_i \mid c_i \in \text{ec-all}(s_i)\} \cup \{\{\hat{s}\}\} \\
&\cup \{\{\hat{s}\} \cup \bigcup_{i \subseteq \{1,..,n\}} c_i \cup \{\hat{s}'\} \mid c_i \in \text{ec-all}(s_i)\}
\end{aligned}$$

  that is, by definition it is possible to split the set of configurations into two: one, those in which s' is not "selected" and the other, those in which s' is indeed selected.

- inside-and. Assume $s = [\hat{s}, (s_{1..k})]$. Then $[\hat{s}, (s_{1..k})] \prec_1 [\hat{s}, (s_{1..k})_{[s_i/\tilde{s}_i]}]$, where $s_i \prec_1 \tilde{s}_i$. Here the inductive hypothesis is used, that is, $\text{ec-all}(s_i) \subseteq \text{ec-all}(\tilde{s}_i)$, since $s_i \prec_1 \tilde{s}_i$. Now,

$$\begin{aligned}
\text{ec-all}([\hat{s}, (s_{1..k})]) &= \{\{\hat{s}\} \cup \bigcup_{j \subseteq \{1,..,n\}} c_j \mid c_j \in \text{ec-all}(s_j)\} \cup \{\{\hat{s}\}\} \\
&= \{\{\hat{s} \cup \bigcup_{j \subseteq (\{1,..,k\} - \{i\})} c_j \cup c \mid c_j \in \text{ec-all}(s_j), c \in \text{ec-all}(s_i)\}\} \\
\text{ec-all}([\hat{s}, (s_{1..k})_{[s_i/\tilde{s}_i]}]) &= \{\{\hat{s}\} \cup \bigcup_{j \subseteq \{1,..,n\}} c_j \mid c_j \in \text{ec-all}(\tilde{s}_j)\} \cup \{\{\hat{s}\}\} \\
&= \{\{\hat{s} \cup \bigcup_{j \subseteq (\{1,..,k\} - \{i\}} c_j \cup c \mid c_j \in \text{ec-all}(s_j), c \in \text{ec-all}(s_i)\}\} \\
&\cup \{\{\hat{s}\} \cup \bigcup_{j \subseteq (\{1,..,k\} - \{i\})} c_j \cup c' \mid c_j \in \text{ec-all}(s_j), c' \in (\text{ec-all}(\tilde{s}_i) - \text{ec-all}(s_i))\} \\
&\cup \{\{\hat{s}\}\}
\end{aligned}$$

  Finally, $\text{ec-all}([\hat{s}, (s_{1..k})]) \subseteq \text{ec-all}([\hat{s}, (s_{1..k})_{[s_i/\tilde{s}_i]}])$.

- ext-or1. Assume $s = [\hat{s}]$. Then $[\hat{s}] \prec_1 [\hat{s}, (s'), 1, 1, \emptyset]$, where $s' \in \mathsf{SC}_B$. Finally,

$$\begin{aligned}
\text{ec-all}([\hat{s}]) &= \{\{\hat{s}\}\} \\
\text{ec-all}([\hat{s}, (s'), 1, 1, \emptyset]) &= \{\{\hat{s}, \hat{s}'\}, \{\hat{s}\}\}, \text{ since conf-all}(s') = \{\{\hat{s}'\}\}
\end{aligned}$$

- ext-or2. Assume $s = [\hat{s}, (s_{1..k}), l, T]$. Then $[\hat{s}, (s_{1..k}), l, T] \prec_1 [\hat{s}, (s_{1..k}, s'), l, T]$, where $s' \in \mathsf{SC}_B$. Finally,

$$\begin{aligned}
\text{ec-all}([\hat{s}, (s_{1..k}), l, T]) &= \{\{\hat{s}\} \cup c \mid \exists i \in \{1..k\}. c \in \text{ec-all}(s_i)\} \cup \{\{\hat{s}\}\} \\
&= \{\{\hat{s}\} \cup c \mid c \in \text{ec-all}(s_1)\} \cup \ldots \cup \{\{\hat{s}\} \cup c \mid c \in \text{ec-all}(s_k)\} \\
&\cup \{\{\hat{s}\}\} \\
\text{ec-all}([\hat{s}, (s_{1..k}, s'), l, T]) &= \{\{\hat{s}\} \cup c \mid \exists i \in \{1..k\}. c \in \text{ec-all}(s_i) \vee c \in \text{ec-all}(s')\} \cup \{\{\hat{s}\}\} \\
&\text{now using the fact that ec-all}(s') = \{\{\hat{s}'\}\}, \\
&= \{\{\hat{s}\} \cup c \mid c \in \text{ec-all}(s_1)\} \cup \ldots \cup \{\{\hat{s}\} \cup c \mid c \in \text{ec-all}(s_k)\} \\
&\cup \ \{\{\hat{s}, \hat{s}'\}\} \cup \{\{\hat{s}\}\}
\end{aligned}$$

  so $\text{ec-all}([\hat{s}, (s_{1..k}), l, T]) \subseteq \text{ec-all}([\hat{s}, (s_{1..k}, s'), l, T])$.

- inside-or. Assume $s = [\hat{s}, (s_{1..k}), l, T]$. Then $[\hat{s}, (s_{1..k}), l, T] \prec_1 [\hat{s}, (s_{1..k})_{[s_i/\tilde{s}_i]}, l, T]$, where $s_i \prec_1 \tilde{s}_i$. Here the inductive hypothesis is used, that is, $\text{ec-all}(s_i) \subseteq \text{ec-all}(\tilde{s}_i)$, since

$s_i \prec_1 \tilde{s}_i$. Now,

$$\begin{aligned}\text{ec-all}([\hat{s},(s_{1..k}),l,T]) &= \{\{\hat{s}\} \cup c | c \in \text{ec-all}(s_1)\} \cup \ldots \cup \{\{\hat{s}\} \cup c | c \in \text{ec-all}(s_i)\} \\ &\cup \ldots \cup \{\{\hat{s}\} \cup c | c \in \text{ec-all}(s_k)\} \cup \{\{\hat{s}\}\} \\ \text{ec-all}([\hat{s},(s_{1..k})_{[s_i/\tilde{s}_i]},l,T]) &= \{\{\hat{s}\} \cup c | c \in \text{ec-all}(s_1)\} \cup \ldots \cup \{\{\hat{s}\} \cup c | c \in \text{ec-all}(\tilde{s}_i)\} \\ &\cup \ldots \cup \{\{\hat{s}\} \cup c | c \in \text{ec-all}(s_k)\} \cup \{\{\hat{s}\}\}\end{aligned}$$

Finally, $\text{ec-all}([\hat{s},(s_{1..k}),l,T]) \subseteq \text{ec-all}([\hat{s},(s_{1..k})_{[s_i/\tilde{s}_i]},l,T])$.

- The cases add-trans, ext-act-en, ext-act-ex and ext-act-trans are trivial because the state structure is not modified, so the set of possible configurations is the same.

□

6.1.1 Monotonicity of next Function

In this section, a proof that the function next applied to a extended state computes a state which indeed is an extension of the next state computed before the extension is given. That is, the function next is monotone with respect to the extension relation. This result is proven in lemma 5. In order to achieve that, two additional lemmas, showing that the functions def and next-stop are also monotone (lemmas 3 and 4 respectively), are presented.

Lemma 3. *For all $s, \tilde{s} \in SC$, if $s \prec_1 \tilde{s}$ then $\mathit{def}(s) \prec_1 \mathit{def}(\tilde{s})$*

Proof. By induction on $s \prec_1 \tilde{s}$. The main idea of the proof is the following. First, assume the form of s, then according to the extension rule applied, deduce the form of $\tilde{s}$. Then, compute $\text{def}(s)$ and $\text{def}(\tilde{s})$ and check that $\text{def}(s) \prec_1 \text{def}(\tilde{s})$.

- ext-and1. Assume $s = [\hat{s}]$. Then $[\hat{s}] \prec_1 [\hat{s},(s')]$, where $s' \in \text{SC}_B$. Now $\text{def}([\hat{s},(s')]) = [\hat{s},(\text{def}(s'))] = [\hat{s},(s')]$, since $s' \in \text{SC}_B$, which indeed is a one step refinement through ext-and1 of $\text{def}([\hat{s}]) = [\hat{s}]$.

- ext-and2. Assume $s = [\hat{s},(s_1,..,s_k)]$. Then $[\hat{s},(s_1,..,s_k)] \prec_1 [\hat{s},(s_1,..,s_k,s')]$, where $s' \in \text{SC}_B$. Now $\text{def}([\hat{s},(s_1,..,s_k,s')]) = [\hat{s},(\text{def}(s_1),..,\text{def}(s_k),\text{def}(s'))] = [\hat{s},(\text{def}(s_1),..,\text{def}(s_k),s')]$, since $s' \in \text{SC}_B$, which indeed is a one step refinement through ext-and2 of $\text{def}([\hat{s},(s_1,..,s_k)]) = [\hat{s},(\text{def}(s_1),..,\text{def}(s_k))]$.

- inside-and. Assume $s = [\hat{s},(s_{1..k})]$. Then $[\hat{s},(s_{1..k})] \prec_1 [\hat{s},(s_{1..k})_{[s_i/\tilde{s}_i]}]$, where $s_i \prec_1 \tilde{s}_i$. Here the inductive hypothesis is applied, that is, $\text{def}(s_i) \prec_1 \text{def}(\tilde{s}_i)$ since $s_i \prec_1 \tilde{s}_i$. Then, $\text{def}([\hat{s},(s_{1..k})_{[s_i/\tilde{s}_i]}]) = [\hat{s},(\text{def}(s_1),..,\text{def}(\tilde{s}_i),..\text{def}(s_k))]$, which indeed is a one step refinement through inside-and of $\text{def}([\hat{s},(s_{1..k})]) = [\hat{s},(\text{def}(s_1),..,\text{def}(s_i),..,\text{def}(s_k))]$ by the inductive hypothesis.

- ext-or1. Assume $s = [\hat{s}]$. Then $[\hat{s}] \prec_1 [\hat{s},(s'),1,1,\emptyset]$, where $s' \in \text{SC}_B$. Now $\text{def}([\hat{s},(s'),1,1,\emptyset]) = [\hat{s},(s')_{[s'/\text{def}(s')]},1,1,\emptyset] = [\hat{s},(s'),1,1,\emptyset]$ since $s' \in \text{SC}_B$, which indeed is a one step refinement of $\text{def}([\hat{s}]) = [\hat{s}]$ through ext-or1.

- ext-or2. Assume $s = [\hat{s},(s_{1..k}),d,l,T]$. Then $[\hat{s},(s_{1..k}),d,l,T] \prec_1 [\hat{s},(s_{1..k},s'),d,l,T]$, where $s' \in \text{SC}_B$. Now $\text{def}([\hat{s},(s_{1..k},s'),d,l,T]) = [\hat{s},(s_{1..k},s')_{[s_d/\text{def}(s_d)]},d,d,T]$, which indeed is a one step refinement of

  $\text{def}([\hat{s},(s_{1..k}),d,l,T]) = [\hat{s},(s_{1..k})_{[s_d/\text{def}(s_d)]},d,d,T]$.

- inside-or. Assume $s = [\hat{s},(s_{1..k}),l,T]$. Then $[\hat{s},(s_{1..k}),l,T] \prec_1 [\hat{s},(s_{1..k})_{[s_i/\tilde{s}_i]},l,T]$, where $s_i \prec_1 \tilde{s}_i$. Here the inductive hypothesis is applied, that is, $\text{def}(s_i) \prec_1 \text{def}(\tilde{s}_i)$, since $s_i \prec_1 \tilde{s}_i$. Two cases must be taken into account,

1. If $i \neq d$, then $\mathsf{def}([\hat{s},(s_{1..k})_{[s_i/\tilde{s}_i]},l,T]) = [\hat{s},(s_{1..k})_{[s_i/\tilde{s}_i][s_d/\mathsf{def}(s_d)]},d,T]$ $= [\hat{s},(s_{1..k})_{[s_d/\mathsf{def}(s_d)][s_i/\tilde{s}_i]},d,T]$, which indeed is a one step refinement of $\mathsf{def}([\hat{s},(s_{1..k}),l,T]) = [\hat{s},(s_{1..k})_{[s_d/\mathsf{def}(s_d)]},d,T]$.
2. If $i = d$, then $\mathsf{def}([\hat{s},(s_{1..k})_{[s_i/\tilde{s}_i]},l,T]) = [\hat{s},(s_{1..k})_{[s_d/\tilde{s}_d][\tilde{s}_d/\mathsf{def}(\tilde{s}_d)]},d,T]$ $= [\hat{s},(s_{1..k})_{[s_d/\mathsf{def}(\tilde{s}_d)]},d,T]$, which indeed is a one step refinement of $\mathsf{def}([\hat{s},(s_{1..k})_{[s_d/\mathsf{def}(s_d)]},d,T])$ by the inductive hypothesis.

- Extensions add-trans, ext-act-en, ext-act-ex and ext-act-trans are trivial because def does not depend on the set of transitions T nor exit nor entry actions.

□

Lemma 4. *For all $s, \tilde{s} \in SC$, if $s \prec_1 \tilde{s}$ then next-stop$(ht, s) \prec_1$ next-stop$(ht, \tilde{s})$, $\forall ht \in HT$*

Proof. By induction on $s \prec_1 \tilde{s}$. The main idea of the proof is analogous to the previous one. Please note that $\tilde{s} \in \mathsf{SC}_O$, because $s \in \mathsf{SC}_O$ by hypothesis. Then only OR rules apply,

- ext-or2. Assume $s = [\hat{s},(s_{1..k}),d,l,T]$. Then $[\hat{s},(s_{1..k}),d,l,T] \prec_1 [\hat{s},(s_{1..k},s'),d,l,T]$, where $s' \in \mathsf{SC}_B$. In each case, next-stop$(ht,\tilde{s})$ is indeed a one step refinement of next-stop(ht,s), through rule ext-or2, as follows.

$$\mathsf{next\text{-}stop}(ht,[\hat{s},(s_{1..k},s'),d,l,T]) = \begin{cases} [\hat{s},(s_{1..k},s'),d,l,T] & \text{if } ht = \mathsf{deep} \\ [\hat{s},(s_{1..k},s')_{[s_d/\mathsf{def}(s_d)]},d,d,T] & \text{if } ht = \mathsf{none}, d \in \{1..k\} \\ [\hat{s},(s_{1..k},s')_{[s_l/\mathsf{def}(s_l)]},d,l,T] & \text{if } ht = \mathsf{shallow}, l \in \{1..k\} \end{cases}$$

- inside-or. Assume $s = [\hat{s},(s_{1..k}),d,l,T]$. Then $[\hat{s},(s_{1..k}),d,l,T] \prec_1 [\hat{s},(s_{1..k})_{[s_i/\tilde{s}_i]},d,l,T]$, where $s_i \prec_1 \tilde{s}_i$. In each case, next-stop$(ht,\tilde{s})$ is indeed a one step refinement, through rule inside-or, as follows.

$$\mathsf{next\text{-}stop}(ht,[\hat{s},(s_{1..k})_{[s_i/\tilde{s}_i]},d,l,T]) = \begin{cases} [\hat{s},(s_{1..k})_{[s_i/\tilde{s}_i]},d,l,T] & \text{if } ht = \mathsf{deep} \\ [\hat{s},(s_{1..k})_{[s_i/\tilde{s}_i][s_d/\mathsf{def}(s_d)]},d,d,T] & \text{if } ht = \mathsf{none} \\ [\hat{s},(s_{1..k})_{[s_i/\tilde{s}_i][s_l/\mathsf{def}(s_l)]},d,l,T] & \text{if } ht = \mathsf{shallow} \end{cases}$$

The second and third cases require some explanation. In the second one, there are two subcases:
• $i \neq d$. Since $[\hat{s},(s_{1..k})_{[s_i/\tilde{s}_i][s_d/\mathsf{def}(s_d)]},d,d,T] = [\hat{s},(s_{1..k})_{[s_d/\mathsf{def}(s_d)][s_i/\tilde{s}_i]},d,d,T]$, then $[\hat{s},(s_{1..k})_{[s_d/\mathsf{def}(s_d)]},d,d,T] \prec_1 [\hat{s},(s_{1..k})_{[s_d/\mathsf{def}(s_d)][s_i/\tilde{s}_i]},d,d,T]$ through rule inside-or.
• $i = d$. It is easy to see that $[\hat{s},(s_{1..k})_{[s_i/\tilde{s}_i][\tilde{s}_d/\mathsf{def}(\tilde{s}_d)]},d,d,T]$ $= [\hat{s},(s_{1..k})_{[s_d/\tilde{s}_d][\tilde{s}_d/\mathsf{def}(\tilde{s}_d)]},d,d,T] = [\hat{s},(s_{1..k})_{[s_d/\mathsf{def}(\tilde{s}_d)]},d,d,T]$.
Now, by lemma 3, $[\hat{s},(s_{1..k})_{[s_d/\mathsf{def}(s_d)]},d,d,T] \prec_1 [\hat{s},(s_{1..k})_{[s_d/\mathsf{def}(\tilde{s}_d)]},d,d,T]$ through rule inside-or. In the third case, there are two subcases:
• $i \neq l$. Since $[\hat{s},(s_{1..k})_{[s_i/\tilde{s}_i][s_l/\mathsf{def}(s_l)]},l,T] = [\hat{s},(s_{1..k})_{[s_l/\mathsf{def}(s_l)][s_i/\tilde{s}_i]},l,T]$, then $[\hat{s},(s_{1..k})_{[s_l/\mathsf{def}(s_l)]},l,T] \prec_1 [\hat{s},(s_{1..k})_{[s_l/\mathsf{def}(s_l)][s_i/\tilde{s}_i]},l,T]$ through rule inside-or.
• $i = l$. It is easy to see that $[\hat{s},(s_{1..k})_{[s_i/\tilde{s}_i][s_l/\mathsf{def}(s_l)]},l,T]$ $= [\hat{s},(s_{1..k})_{[s_l/\tilde{s}_l][s_l/\mathsf{def}(s_l)]},l,T] = [\hat{s},(s_{1..k})_{[s_l/\mathsf{def}(\tilde{s}_l)]},l,T]$.
Now, by lemma 3, $[\hat{s},(s_{1..k})_{[s_l/\mathsf{def}(s_l)]},l,T] \prec_1 [\hat{s},(s_{1..k})_{[s_l/\mathsf{def}(\tilde{s}_l)]},l,T]$ through rule inside-or.

- add-trans. Assume $s = [\hat{s}, (s_{1..k}), l, T]$. Then, $[\hat{s}, (s_{1..k}), l, T] \prec_1 [\hat{s}, (s_{1..k}), l, T \cup \{t^*\}]$, where $t^* \in \mathsf{T}$. This case is analogous to ext-or2 case, but the refinement is through rule add-trans.

- Extensions ext-act-en, ext-act-ex and ext-act-trans are trivial because next-stop does not depend on the set of transitions T nor exit nor entry actions.

□

Lemma 5. *For all $s, \tilde{s} \in SC$, if $s \prec_1 \tilde{s}$ then $\mathsf{next}(ht, T_d, s) \prec_1 \mathsf{next}(ht, T_d, \tilde{s})$, $\forall\ T_d \in \mathsf{ec\text{-}all}(s)$, $\forall\ ht \in HT$*

Proof. First note that $T_d \in \mathsf{ec\text{-}all}(s) \Rightarrow T_d \in \mathsf{ec\text{-}all}(\tilde{s})$ by lemma 2. Then, use induction on $s \prec_1 \tilde{s}$.

- ext-and1. Assume $s = [\hat{s}]$. Then $[\hat{s}] \prec_1 [\hat{s}, (s')]$, where $s' \in \mathsf{SC}_B$. Now $\mathsf{next}(ht, T_d, [\hat{s}, (s')]) = [\hat{s}, (\mathsf{next}(ht, T_d, s'))] = [\hat{s}, (s')]$ because $s' \in \mathsf{SC}_B$, which indeed is a one step refinement of $\mathsf{next}(ht, T_d, s)$ through ext-and1.

- ext-and2. Assume $s = [\hat{s}, (s_1, .., s_k)]$. Then $[\hat{s}, (s_1, .., s_k)] \prec_1 [\hat{s}, (s_1, .., s_k, s')]$, where $s' \in \mathsf{SC}_B$.
Now $\mathsf{next}(ht, T_d, [\hat{s}, (s_1, .., s_k, s')]) = [\hat{s}, (\mathsf{next}(ht, T_d, s_1), .., \mathsf{next}(ht, T_d, s_k), s')]$ because $s' \in \mathsf{SC}_B$, which indeed is a one step refinement of
$[\hat{s}, (\mathsf{next}(ht, T_d, s_1), .., \mathsf{next}(ht, T_d, s_k))]$ through ext-and2.

- inside-and. Assume $s = [\hat{s}, (s_{1..k})]$. Then $[\hat{s}, (s_{1..k})] \prec_1 [\hat{s}, (s_{1..k})_{[s_i/\tilde{s}_i]}]$, where $s_i \prec_1 \tilde{s}_i$. Here the inductive hypothesis is applied, that is,
$\mathsf{next}(ht, T_d, s_i) \prec_1 \mathsf{next}(ht, T_d, \tilde{s}_i)$, since $s_i \prec_1 \tilde{s}_i$ and lemma 2. Finally,
$[\hat{s}, (\mathsf{next}(ht, T_d, s_1), .., \mathsf{next}(ht, T_d, s_i), .., \mathsf{next}(ht, T_d, s_k))]$
$\prec_1$ $[\hat{s}, (\mathsf{next}(ht, T_d, s_1), .., \mathsf{next}(ht, T_d, \tilde{s}_i), .., \mathsf{next}(ht, T_d, s_k))]$ through inside-and.

- ext-or1. Assume $s = [\hat{s}]$. Then $[\hat{s}] \prec_1 [\hat{s}, (s'), 1, 1, \emptyset]$, where $s' \in \mathsf{SC}_B$. Two cases must be taken into account,

  1. If $\exists \nu \in T_d.\nu = \hat{s}'$, then $\mathsf{next}(ht, T_d, [\hat{s}, (s'), 1, 1, \emptyset]) = [\hat{s}, (s')_{[s'/\mathsf{next}(ht, T_d, s')]}, 1, 1, \emptyset]$ $= [\hat{s}, (s'), 1, 1, \emptyset]$ because $s' \in \mathsf{SC}_B$, which indeed is a one step refinement of s through ext-or1.

  2. Otherwise, $\mathsf{next}(ht, T_d, [\hat{s}]) = \mathsf{next\text{-}stop}(ht, [\hat{s}])$. Then, by lemma 4,
  $\mathsf{next}(ht, T_d, [\hat{s}]) \prec_1$
  $\mathsf{next}(ht, T_d, [\hat{s}, (s'), 1, 1, \emptyset])$.

- ext-or2. Assume $s = [\hat{s}, (s_{1..k}), l, T]$. Then $[\hat{s}, (s_{1..k}), l, T] \prec_1 [\hat{s}, (s_{1..k}, s'), l, T]$, where $s' \in \mathsf{SC}_B$. Two cases must be taken into account,

  1. If $\exists \nu \in T_d.\nu = \hat{s}_j$, then $\mathsf{next}(ht, T_d, [\hat{s}, (s_{1..k}, s'), l, T])$
  $= [\hat{s}, (s_{1..k}, s')_{[s_j/\mathsf{next}(ht, T_d, s_j)]}, j, T]$, and since $j \in \{1..k\}$, it is a one step refinement of $[\hat{s}, (s_{1..k})_{[s_j/\mathsf{next}(ht, T_d, s_j)]}, j, T]$ through ext-or2.

  2. Otherwise, $\mathsf{next}(ht, T_d, [\hat{s}, (s_{1..k}), l, T]) = \mathsf{next\text{-}stop}(ht, [\hat{s}, (s_{1..k}), l, T])$. Then by lemma 4, $\mathsf{next}(ht, T_d, [\hat{s}, (s_{1..k}), l, T]) \prec_1 \mathsf{next}(ht, T_d, [\hat{s}, (s_{1..k}, s'), l, T])$.

- inside-or. Assume $s = [\hat{s}, (s_{1..k}), l, T]$. Then $[\hat{s}, (s_{1..k}), l, T] \prec_1 [\hat{s}, (s_{1..k})_{[s_i/\tilde{s}_i]}, l, T]$, where $s_i \prec_1 \tilde{s}_i$. Here the inductive hypothesis is applied, that is,
$\mathsf{next}(ht, T_d, s_i) \prec_1 \mathsf{next}(ht, T_d, \tilde{s}_i)$, since $s_i \prec_1 \tilde{s}_i$. Two cases must be taken into account,
1. If $\exists \nu \in T_d.\nu = \hat{s}_j$, then two subcases must be taken into account:

  (a) If $i \neq j$, then $\mathsf{next}(ht, T_d, [\hat{s}, (s_{1..k})_{[s_i/\tilde{s}_i]}, l, T]) = [\hat{s}, (s_{1..k})_{[s_i/\tilde{s}_i][s_j/\mathsf{next}(ht, T_d, s_j)]}, j, T]$
  $= [\hat{s}, (s_{1..k})_{[s_j/\mathsf{next}(ht, T_d, s_j)][s_i/\tilde{s}_i]}, j, T]$, which is a one step refinement of
  $[\hat{s}, (s_{1..k})_{[s_j/\mathsf{next}(ht, T_d, s_j)]}, j, T]$ through inside-or.

(b) If $i = j$, then $\mathsf{next}(ht, T_d, [\hat{s}, (s_{1..k})_{[s_i/\tilde{s}_i]}, l, T]) = [\hat{s}, (s_{1..k})_{[s_i/\tilde{s}_i][s_j/\mathsf{next}(ht,T_d,s_j)]}, j, T]$ $= [\hat{s}, (s_{1..k})_{[s_j/\tilde{s}_j][\tilde{s}_j/\mathsf{next}(ht,T_d,\tilde{s}_j)]}, j, T] = [\hat{s}, (s_{1..k})_{[s_j/\mathsf{next}(ht,T_d,\tilde{s}_j)]}, j, T]$. Then, by inductive hypothesis, $[\hat{s}, (s_{1..k})_{[s_j/\mathsf{next}(ht,T_d,\tilde{s}_j)]}, j, T]$ is a one step refinement of $[\hat{s}, (s_{1..k})_{[s_j/\mathsf{next}(ht,T_d,s_j)]}, j, T]$.

2. Otherwise, $\mathsf{next}(ht, T_d, [\hat{s}, (s_{1..k}), l, T]) = \mathsf{next\text{-}stop}(ht, [\hat{s}, (s_{1..k}), l, T])$. Then by lemma 4, $\mathsf{next}(ht, T_d, [\hat{s}, (s_{1..k}), l, T]) \prec_1 \mathsf{next}(ht, T_d, [\hat{s}, (s_{1..k})_{[s_i/\tilde{s}_i]}, l, T])$.

- Extensions add-trans, ext-act-en, ext-act-ex and ext-act-trans are trivial because next does not depend on the set of transitions T nor exit nor entry actions.

□

6.2 Extension Relation as a Behavioral Refinement

Due to the priority mechanism for transitions specified by UML, it cannot be expected that every extension preserves the semantics. The conflict arises when an inner transition with the same triggering event as an existing outer transition is added to a statechart. As the inner transition has priority over the outer one, the semantics are not preserved, since the outer transition will not take place in the extended statechart. For that reason, we must introduce the concept of safe extension: an extension is defined to be *safe* if no inner transitions are added with the same event as an existing outer transition. Formally:

Definition 5. *Let $s = [\hat{s}, (s_{1..k}), l, T], \tilde{s} = [\hat{s}, (s_{1..k})_{[s_i/\tilde{s}_i]}, l, T]$, $s_i \prec_1 \tilde{s}_i$. $s \prec_1 \tilde{s}$ is a* ***safe extension*** *iff*

$$\forall e \in \mathsf{E}: \ (\exists s' \in \mathsf{SC}.\ (s \xrightarrow{e}{}^{1} s' \wedge s_i \not\xrightarrow{e}{}^{1}\ ) \Rightarrow \tilde{s}_i \not\xrightarrow{e}{}^{1}\ )$$

From now on, all extensions are required to be safe. The main result of this work follows:

Theorem 1. *For all $s, s', t \in \mathsf{SC}, e \in \mathsf{E}$, if $s \xrightarrow{e}{}^{1} s'$ and $s \prec_1 t$, then $\exists t'$ such that $t \xrightarrow{e}{}^{1} t'$ and $s' \prec_1 t'$.*

This theorem states that the extension relation is indeed a behavioral refinement, since the extended statechart preserves all the reachable states of the original one. Graphically,

$$\begin{array}{ccc} s & \xrightarrow{e}{}^{1} & s' \\ \curlyvee_1 & & \curlyvee_1 \\ t & \xrightarrow{e}{}^{1} & \exists\, t' \end{array}$$

Note that the theorem assumes that the stuttering flag is equal to 1. Since a statechart transition takes place when the stuttering flag is equal to 1, idle steps (f=0) are not taken into account. As mentioned previously, the stuttering flag is needed to assure that idle steps can only occur if no non-idle step is possible. Basically, it allows the semantics to fullfill the maximality condition of statecharts, since when no statechart transition can be taken, a stuttering step (loop) can be done. It is important to remark again that statechart transitions are different from semantic transitions.

Proof of theorem 1. By induction on $s \prec_1 t$.

- ext-and1. Let $[\hat{s}] \prec_1 [\hat{s}, (\tilde{s})]$, where $\tilde{s} \in \mathsf{SC}_B$. Then $[\hat{s}] \xrightarrow{e}{}^{0} [\hat{s}]$ because the only rule that can be applied to a BASIC state is BAS. Then, the hypothesis of the theorem does not hold, because there is a transition with flag equal to 0.

- ext-and2. Let $[\hat{s},(s_1,..,s_k)] \prec_1 [\hat{s},(s_1,..,s_k,\tilde{s})]$, where $\tilde{s} \in \mathsf{SC}_B$ and assume that there is at least one substate that performs a transition with $f = 1$ (otherwise the hypothesis of the theorem does not hold), that is, $\exists\ j.f_j = 1$:

$$\frac{\forall_{j\in\{1,\ldots,k\}}.s_j \overset{e}{\to}^{f_j} s'_j}{[\hat{s},(s_{1..k})] \overset{e}{\to}^{1} [\hat{s},(s'_{1..k})] = s'}\ \mathsf{AND}$$

Since $\tilde{s} \in \mathsf{SC}_B$, then $[\tilde{s}] \overset{e}{\to}^{0} [\tilde{s}]$. Now

$$\frac{\forall_{j\in\{1,\ldots,k\}}.s_j \overset{e}{\to}^{f_j} s'_j \quad \tilde{s} \overset{e}{\to}^{0} \tilde{s}}{[\hat{s},(s_{1..k},\tilde{s})] \overset{e}{\to}^{1} [\hat{s},(s'_{1..k},\tilde{s})]{=}t'}\ \mathsf{AND}$$

Then t' is indeed a refinement of s', by rule ext-and2.

- inside-and. Let $[\hat{s},(s_{1..k})] \prec_1 [\hat{s},(s_{1..k})_{[s_i/\tilde{s}_i]}]$, where $s_i \prec_1 \tilde{s}_i$.
Let $[\hat{s},(s_{1..k})] \overset{e}{\to}^{f} [\hat{s},(s'_{1..k})]{=}s'$ with $s_j \overset{e}{\to}^{f_j} s'_j\ \forall j{=}1,..,k$. By inductive hypothesis: if $s_i \overset{e}{\to}^{1} s'_i$ and $s_i \prec_1 \tilde{s}_i$ then $\exists \tilde{s}'_i$ such that $\tilde{s}_i \overset{e}{\to}^{1} \tilde{s}'_i$ and $s'_i \prec_1 \tilde{s}'_i$. Then,

$$\frac{\forall_{j=1..i-1,i+1..k}.s_j \overset{e}{\to}^{f_j} s'_j, \qquad \dfrac{s_i \overset{e}{\to}^{1} s'_i, \quad s_i \prec_1 \tilde{s}_i}{\tilde{s}_i \overset{e}{\to}^{1} \tilde{s}'_i}\ \mathsf{I.H.}}{t = [\hat{s},(s_{1..k})_{[s_i/\tilde{s}_i]}] \overset{e}{\to}^{1} [\hat{s},(s'_{1..k})_{[s'_i/\tilde{s}'_i]}] = t'}\ \mathsf{AND}$$

And $[\hat{s},(s'_{1..k})] \prec_1 [\hat{s},(s'_{1..k})_{[s'_i/\tilde{s}'_i]}]$ by inside-and.

- ext-or1. Let $[\hat{s}] \prec_1 [\hat{s},(\tilde{s}),1,1,\emptyset]$, where $\tilde{s} \in \mathsf{SC}_B$. Then $[\hat{s}] \overset{e}{\to}^{0} [\hat{s}]$ because the only rule that can be applied to a BASIC state is BAS. Then, the hypothesis of the theorem does not hold.

- ext-or2. Let $[\hat{s},(s_{1..k}),l,T] \prec_1 [\hat{s},(s_{1..k},\tilde{s}),l,T]$, where $\tilde{s}{\in}\mathsf{SC}_B$. Three rules can be applied in this case:
OR-1 rule: Let $[\hat{s},(s_{1..k}),l,T] \overset{e}{\to}^{1} [\hat{s},(s_{1..k})_{[s_p/\mathsf{next}(ht,T_d,s_p)]},p,T]{=}s'$ by OR-1 rule, where $\langle \hat{t}{,}l{,}_{,}e{,}_{,}T_d{,}p{,}ht\rangle \in T$. Since the extension adds a new state, it cannot be the current active one. Then $t' = [\hat{s},(s_{1..k[s_p/\mathsf{next}(ht,T_d,s_p)]},\tilde{s}),t,T]$ is chosen. By rule OR-1, the state t' can be reached from $t = [\hat{s},(s_{1..k},\tilde{s}),l,T]$. That is,

$$\frac{\langle \hat{t}{,}l{,}S_r{,}e{,}\alpha{,}T_d{,}p{,}ht\rangle \in T, \quad S_r \subseteq \mathsf{conf}(s_l), \quad s_l \overset{e}{\not\to}^{1}}{t{=}[\hat{s},(s_{1..k},\tilde{s}),l,T] \overset{e}{\to}^{1} [\hat{s},(s_{1..k[s_p/\mathsf{next}(ht,T_d,s_p)]},\tilde{s}),p,T]{=}t'}\ \mathsf{OR\text{-}1}$$

And $[\hat{s},(s_{1..k})_{[s_p/\mathsf{next}(ht,T_d,s_p)]},p,T] \prec_1 [\hat{s},(s_{1..k[s_p/\mathsf{next}(ht,T_d,s_p)]},\tilde{s}),p,T]$ by ext-or2.
OR-2 rule: Let $[\hat{s},(s_{1..k}),l,T] \overset{e}{\to}^{1} [\hat{s},(s_{1..k})_{[s_l/s'_l]},l,T]{=}s'$, when $s_l \overset{e}{\to}^{1} s'_l$. Then,

$$\frac{s_l \overset{e}{\to}^{1} s'_l}{t{=}[\hat{s},(s_{1..k},\tilde{s}),l,T] \overset{e}{\to}^{1} [\hat{s},(s_{1..k[s_l/s'_l]},\tilde{s}),l,T]{=}t'}\ \mathsf{OR\text{-}2}$$

And $[\hat{s},(s_{1..k})_{[s_l/s'_l]},l,T] \prec_1 [\hat{s},(s_{1..k[s_l/s'_l]},\tilde{s}),l,T]$ by ext-or2.
OR-3 rule: Let $[\hat{s},(s_{1..k}),l,T] \overset{e}{\to}^{0} [\hat{s},(s_{1..k}),l,T]$, when $[\hat{s},(s_{1..k}),l,T] \overset{e}{\not\to}^{1}$. Then, the hypothesis of the theorem does not hold.

- add-trans. Let $[\hat{s},(s_{1..k}),l,T]\prec_1[\hat{s},(s_{1..k}),l,T\cup\{t^*\}]$, where $t^*\in\mathsf{T}$. Again, three rules can be applied:

  OR-1 rule: Let $[\hat{s},(s_{1..k}),l,T]\xrightarrow{e}^1[\hat{s},(s_{1..k})_{[s_m/\mathsf{next}(ht,T_d,s_m)]},m,T]{=}s'$ by OR-1 rule, where $\langle\hat{t},l,_,e,_,T_d,m,ht\rangle\neq t^*\in T$. Then,

$$\frac{\langle\hat{t},l,S_r,e,_,T_d,m,ht\rangle\in T,\quad S_r\subseteq\mathsf{conf}(s_l),\quad s_l\stackrel{e}{\not\to}^1}{t{=}[\hat{s},(s_{1..k}),l,T\cup\{t^*\}]\xrightarrow{e}^1[\hat{s},(s_{1..k})_{[s_m/\mathsf{next}(ht,T_d,s_m)]},m,T\cup\{t^*\}]{=}t'}\ \text{OR-1}$$

  So $[\hat{s},(s_{1..k})_{[s_m/\mathsf{next}(ht,T_d,s_m)]},m,T]\prec_1[\hat{s},(s_{1..k})_{[s_m/\mathsf{next}(ht,T_d,s_m)]},m,T\cup\{t^*\}]$ by add-trans.

  OR-2 rule: Let $[\hat{s},(s_{1..k}),l,T]\xrightarrow{e}^1[\hat{s},(s_{1..k})_{[s_l/s_l']},l,T]{=}s'$, when $s_l\xrightarrow{e}^1 s_l'$. Then,

$$\frac{s_l\xrightarrow{e}^1 s_l'}{t=[\hat{s},(s_{1..k}),l,T\cup\{t^*\}]\xrightarrow{e}^1[\hat{s},(s_{1..k})_{[s_l/s_l']},l,T\cup\{t^*\}]=t'}\ \text{OR-2}$$

  Here $[\hat{s},(s_{1..k})_{[s_l/s_l']},l,T]\prec_1[\hat{s},(s_{1..k})_{[s_l/s_l']},l,T\cup\{t^*\}]$ by add-trans.

  OR-3 rule: Let $[\hat{s},(s_{1..k}),l,T]\xrightarrow{e}^0[\hat{s},(s_{1..k}),l,T]$, when $[\hat{s},(s_{1..k}),l,T]\stackrel{e}{\not\to}^1$. Then, the hypothesis of the theorem does not hold.

- inside-or. Here, $[\hat{s},(s_{1..k}),l,T]\prec_1[\hat{s},(s_{1..k})_{[s_i/\tilde{s}_i]},l,T']$, where $s_i\prec_1\tilde{s}_i$. For the sake of clarity, two cases are considered:

  1. When $i\neq l$.

  OR-1 rule: Let $[\hat{s},(s_{1..k}),l,T]\xrightarrow{e}^1[\hat{s},(s_{1..k})_{[s_m/\mathsf{next}(ht,T_d,s_m)]},m,T]{=}s'$, where $\langle\hat{t},l,_,_,_,T_d,m,ht\rangle\in T$. Here two subcases are taken into account:

  (a) When $i\neq m$.

$$\frac{\langle\hat{t},l,S_r,e,_,T_d,m,ht\rangle\in T,\quad S_r\subseteq\mathsf{conf}(s_l),\quad s_l\stackrel{e}{\not\to}^1}{t{=}[\hat{s},(s_{1..k})_{[s_i/\tilde{s}_i]},l,T]\xrightarrow{e}^1[\hat{s},(s_{1..k})_{[s_i/\tilde{s}_i][s_m/\mathsf{next}(ht,T_d,s_m)]},m,T]{=}t'}\ \text{OR-1}$$

  Here $[\hat{s},(s_{1..k})_{[s_m/\mathsf{next}(ht,T_d,s_m)]},m,T]\prec_1[\hat{s},(s_{1..k})_{[s_i/\tilde{s}_i][s_m/\mathsf{next}(ht,T_d,s_m)]},m,T]$ by inside-or.

  (b) When $i=m$.

$$\frac{\langle\hat{t},l,S_r,e,_,T_d,m,ht\rangle\in T,\quad S_r\subseteq\mathsf{conf}(s_l),\quad s_l\stackrel{e}{\not\to}^1}{t=[\hat{s},(s_{1..k})_{[s_i/\tilde{s}_i]},l,T]\xrightarrow{e}^1[\hat{s},(s_{1..k})_{[s_i/\tilde{s}_i][\tilde{s}_m/\mathsf{next}(ht,T_d,\tilde{s}_m)]},m,T]=t'}\ \text{OR-1}$$

  Here $[\hat{s},(s_{1..k})_{[s_m/\mathsf{next}(ht,T_d,s_m)]},m,T]\prec_1[\hat{s},(s_{1..k})_{[s_i/\tilde{s}_i][\tilde{s}_m/\mathsf{next}(ht,T_d,\tilde{s}_m)]},m,T]$ by inside-or, since $\mathsf{next}(ht,T_d,s_m)\prec_1\mathsf{next}(ht,T_d,\tilde{s}_m)$, by lemma 5.

  OR-2 rule: Let $[\hat{s},(s_{1..k}),l,T]\xrightarrow{e}^1[\hat{s},(s_{1..k})_{[s_l/s_l']},l,T]{=}s'$, when $s_l\xrightarrow{e}^1 s_l'$. Then,

$$\frac{s_l\xrightarrow{e}^1 s_l'}{t=[\hat{s},(s_{1..k})_{[s_i/\tilde{s}_i]},l,T]\xrightarrow{e}^1[\hat{s},(s_{1..k})_{[s_i/\tilde{s}_i][s_l/s_l']},l,T]=t'}\ \text{OR-2}$$

  Since $i{\neq}l$, $[\hat{s},(s_{1..k})_{[s_l/s_l']},l,T]\prec_1[\hat{s},(s_{1..k})_{[s_i/\tilde{s}_i][s_l/s_l']},l,T]$ by inside-or.

  OR-3 rule: The hypothesis of the theorem does not hold.

  2. When $i=l$. Note that if $s_i\stackrel{e}{\not\to}^1$ then $\tilde{s}_i\stackrel{e}{\not\to}^1$, because the extensions are safe. Three OR rules apply:

OR-1 rule: Let $[\hat{s},(s_{1..k}),i,T] \xrightarrow{e}^1 [\hat{s},(s_{1..k})_{[s_m/\mathsf{next}(ht,T_d,s_m)]},m,T]{=}s'$, where $\langle \hat{t},i,S_r,e,_,T_d,m,ht\rangle \in T$. Two subcases must be taken into account:

(a) When $i \neq m$.

$$\frac{\langle \hat{t},i,S_r,e,_,T_d,m,ht\rangle \in T, \quad S_r \subseteq \mathsf{conf}(\tilde{s}_i), \quad \tilde{s}_i \not\xrightarrow{e}^1}{t{=}[\hat{s},(s_{1..k})_{[s_i/\tilde{s}_i]},i,T] \xrightarrow{e}^1 [\hat{s},(s_{1..k})_{[s_i/\tilde{s}_i][s_m/\mathsf{next}(ht,T_d,s_m)]},m,T]{=}t'} \ \textsf{OR-1}$$

And $[\hat{s},(s_{1..k})_{[s_m/\mathsf{next}(ht,T_d,s_m)]},m,T] \prec_1 [\hat{s},(s_{1..k})_{[s_i/\tilde{s}_i][s_m/\mathsf{next}(ht,T_d,s_m)]},m,T]$ by inside-or. Note that $S_r \subseteq \mathsf{conf}(s_i) \Rightarrow S_r \subseteq \mathsf{conf}(\tilde{s}_i)$ holds by lemma 1.

(b) When $i = m$.

$$\frac{\langle \hat{t},i,S_r,e,_,T_d,m,ht\rangle \in T, \quad S_r \subseteq \mathsf{conf}(\tilde{s}_i), \quad \tilde{s}_i \not\xrightarrow{e}^1}{t{=}[\hat{s},(s_{1..k})_{[s_i/\tilde{s}_i]},i,T] \xrightarrow{e}^1 [\hat{s},(s_{1..k})_{[s_i/\tilde{s}_i][\tilde{s}_m/\mathsf{next}(ht,T_d,\tilde{s}_m)]},m,T]{=}t'} \ \textsf{OR-1}$$

And $[\hat{s},(s_{1..k})_{[s_m/\mathsf{next}(ht,T_d,s_m)]},m,T] \prec_1 [\hat{s},(s_{1..k})_{[s_i/\tilde{s}_i][\tilde{s}_m/\mathsf{next}(ht,T_d,\tilde{s}_m)]},m,T]$ by inside-or, since $\mathsf{next}(ht,T_d,s_m) \prec_1 \mathsf{next}(ht,T_d,\tilde{s}_m)$ by lemma 5.

OR-2 rule: Let $[\hat{s},(s_{1..k}),i,T] \xrightarrow{e}^1 [\hat{s},(s_{1..k})_{[s_i/s_i']},i,T]{=}s'$, when $s_l \xrightarrow{e}^1 s_l'$. Then,

$$\frac{\dfrac{s_i \xrightarrow{e}^1 s_i', \quad s_i \prec_1 \tilde{s}_i}{\tilde{s}_i \xrightarrow{e}^1 \tilde{s}_i'} \ \textsf{I.H.}}{t = [\hat{s},(s_{1..k})_{[s_i/\tilde{s}_i]},i,T] \xrightarrow{e}^1 [\hat{s},(s_{1..k})_{[s_i/\tilde{s}_i][\tilde{s}_i/\tilde{s}_i']},i,T] = t'} \ \textsf{OR-2}$$

Here $[\hat{s},(s_{1..k})_{[s_i/s_i']},i,T] \prec_1 [\hat{s},(s_{1..k})_{[s_i/\tilde{s}_i][\tilde{s}_i/\tilde{s}_i']},i,T]$ by inside-or.

OR-3 rule: Then, the hypothesis of the theorem does not hold.

- ext-act-en, ext-act-ex and ext-act-trans are trivial: In those cases, the same refinement applied to s' is applied to s in order to obtain t'.

□

6.3 On the Actions Generated by the SO Semantics

In Theorem 1 it is proved that the extension relation preserves the transitions defined in the SO semantics, and thus it can be considered as a behavioral refinement. The next step is to assure that the set of actions generated by a statechart is preserved in an extended statechart. In this section a theorem is proved, together with two auxiliary lemmas, which take into account the actions generated by the SO semantics. That theorem proves that the set of possible actions generated by a statechart are preserved by the refinement.

Lemma 6. *For all $s, \tilde{s} \in \mathsf{SC}$. if $s \prec_1 \tilde{s}$ then $\forall \alpha \in \mathsf{exit}(s)$. $\exists \tilde{\alpha} \in \mathsf{exit}(\tilde{s})$. $\alpha \lhd \tilde{\alpha}$*

Proof. By induction on $s \prec_1 \tilde{s}$. The main idea of the proof is the following. First, assume the form of s, then according to the extension rule applied, deduce the form of $\tilde{s}$. Then compute the sets $\mathsf{exit}(s)$ and $\mathsf{exit}(\tilde{s})$. Finally, for each element of $\mathsf{exit}(s)$ check that there exists the associated one in $\mathsf{exit}(\tilde{s})$, such that this is a substring of the former one.

- ext-and1. Assume $s = [\hat{s}]$. Then $[\hat{s}] \prec_1 [\hat{s},(s')]$, where $s' \in \mathsf{SC}_B$. Then, $\mathsf{exit}([\hat{s}]) = \{ex\}$ and $\mathsf{exit}([\hat{s},(s')]) = \{ex'{::}ex\}$ where $ex' = \mathsf{exit}(s')$. The thesis holds because $\{ex\}$ is a suffix of $\{ex'{::}ex\}$.

- ext-and2. Assume $s = [\hat{s}, (s_{1..k})]$. Then $[\hat{s}, (s_{1..k})] \prec_1 [\hat{s}, (s_{1..k}, s')] = \tilde{s}$, where $s' \in \mathsf{SC}_B$. So $\mathsf{exit}(s) = \{m_1::..::m_k::ex \mid \exists p:\{1..k\} \leftrightarrow \{1..k\}.m_i \in \mathsf{exit}(s_{p(i)}) \forall_i\}$. Let p_i be one bijecton (permutation) $p_i = p(i)$ such that $\mathsf{exit}(s)_{p_i} = \{m_1::..::m_k::ex \mid m_i \in \mathsf{exit}(s_{p_i}) \forall_i\}$. Now, $\mathsf{exit}(s)_{p_i}$ is the set of all possible combinations of $m_i \in \mathsf{exit}(s_{p_i})$. Now define, $\mathsf{exit}(\tilde{s})_{p_i} = \{m_1::..::m_k::ex'::ex \mid m_i \in \mathsf{exit}(s_{p_i}) \forall_i, ex' = \mathsf{exit}(s')\}$. It is easy to see that for all $\alpha \in \mathsf{exit}(s)_{p_i}$ exists one $\tilde{\alpha} \in \mathsf{exit}(\tilde{s}_{p_i})$ s.t. $\alpha \lhd \tilde{\alpha}$.

- inside-and. Assume $s = [\hat{s}, (s_{1..k})]$. Then $[\hat{s}, (s_{1..k})] \prec_1 [\hat{s}, (s_{1..k})_{[s_i/\tilde{s}_i]}]$, where $s_i \prec_1 \tilde{s}_i$. Then, $\mathsf{exit}(s) = \{m_1::..::m_k::ex \mid \exists p:\{1..k\} \leftrightarrow \{1..k\}.m_i \in \mathsf{exit}(s_{p(i)}) \forall_i\}$. Let p_i be one bijecton (permutation) $p_i = p(i)$ such that $\mathsf{exit}(s)_{p_i} = \{m_1::..::m_k::ex \mid m_i \in \mathsf{exit}(s_{p_i}) \forall_i\}$. Now, $\mathsf{exit}(s)$ is the set of all possible combinations of $m_i \in \mathsf{exit}(s_{p_i})$. Let $p^{-1} = r$ be the inverse of p, such that $r(p_i) = i$. Take $j = p_i$ (note that $r_j = i$) and using the inductive hypothesis, that is, $(\forall m_{r_j} \in \mathsf{exit}(s_j), \exists \tilde{m}_{r_j} \in \mathsf{exit}(\tilde{s}_j))$. $m_{r_j} \lhd \tilde{m}_{r_j}$, the string $\tilde{m}_i$ is obtained. Now, the string $m_1::..::m_i::..::m_k \lhd m_1::..::\tilde{m}_i::..::m_k$ because it is a subsequence. Since the reasoning is valid for any permutation, the thesis holds.

- ext-or1. Assume $s = [\hat{s}]$. Then $[\hat{s}] \prec_1 [\hat{s}, (s'), 1, \emptyset]$, where $s' \in \mathsf{SC}_B$. Finally, $\mathsf{exit}(s) = \{ex\}$ and $\mathsf{exit}(\tilde{s}) = \{ex'::ex \mid ex' \in \mathsf{exit}(s')\} = \{ex'::ex\}$ because $s' \in \mathsf{SC}_B$.

- ext-or2. Assume $s = [\hat{s}, (s_{1..k}), l, T]$. Then $[\hat{s}, (s_{1..k}), l, T] \prec_1 [\hat{s}, (s_{1..k}, s'), l, T]$, where $s' \in \mathsf{SC}_B$. Finally, $\mathsf{exit}(s) = \{ex'::ex \mid ex' \in \mathsf{exit}(s_l)\} = \mathsf{exit}(\tilde{s})$, because $l \in \{1..k\}$.

- inside-or. Assume $s = [\hat{s}, (s_{1..k}), l, T]$. Then $[\hat{s}, (s_{1..k}), l, T] \prec_1 [\hat{s}, (s_{1..k})_{[s_i/\tilde{s}_i]}, l, T]$, where $s_i \prec_1 \tilde{s}_i$. Two cases must be taken into account,
  1. If $i \neq l$. Then, $\mathsf{exit}(s) = \{ex'::ex \mid ex' \in \mathsf{exit}(s_l)\} = \mathsf{exit}(\tilde{s})$.
  2. If $i = l$. Apply the inductive hypothesis, that is, $(\forall m_i \in \mathsf{exit}(s_l), \exists \tilde{m}_l \in \mathsf{exit}(\tilde{s}_l))$. $m_l \lhd \tilde{m}_l$. Then $m_1::..::m_l::..::m_k \lhd m_1::..::\tilde{m}_l::..::m_k$.

- ext-act-ex. Three cases must be taken into account,
  1. Assume $s = [\hat{s}, (en, ex)]$. Then $[\hat{s}, (en, ex)] \prec_1 [\hat{s}, (en, \tilde{ex})]$, where $ex \lhd \tilde{ex}$. Finally, $\mathsf{exit}(s) = \{ex\}$ and $\mathsf{exit}(\tilde{s}) = \{\tilde{ex}\}$.
  2. Assume $s = [\hat{s}, (s_{1..k}), l, T, (en, ex)]$. Then $[\hat{s}, (s_{1..k}), l, T, (en, ex)] \prec_1$ $[\hat{s}, (s_{1..k}), l, T, (en, \tilde{ex})]$, where $ex \lhd \tilde{ex}$. Finally, $\mathsf{exit}(s) = \mathsf{exit}(s) = \{ex'::ex \mid ex' \in \mathsf{exit}(s_l)\}$ and $\mathsf{exit}(\tilde{s}) = \mathsf{exit}(s) = \{ex'::\tilde{ex} \mid ex' \in \mathsf{exit}(s_l)\}$.
  3. Assume $s = [\hat{s}, (s_{1..k}), (en, ex)]$. Then $[\hat{s}, (s_{1..k}), (en, ex)] \prec_1 [\hat{s}, (s_{1..k}), (en, \tilde{ex})]$, where $ex \lhd \tilde{ex}$. Finally, $\mathsf{exit}(s) = \{m_1::..::m_k::ex \mid \exists p:\{1..k\} \leftrightarrow \{1..k\}.m_i \in \mathsf{exit}(s_{p(i)}) \forall_i\}$ and $\mathsf{exit}(\tilde{s}) = \{m_1::..::m_k::\tilde{ex} \mid \exists p:\{1..k\} \leftrightarrow \{1..k\}.m_i \in \mathsf{exit}(s_{p(i)}) \forall_i\}$.

- Extensions add-trans, ext-act-en and ext-act-trans are trivial because exit does not depend on the set of transitions T nor entry actions.

□

Lemma 7. *For all $s, \tilde{s} \in SC$. if $s \prec_1 \tilde{s}$ then $\forall \alpha \in \mathit{entry}(s_i)$. $\exists \tilde{\alpha} \in \mathit{entry}(\tilde{s}'_i)$. $\alpha \lhd \tilde{\alpha}$*

Proof. By induction on $s \prec_1 \tilde{s}$. Analogous to lemma 6. □

Theorem 2. *Let $s, s', t, t' \in SC$ where theorem 1 holds and $s \rightarrow_\alpha s'$, $t \rightarrow_{\alpha'} t'$ then $\alpha \lhd \alpha'$.*

Proof. By induction on $s \prec_1 t$.

- ext-and. Let $[\hat{s}] \prec_1 [\hat{s}, (s_1, .., s_k)]$, where $s_1, .., s_k \in \mathsf{SC}_B$. The only rule that can be applied is BAS, then the hypothesis of the theorem does not hold.

- inside-and. Let $[\hat{s},(s_{1..k})] \prec_1 [\hat{s},(s_{1..k})_{[s_i/\tilde{s}_i]}]$, where $s_i \prec_1 \tilde{s}_i$. Let $[\hat{s},(s_{1..k})] \xrightarrow{e}{}^{f}_{\alpha} [\hat{s},(s'_{1..k})]=s'$ with $s_j \xrightarrow{e}{}^{f_j}_{\alpha_j} s'_j$ $\forall j{=}1,..,k$. By inductive hypothesis: if $s_i \xrightarrow{e}{}^{1}_{\alpha_i} s'_i$ and $s_i \prec_1 \tilde{s}_i$ then $\exists \tilde{s}'_i$ such that $\tilde{s}_i \xrightarrow{e}{}^{1}_{\alpha'_i} \tilde{s}'_i$ and $s'_i \prec_1 \tilde{s}'_i$ where $\alpha_i \lhd \alpha'_i$. Then,

$$\dfrac{\forall_{j=1..i-1,i+1..k}.s_j \xrightarrow{e}{}^{f_j}_{\alpha_j} s'_j, \qquad \dfrac{s_i \xrightarrow{e}{}^{1}_{\alpha_i} s'_i, \quad s_i \prec_1 \tilde{s}_i}{\tilde{s}_i \xrightarrow{e}{}^{1}_{\alpha'_i} \tilde{s}'_i}\ \textsf{I.H.}}{t = [\hat{s},(s_{1..k})_{[s_i/\tilde{s}_i]}] \xrightarrow{e}{}^{1}_{\alpha'} [\hat{s},(s'_{1..k})_{[s'_i/\tilde{s}'_i]}] = t'}\ \textsf{AND}$$

  Given a permutation we get $\alpha = \alpha_{p_1}::..::\alpha_{p_j}::..::\alpha_{p_k}$. Take $p_j = i$, then $\alpha_1::..::\alpha_i::..::\alpha_k \lhd \alpha_1::..::\alpha'_i::..::\alpha_k = \alpha'$.

- ext-or1. Let $[\hat{s}] \prec_1 [\hat{s},(\tilde{s}),1,\emptyset]$, where $\tilde{s} \in \mathsf{SC}_B$. The only rule that can be applied is BAS. Then, the hypothesis of the theorem does not hold.

- ext-or2. Let $[\hat{s},(s_{1..k}),l,T] \prec_1 [\hat{s},(s_{1..k},\tilde{s}),l,T]$, where $\tilde{s} \in \mathsf{SC}_B$. Three cases must be taken into account:

  OR-1 rule: Let $[\hat{s},(s_{1..k}),l,T] \xrightarrow{e}{}^{1}_{ex::\gamma::en} [\hat{s},(s_{1..k})_{[s_p/\mathsf{next}(ht,T_d,s_p)]},p,T]=s'$ by OR-1 rule, where $\langle \hat{t},l,_,e,\gamma,T_d,p,ht \rangle \in T$, $ex \in \mathsf{exit}(s_l)$ and $en \in \mathsf{entry}(\mathsf{next}(ht,T_d,s_p))$. Since the extension adds a new state, it cannot be the current active one. Then,

$$\dfrac{\langle \hat{t},l,S_r,e,\gamma,T_d,p,ht \rangle \in T, \quad S_r \subseteq \mathsf{conf}(s_l), \quad s_l \not\xrightarrow{e}{}^{1}}{t{=}[\hat{s},(s_{1..k},\tilde{s}),l,T] \xrightarrow{e}{}^{1}_{ex::\gamma::en} [\hat{s},(s_{1..k[s_p/\mathsf{next}(ht,T_d,s_p)]},\tilde{s}),p,T]{=}t'}\ \textsf{OR-1}$$

  then the output actions are unchanged.

  OR-2 rule: Let $[\hat{s},(s_{1..k}),l,T] \xrightarrow{e}{}^{1}_{\alpha} [\hat{s},(s_{1..k})_{[s_l/s'_l]},l,T]=s'$, when $s_l \xrightarrow{e}{}^{1}_{\alpha} s'_l$. Then,

$$\dfrac{s_l \xrightarrow{e}{}^{1}_{\alpha} s'_l}{t{=}[\hat{s},(s_{1..k},\tilde{s}),l,T] \xrightarrow{e}{}^{1}_{\alpha} [\hat{s},(s_{1..k[s_l/s'_l]},\tilde{s}),l,T]{=}t'}\ \textsf{OR-2}$$

  then the output actions are unchanged.

  OR-3 rule: Let $[\hat{s},(s_{1..k}),l,T] \xrightarrow{e}{}^{0} [\hat{s},(s_{1..k}),l,T]$, when $[\hat{s},(s_{1..k}),l,T] \not\xrightarrow{e}{}^{1}$. Then, the hypothesis of the theorem does not hold.

- add-trans. Let $[\hat{s},(s_{1..k}),l,T] \prec_1 [\hat{s},(s_{1..k}),l,T \cup \{t^*\}]$, where $t^* \in \mathsf{T}$. Again, three rules can be applied:

  OR-1 rule: Let $[\hat{s},(s_{1..k}),l,T] \xrightarrow{e}{}^{1}_{ex::\gamma::en} [\hat{s},(s_{1..k})_{[s_m/\mathsf{next}(ht,T_d,s_m)]},m,T]=s'$ by OR-1 rule, where $\langle \hat{t},l,_,e,\gamma,T_d,m,ht \rangle \neq t^* \in T$, $ex \in \mathsf{exit}(s_l)$ and $en \in \mathsf{entry}(\mathsf{next}(ht,T_d,s_m))$. Then,

$$\dfrac{\langle \hat{t},l,S_r,e,\gamma,T_d,m,ht \rangle \in T, \quad S_r \subseteq \mathsf{conf}(s_l), \quad s_l \not\xrightarrow{e}{}^{1}}{t{=}[\hat{s},(s_{1..k}),l,T \cup \{t^*\}] \xrightarrow{e}{}^{1}_{ex::\gamma::en} [\hat{s},(s_{1..k})_{[s_m/\mathsf{next}(ht,T_d,s_m)]},m,T \cup \{t^*\}]{=}t'}\ \textsf{OR-1}$$

  then the output actions are unchanged.

  OR-2 rule: Let $[\hat{s},(s_{1..k}),l,T] \xrightarrow{e}{}^{1}_{\alpha} [\hat{s},(s_{1..k})_{[s_l/s'_l]},l,T]=s'$, when $s_l \xrightarrow{e}{}^{1}_{\alpha} s'_l$. Then,

$$\dfrac{s_l \xrightarrow{e}{}^{1}_{\alpha} s'_l}{t{=}[\hat{s},(s_{1..k}),l,T \cup \{t^*\}] \xrightarrow{e}{}^{1}_{\alpha} [\hat{s},(s_{1..k})_{[s_l/s'_l]},l,T \cup \{t^*\}]{=}t'}\ \textsf{OR-2}$$

then the output actions are unchanged.

OR-3 rule: Let $[\hat{s}, (s_{1..k}), l, T] \xrightarrow{e}{}^{0} [\hat{s}, (s_{1..k}), l, T]$, when $[\hat{s}, (s_{1..k}), l, T] \overset{e}{\not\to}{}^{1}$. Then, the hypothesis of the theorem does not hold.

- inside-or. In this case, $[\hat{s}, (s_{1..k}), l, T] \prec_1 [\hat{s}, (s_{1..k})_{[s_i/\tilde{s}_i]}, l, T']$, where $s_i \prec_1 \tilde{s}_i$. For the sake of clarity, the proof is split in two cases:

1. When $i \neq l$.

OR-1 rule: Let $[\hat{s}, (s_{1..k}), l, T] \xrightarrow{e}{}^{1}_{ex::\gamma::en} [\hat{s}, (s_{1..k})_{[s_m/\mathsf{next}(ht,T_d,s_m)]}, m, T] = s'$, where $\langle \hat{t}, l, _, _, \gamma, T_d, m, ht \rangle \in T$, $ex \in \mathsf{exit}(s_l)$ and $en \in \mathsf{entry}(\mathsf{next}(ht, T_d, s_m))$. Two cases must be taken into account:

(a) When $i \neq m$.

$$\frac{\langle \hat{t}, l, S_r, e, \gamma, T_d, m, ht \rangle \in T, \quad S_r \subseteq \mathsf{conf}(s_l), \quad s_l \overset{e}{\not\to}{}^{1}}{t = [\hat{s}, (s_{1..k})_{[s_i/\tilde{s}_i]}, l, T] \xrightarrow{e}{}^{1}_{ex::\gamma::en} [\hat{s}, (s_{1..k})_{[s_i/\tilde{s}_i][s_m/\mathsf{next}(ht,T_d,s_m)]}, m, T] = t'} \text{ OR-1}$$

then the output actions are unchanged.

(b) When $i = m$.

$$\frac{\langle \hat{t}, l, S_r, e, \gamma, T_d, m, ht \rangle \in T, \quad S_r \subseteq \mathsf{conf}(s_l), \quad s_l \overset{e}{\not\to}{}^{1}}{t = [\hat{s}, (s_{1..k})_{[s_i/\tilde{s}_i]}, l, T] \xrightarrow{e}{}^{1}_{ex::\gamma::en'} [\hat{s}, (s_{1..k})_{[s_i/\tilde{s}_i][\tilde{s}_m/\mathsf{next}(ht,T_d,\tilde{s}_m)]}, m, T] = t'} \text{ OR-1}$$

By lemma 7, $en \lhd en'$ since $\mathsf{next}(ht, T_d, s_m) \prec_1 \mathsf{next}(ht, T_d, \tilde{s}_m)$, by lemma 5. Then $ex::\gamma::en \lhd ex::\gamma::en'$.

OR-2 rule: Let $[\hat{s}, (s_{1..k}), l, T] \xrightarrow{e}{}^{1}_{\alpha} [\hat{s}, (s_{1..k})_{[s_l/s'_l]}, l, T] = s'$, when $s_l \xrightarrow{e}{}^{1}_{\alpha} s'_l$. Then,

$$\frac{s_l \xrightarrow{e}{}^{1}_{\alpha} s'_l}{t = [\hat{s}, (s_{1..k})_{[s_i/\tilde{s}_i]}, l, T] \xrightarrow{e}{}^{1}_{\alpha} [\hat{s}, (s_{1..k})_{[s_i/\tilde{s}_i][s_l/s'_l]}, l, T] = t'} \text{ OR-2}$$

then the output actions are unchanged.

OR-3 rule: Then, the hypothesis of the theorem does not hold.

2. When $i = l$. Note that if $s_i \overset{e}{\not\to}{}^{1}$ then $\tilde{s}_i \overset{e}{\not\to}{}^{1}$, because all the extensions are safe. Three OR rules applies:

OR-1 rule: Let $[\hat{s}, (s_{1..k}), i, T] \xrightarrow{e}{}^{1}_{ex::\gamma::en} [\hat{s}, (s_{1..k})_{[s_m/\mathsf{next}(ht,T_d,s_m)]}, m, T] = s'$, where $\langle \hat{t}, i, S_r, e, \gamma, T_d, m, ht \rangle \in T$, $ex \in \mathsf{exit}(s_l)$ and $en \in \mathsf{entry}(\mathsf{next}(ht, T_d, s_m))$. Two sub-cases must be taken into account,

(a) When $i \neq m$.

$$\frac{\langle \hat{t}, i, S_r, e, \gamma, T_d, m, ht \rangle \in T, \quad S_r \subseteq \mathsf{conf}(\tilde{s}_i), \quad \tilde{s}_i \overset{e}{\not\to}{}^{1}}{t = [\hat{s}, (s_{1..k})_{[s_i/\tilde{s}_i]}, i, T] \xrightarrow{e}{}^{1}_{ex'::\gamma::en} [\hat{s}, (s_{1..k})_{[s_i/\tilde{s}_i][s_m/\mathsf{next}(ht,T_d,s_m)]}, m, T] = t'} \text{ OR-1}$$

By lemma 6, $ex \lhd ex'$. Then $ex::\gamma::en \lhd ex'::\gamma::en$. Note that $S_r \subseteq \mathsf{conf}(s_i) \Rightarrow S_r \subseteq \mathsf{conf}(\tilde{s}_i)$ holds by lemma 1.

(b) When $i = m$.

$$\frac{\langle \hat{t}, i, S_r, e, _, T_d, m, ht \rangle \in T, \quad S_r \subseteq \mathsf{conf}(\tilde{s}_i), \quad \tilde{s}_i \overset{e}{\not\to}{}^{1}}{t = [\hat{s}, (s_{1..k})_{[s_i/\tilde{s}_i]}, i, T] \xrightarrow{e}{}^{1}_{ex'::\gamma::en'} [\hat{s}, (s_{1..k})_{[s_i/\tilde{s}_i][\tilde{s}_m/\mathsf{next}(ht,T_d,\tilde{s}_m)]}, m, T] = t'} \text{ OR-1}$$

By lemma 6, $ex \lhd ex'$. By lemma 7, $en \lhd en'$ since $\mathsf{next}(ht, T_d, s_m) \prec_1 \mathsf{next}(ht, T_d, \tilde{s}_m)$, by lemma 5. Then $ex{::}\gamma{::}en \lhd ex'{::}\gamma{::}en'$.

OR-2 rule: Let $[\hat{s}, (s_{1..k}), i, T] \xrightarrow{e}{}^{1}_{\alpha} [\hat{s}, (s_{1..k})_{[s_i/s'_i]}, i, T]{=}s'$, when $s_i \xrightarrow{e}{}^{1}_{\alpha_i} s'_i$. Inductive hypothesis is applied to get,

$$\dfrac{\dfrac{s_i \xrightarrow{e}{}^{1}_{\alpha_i} s'_i, \quad s_i \prec_1 \tilde{s}_i}{\tilde{s}_i \xrightarrow{e}{}^{1}_{\alpha'_i} \tilde{s}'_i}\ \text{I.H.}}{t{=}[\hat{s}, (s_{1..k})_{[s_i/\tilde{s}_i]}, i, T] \xrightarrow{e}{}^{1}_{\alpha'} [\hat{s}, (s_{1..k})_{[s_i/\tilde{s}_i][\tilde{s}_i/\tilde{s}'_i]}, i, T]{=}t'}\ \text{OR-2}$$

Since $\alpha_i \lhd \alpha'_i$ by IH, $\alpha \lhd \alpha'$.

OR-3 rule: Then, the hypothesis of the theorem does not hold.

- ext-act-trans. Let $[\hat{s}, (s_1,..,s_n), T] \prec_1 [\hat{s}, (s_1,..,s_n), T_{[r/r']}]$, where $r = \langle \hat{t}, l, S_r, e, \gamma, T_d, p, ht \rangle \in T$, $r' = \langle \hat{t}, l, S_r, e, \tilde{\gamma}, T_d, p, ht \rangle$ and $\gamma \lhd \tilde{\gamma}$. Three rules can be applied in this case:

  OR-1 rule: Let $[\hat{s}, (s_{1..k}), l, T] \xrightarrow{e}{}^{1}_{ex::\gamma::en} [\hat{s}, (s_{1..k})_{[s_p/\mathsf{next}(ht, T_d, s_p)]}, p, T]{=}s'$ by OR-1 rule, where $\langle \hat{t}, l, _, e, \gamma, T_d, p, ht \rangle \in T$, $ex \in \mathsf{exit}(s_l)$ and $en \in \mathsf{entry}(\mathsf{next}(ht, T_d, s_p))$. Then,

  $$\dfrac{\langle \hat{t}, l, S_r, e, \tilde{\gamma}, T_d, p, ht \rangle \in T, \quad S_r \subseteq \mathsf{conf}(s_l), \quad s_l \not\xrightarrow{e}{}^{1}}{t{=}[\hat{s}, (s_{1..k}, \tilde{s}), l, T] \xrightarrow{e}{}^{1}_{ex::\tilde{\gamma}::en} [\hat{s}, (s_1,..,s_n), T_{[r/r']}]{=}t'}\ \text{OR-1}$$

  then $ex{::}\gamma{::}en \lhd ex{::}\tilde{\gamma}{::}en$.

  OR-2 rule: Let $[\hat{s}, (s_{1..k}), l, T] \xrightarrow{e}{}^{1}_{\alpha} [\hat{s}, (s_{1..k})_{[s_l/s'_l]}, l, T]{=}s'$, when $s_l \xrightarrow{e}{}^{1}_{\alpha} s'_l$. The output actions are unchanged.

  OR-3 rule: Let $[\hat{s}, (s_{1..k}), l, T] \xrightarrow{e}{}^{0} [\hat{s}, (s_{1..k}), l, T]$, when $[\hat{s}, (s_{1..k}), l, T] \not\xrightarrow{e}{}^{1}$. Then, the hypothesis of the theorem does not hold.

- ext-act-en and ext-act-ex. These are trivial, because entry and exit actions of the super-state do not take part in the SOS rules. □

Chapter 7

The SC* Modeler

In this chapter, a prototype implementation of some of the ideas presented in this work is described. The prototype was constructed basically to check the adequacy of the definitions and to become the first step in the construction of a software engineering tool for modeling the behavior of Software Product Lines. This prototype is called “SC* Modeler”. First, a brief description of the modeling approach implied by the ideas of this work is presented. Then, the architecture of the prototype is described. Finally, some details on the inference engine are given.

7.1 Modeling Approach

Formal specifications provide a precise supplement to natural language specifications and can be rigorously validated and verified, leading to the early detection of ambiguities and inconsistencies. Although interest in formal methods captured the attention of many research groups, it has been limited within industry[SB06]. In general, designers are reluctant to the idea of formal methods being used to develop products, and usually UML is preferred instead. One possibility for overcoming this barrier is to use some kind of “transparent formality”. This is one of the goals of the SC* Modeler, in which the designer can model the system using a “point and click” tool, and then these designs can be subsequently refined and even converted into more specific ones, like executable machine code.

A model enable designers to concentrate on the significant system aspects, allowing to handle more complexity through some form of abstraction. One form of abstraction is classification, which is the process of identifying types, also known as concepts. Classification is the basic form of abstraction found in object-oriented modeling [BG05]. Features are “a distinguishable characteristic of a concept (e.g., system, component and so on) that is relevant to some stakeholder of the concept” [SHTB07]. Feature Diagrams (see section 5.1) support this form of abstraction, by describing the capabilities of the product line and defining the constraints for potential products. Among all the possible combinations of features, a Feature Diagram configuration describes the selected features for one particular product, because it no longer contains variability since all configuration decisions have been made. Moreover, it can be checked for conformity with the Feature Diagram. On the other hand, a UML statechart describes the elements used in the behavior implementation of the products. Finally, given a configuration, the SC* determines which statecharts have been selected for a particular product.

This method is based on a stepwise refinement and decomposition of a problem. After the initial specification of behavioral requirements, an abstraction is made in order to capture the most essential behavioral properties of the modeled system. Then, more detail is added to this specification in small steps of two types. First, the specification is decomposed into subsystems through the use of features, and second by refining the statecharts

which are used to represent the behavior associated with each feature.

7.2 Architecture

The architecture of the prototype is divided in three layers: i) User Interface, ii) Metamodel and iii) Inference Engine, as can be seen in figure 7.1. Each layer comprises a group of functionalities. The main benefit of this architecture is the separation between the user interface and the system kernel (metamodel and inference engine), in such a way that it is possible to completely reuse the inference engine, which is the main deliverable of this prototype. The ultimate goal is to implement it as an Eclipse plug-in using Eclipse Modeling Framework [SBPM08], which is a common platform used to implement modeling tools.

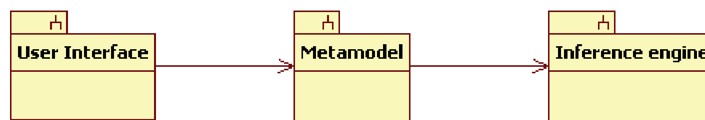


Figure 7.1: SC* Modeler architecture

User Interface

This layer allows the interaction between the system and the user, which is assumed to be a software engineer (or designer). The user interface helps the user in the construction of statecharts, feature diagrams and statecharts with variabilities.

Metamodel

This layer allows the abstraction of the Inference Engine from the User Interface.

Inference Engine

In this layer reside the main modules of the prototype, which basically perform the computation of the union of two statecharts (see section 4.2), in order to compute the statechart which implements a given product of the line. The language used to implement it is Prolog[1]. In Prolog, program logic is expressed in terms of relations, and a computation is initiated by running a query over these relations. Relations and queries are constructed using Prolog's main data type, the term. Given a query, the Prolog engine attempts to find a resolution refutation of the negated query. If the negated query can be refuted, i.e., an instantiation for all free variables is found that makes the union of clauses and the singleton set consisting of the negated query false, it follows that the original query, with the found instantiation applied, is a logical consequence of the program. For more references to programming in logic the reader is referred to [SS94, Kow79, Llo87]. The features fully implemented in this prototype are the verification of the extension relation and the calculation of the union.

[1]In particular, SWI-PROLOG [Wie09]

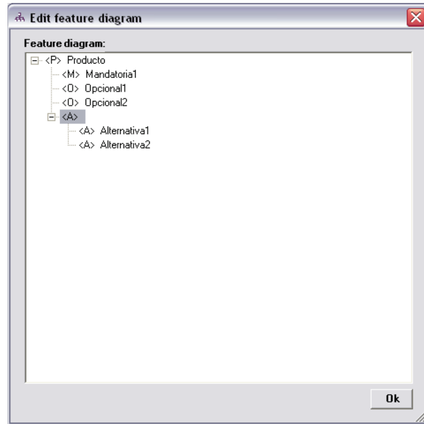


Figure 7.2: Constructing a Feature Diagram

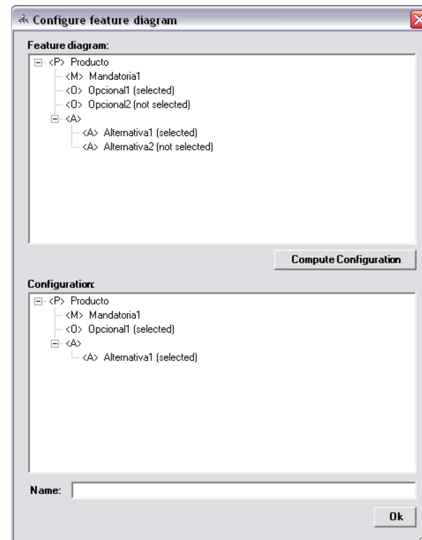


Figure 7.3: Configuring a Feature Diagram

Languages and Technologies Used

The prototype was constructed using Microsoft's .NET framework. A framework is a special case of software library, whose main distinguishing feature is that it can be extended by the user. In particular, the key benefit of this framework are its Rapid Application Development (RAD) capabilities, which allowed us an early validation of the inference engine.

As previously mentioned, the inference engine is implemented in Prolog. The two main reasons for this are:

1. Allows a direct implementation of the definitions presented in chapters 3, 4, and 5.
2. Allows non deterministic programming, needed for computing the union (section 4.2).

Moreover, the inference engine can be compiled for .NET, using P# [Coo] and for Java, using PrologCafe system [BT].

7.2.1 User Interface

The user interface is structured as follows:

- **Construction of Feature Diagrams:** Allows the construction of a Feature Diagram with mandatory, optional and alternative features (figure 7.2).
- **Configuration of Feature Diagrams:** Allows the selection of features of a Feature Diagram (optional and alternativeso), that is, a configuration (figure 7.3).
- **Construction and extension of statecharts:** Allows the construction of Basic, And and Or statecharts as well as the extension of an already constructed one (figure7.4).
- **Construction of statecharts with variabilities:** Allows the user to associate statecharts and features Features (figure 7.5).
- **Statechart derivation:** Allows the user to obtain the statechart determined by a specific Feature Diagram configuration (product), as described in section 5.2.

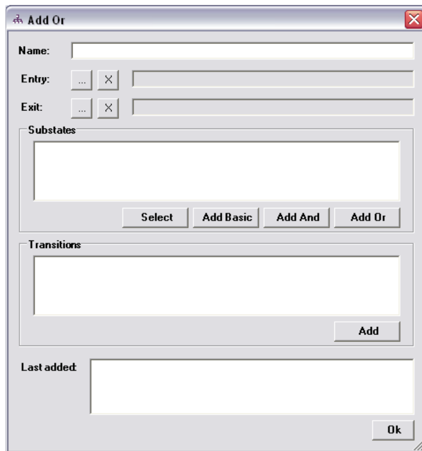


Figure 7.4: Constructing an Or statechart

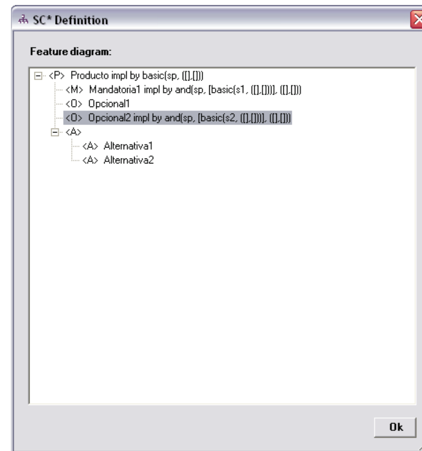


Figure 7.5: Constructing a statechart with variabilities

Its simplified metamodel is shown in figure 7.6.

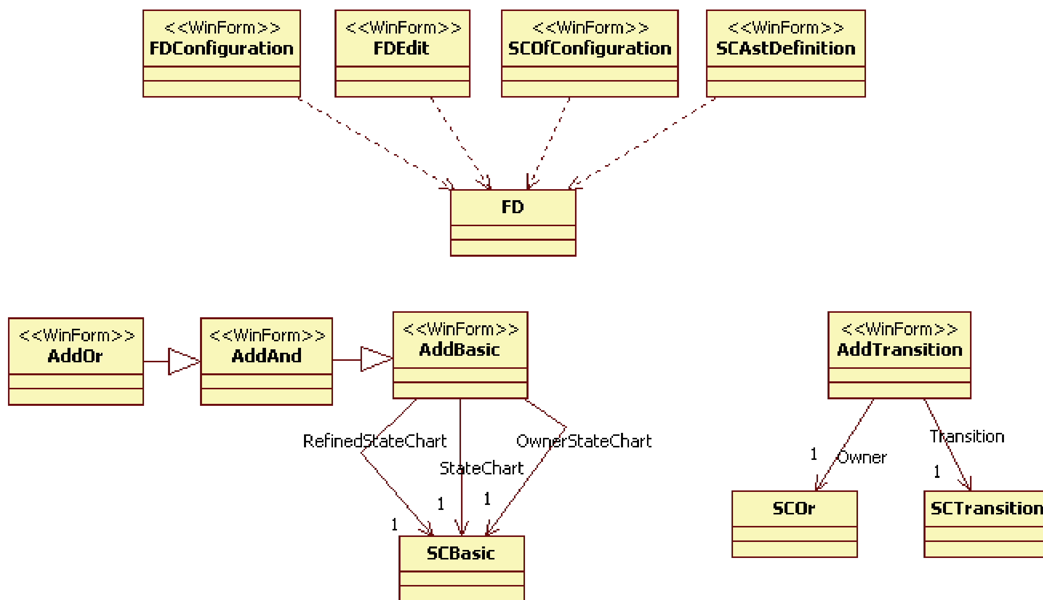


Figure 7.6: User interface diagram

7.2.2 Metamodel

Feature Diagrams

In section 5.1 the set of Feature Diagrams is defined. Its simplified metamodel is shown in figure 7.7. As before, the constraints that are not reflected in the metamodel are validated at the inference engine layer.

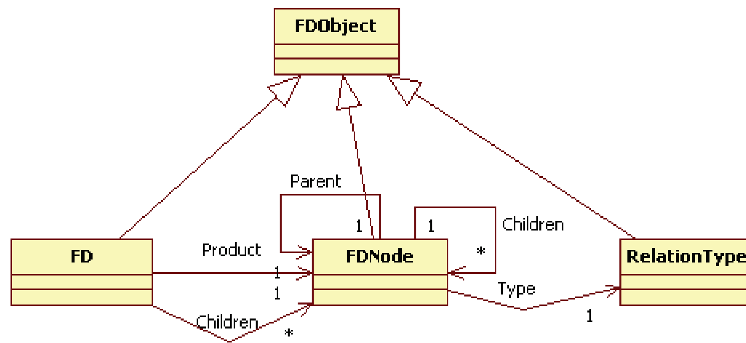


Figure 7.7: Feature Diagrams metamodel

Statecharts

In section 3.1 the set SC is inductively defined. Its simplified metamodel is shown in figure 7.8. Observe that, from the structural point of view, an Or statechart can be modeled as an And statechart but adding more components (the set of transitions, for example). The same happens between an And statechart and a basic one. The constraints that are not reflected in the metamodel are validated at the inference engine layer (for example, given the And statechart $[\hat{s}, (s_1, ..s_n)]$, the following constraint must hold: $\hat{s} \neq \hat{s}_i \ \forall_{i=1..n}$).

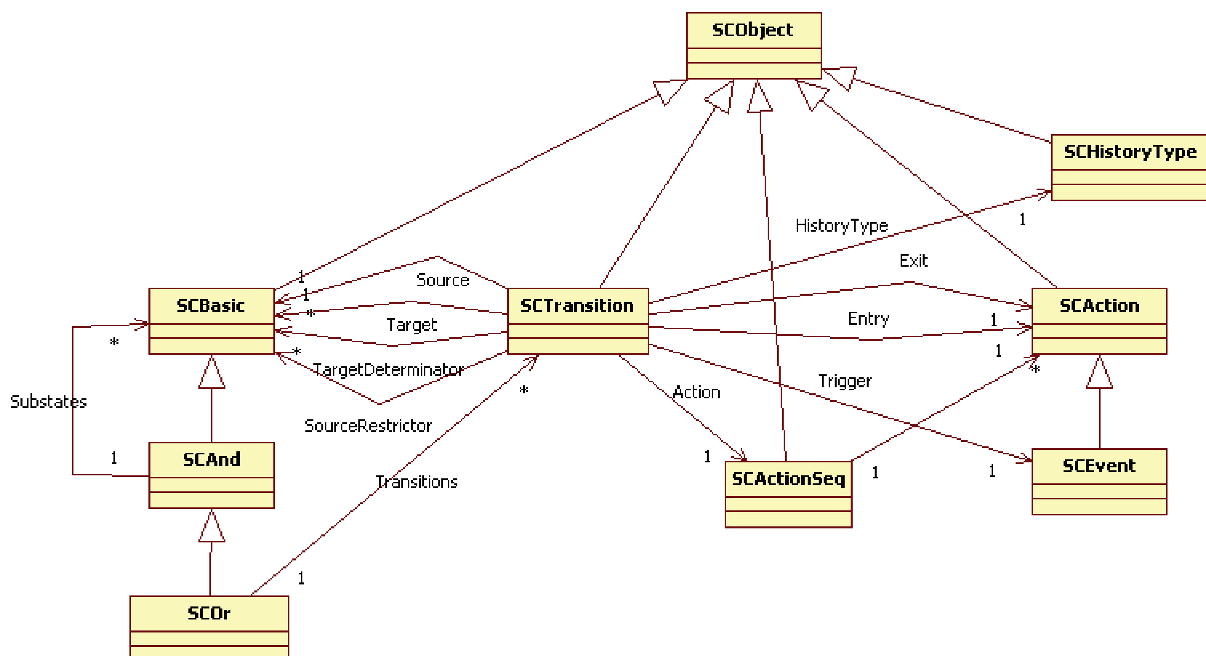


Figure 7.8: Statecharts metamodel

7.2.3 Inference Engine

The inference engine implements the intelligence of the system. The syntax used to represent statecharts is the same as the abstract syntax used in the previous section. For instance, the basic statechart $[\hat{s}, (en, ex)]$ is represented by the Prolog term `basic(St,(En,Ex))`.

The main predicates implemented are:

- 1-step Extension relation and its closure (n-step).
- Statechart union.

We define a predicate for each rule of the definition in figure 4.1. We show some examples.

1-step extension relation

Using Prolog, it is straightforward to implement the definitions of $\prec_1$ and $\cup$. A possible extension of a Basic statechart is adding to it a Basic substate, and then obtain an And statechart:

$$\frac{s' \in \mathsf{SC}_B, \quad \hat{s}' \neq \hat{s}}{[\hat{s}]\prec_1 [\hat{s}, (s')]} \ \textsf{ext-and1}$$

This is implemented in Prolog with the rule:

```
addp(basic(St, (En,Ex)), Sp, and(St, [Sp], (En,Ex))) :- is_state(Sp),
                                                         sname(Sp, Stp),
                                                         St \== Stp.
```

Another possible extension of an And statechart is extending a substate:

$$\frac{s_i \prec_1 s_i', \quad \hat{s}_i' \neq \hat{s}_j \ \forall_{j=1..i-1,i+1,..n}}{[\hat{s}, (s_1, .., s_n), l, T]\prec_1 [\hat{s}, (s_1, .., s_n)_{[s_i/s_i']}, l, T]} \ \textsf{inside-or}$$

This is implemented in Prolog with the rule:

```
adds(or(St, Ss, Ts, (En,Ex)), Sip, or(St, Ss2, Ts, (En,Ex)))
                                       :- is_state(Sip),
                                          my_select(Si, Ss, Ss1),
                                          extends(Si, Sip),
                                          my_select(Sip, Ss2, Ss1).
```

1-step union

We define the predicate `union/4` to calculate the union of two statecharts. As an example, one possible case for the union between the And statecharts is: if $s = [\hat{s}, (s_1, .., s_n), (en, ex)]$,

$$r_1 = [\hat{s}, (s_1, .., \tilde{s}_i, .., s_n), (en, ex)] \text{ with } s_i \prec \tilde{s}_i, \quad r_2 = [\hat{s}, (s_1, .., s_n, s'), (en, ex)]$$

$$\Rightarrow r_1 \cup r_2 := [\hat{s}, (s_1, .., \tilde{s}_i, .., s_n, s'), (en, ex)]$$

This is implemented in Prolog with the rule:

```
union(and(St, Ss, (En,Ex)), and(St, Ss1, (En,Ex)),
                            and(St, Ss2, (En,Ex)),
                            and(St, SsR, (En,Ex)))
                            :- my_select(Sp, Ss1, Ss),
                               my_select(Si, Ss, SsT),
                               my_select(Sip, Ss2, SsT),
                               extends(Si, Sip),
                               disj_names(Sip, Sp),
                               my_select(Sp, SsR, SsTT),
                               extends(Si, Sip),
                               my_select(Sip, SsTT, SsT).
```

N-step relations

N-step relations are implemented using a general problem solving strategy called "state space search" [SS94, Llo87, Bra00]. A state space is represented with a graph whose vertices correspond to problem states, and the solution of a given problem is reduced to finding a path in this graph. In this case, vertices correspond to statecharts and the edges are induced by the extension relation. The search strategy used is a depth-first search (DFS). This implementation of the algorithm takes time $O(|V|+|E|)$, that is, linear in the size of the graph. It also uses space $O(|V|)$ in the worst case to store the stack of vertices on the current search path as well as the set of already-visited vertices.

For example, given $s, \tilde{s} \in \mathsf{SC}$, the designer wants to determine if $s \prec_1 \tilde{s}$. The predicate `extends` tests if the concrete statechart `S2` is a refinement of `S1`. For this to hold, first, they should have the same name, then the predicate `extends` returns in `Es` one possible sequence of refinements in order to reach `S2` from `S1`. Basically, the idea is to start from `S1` and generate all the possible extensions to it until reach `S2`, or the state is "too big" to be `S2` (for example if s is an or-state with 2 substates, $\tilde{s}$ cannot have 3 substates). In some sense, those are "prune" rules.

```
extends(S1, S2, Es) :- is_state(S1),
                       is_state(S2),
                       sname(S1, N),
                       sname(S2, N),
                       extends_(S1, S2, Es).

extends_(S1, S2, [reflexive]) :- eq_states(S1, S2).
extends_(Sc, Sf, [E|Es]) :- next_state(Sc, Sf, Sn, E),
                            extends_(Sn, Sf, Es).
```

The DFS is more amenable to the recursive style of programming in Prolog. The reason for this is that Prolog itself, when executing goals, explores alternatives in a DFS fashion. The predicate `next_state` generates a possible extension of `Sc`, taking into account that the target state is `Sf`. The target state is needed in order to stop making extensions on `Sc` once it is determined that it is not possible to reach `Sf`.

Chapter 8

Case study

The formalism presented in the previous chapters, together with the prototype implemented, follow a stepwise refinement approach to software design, allowing the user to start form a simple specification of the kernel features of the product line, and to progressively add new ones to the line, specifying how each feature contributes to the behavior of the whole line. Analyzing the proposal from a practical point of view is of most importance for a fine-tuning and a careful utilization of it. In this chapter, a case study for a product line of microwave oven systems, adapted from [Gom05][1], is presented. First, the formal description of the involved statecharts are given together with the Feature Diagram of the product line. The impact of each feature in the product line is analyzed and the UML statechart with variabilities is given. Finallly, specific products for the line are defined.

8.1 Problem Description

The basic microwave oven system has input buttons for selecting Cooking Time, Start, and Cancel, as well as a numeric keypad. It also has a display to show the cooking time left. In addition, the oven has a microwave heating element for cooking the food, a door sensor to sense when the door is open, and a weight sensor to detect if there is an object in the oven. Cooking is possible only when the door is closed and when there is something inside the oven.

Options available for more advanced ovens are: a beeper to indicate when cooking is finished, a light that is switched on when the door is open and when food is being cooked, and a turntable that turns during cooking. The microwave oven displays messages to the user such as prompts and warning messages. The basic oven has a one-line display; more-advanced ovens can have multi-line displays.

The top-of-the-line oven has a recipe cooking feature, which needs an analog weight sensor in place of the basic boolean weight sensor, the multi-line display feature, and a multi-level power feature (high, medium, low) in place of the basic on/off power feature.

The example is adapted from [Gom05]. Some features are removed for space reasons but main complexity is mantained.

8.2 Use Cases

In the use case approach, functional requirements are defined in terms of actors, which are users of the system, and use cases. An actor participates in a use case. A use case defines a sequence of interactions between one or more actors and the system. The use case model

[1] This reference is the first book on modeling Software Product Lines with UML.

describes the interactions between the actor(s) and the system in a narrative form consisting of user inputs and system responses. Each use case defines the functional behavior of one part of the system without revealing its internal structure [Gom05]. In this example, commonality can be captured by two use cases, `Cook Food` and `Cook Food with Recipe`, as follows:

Use case name: Cook Food.

Summary: User puts food in oven, and microwave oven cooks food.

Actors: User (primary), Timer (secondary).

Precondition: Microwave oven is idle.

Description:

1. User opens the door, puts food in the oven, and closes the door.
2. User presses `Cooking Time` button.
3. System prompts for cooking time.
4. User enters the cooking time on the numeric keypad and presses `Start`.
5. System starts cooking the food.
6. System continually displays the cooking time remaining.
7. Timer elapses and notifies the system.
8. System stops cooking the food and displays the end message.
9. User opens the door, removes the food from the oven, and closes the door.
10. System clears the display.

Alternatives:

4. User presses `Start` when the door is closed and the cooking time is equal to zero. System does not start cooking.
6. User opens door during cooking. System stops cooking. User removes food and presses `Cancel`, or user closes door and presses `Start` to resume cooking.
6. User presses `Cancel`. System stops cooking. User may press `Start` to resume cooking. Alternatively, user may press `Cancel` again; system then cancels timer and clears display.

Postcondition: Microwave oven has cooked the food.

The `Recipe` feature has a major impact, and is described in the `Cook Food with Recipe` use case, as follows:

Use case name: Cook Food with Recipe.

Summary: User puts food in microwave oven and cooks food, using recipe.

Actors: User (primary), Timer (secondary).

Precondition: Microwave oven is idle.

Description:

1. User opens the door, puts food in the oven, and closes the door.
2. User presses the desired recipe button from the recipe buttons on the keypad.
3. System displays the recipe name. Recipe has name, power level (p), fixed time ($t1$), and time per unit weight ($t2$).
4. User presses the `Start` button.
5. System starts cooking the food for a time given by the following equation: $CookingTime = t1 + w\ t2$, where $t1$ and $t2$ are times specified in the recipe and w is the weight of the item, and the power level p is specified in the recipe.

6. System continually displays the cooking time remaining.
7. Timer elapses and notifies the system.
8. System stops cooking the food and displays the end message.
9. User opens the door, removes the food from the oven, and closes the door.
10. System clears the display.

Alternatives: 2. User presses `Start` when the door is closed and a recipe has not been chosen. System does not start cooking.

4. User presses `Cancel`. System cancels recipe and clears display.
6. User opens the door during cooking. System stops cooking. User removes food and presses `Cancel`, or user closes the door and presses `Start` to resume cooking.
6. User presses `Cancel`. System stops cooking. User may press `Start` to resume cooking. Alternatively, user may press `Cancel` again; system then cancels the recipe and clears the display.
7. If the recipe has more than one step, system completes one step, cooking food for the computed time and specified power level, and then proceeds to the next step.

8.3 Formal Description

In this section the Feature Diagram and the associated statecharts in order to formally describe the system are presented. The first one describes the variability of the product line in terms of its features, and then, for each feature, the statechart which models its behavior is defined.

8.3.1 Features

The name of the product line: $P =$ Microwave oven

Its features are[2]:

- f_{mp} = Minute Plus
- f_l = Light[3]
- $f_{l\bar{r}}$ = Light w/o Recipe
- f_{lr} = Light w Recipe
- f_t = Turntable
- f_b = Beeper
- f_{dm} = Display Unit
- f_{mld} = Multi-line Display
- f_{old} = One-line Display
- f_{ws} = Weight Sensor
- f_{bw} = Boolean Weight
- f_{aw} = Analog Weight

[2]Parameterized features in the original example are modeled as alternative features. Features with more than one parent are modeled as new feature with certain constraints over features.

[3]The impact of this feature depends on the presence of feature f_r. Each case is reflected in two optional features, $f_{l\bar{r}}$ and f_{lr}.

- f_{he} = Heating Element
- f_{oh} = One-level Heating
- f_{mh} = Multi-level Heating
- f_{pl} = Power Level
- f_r = Recipe

$\mathcal{F} = \{P, f_{mp}, f_l, f_{l\bar{r}}, f_{lr}, f_t, f_b, f_{dm}, f_{mld}, f_{old}, f_{ws}, f_{bw}, f_{aw}, f_{he}, f_{oh}, f_{mh}, f_{pl}, f_r\}$

$$\Upsilon = \langle P, \mathcal{F} - \{P\},$$
$$\{(\neg f_r \wedge f_{l\bar{r}}) \vee (f_r \wedge f_{lr} \wedge f_{aw} \wedge f_{mld} \wedge f_{mh}))\},$$
$$\{\langle P,f_{dm}\rangle, \langle P,f_{ws}\rangle, \langle P,f_{he}\rangle\},$$
$$\{\langle P,f_{mp}\rangle, \langle P,f_l\rangle, \langle P,f_t\rangle, \langle P,f_b\rangle, \langle P,f_r\rangle, \langle f_{mld},f_{511}\rangle, \langle f_{mh},f_{pl}\rangle\},$$
$$\{\langle f_l,\{f_{l\bar{r}}, f_{lr}\}\rangle, \langle f_{dm},\{f_{mld}, f_{old}\}\rangle, \langle f_{ws},\{f_{bw}, f_{aw}\}\rangle, \langle f_{he},\{f_{oh}, f_{mh}\}\rangle\}\rangle$$

Note that the first constraint in the feature diagram reflects the fact that the recipe feature needs an analog weight sensor, a multi-line display feature and a multi-level power feature. The second one reflects the fact that the impact of the feature f_l depends on the presence of the feature f_r.

Graphically:

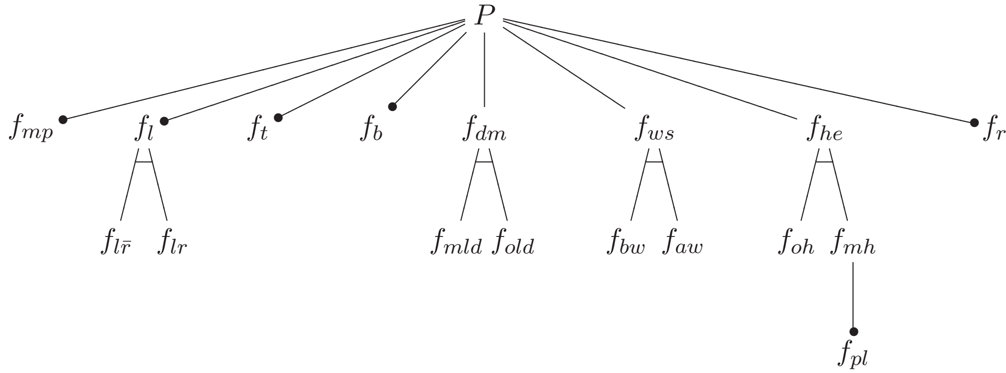


Figure 8.1: Feature Diagram Microwave oven

8.3.2 Statecharts

The statecharts describing the behavior of the microwave oven system are defined by the following:

A = { `UpdateCookingTime`, `ClearCookingTime`, `CookingTimeEntered`, `DisplayCookingTime`, `ClearDisplay`, `CookingTimeSelected`, `PromptForTime`, `TimerExpired`, `StartTimer`, `StopTimer`, `CancelTimer`, `Start`, `Cancel`, `DoorOpened`, `DoorClosed`, `ItemPlaced`, `ItemRemoved`, `DoorClosedAndTimeRemaining`, `DoorClosedAndZeroTime`, `StartCooking`, `StopCooking` }

E = A − { `StartTimer`, `CancelTimer`, `StopTimer`, `StartCooking`, `StopCooking`, `PromptForTime`, `ClearDisplay`, `DisplayCookingTime`, `ClearCookingTime` }

$\mathcal{S}$ = { `MicrowaveOvenControl`, `MicrowaveOvenSequencing`, `CookingTimeCondition`, `ZeroTime`, `TimeRemaining`, `DoorShut`, `DoorOpen`, `DoorOpenWithItem`, `Cooking`, `DoorShutWithItem`, `WaitingUser`, `WaitingCookingTime`, `ReadyToCook`} = $\{moc, mos, ctc, zt, tr, ds, do, doi, c, dsi, wu, wct, rc\}$

$$\mathcal{T} =^{4} \{\ t_{zu\hat{\ }_{CT}},\ t_{tr\hat{\ }_{TE}},\ t_{tr\hat{\ }_{CT}},\ t_{ds\hat{\ }_{C}},\ t_{ds\hat{\ }_{DO}},\ t_{do\hat{\ }_{DC}},\ t_{do\hat{\ }_{IP}},\ t_{doi\hat{\ }_{IR}},\ t_{doi\hat{\ }_{DCTR}},\ t_{rc\hat{\ }_{DO}},\ t_{doi\hat{\ }_{DCZT}},\ t_{dsi\hat{\ }_{DO}},\ t_{c\hat{\ }_{DO}},\ t_{dsi\hat{\ }_{CTE}},\ t_{c\hat{\ }_{TE}},\ t_{rc\hat{\ }_{C}},\ t_{c\hat{\ }_{C}},\ t_{rc\hat{\ }_{S}},\ t_{rc\hat{\ }_{CTE}},\ t_{wu\hat{\ }_{CTS}},\ t_{wct\hat{\ }_{C}}\ \}$$

The core statechart of the product line is:

$$moc := [\texttt{MicrowaveOvenControl}, (mos, ctc), ((),())]$$

$$ctc := [\texttt{CookingTimeCondition}, (zt, tr), 1, 1, T_{ctc}, ((),())] \text{ where}$$

$$\begin{aligned}T_{ctc} = \{\ & \langle t_{zt\hat{\ }_{UCT}}, zt, \emptyset, \texttt{UpdateCookingTime}, (), \emptyset, tr, \text{none}\rangle,\\ & \langle t_{tr\hat{\ }_{TE}}, tr, \emptyset, \texttt{TimerExpired}, (), \emptyset, zt, \text{none}\rangle,\\ & \langle t_{tr\hat{\ }_{CT}}, tr, \emptyset, \texttt{CancelTimer}, \texttt{ClearDisplay::ClearCookingTime}, \emptyset, zt, \text{none}\rangle\ \}\end{aligned}$$

$$mos := [\texttt{MicrowaveOvenSequencing}, (ds, do, doi, c, dsi, rc), 1, 1, T_{mos}, ((),())] \text{ where}$$

$$\begin{aligned}T_{mos} = \{\ & \langle t_{ds\hat{\ }_{C}}, ds, \emptyset, \texttt{Cancel}, \texttt{CancelTimer}, \emptyset, ds, \text{none}\rangle,\\ & \langle t_{ds\hat{\ }_{DO}}, ds, \emptyset, \texttt{DoorOpened}, (), \emptyset, do, \text{none}\rangle,\\ & \langle t_{do\hat{\ }_{DC}}, do, \emptyset, \texttt{DoorClosed}, (), \emptyset, ds, \text{none}\rangle,\\ & \langle t_{do\hat{\ }_{IP}}, do, \emptyset, \texttt{ItemPlaced}, (), \emptyset, doi, \text{none}\rangle,\\ & \langle t_{doi\hat{\ }_{IR}}, doi, \emptyset, \texttt{ItemRemoved}, (), \emptyset, do, \text{none}\rangle,\\ & \langle t_{doi\hat{\ }_{DCTR}}, doi, \emptyset, \texttt{DoorClosedAndTimeRemaining}, (), \emptyset, rc, \text{none}\rangle,\\ & \langle t_{rc\hat{\ }_{DO}}, rc, \emptyset, \texttt{DoorOpened}, (), \emptyset, doi, \text{none}\rangle,\\ & \langle t_{doi\hat{\ }_{DCZT}}, doi, \emptyset, \texttt{DoorClosedAndZeroTime}, (), \emptyset, dsi, \text{shallow}\rangle,\\ & \langle t_{dsi\hat{\ }_{DO}}, dsi, \emptyset, \texttt{DoorOpened}, (), \emptyset, doi, \text{none}\rangle,\\ & \langle t_{c\hat{\ }_{DO}}, c, \emptyset, \texttt{DoorOpened}, \texttt{StopTimer}, \emptyset, doi, \text{none}\rangle,\\ & \langle t_{dsi\hat{\ }_{CTE}}, dsi, \{\hat{wct}\}, \texttt{CookingTimeEntered}, \texttt{DisplayCookingTime}, \emptyset, rc, \text{none}\rangle,\\ & \langle t_{c\hat{\ }_{TE}}, c, \emptyset, \texttt{TimerExpired}, (), \{\hat{wu}\}, dsi, \text{none}\rangle,\\ & \langle t_{rc\hat{\ }_{C}}, rc, \emptyset, \texttt{Cancel}, \texttt{TimerExpired}, \{\hat{wu}\}, dsi, \text{none}\rangle,\\ & \langle t_{c\hat{\ }_{C}}, c, \emptyset, \texttt{Cancel}, \texttt{StopTimer}, \emptyset, rc, \text{none}\rangle,\\ & \langle t_{rc\hat{\ }_{S}}, rc, \emptyset, \texttt{Start}, \texttt{StartTimer}, \emptyset, c, \text{none}\rangle,\\ & \langle t_{rc\hat{\ }_{CTE}}, rc, \emptyset, \texttt{CookingTimeEntered}, \texttt{DisplayCookingTime}, \emptyset, rc, \text{none}\rangle\ \}\end{aligned}$$

$$ds := [\texttt{DoorShut}, ((),())]$$

$$do := [\texttt{DoorOpen}, ((),())]$$

$$doi := [\texttt{DoorOpenWithItem}, ((),())]$$

$$c := [\texttt{Cooking}, (\texttt{StartCooking}, \texttt{StopCooking})]$$

$$dsi := [\texttt{DoorShutWithItem}, (wu, wct), 1, 1, T_{dsi}, ((),())] \text{ with}$$

$$\begin{aligned}T_{dsi} = \{\ & \langle t_{wu\hat{\ }_{CTS}}, wu, \emptyset, \texttt{CookingTimeSelected}, \texttt{PromptForTime}, \emptyset, wct, \text{none}\rangle,\\ & \langle t_{wct\hat{\ }_{C}}, wct, \emptyset, \texttt{Cancel}, \texttt{ClearDisplay}, \emptyset, wu, \text{none}\rangle\ \}\end{aligned}$$

$$rc := [\texttt{ReadyToCook}, ((),())]$$

In figures 8.2 and 8.3 the core statechart is shown.

8.3.3 Feature Impact Description

In this section, the impact of the features on *moc* statechart is described. It can be seen that it is easy to unambiguously describe the impact of the features in the product line using the present formalism.

Minute Plus: If Minute Plus is pressed after cooking has started, then the cooking time is updated. If Minute Plus is pressed before cooking has started, then the cooking time is updated and cooking is started (assuming that the oven is ready to start cooking). This feature adds two new transitions to *mos*: from `Cooking` to `Cooking` and from `WaitingUser` to `Cooking`. Formally:

[4] The following name convention is used for transitions: The transition which has source state `DoorShut`, and event `StartTimer` will be named $t_{ds\hat{\ }_{ST}}$. This transition is represented as $t_{ds_{ST}}$

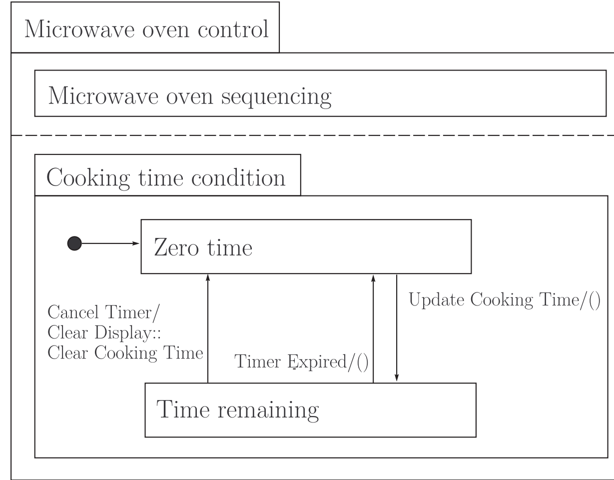


Figure 8.2: Microwave oven statechart

$mp_1 = \textsf{add-trans}(mos,\langle t_{\widehat{dsi_{MP}}},dsi,\{\hat{wu}\},\texttt{MinutePlus},\texttt{StartMinute},\emptyset,c,\textsf{none}\rangle)$

$mp_2 = \textsf{add-trans}(mos,\langle t_{\widehat{c_{MP}}},c,\emptyset,\texttt{MinutePlus},\texttt{AddMinute},\emptyset,c,\textsf{none}\rangle)$

the result is $mp = \textsf{inside-and}(moc, mos, mp_1 \cup mp_2)$

Light: The lamp is switched on whenever the door is opened or food is cooking. It is switched off whenever the door is closed or cooking stopped. This feature adds `SwitchOn` and `SwitchOff` actions in cooking states. Formally:

$mos_l = \textsf{inside-or}(mos, c, \textsf{ext-act-en-a}(c,\texttt{SwitchOn}))$

$l_1 = \textsf{ext-act-trans-a}(mos_l,t_{ds_{DO}},\texttt{SwitchOn})$

$l_2 = \textsf{ext-act-trans-a}(mos_l,t_{rc_{DO}},\texttt{SwitchOn})$

$l_3 = \textsf{ext-act-trans-a}(mos_l,t_{dsi_{DO}},\texttt{SwitchOn})$

$l_4 = \textsf{ext-act-trans-a}(mos_l,t_{do_{DC}},\texttt{SwitchOff})$

$l_5 = \textsf{ext-act-trans-a}(mos_l,t_{doi_{DCZT}},\texttt{SwitchOff})$

$l_6 = \textsf{ext-act-trans-a}(mos_l,t_{c_{TE}},\texttt{SwitchOff})$

$l_7 = \textsf{ext-act-trans-a}(mos_l,t_{c_C},\texttt{SwitchOff})$

$l_8 = \textsf{ext-act-trans-a}(mos_l,t_{doi_{DCTR}},\texttt{SwitchOff})$

the result is $l = \textsf{inside-or}(moc, mos, l_r)$, where $l_r = l_1 \cup l_2 \cup ... \cup l_8$

in the case that the recipe feature is present (see below), the result is
$l_r = \textsf{inside-and}(moc, mos, l_r \cup \textsf{ext-act-trans-a}(mos_l,t_{r_{TE}},\texttt{SwitchOff}))$

Turntable: The turntable needs to turn when food is cooking and to be stationary when food is not cooking. This feature adds `StartTurning` and `StopTurning` actions. Formally:

$t_1 = \textsf{ext-act-en-p}(c,\texttt{StartTurning})$

$t_2 = \textsf{ext-act-ex-a}(c,\texttt{StopTurning})$

the result is $t = \textsf{inside-and}(moc, mos, \textsf{inside-or}(mos, c, t_1 \cup t_2))$

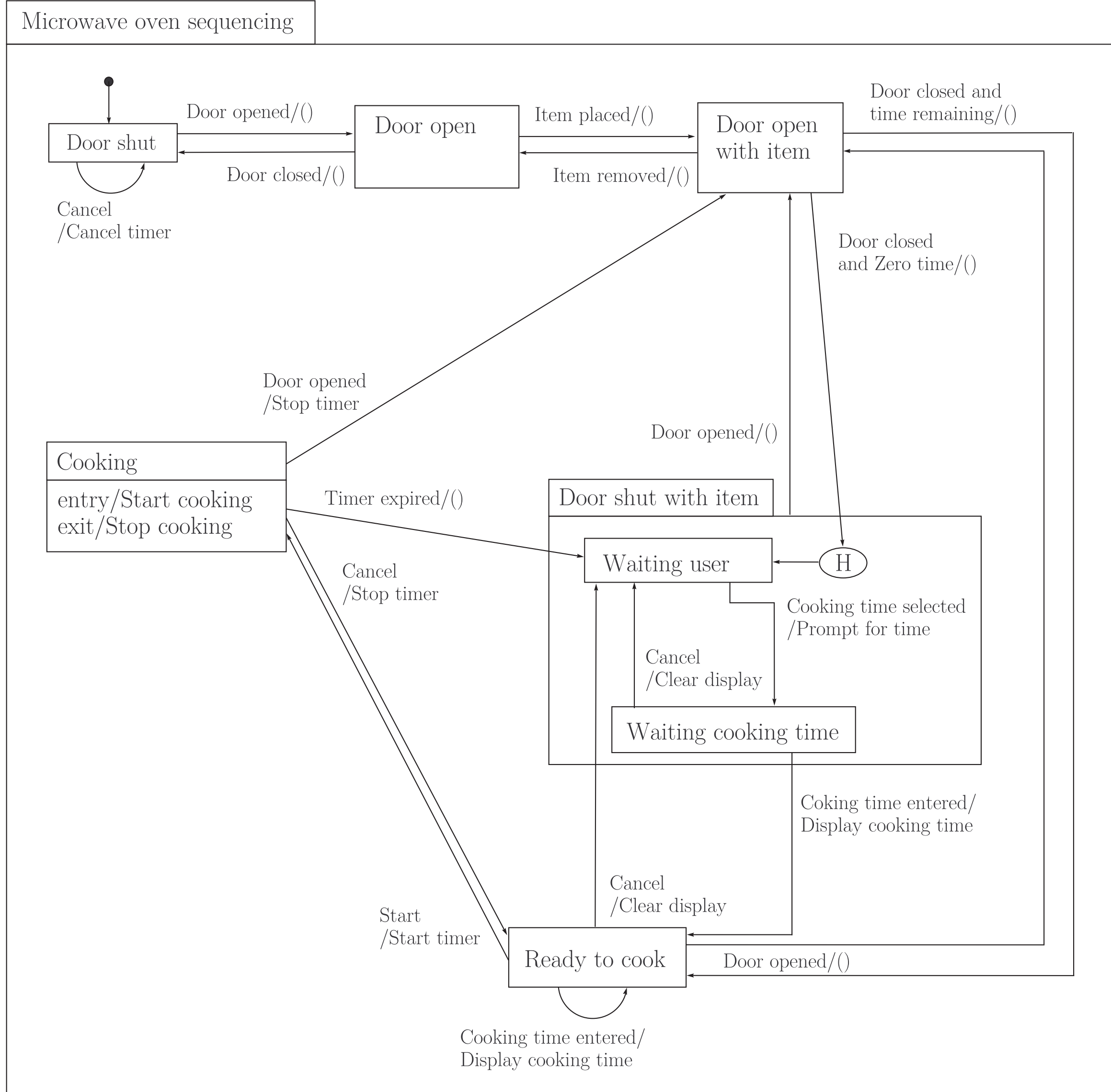


Figure 8.3: Microwave oven sequencing statechart

Beeper: The beeper is switched on when cooking has finished. This feature adds `Beep` exit action in cooking state. Formally:

the result is $b = \textsf{inside-and}(moc, mos, \textsf{inside-or}(mos, c, \textsf{ext-act-ex-a}(c, \texttt{Beep})))$

Power Level: This feature allows the user to select the power level. The impact of this feature is to add a new substate in `DoorShutWithItem` called `WaitingUserAfterPL`. The new substate is entered when the `PLSelected` event is received during the `WaitingUser` state. The actions are `UpdatePL` and `DisplayPL`. Formally:

$p_1 = \textsf{ext-or}(dsi, wupl)$, where $wupl := [\texttt{WaitingUserAfterPL}, ((), ())]$

$p_2 = \textsf{add-trans}(p_1, \langle t_{\hat{wupl}_{PLS}}, wupl, \emptyset, \texttt{PLSelected}, \texttt{UpdatePL::DisplayPL}, \emptyset, wupl, \textsf{none}\rangle)$

$p_3 = \textsf{add-trans}(p_1, \langle t_{\hat{wu}_{PLS}}, wu, \emptyset, \texttt{PLSelected}, \texttt{UpdatePL::DisplayPL}, \emptyset, wupl, \textsf{none}\rangle)$

$p_4 = \textsf{add-trans}(p_1, \langle t_{wu\hat{p}l_{CD}}, wupl, \emptyset, \texttt{ClearDisplay}, \texttt{ClearPL}, \emptyset, wu, \textsf{none}\rangle)$

$p_5 = \textsf{add-trans}(p_1, \langle t_{wu\hat{p}l_{CTS}}, wupl, \emptyset, \texttt{CookingTimeSelected}, \texttt{PromptForTime}, \emptyset, wtc, \textsf{none}\rangle)$

$p_6 = \textsf{ext-act-trans-a}(p_1, t_{wu_{CTS}}, \texttt{SetDefaultPower})$

$p_7 = \textsf{ext-act-trans-p}(mos, t_{rc_C}, \texttt{ClearPL})$

$p_8 = \textsf{inside-or}(mos, dsi, p_2 \cup ... \cup p_6)$

the result is $pl = \textsf{inside-and}(moc, mos, p_7 \cup p_8)$

Recipe: This feature adds a new substate in mos. Formally:

$r_1 = \textsf{ext-or}(mos, r)$, where $r := [\texttt{Recipe}, ((), ())]$

$r_2 = \textsf{add-trans}(r_1, \langle t_{ds\hat{i}_{RE}}, dsi, \emptyset, \texttt{RecipeEntered}, \texttt{SelectRecipe::DisplayRecipe}, \emptyset, r, \textsf{none}\rangle)$

$r_3 = \textsf{add-trans}(r_1, \langle t_{\hat{r}_C}, r, \emptyset, \texttt{Cancel}, \texttt{CancelRecipe::DisplayRecipeCanceled}, \emptyset, dsi, \textsf{none}\rangle)$

$r_4 = \textsf{add-trans}(r_1, \langle t_{\hat{r}_{TE}}, r, \emptyset, \texttt{TimerExpired}, \texttt{ClearRecipe}, \emptyset, dsi, \textsf{none}\rangle)$, in the case that light feature is present: $r_4 = \textsf{add-trans}(r_1, \langle t_{\hat{r}_{TE}}, r, \emptyset, \texttt{TimerExpired}, \texttt{ClearRecipe::SwitchOff}, \emptyset, dsi, \textsf{none}\rangle)$

$r_5 = \textsf{add-trans}(r_1, \langle t_{\hat{r}_{IR}}, r, \emptyset, \texttt{ItemRemoved}, \texttt{CancelRecipe::DisplayRecipeCanceled}, \emptyset, do, \textsf{none}\rangle)$

the result is $r = \textsf{inside-and}(moc, mos, r_2 \cup ... \cup r_5)$

8.3.4 Statechart with Variabilities

Now, the formal description of the example will be completed. In section 5.2, the set SC* of statecharts with variabilities is defined, given a Feature Diagram $\Upsilon = \langle \mathsf{L}, \mathsf{N}, \mathsf{N_C}, \mathsf{R_M}, \mathsf{R_O}, \mathsf{R_A} \rangle$, as a function Ψ: $\mathsf{N} \cup \{\mathsf{L}\} \rightarrow \mathsf{SC}$ that associates each feature of Υ with a statechart. In this example we have: $\Psi(P) = moc$[5]; $\Psi(f_{mp}) = mp$; $\Psi(f_l) = \Psi(f_{l\bar{r}}) = l$; $\Psi(f_{lr}) = l_r$; $\Psi(f_t) = t$; $\Psi(f_b) = b$; $\Psi(f_{pl}) = pl$ and $\Psi(f_r) = r$.

8.3.5 Examples of Products

In order to obtain specific products of the line defined the Feature Diagram Υ, the possible configurations of Υ are defined as the instances of the tree that are consistent with the relations amongst its features and the constraints of Υ. Feature Diagram configurations are defined in section 5.1.2. A configuration of a Feature Diagram is determined by the set of optional and alternative features that are selected for the product. Two possible valid configurations of Υ are:

$$\begin{aligned} C_1 &= \textsf{conf}_\Upsilon(\{f_{mp}, f_{lr}, f_{mld}, f_{aw}, f_{mh}, f_r\}) \\ &= \langle P, \{f_{mp}, f_{lr}, f_{mld}, f_{aw}, f_{mh}, f_r\}, \{\langle P, f_{mp}\rangle, \langle P, f_l\rangle, \langle f_l, f_{lr}\rangle, \langle P, f_{dm}\rangle, \langle f_{dm}, f_{mld}\rangle, \\ &\quad \langle P, f_{ws}\rangle, \langle f_{ws}, f_{aw}\rangle, \langle P, f_7\rangle, \langle f_{he}, f_{mh}\rangle, \langle P, f_r\rangle\}\rangle \end{aligned}$$

and

$$\begin{aligned} C_2 &= \textsf{conf}_\Upsilon(\{f_{l\bar{r}}, f_3, f_4, f_{old}, f_{bw}, f_{pl}\}) \\ &= \langle P, \{f_{l\bar{r}}, f_3, f_4, f_{old}, f_{bw}, f_{pl}\}, \{\langle P, f_l\rangle, \langle f_l, f_{l\bar{r}}\rangle, \langle P, f_3\rangle, \langle P, f_4\rangle, \langle P, f_{dm}\rangle, \\ &\quad \langle f_{dm}, f_{old}\rangle, \langle P, f_6\rangle, \langle f_{ws}, f_{bw}\rangle, \langle P, f_7\rangle, \langle f_{he}, f_{mh}\rangle, \langle f_{mh}, f_{pl}\rangle\}\rangle \end{aligned}$$

The statechart that specifies the product corresponding to the configuration C_1 (Microwave with Recipe and Minute Plus, shown in figure 8.4) is $moc \cup mp \cup l_r \cup r$, that is,

[5] the same applies for $f_{dm}, f_{mld}, f_{old}, f_{ws}, f_{bw}, f_{aw}, f_{he}, f_{oh}, f_{mh}$, since those features do not affect any statechart.

$moc := [\texttt{MicrowaveOvenControl}, (mos, ctc), ((), ())]$

ctc remains unchanged.

$mos := [\texttt{MicrowaveOvenSequencing}, (ds, do, doi, c, dsi, rc, r), 1, 1, T_{mos}, ((), ())]$ where

$$
\begin{aligned}
T_{mos} = \{ & \langle \hat{t_{ds_C}}, ds, \emptyset, \texttt{Cancel}, \texttt{CancelTimer}, \emptyset, ds, \textsf{none}\rangle, \\
& \langle \hat{t_{ds_{DO}}}, ds, \emptyset, \texttt{DoorOpened}, \texttt{SwitchOn}, \emptyset, do, \textsf{none}\rangle, \\
& \langle \hat{t_{do_{DC}}}, do, \emptyset, \texttt{DoorClosed}, \texttt{SwitchOff}, \emptyset, ds, \textsf{none}\rangle, \\
& \langle \hat{t_{do_{IP}}}, do, \emptyset, \texttt{ItemPlaced}, (), \emptyset, doi, \textsf{none}\rangle, \\
& \langle \hat{t_{doi_{IR}}}, doi, \emptyset, \texttt{ItemRemoved}, (), \emptyset, do, \textsf{none}\rangle, \\
& \langle \hat{t_{doi_{DCTR}}}, doi, \emptyset, \texttt{DoorClosedAndTimeRemaining}, \texttt{SwitchOff}, \emptyset, rc, \textsf{none}\rangle, \\
& \langle \hat{t_{rc_{DO}}}, rc, \emptyset, \texttt{DoorOpened}, \texttt{SwitchOn}, \emptyset, doi, \textsf{none}\rangle, \\
& \langle \hat{t_{doi_{DCZT}}}, doi, \emptyset, \texttt{DoorClosedAndZeroTime}, \texttt{SwitchOff}, \emptyset, dsi, \textsf{shallow}\rangle, \\
& \langle \hat{t_{dsi_{DO}}}, dsi, \emptyset, \texttt{DoorOpened}, \texttt{SwitchOn}, \emptyset, doi, \textsf{none}\rangle, \\
& \langle \hat{t_{c_{DO}}}, c, \emptyset, \texttt{DoorOpened}, \texttt{StopTimer}, \emptyset, doi, \textsf{none}\rangle, \\
& \langle \hat{t_{dsi_{CTE}}}, dsi, \{\hat{wct}\}, \texttt{CookingTimeEntered}, \texttt{DisplayCookingTime}, \emptyset, rc, \textsf{none}\rangle, \\
& \langle \hat{t_{c_{TE}}}, c, \emptyset, \texttt{TimerExpired}, \texttt{SwitchOff}, \{\hat{wu}\}, dsi, \textsf{none}\rangle, \\
& \langle \hat{t_{rc_C}}, rc, \emptyset, \texttt{Cancel}, \texttt{TimerExpired}, \{\hat{wu}\}, dsi, \textsf{none}\rangle, \\
& \langle \hat{t_{c_C}}, c, \emptyset, \texttt{Cancel}, \texttt{StopTimer::SwitchOff}, \emptyset, rc, \textsf{none}\rangle, \\
& \langle \hat{t_{rc_S}}, rc, \emptyset, \texttt{Start}, \texttt{StartTimer}, \emptyset, c, \textsf{none}\rangle, \\
& \langle \hat{t_{rc_{CTE}}}, rc, \emptyset, \texttt{CookingTimeEntered}, \texttt{DisplayCookingTime}, \emptyset, rc, \textsf{none}\rangle, \\
& \langle \hat{t_{dsi_{MP}}}, dsi, \{\hat{wu}\}, \texttt{MinutePlus}, \texttt{StartMinute}, \emptyset, c, \textsf{none}\rangle, \\
& \langle \hat{t_{c_{MP}}}, c, \emptyset, \texttt{MinutePlus}, \texttt{AddMinute}, \emptyset, c, \textsf{none}\rangle, \\
& \langle \hat{t_{dsi_{RE}}}, dsi, \emptyset, \texttt{RecipeEntered}, \texttt{SelectRecipe::DisplayRecipe}, \emptyset, r, \textsf{none}\rangle, \\
& \langle \hat{t_{r_C}}, r, \emptyset, \texttt{Cancel}, \texttt{CancelRecipe::DisplayRecipeCanceled}, \emptyset, dsi, \textsf{none}\rangle, \\
& \langle \hat{t_{r_{TE}}}, r, \emptyset, \texttt{TimerExpired}, \texttt{ClearRecipe::SwitchOff}, \emptyset, dsi, \textsf{none}\rangle, \\
& \langle \hat{t_{r_{IR}}}, r, \emptyset, \texttt{ItemRemoved}, \texttt{CancelRecipe::DisplayRecipeCanceled}, \emptyset, do, \textsf{none}\rangle \ \}
\end{aligned}
$$

where

ds remains unchanged.

do remains unchanged.

doi remains unchanged.

$c := [\texttt{Cooking}, (\texttt{StartCooking::SwitchOn}, \texttt{StopCooking})]$

dsi remains unchanged.

rc remains unchanged.

$r := [\texttt{Recipe}, ((), ())]$

and the corresponding to the configuration C_2 (Microwave with Turntable and Power Level, shown in figure 8.5) is $moc \cup l \cup t \cup b \cup pl$, that is,

$moc := [\texttt{MicrowaveOvenControl}, (mos, ctc), ((), ())]$

ctc remains unchanged.

$mos := [\texttt{MicrowaveOvenSequencing}, (ds, do, doi, c, dsi, rc), 1, 1, T_{mos}, ((), ())]$ where

$$
\begin{aligned}
T_{mos} = \{ & \langle \hat{t_{ds_C}}, ds, \emptyset, \texttt{Cancel}, \texttt{CancelTimer}, \emptyset, ds, \textsf{none}\rangle, \\
& \langle \hat{t_{ds_{DO}}}, ds, \emptyset, \texttt{DoorOpened}, \texttt{SwitchOn}, \emptyset, do, \textsf{none}\rangle, \\
& \langle \hat{t_{do_{DC}}}, do, \emptyset, \texttt{DoorClosed}, \texttt{SwitchOff}, \emptyset, ds, \textsf{none}\rangle, \\
& \langle \hat{t_{do_{IP}}}, do, \emptyset, \texttt{ItemPlaced}, (), \emptyset, doi, \textsf{none}\rangle, \\
& \langle \hat{t_{doi_{IR}}}, doi, \emptyset, \texttt{ItemRemoved}, (), \emptyset, do, \textsf{none}\rangle, \\
& \langle \hat{t_{doi_{DCTR}}}, doi, \emptyset, \texttt{DoorClosedAndTimeRemaining}, \texttt{SwitchOff}, \emptyset, rc, \textsf{none}\rangle, \\
& \langle \hat{t_{rc_{DO}}}, rc, \emptyset, \texttt{DoorOpened}, \texttt{SwitchOn}, \emptyset, doi, \textsf{none}\rangle,
\end{aligned}
$$

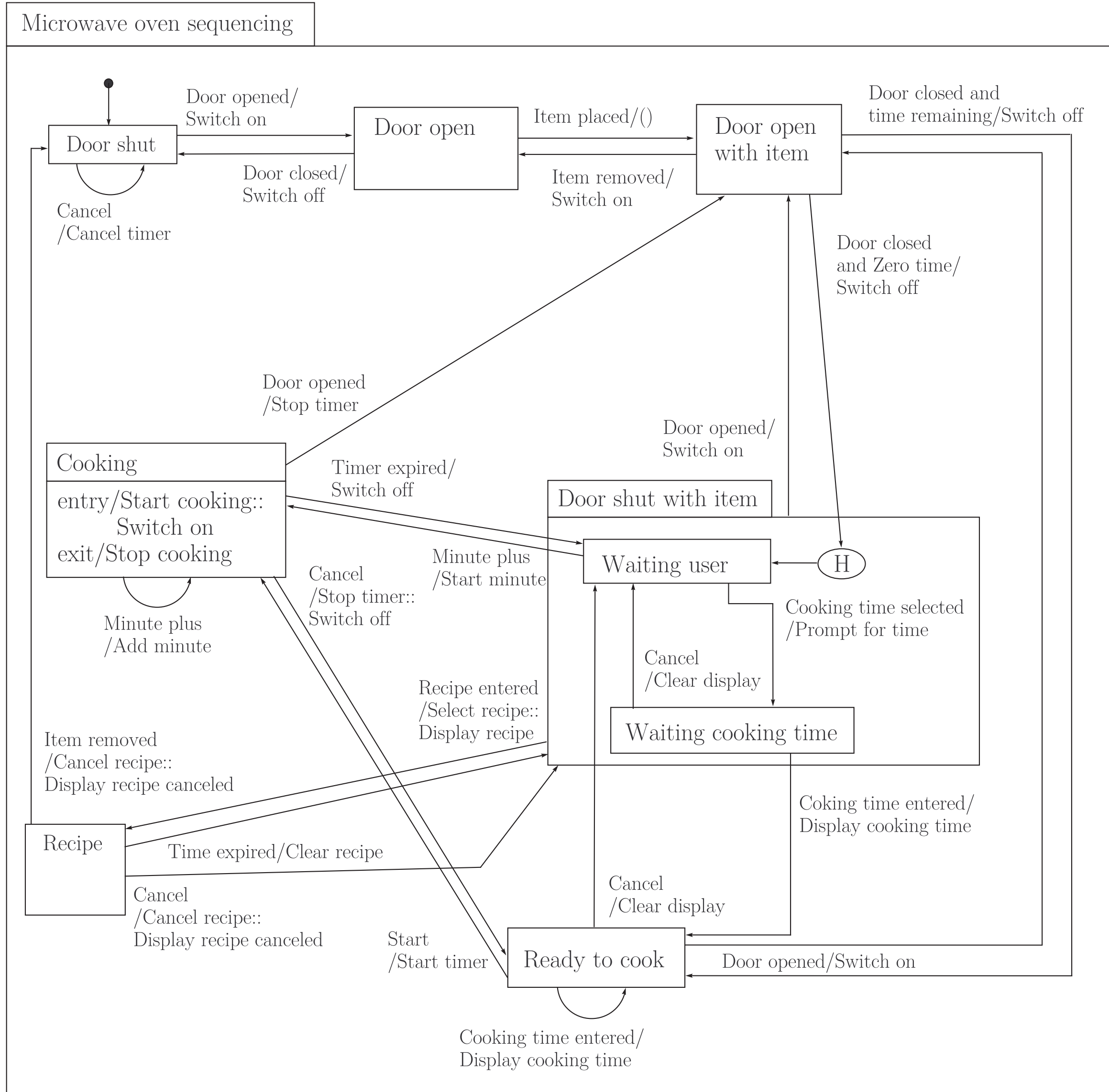


Figure 8.4: Microwave oven sequencing with Recipe and Minute Plus product statechart

$\langle t_{\widehat{doi_{DCZT}}}$,$doi$,$\emptyset$,`DoorClosedAndZeroTime`,`SwitchOff`,$\emptyset$,$dsi$,shallow$\rangle$,
$\langle t_{\widehat{dsi_{DO}}}$,$dsi$,$\emptyset$,`DoorOpened`,`SwitchOn`,$\emptyset$,$doi$,none$\rangle$,
$\langle t_{\widehat{c_{DO}}}$,$c$,$\emptyset$,`DoorOpened`,`StopTimer`,$\emptyset$,$doi$,none$\rangle$,
$\langle t_{\widehat{dsi_{CTE}}}$,$dsi$,$\{\widehat{wct}\}$,`CookingTimeEntered`,`DisplayCookingTime`,$\emptyset$,$rc$,none$\rangle$,
$\langle t_{\widehat{c_{TE}}}$,$c$,$\emptyset$,`TimerExpired`,`SwitchOff`,$\{\widehat{wu}\}$,$dsi$,none$\rangle$,
$\langle t_{\widehat{rc_C}}$,$rc$,$\emptyset$,`Cancel`,`ClearPL::TimerExpired`,$\{\widehat{wu}\}$,$dsi$,none$\rangle$,
$\langle t_{\widehat{c_C}}$,$c$,$\emptyset$,`Cancel`,`StopTimer::SwitchOff`,$\emptyset$,$rc$,none$\rangle$,
$\langle t_{\widehat{rc_S}}$,$rc$,$\emptyset$,`Start`,`StartTimer`,$\emptyset$,$c$,none$\rangle$,
$\langle t_{\widehat{rc_{CTE}}}$,$rc$,$\emptyset$,`CookingTimeEntered`,`DisplayCookingTime`,$\emptyset$,$rc$,none$\rangle$ }

ds remains unchanged.

do remains unchanged.

doi remains unchanged.

$c := [\texttt{Cooking}, (\texttt{StartTurning::StartCooking::SwitchOn}, \texttt{StopCooking::StopTurning::Beep})]$

$dsi := [\texttt{DoorShutWithItem}, (wu, wct), 1, 1, T_{dsi}, ((), ())]$ with

$$\begin{aligned}
T_{dsi} = \{ & \langle t_{\widehat{wu}_{CTS}}, wu, \emptyset, \texttt{CookingTimeSelected}, \texttt{PromptForTime::SetDefaultPower}, \emptyset, wct, \mathsf{none} \rangle, \\
& \langle t_{\widehat{wct}_{C}}, wct, \emptyset, \texttt{Cancel}, \texttt{ClearDisplay}, \emptyset, wu, \mathsf{none} \rangle, \\
& \langle t_{\widehat{wupl}_{PLS}}, wupl, \emptyset, \texttt{PLSelected}, \texttt{UpdatePL::DisplayPL}, \emptyset, wupl, \mathsf{none} \rangle, \\
& \langle t_{\widehat{wu}_{PLS}}, wu, \emptyset, \texttt{PLSelected}, \texttt{UpdatePL::DisplayPL}, \emptyset, wupl, \mathsf{none} \rangle, \\
& \langle t_{\widehat{wupl}_{CD}}, wupl, \emptyset, \texttt{ClearDisplay}, \texttt{ClearPL}, \emptyset, wu, \mathsf{none} \rangle, \\
& \langle t_{\widehat{wupl}_{CTS}}, wupl, \emptyset, \texttt{CookingTimeSelected}, \texttt{PromptForTime}, \emptyset, wtc, \mathsf{none} \rangle \ \}
\end{aligned}$$

rc remains unchanged.

$wupl := [\texttt{WaitingUserAfterPL}, ((), ())]$

This case study shows that the extension relation and the union operation constitute a simple and precise notation useful to specify the behavior of a product of the line.

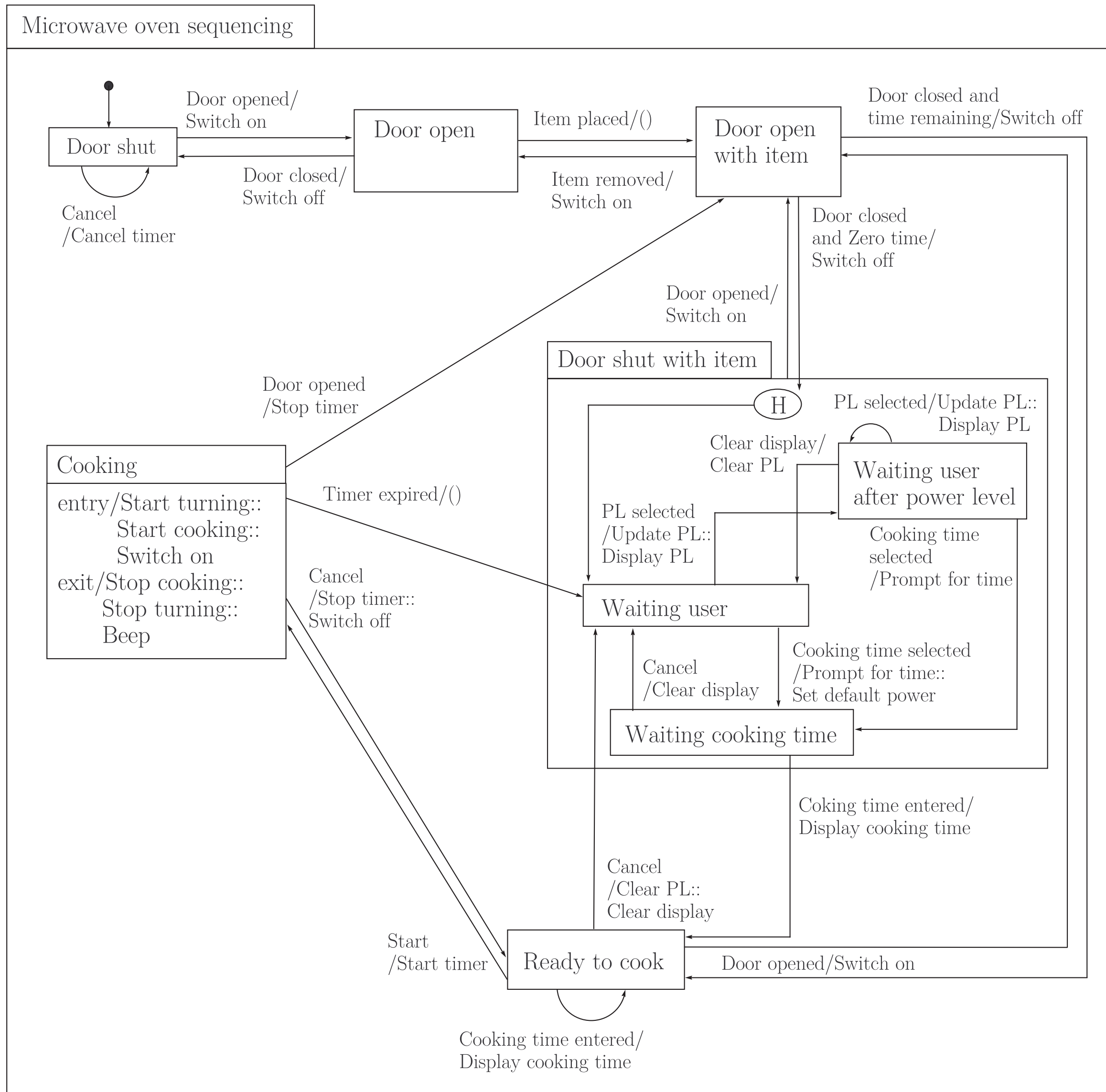


Figure 8.5: Microwave oven sequencing with Turntable and Power Level product statechart

Chapter 9

Conclusions and Further Work

In this work an extension for an UML Statecharts semantics was developed, together with an extensive analysis of their properties from a formal and practical point of view, including an implementation of the main ideas. In this chapter, the main results and conclusions are presented in section 9.1 and then future work is outlined in section 9.2.

9.1 Summary and Conclusions

The main achievement of this work is the advance of the state of the art towards a complete formal semantics of UML Statecharts with Variabilities, applied to a Software Product Line context. In particular, it shows that the structured operational semantics for UML Statecharts presented in [vdB02], can be extended to define the semantics of a family of systems. The extension of the semantics is realized in several steps. First, a partial order relation $\prec$ between statecharts is defined, in such a way that $s_1 \prec s_2$ means that s_2 enriches syntactically the substates or transitions of s_1. This determines a natural way of incremental design. Using the formalism presented in the previous chapters, the software engineer or designer can apply a stepwise refinement approach to software design. The idea is to start form a simple specification of the kernel features of the product line, and then progressively add new ones, specifying how each feature contributes to the behavior of the whole line.

The behavioral description of the kernel features of a given line is given by a root statechart. Then, in order to obtain the description of a particular product, a specification of the contribution of each additional feature must be formally described. The application of the rules of $\prec$, provides this formalism. Each rule specifies how each feature contributes to the behavior of the whole line. This kind of transformational approach is of most importance in Model-Driven Development approaches, because they aim at reducing design complexity by focusing on modeling than implementation. In MDD, an initial system model is incrementally refined by adding relevant details.

An important property of the aformentioned extension relation is that it does not permit any inconsistency within the statecharts during the extension process. We consider this formalization as a genuine contribution. Once a relation is defined among the set of statecharts, we need to combine the structure of two different extensions of a given statechart, in a union type operation. Then, given the two statecharts s_1 and s_2 that are extensions of the same statechart s (i.e., $s \prec s_1, s \prec s_2$), the union computes a new statechart $s_1 \cup s_2$ such that $s_1 \prec s_1 \cup s_2$ and $s_2 \prec s_1 \cup s_2$. Moreover, a minimum amount of extending steps as possible is desired, i.e., $\forall s_3 \in \mathsf{SC}.\ (s_1 \prec s_3 \wedge s_2 \prec s_3) \Rightarrow s_1 \cup s_2 \prec s_3$. As it is not possible to maintain consistency in all cases, it is needed to handle inconsistent statecharts, for which the concept of an overspecified statechart $\top$, is introduced.

The next step is now relate this process of statechart extension with the common and variant functionalities of a family of products. In this work, Feature Diagrams (FDs) are used to model these, presenting a formal syntax for Feature Diagrams and their configurations. With these notions, and given the description of a family of products as a FD, an UML Statecharts with Variabilities is defined as a function that associates each feature of the FD with a statechart. The mapping must comply with the hierarchical structure and the feature restrictions, i.e., the more features a product has, the richer the statechart that models it must be. In this way, it is possible to describe the effect that each feature has on the products in which it is present. This definition provides a very simple and flexible way to obtain the specification of the behavior of any configuration of the product line as the combination of the statecharts that implement all the features present in that product. We consider both the extension relation and the definition of UML Statecharts with variabilities authentic contributions.

Then, it is proved that, when a UML statechart is extended, it is still possible to perform the same semantic transitions on it as before. As a consequence of this fact, the extension relation can be considered as a behavioral refinement, thus allowing the representation of the common and variant functionalities of a family of products in conjunction with Feature Diagrams as an incremental process of statechart structure enrichment. Moreover, it is also proved that the set of actions generated by the SO semantics of a given statechart is preserved with any possible extension of it. We also consider this as a genuine contribution.

Finally, the practical point of view of this proposal is addressed, through the construction of a tool prototype, and the application of the formalism to a case study taken from the SPL literature.

9.2 Further work

In a more abstract perspective, in this work we deal essentially with sets of modeling elements and relations between them. For example, let $[\hat{s}, (s_1, ..., s_n), \hat{s}_d, \hat{s}_l, T, (en, ex)] \in \mathsf{SC}_O$. In this case the modeling elements are $\hat{s}, s, s_1, .., s_n, \hat{s}_d, \hat{s}_l, T, en, ex$. Some relations already defined are $\mathsf{name}(s) = \hat{s}$, $\mathsf{act\text{-}en}(s) = en$, $\mathsf{act\text{-}ex}(s) = ex$. The rest can be simply represented by $(s, s_1), ..., (s, s_n)$, $(s, \hat{s}_d)$, $(s, \hat{s}_l)$, and (s, T). T is in turn a relation (possibly labeled) among the substates $s_1, ..., s_2$. Then, we focus into ways of extending those models and relations, under certain conditions. The possible extensions are: i) modify an element, ii) add a new element, iii) remove an element, iv) add a new relation, v) remove a relation. In this work we did not take into account extensions of type iii) and v), and left them for future work. The relation $\prec$ can be interpreted as an extension operation that only allows extensions of type i), ii) and iv). Together with $\prec$, a complementary operation $\cup$ is defined, which "glues" different extensions of the models and relations. On the other hand, in order to model variability, features (modeling elements) are structured in a Feature Diagram (relation between features). Furthermore, the definition of SC^* relates features and statecharts. Finally, an FD can be configured into valid combinations of features.

Summarizing, we have modeling elements, relations between them, an extension operation which modifies those models and relations (under certain conditions), a join operation which is able to combine those different extensions, and finally configurations, i.e., a rule for choosing which modeling elements and relations implement a certain product of the line. We think that those ideas can be further generalized into a formal variability modeling framework which could include more UML languages for modeling behaviour, such as Use Cases, Activities, and Interactions.

Additionally, there are three more lines of further work: The first one involves extending the syntax and semantics of UML Statecharts in order to fully cover the UML 2.0 specification. Secondly, exploring the advantages that can be obtained from the relation-

ship between our formalism and lattice theory. Finally, evolve the SC* Modeler prototype into an Integrated Development Environment (IDE) plugin.

9.2.1 UML enrichments

In this section possible extensions of the syntax and semantics presented in chapter 3 are discussed. As already mentioned, the work presented is based on von der Beeck's [vdB02] semantics, which provides a reasonable coverage of UML 1.4 statecharts features. Some of the features of UML1.4 are not covered, and for obvious reasons, none of the UML 2.0.

do actions

According to the UML specification[Gro05], "do activity" is performed as long as the state that contains it is active or until the computation specified by the expression is completed. In the semantics of this work, a state can execute *do* actions of active states, or can dispatch an event from the queue. In order to take account of do activity, the complete semantics (see section 3.5) is modified adding the rule:

$$\frac{s \overset{e}{\underset{\alpha}{\longrightarrow}}_f s'}{(s,\epsilon) \Longrightarrow (s',\epsilon')} \;\mathsf{GD}\;\; (\exists (s',e) \in \mathsf{sel\text{-}act}(s), \exists(\alpha,\epsilon,\epsilon') \in \mathsf{join})$$

Where:

- $\mathsf{sel\text{-}act}$: $\mathsf{SC} \hookrightarrow \mathcal{P}((\mathsf{SC} \times A))$ models the separation of a *do* action from a given statechart.
- $\mathsf{join} \subseteq (A^* \times A^*) \times A^*$, composes 2 sequence of events.

Note that *do* actions will not change auxiliary semantics. This can be seen as an optional queue of events to execute.

Final Pseudostates

According to UML specification [Gro05], when the final pseudostate is entered, its containing region is completed, which means that it satisfies the completion condition. If the region is directly contained in a state machine and all other regions in the state machine also are completed, then it means that the entire state machine is completed. Final pseudostates cannot have exit transitions, and no entry, exit, or do actions. Intralevel transitions to final pseudostates are not allowed. For modeling this, the set of SC statecharts is augmented with a new state representing a final pseudostate, denoted with $\Diamond$. The transition space should be modified to allow a final state only in the target field. Then, the definition of SC is modified in order to only allow an or-state to have a final pseudostate. Then, in each corresponding definition a new case is added:

- $\mathsf{conf}(\Diamond) := \emptyset$ defined in section 3.2
- $\mathsf{ec\text{-}all}(\Diamond) := \{\emptyset\}$ defined in section 3.2
- $\mathsf{next}(_,_,\Diamond) := \Diamond$ defined in section 3.2
- $\mathsf{def}(\Diamond) := \Diamond$ defined in section 3.2
- $\mathsf{entry}(\Diamond) := \{\langle\rangle\}$ defined in section 3.2
- $\mathsf{exit}(\Diamond) := \{\langle\rangle\}$ defined in section 3.2

In principle, there is no need to change any semantic rule.

Entry/Exit Pseudostates

Entry and exit pseudostates (points) can be modeled as a new component into an or-statechart. Entry points have only one outgoing transition. If a transition enters a composite state through an entry point, then the entry behavior is executed before the action associated with the internal transition emanating from the entry point. In the exit from a composite state occurs throught an exit point the exit behavior of the composite state occurs after the behavior associated with the transition incoming to the exit point. In order to model that, the rule OR-1 from the SO semantics (section 3.4) must be modified. Since entry and exit pseudostates are not allowed in intralevel transitions, the transition space must be modified too. In principle, there is no need to modify any semantic rule.

9.2.2 Relations with Lattice Theory

Since it can be shown that $(\mathsf{SC}, \cup, \prec)$ is a join-semilattice with respect to the root name of the containing statechart, a plethora of theorems from this vast research area can be explored in order to find relevant applications to variability modeling [Bly10, Rom10]. For example, using the duality principle we can assert that to every statement that concerns an order on a set SC there is a dual statement that concerns the corresponding dual order on SC. Using this important theorem, we can consider for example the removal of transitions or substates from a given statechart.

9.2.3 SC* Modeler and Integrated Development Environments

As it is well known, the availability of a certain modeling technique in an Integrated Development Environment significantly increases the likelihood that software engineers integrates them into their daily work. In the case of complex software systems as Software Product Lines, the advantages of automating different analysis tasks are obvious. For UML Statecharts, several analyses can be found in the literature [CD05], for example, Syntax checking (well formedness), Consistency Checking (i.e., verifying whether a given statechart satisfies assertions on its related class diagram) and Model Checking (i.e., determining whether certain properties hold for all executions of a given statechart). Those kinds of analysis already have tool support, but they were not investigated in this work.

As is mentioned in chapter 7, the current prototype allows the construction of statecharts, feature diagrams and statecharts with variabilities. Also, it is possible to configure a Feature Diagram and then obtain the statechart that implements it. Since the layered architecture of the prototype allows to completely reuse the inference engine, the next step is to implement it as an Eclipse plug-in [SBPM08], which is a modular IDE commonly used to implement modeling tools. Although the User Interface needs to be rebuilded in order to use the graphical capabilities of the Eclipse IDE, none of the core computing modules need to be rebuilt.

Bibliography

[Bat05] D. Batory, *Feature Models, Grammars, and Propositional Formulas*, to be published at Sofware Product Line Conference (SPLC 2005), 2005.

[BCD+06] Manfred Broy, Michelle L. Crane, Juergen Dingel, Alan Hartman, Bernhard Rumpe, and Bran Selic, *2nd UML 2 semantics symposium: Formal semantics for UML*, MoDELS 2006 Workshops (T. Kühne, ed.), LNCS, vol. 4364, Springer, 2006, pp. 318–323.

[BCR] E. Borger, A. Cavarra, and E. Riccobene, *An ASM Semantics for UML Activity Diagrams.*

[BCR00] Egon Börger, Alessandra Cavarra, and Elvinia Riccobene, *An ASM semantics for UML activity diagrams*, Proceedings Algebraic Methodology and Software Technology, 8th International Conference, AMAST 2000, Iowa City, Iowa, USA, May 2000 (T. Rus, ed.), LNCS, vol. 1816, Springer, 2000, pp. 293–308.

[BCR04] Egon Brger, Alessandra Cavarra, and Elvinia Riccobene, *On formalizing uml state machines using asms*, Information and Software Technology **46** (2004), no. 5, 287 – 292, Special Issue on Software Engineering, Applications, Practices and Tools from the ACM Symposium on Applied Computing 2003.

[BDM02] Simona Bernardi, Susanna Donatelli, and José Merseguer, *From uml sequence diagrams and statecharts to analysable petrinet models*, Workshop on Software and Performance, 2002, pp. 35–45.

[Bee94] Michael von der Beeck, *A comparison of statecharts variants*, Proceedings of the Third International Symposium Organized Jointly with the Working Group Provably Correct Systems on Formal Techniques in Real-Time and Fault-Tolerant Systems (London, UK), Springer-Verlag, 1994, pp. 128–148.

[BG05] Sami Beydeda and Volker Gruhn, *Model-Driven Software Development*, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[BLMF00] Jean-Michel Bruel, Johan Lilius, Ana Moreira, and Robert B. France, *Defining precise semantics for UML*, Proceedings ECOOP 2000 Workshops, Panels, and Posters, Sophia Antipolis and Cannes, France, June 2000 (J. Malenfant, S. Moisan, and A. Moreira, eds.), LNCS, vol. 1964, Springer, 2000, pp. 113–122.

[Bly10] T.S. Blyth, *Lattices and ordered algebraic structures (universitext)*, softcover reprint of hardcover 1st ed. 2005 ed., Springer, 11 2010.

[BN98] Franz Baader and Tobias Nipkow, *Term rewriting and all that*, Cambridge University Press, New York, NY, USA, 1998.

[BP01] Luciano Baresi and Mauro Pezzè, *On formalizing uml with high-level petri nets*, Concurrent Object-Oriented Programming and Petri Nets (Gul Agha, Fiorella de Cindio, and Grzegorz Rozenberg, eds.), Lecture Notes in Computer Science, vol. 2001, Springer, 2001, pp. 276–304.

[BP05] Luciano Baresi and Mauro Pezz, *Petri nets as semantic domain for diagram notations*, Electronic Notes in Theoretical Computer Science **127** (2005), no. 2, 29 – 44, Proceedings of the Workshop on Petri Nets and Graph Transformations (PNGT 2004).

[Bra00] Ivan Bratko, *Prolog programming for artificial intelligence*, 3rd ed., Addison Wesley, 9 2000.

[BS03] Egon Boerger and Robert Staerk, *Abstract state machines: A method for high-level system design and analysis*, 1 ed., Springer, 6 2003.

[BT] Mutsunori Banbara and Naoyuki Tamura, *Prolog cafe*, http://kaminari.scitec.kobe-u.ac.jp/PrologCafe/.

[CABA09] Lianping Chen, Muhammad Ali Babar, and Nour Ali, *Variability management in software product lines: a systematic review*, Proceedings of the 13th International Software Product Line Conference (Pittsburgh, PA, USA), SPLC '09, Carnegie Mellon University, 2009, pp. 81–90.

[CAK^{+}05] Krzysztof Czarnecki, Michal Antkiewicz, Chang Hwan Peter Kim, Sean Lau, and Krzysztof Pietroszek, *Model-driven software product lines*, Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (New York, NY, USA), OOPSLA '05, ACM, 2005, pp. 126–127.

[CD05] Michelle L. Crane and Juergen Dingel, *On the semantics of uml state machines: Categorization and comparision*, In Technical Report 2005-501, School of Computing, Queen's, 2005.

[CGW05] M. Cengarle, P. Graubmann, and S. Wagner, *Semantics of uml 2.0 interactions with variabilities*, International Workshop on Formal Aspects of Component Software (FACS05), 2005.

[CGW06] Mara Victoria Cengarle, Peter Graubmann, and Stefan Wagner, *Semantics of uml 2.0 interactions with variabilities*, Electronic Notes in Theoretical Computer Science **160** (2006), no. 0, 141 – 155, ¡ce:title¿Proceedings of the International Workshop on Formal Aspects of Component Software (FACS 2005)¡/ce:title¿ ¡xocs:full-name¿Proceedings of the International Workshop on Formal Aspects of Component Software (FACS 2005)¡/xocs:full-name¿.

[CHE05a] K. Czarnecki, S. Helsen, and U. Eisenecker, *Formalizing cardinality-based feature models and their specialization*, Software Process: Improvement and Practice **10** (2005), no. 1, 7–29.

[CHE05b] Krzysztof Czarnecki, Simon Helsen, and Ulrich W. Eisenecker, *Staged configuration through specialization and multilevel configuration of feature models*, Software Process: Improvement and Practice **10** (2005), no. 2, 143–169.

[CHS00] Kevin Compton, James Huggins, and Wuwei Shen, *A semantic model for the state machine in the unified modeling language*, In Proceeding of Dynamic Behavior in UML Models: Semantic Questions, UML 2000 workshop, Springer Verlag, 2000, pp. 25–31.

[CK01] María Victoria Cengarle and Alexander Knapp, *A formal semantics for OCL 1.4*, UML 2001 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools. 4th International Conference, Toronto, Canada, October 2001, Proceedings (Martin Gogolla and Cris Kobryn, eds.), LNCS, vol. 2185, Springer, 2001, pp. 118–133.

[CK04] Mar'ıa V. Cengarle and Alexander Knapp, *OCL 1.4/5 vs. 2.0 Expressions Formal semantics and expressiveness*, Software and Systems Modeling **V3** (2004), no. 1, 9–30.

[CKTW08] María Cengarle, Alexander Knapp, Andrzej Tarlecki, and Martin Wirsing, *A Heterogeneous Approach to UML Semantics*, 2008, pp. 383–402.

[CN02] P. Clements and L. Northrop, *Software product lines: Practices and patterns*, Addison Wesley, 2002.

[Coo] John L. Cook, *P#*, http://homepages.inf.ed.ac.uk/stg/research/Psharp/.

[CR00] A. Cavarra and E. Riccobene, *Modeling the dynamics of UML behavioral diagrams*, FORTE/PSTV 2000 (Pisa), OCT 2000, Poster.

[CR09] Christine Choppy and Gianna Reggio, *A method for developing uml state machines*, Proceedings of the 2009 ACM symposium on Applied Computing (New York, NY, USA), SAC '09, ACM, 2009, pp. 382–388.

[Cza98] K. Czarnecki, *Generative programming: Principles and techniques of software engineering based on automated configuration and fragment-based component models*, Ph.D. thesis, Technical University of Ilmenau, 1998.

[Cza05] Krzysztof Czarnecki, *Mapping features to models: A template approach based on superimposed variants*, GPCE'05, volume 3676 of LNCS, 2005, pp. 422–437.

[DJPV02] Werner Damm, Bernhard Josko, Amir Pnueli, and Angelika Votintseva, *Understanding uml: A formal semantics of concurrency and communication in real-time uml*, FMCO, 2002, pp. 71–98.

[dJVI02] Merijn de Jonge, Joost Visser, and Sourcelang Impllang, *Grammars as feature diagrams*, Proceedings of Workshop on Generative Programming, 2002, pp. 23–24.

[EEPT06] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer, *Fundamentals of Algebraic Graph Transformation*, EATCS Monographs in Theoretical Computer Science, Springer, 2006.

[Esh09] Rik Eshuis, *Reconciling statechart semantics*, Science of Computer Programming **74** (2009), no. 3, 65 – 99.

[EW00] Rik Eshuis and Roel Wieringa, *Requirements-level semantics for uml statecharts*, Fourth International Conference on Formal methods for open object-based distributed systems IV (Norwell, MA, USA), Kluwer Academic Publishers, 2000, pp. 121–140.

[FG08] A. Fantechi and S. Gnesi, *Formal modeling for product families engineering*, Software Product Line Conference, 2008. SPLC '08. 12th International, sept. 2008, pp. 193 –202.

[FK05] W. B. Frakes and Kyo Kang, *Software Reuse Research: Status and Future*, Software Engineering, IEEE Transactions on **31** (2005), no. 7, 529–536.

[FKS05] Harald Fecher, Marcel Kyas, and Jens Schönborn, *Semantic issues in UML 2.0 state machines*, Tech. Report 0507, Christian-Albrechts-Universität zu Kiel, 2005.

[FS06] Harald Fecher and Jens Schönborn, *Uml 2.0 state machines: Complete formal semantics via core state machine*, FMICS/PDMC (Lubos Brim, Boudewijn R. Haverkort, Martin Leucker, and Jaco van de Pol, eds.), Lecture Notes in Computer Science, vol. 4346, Springer, 2006, pp. 244–260.

[FSKdR05] Harald Fecher, Jens Schnborn, Marcel Kyas, and Willem-Paul de Roever, *29 new unclarities in the semantics of uml 2.0 state machines*, Formal Methods and Software Engineering (Kung-Kiu Lau and Richard Banach, eds.), Lecture Notes in Computer Science, vol. 3785, Springer Berlin / Heidelberg, 2005, pp. 52–65.

[GL10] A. Gonzalez and C. Luna, *Specification of products and product lines*, CoRR **abs/1001.4436** (2010).

[GLM02] Stefania Gnesi, Diego Latella, and Mieke Massink, *Modular semantics for a uml statechart diagrams kernel and its extension to multicharts and branching time model-checking*, Journal of Logic and Algebraic Programming **51** (2002), no. 1, 43 – 75.

[Gom05] H. Gomaa, *Designing software product lines with uml*, Addison Wesley, 2005.

[Gro05] Object Management Group, *UML 2.0 Superstructure Specification*, Tech. report, August 2005.

[Gro11] ______, *Object management group*, June 2011.

[GV01] Claude Girault and Rudiger Valk, *Petri nets for system engineering: A guide to modeling, verification, and applications*, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2001.

[GZ09] Feng Guo and Meng Zhang, *Translating uml statechart diagrams to x-nets*, Information Science and Engineering (ICISE), 2009 1st International Conference on, 2009, pp. 5279 –5282.

[GZK02] Martin Gogolla, Paul Ziemann, and Sabine Kuske, *Towards an integrated graph based semantics for uml*, GT-VMT'2002 Graph Transformation and Visual Modeling Techniques, Barcelona, Spain, 11-12 October 2002 (Paolo Bottoni and Mark Minas, eds.), ENTCS, vol. 72(3), Elsevier, 2002.

[Har87] D. Harel, *Statecharts: A visual formalism for complex systems*, North-Holland (1987).

[Har97] David Harel, *Some thoughts on statecharts, 13 years later*, Computer Aided Verification (Orna Grumberg, ed.), Lecture Notes in Computer Science, vol. 1254, Springer Berlin / Heidelberg, 1997, pp. 226–231.

[Hol03] Gerard J. Holzmann, *The SPIN Model Checker: Primer and Reference Manual*, Addison-Wesley Professional, September 2003.

[HR00] D. Harel and B. Rumpe, *Modeling Languages: Syntax, Semantics and All That Stuff, Part I:The Basic Stuff*, Tech. report, Jerusalem, Israel, Israel, 2000.

[HU79] John E. Hopcroft and Jeffrey D. Ullman, *Introduction to automata theory, languages and computation*, Addison-Wesley, 1979.

[JÖ2] Jan Jürjens, *A uml statecharts semantics with message-passing*, Proceedings of the 2002 ACM symposium on Applied computing (New York, NY, USA), SAC '02, ACM, 2002, pp. 1009–1013.

[Jac96] Jonathan Jacky, *The way of z: practical programming with formal methods*, Cambridge University Press, New York, NY, USA, 1996.

[JB08] Mikol Janota and Goetz Botterweck, *Formal approach to integrating feature and architecture models*, Fundamental Approaches to Software Engineering (Jos Fiadeiro and Paola Inverardi, eds.), Lecture Notes in Computer Science, vol. 4961, Springer Berlin / Heidelberg, 2008, pp. 31–45.

[JEJ04] Yan Jin, Robert Esser, and Jörn W. Janneck, *A method for describing the syntax and semantics of uml statecharts*, Software and System Modeling **3** (2004), no. 2, 150–163.

[JKB08] Mikoláš Janota, Joseph Kiniry, and Goetz Botterweck, *Formal methods in Software Product Lines: Concepts, survey, and guidelines*, Tech. Report Lero-TR-SPL-2008-02, Lero, University of Limerick, May 2008.

[Jür02] Jan Jürjens, *Formal semantics for interacting UML subsystems*, 5th International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS 2002) (B. Jacobs and A. Rensink, eds.), Kluwer Academic Publishers, March 2002, Proceedings of the First Annual Working Conference on Network Security (I-NetSec '01), pp. 29–44.

[Jür04] ———, *Model-based security engineering with uml*, FOSAD (Alessandro Aldini, Roberto Gorrieri, and Fabio Martinelli, eds.), Lecture Notes in Computer Science, vol. 3655, Springer, 2004, pp. 42–77.

[Kan10] Kyo C. Kang, *Foda: Twenty years of perspective on feature modeling*, VaMoS (David Benavides, Don S. Batory, and Paul Grünbacher, eds.), ICB-Research Report, vol. 37, Universität Duisburg-Essen, 2010, p. 9.

[KCH+90] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson, *Feature–Oriented Domain Analysis (FODA) Feasibility Study*, Tech. Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, November 1990.

[KG10] Jens Kohlmeyer and Walter Guttmann, *Unifying the semantics of uml 2 state, activity and interaction diagrams*, Perspectives of Systems Informatics (Amir Pnueli, Irina Virbitskaite, and Andrei Voronkov, eds.), Lecture Notes in Computer Science, vol. 5947, Springer Berlin / Heidelberg, 2010, pp. 206–217.

[KKL+98] Kyo 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** (1998), 143–168, 10.1023/A:1018980625587.

[KMHC05] Soo Dong Kim, Hyun Gi Min, Jin Sun Her, and Soo Ho Chang, *Dream: a practical product line engineering using model driven architecture*, Information Technology and Applications, 2005. ICITA 2005. Third International Conference on, vol. 1, july 2005, pp. 70 – 75 vol.1.

[Kow79] Robert Kowalski, *Logic for problem solving*, Appleton & Lange, 12 1979.

[KP00] Peter J. B. King and Rob Pooley, *Derivation of petri net performance models from uml specifications of communications software*, Computer Performance Evaluation / TOOLS (Boudewijn R. Haverkort, Henrik C. Bohnenkamp, and Connie U. Smith, eds.), Lecture Notes in Computer Science, vol. 1786, Springer, 2000, pp. 262–276.

[Kru92] Charles W. Krueger, *Software reuse*, ACM Comput. Surv. **24** (1992), no. 2, 131–183.

[Krü02] I. H. Krüger, *Towards precise service specification with uml and uml-rt*, Critical Systems Development with UML – Proceedings of the UML'02 workshop (Jan Jürjens, María Victoria Cengarle, Eduardo B. Fernandez, Bernhard Rumpe, and Robert Sandner, eds.), Technische Universität München, Institut für Informatik, 2002, pp. 19–34.

[Kwo00] Gihwon Kwon, *Rewrite rules and operational semantics for model checking uml statecharts*, Proceedings of the 3rd international conference on The unified modeling language: advancing the standard (Berlin, Heidelberg), UML'00, Springer-Verlag, 2000, pp. 528–540.

[LC08] Kevin Lano and David Clark, *Semantics and refinement of behavior state machines*, ICEIS (3-1) (José Cordeiro and Joaquim Filipe, eds.), 2008, pp. 42–49.

[LC09] ———, *Axiomatic semantics of state machines*, John Wiley & Sons, Inc, 2009.

[Llo87] John W. Lloyd, *Foundations of logic programming (symbolic computation / artificial intelligence)*, 2nd ed., Springer, 9 1987.

[LMM99] Diego Latella, Istvan Majzik, and Mieke Massink, *Towards a formal operational semantics of UML statechart diagrams*, Proc. FMOODS'99, IFIP TC6/WG6.1 Third International Conference on Formal Methods for Open Object-Based Distributed Systems, Florence, Italy, February 15-18, 1999, Kluwer, 1999.

[MC02] Jos Merseguer and Javier Campos, *A compositional semantics for uml state machines aimed at performance evaluation*, Proceedings of the 6th International Workshop on Discrete Event Systems, IEEE Computer Society Press, 2002, pp. 295–302.

[MC08] Parastoo Mohagheghi and Reidar Conradi, *An empirical investigation of software reuse benefits in a large telecom product*, ACM Trans. Softw. Eng. Methodol. **17** (2008), no. 3, 1–31.

[Mci69] M. D. Mcilroy, *Mass Produced Software Components*, Tech. report, NATO, 1969.

[Mer02] S. Merz, *From diagrams to semantics – and back*, Critical Systems Development with UML – Proceedings of the UML'02 workshop (Jan Jürjens, María Victoria Cengarle, Eduardo B. Fernandez, Bernhard Rumpe, and Robert Sandner, eds.), Technische Universität München, Institut für Informatik, 2002, p. 1.

[MLG06] Mieke Massink, Diego Latella, and Stefania Gnesi, *On testing uml statecharts*, Journal of Logic and Algebraic Programming **69** (2006), no. 1-2, 1 – 74.

[MMM95] Hafedh Mili, Fatma Mili, and Ali Mili, *Reusing Software: Issues and Research Directions*, Software Engineering **21** (1995), no. 6, 528–562.

[MNB04] Sun Meng, Zhang Naixiao, and Luís Soares Barbosa, *On semantics and refinement of uml statecharts: A coalgebraic view*, SEFM, IEEE Computer Society, 2004, pp. 164–173.

[MP92] Zohar Manna and Amir Pnueli, *The temporal logic of reactive and concurrent systems*, Springer-Verlag New York, Inc., New York, NY, USA, 1992.

[MP95] ———, *Temporal verification of reactive systems: safety*, Springer-Verlag New York, Inc., New York, NY, USA, 1995.

[omg06] omg, *Meta Object Facility (MOF) Core Specification Version 2.0*, 2006.

[Par72] D. L. Parnas, *On the criteria to be used in decomposing systems into modules*, Commun. ACM **15** (1972), 1053–1058.

[PD93] R. Prieto-Diaz, *Status report: software reusability*, Software, IEEE **10** (1993), no. 3, 61 –66.

[Pet62] C. A. Petri, *Kommunikation mit Automaten*, Ph.D. thesis, Institut für instrumentelle Mathematik, Bonn, 1962.

[Pet81] James Lyle Peterson, *Petri net theory and the modeling of systems*, Prentice Hall PTR, Upper Saddle River, NJ, USA, 1981.

[PKGJ08] G. Perrouin, J. Klein, N. Guelfi, and J.-M. Jezequel, *Reconciling automation and flexibility in product derivation*, Software Product Line Conference, 2008. SPLC '08. 12th International, sept. 2008, pp. 339 –348.

[Plo81] G. D. Plotkin, *A Structural Approach to Operational Semantics*, Tech. Report DAIMI FN-19, University of Aarhus, 1981.

[Pre03] Christian Prehofer, *Plug-and-Play Composition of Features and Feature Interactions with Statechart Diagrams*, vol. 2003, 2003, pp. 43–58.

[RACH00] Gianna Reggio, Egidio Astesiano, Christine Choppy, and Heinrich Hussmann, *Analysing uml active classes and associated state machines - a lightweight formal approach*, Proceedings of the Third Internationsl Conference on Fundamental Approaches to Software Engineering: Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 2000 (London, UK), FASE '00, Springer-Verlag, 2000, pp. 127–146.

[RCA01] Gianna Reggio, Maura Cerioli, and Egidio Astesiano, *Towards a rigorous semantics of UML supporting its multiview approach*, Fundamental Approaches to Software Engineering, 4th International Conference, FASE 2001, held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001, Genova, Italy, April 2-6, 2001, Proceedings (Heinrich Hussmann, ed.), LNCS, vol. 2029, Springer, 2001, pp. 171–186.

[RJB04] James Rumbaugh, Ivar Jacobson, and Grady Booch, *Unified modeling language reference manual, (paperback), the (2nd edition)*, 2 ed., Addison-Wesley Professional, 7 2004.

[Rom10] Steven Roman, *Lattices and ordered sets*, softcover reprint of hardcover 1st ed. 2009 ed., Springer, 10 2010.

[RW99] G. Reggio and R.J. Wieringa, *Thirty one problems in the semantics of uml 1.3 dynamics*, In Conference on Object Oriented programming, Systems, Languages and Applications (OOPSLA99) Workshop Rigorous Modeling an Analysis of the UML Challenges and Limitations, 1999, p. nd.

[SB06] Colin Snook and Michael Butler, *Uml-b: Formal modeling and design aided by uml*, ACM Trans. Softw. Eng. Methodol. **15** (2006), 92–122.

[SBPM08] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks, *EMF: Eclipse Modeling Framework (2nd Edition)*, 2 ed., Addison-Wesley Professional, December 2008.

[SCH02] Wuwei Shen, Kevin Compton, and James Huggins, *A toolset for supporting uml static and dynamic model checking*, In Proc. 16th IEEE International Conference on Automated Software Engineering (ASE, IEEE Computer Society, 2002, pp. 147–152.

[SEI11a] Carnegie-Mellon University Software Engineering Institute, *Software product lines*, June 2011.

[SEI11b] ———, *Software product lines case studies*, June 2011.

[SG98] Anthony J.H. Simons and Ian Graham, *37 things that don't work in object-oriented modelling with uml*, Universitat Muchen, 1998, pp. 209–232.

[SH05] Harald Störrle and Jan Hendrik Hausmann, *Towards a formal semantics of uml 2.0 activities*, Software Engineering (Peter Liggesmeyer, Klaus Pohl, and Michael Goedicke, eds.), LNI, vol. 64, GI, 2005, pp. 117–128.

[SHT06] Pierre-Yves Schobbens, Patrick Heymans, and Jean-Christophe Trigaux, *Feature diagrams: A survey and a formal semantics*, RE, IEEE Computer Society, 2006, pp. 136–145.

[SHTB07] Pierre-Yves Schobbens, Patrick Heymans, Jean-Christophe Trigaux, and Yves Bontemps, *Generic semantics of feature diagrams*, Computer Networks **51** (2007), no. 2, 456 – 479, Feature Interaction.

[SK10] Jens Schnborn and Marcel Kyas, *Refinement patterns for hierarchical uml state machines*, Fundamentals of Software Engineering (Farhad Arbab and Marjan Sirjani, eds.), Lecture Notes in Computer Science, vol. 5961, Springer Berlin / Heidelberg, 2010, pp. 371–386.

[SS94] Leon Sterling and Ehud Shapiro, *The art of prolog, second edition: Advanced programming techniques (logic programming)*, 2 ed., The MIT Press, 3 1994.

[Stö03] Harald Störrle, *Semantics of interactions in uml 2.0*, HCC, IEEE Computer Society, 2003, pp. 129–136.

[SV08] N. Szasz and P. Vilanova, *Statecharts and variabilities*, VaMoS (P. Heymans, K. C. Kang, A. Metzger, and K. Pohl, eds.), ICB Research Report, 2008, pp. 131–140.

[SV10] N. Szasz and P. Vilanova, *Behavioral Refinements of UML-Statecharts*, Technical Report No. 13, PEDECIBA Informatica, Montevideo, Uruguay, 2010.

[SZ08] Lijun Shan and Hong Zhu, *A Formal Descriptive Semantics of UML*, Formal Methods and Software Engineering (Shaoying Liu, Tom Maibaum, and Keijiro Araki, eds.), Lecture Notes in Computer Science, vol. 5256, Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 375–396.

[vdB02] M. von der Beeck, *A structured operational semantics for uml-statecharts*, Software and Systems Modeling **1** (2002), no. 2, 130–141.

[VDS07] Tijs Van Der Storm, *Generic feature-based software composition*, Proceedings of the 6th international conference on Software composition (Berlin, Heidelberg), SC'07, Springer-Verlag, 2007, pp. 66–80.

[VG07] M. Voelter and I. Groher, *Product line implementation using aspect-oriented and model-driven software development*, Software Product Line Conference, 2007. SPLC 2007. 11th International, sept. 2007, pp. 233 –242.

[Wie09] Jan Wielemaker, *Logic programming for knowledge-intensive interactive applications*, Ph.D. thesis, University of Amsterdam, 2009, http://dare.uva.nl/en/record/300739.

[ZHJ04a] T. Ziadi, L. Helouet, and J. Jezequel, *Behaviors generation from product lines requirements*, UML2004 Workshop on Software Architecture Description and UML, 2004.

[ZHJ04b] Tewfic Ziadi, Loic Helouet, and Jean-Marc Jezequel, *Revisiting statechart synthesis with an algebraic approach*, Proceedings of the 26th International Conference on Software Engineering (Washington, DC, USA), ICSE '04, IEEE Computer Society, 2004, pp. 242–251.

[ZHJ04c] Tewfik Ziadi, Loïc Hélouët, and Jean-Marc Jézéquel, *Towards a UML Profile for Software Product Lines*, 2004.

[ZJ06] Tewfik Ziadi and Jean-Marc Jezequel, *Software product line engineering with the uml: Deriving products*, Software Product Lines (Timo Kkla and Juan Carlos Duenas, eds.), Springer Berlin Heidelberg, 2006, pp. 557–588.